LECTURE NOTES IN ELECTRICAL ENGINEERING 10

Eugenio Villar
Editor

Embedded Systems Specification and Design Languages

Selected Contributions from FDL '07

Springer

Embedded Systems Specification and Design Languages

Selected contributions from FDL'07

Lecture Notes in Electrical Engineering

Embedded Systems Specification and Design Languages
Villar, Eugenio (Ed.)
2008, Approx. 400 p., Hardcover
ISBN: 978-1-4020-8296-2, Vol. 10

Content Delivery Networks
Buyya, Rajkumar; Pathan, Mukaddim; Vakali, Athena (Eds.)
2008, Approx. 400 p., Hardcover
ISBN: 978-3-540-77886-8, Vol. 9

Unifying Perspectives in Computational and Robot Vision
Kragic, Danica; Kyrki, Ville (Eds.)
2008, 28 illus., Hardcover
ISBN: 978-0-387-75521-2, Vol. 8

Sensor and Ad-Hoc Networks
Makki, S.K.; Li, X.-Y.; Pissinou, N.; Makki, S.; Karimi, M.; Makki, K. (Eds.)
2008, Approx. 350 p. 20 illus., Hardcover
ISBN: 978-0-387-77319-3, Vol. 7

Trends in Intelligent Systems and Computer Engineering
Castillo, Oscar; Xu, Li; Ao, Sio-Iong (Eds.)
2008, Approx. 750 p., Hardcover
ISBN: 978-0-387-74934-1, Vol. 6

Advances in Industrial Engineering and Operations Research
Chan, Alan H.S.; Ao, Sio-Iong (Eds.)
2008, XXVIII, 500 p., Hardcover
ISBN: 978-0-387-74903-7, Vol. 5

Advances in Communication Systems and Electrical Engineering
Huang, Xu; Chen, Yuh-Shyan; Ao, Sio-Iong (Eds.)
2008, V, 615 p., Hardcover
ISBN: 978-0-387-74937-2, Vol. 4

Digital Noise Monitoring of Defect Origin
Aliev T.
2007, XIV, 223 p. 15 illus., Hardcover
ISBN: 978-0-387-71753-1, Vol. 2

Multi-Carrier Spread Spectrum 2007
Plass, S.; Dammann, A.; Kaiser, S.; Fazel, K. (Eds.)
2007, X, 106 p., Hardcover
ISBN: 978-1-4020-6128-8, Vol. 1

Eugenio Villar
Editor

Embedded Systems Specification and Design Languages

Selected contributions from FDL'07

Springer

Editor
Prof. Eugenio Villar
University of Cantabria
Spain

Series Editors
Sio-Iong Ao
IAENG Secretariat
37–39 Hung To Road
Unit 1, 1/F
Hong Kong
People's Republic of China

Li Xu
Zhejiang University
College of Electrical Engineering
Department of Systems Science & Engineering
Yu-Quan Campus
310027 Hangzhou
People's Republic of China

ISBN 978-1-4020-8296-2 e-ISBN 978-1-4020-8297-9

Library of Congress Control Number: 2008921989

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

FDL is the premier European forum to present research results, to exchange experiences, and to learn about new trends in the application of specification and design languages as well as of associated design and modeling methods and tools for complex, heterogeneous HW/SW embedded systems. Modeling and specification concepts push the development of new methodologies for design and verification to system level; thus providing the means for model driven design of complex information processing systems in a variety of application domains. The aim of FDL is to cover several related thematic areas and to give an opportunity to gain up-to-date knowledge in this fast evolving, essential area in system design and verification.

FDL'07 was the tenth of a series of successful events that were held in Lausanne, Lyon, Tübingen, Marseille, Frankfurt am Main, Lille and Darmstad. FDL'07 was held between September 18 and 20, 2007 at the 'Casa de Convalescència', the main Congress facilities of the 'Universitat Autònoma de Barcelona' in the city center of Barcelona, the capital city of Catalonia, Spain.

The high number of submissions to the conference this year allowed the Program Committee to prepare a high quality conference program.

The book includes a selection of the most relevant contributions based on the review made by the program committee members and the quality of the contents of the presentation at the conference. The original content of each paper has been revised and improved by the authors following the comments made by the reviewers.

FDL'07 was organized again around four thematic areas (TA) that cover essential aspects of system-level design methods and tools. The book follows the same structure:

Part I, C/C++ Based System Design, contains seven chapters covering a comparison between Esterel and SystemC, modeling of asynchronous circuits, TLM bus models, SystemC debugging, quality analysis of SystemC test benches and SystemC simulation of a custom configurable architecture.

Part II, Analog, Mixed-Signal, and Heterogeneous System Design, includes three chapters addressing heterogeneous, mixed-signal modeling, extensions to VHDL-AMS for partial differential equations and modeling of configurable CMOS transistors.

Part III, UML-Based System Specification and Design, presents six contributions comparing AADL with MARTE, modeling real-time resources, proposing model transformations to synchronous languages, mapping UML to SystemC, defining a SystemC UML profile with dynamic features and generating SystemC from StateCharts.

Part IV, Formalisms for Property-Driven Design, is composed of three chapters presenting methods for monitoring logical and temporal assertions, for transactor-based formal verification and a case study in property-based synthesis.

The collection of contributions to the book provides an excellent overview of the latest research contributions to the application of languages to the specification, design and verification of complex Embedded Systems. The papers cover the most important aspects in this essential area in Embedded Systems design.

I would like to take this opportunity to thank the member of the program committee who made a tremendous effort in revising and selecting the best papers for the conference and the most outstanding among them for this book. Specially, the four Topic Chairs, Frank Oppenheimer from OFFIS, responsible of C/C++ Based System Design, Sorin Huss from TU Darmstad, responsible of Analog, Mixed-Signal, and Heterogeneous System Design, Pierre Boulet from Lille University, responsible of UML-Based System Specification and Design and Dominique Borrione from TIMA, responsible of Formalisms for Property-Driven Design. I would like to thank also all the authors for the extra work made in revising and improving their contributions to the book.

The objective of the book is to serve as a reference text for researchers and designers interested in the extension and improvement of the application of design and verification languages in the area of Embedded Systems.

Eugenio Villar
FDL'07 General Chair
University of Cantabria

Contents

Chapter 1
How Different Are Esterel and SystemC

Jens Brandt[1] **and Klaus Schneider**[2]

Abstract In this paper, we compare the underlying models of computation of the system description languages SystemC and Esterel. Although these languages have a rather different origin, we show that the execution/simulation of programs written in these languages consists of many corresponding computation steps. As a consequence, we identify different classes of Esterel programs that can be easily translated to SystemC processes and vice versa. Moreover, we identify concepts like preemption in Esterel that are difficult to implement in a structured way in SystemC.

Keywords Synchronous Languages, SystemC, Models of Computation

1.1 Introduction

System description languages like SystemC [11, 13] and synchronous languages [1, 8] like Esterel [2, 4, 5, 12] are becoming more and more popular for the efficient development of modern hardware-software systems. The common goal of these languages is to establish a model-based design flow, where different design tasks like simulation, verification and code generation (for both hardware and software) can be performed on the basis of a single system description.

While the overall goal of SystemC and Esterel is therefore the same, there are many differences between these languages. In particular, these languages have different underlying models of computation.

As a synchronous language, the execution of an Esterel program is divided into macro steps that correspond with single reactions that are triggered by a common clock of a hardware circuit. Each macro step is divided into finitely many microsteps that are all executed in zero time and within the same variable environment.

[1]Embedded Systems Group, University of Kaiserslautern, Email: brandt@informatik.uni-kl.de

[2]Embedded Systems Group, University of Kaiserslautern,
Email: klaus.schneider@informatik.uni-kl.de

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Hence, the execution of Esterel programs are driven in a cycle-based fashion. Due to the instantaneous reaction of microsteps, causality problems may occur if actions modify variables whose values are responsible for triggering the action. In order to analyze the causality of programs, a fixpoint iteration may be performed to compute the reaction of a macrostep. It is well-known that this fixpoint iteration is the ternary simulation [6] of the corresponding hardware circuits. However, it has to be remarked that Esterel compilers usually perform this fixpoint analysis at compile time, so that (1) more efficient code is generated and (2) it is known at compile time that the iteration finally terminates with known values.

SystemC follows the discrete-event semantics that are well-known from hardware description languages like VHDL [9] and Verilog [10]. A SystemC program consists of a set of processes that run in parallel. SystemC distinguishes thereby between three classes of processes, namely 'methods', asynchronous processes and synchronous processes. Methods are special cases of asynchronous processes that do not have **wait** statements. Asynchronous processes are triggered by events, i.e., by changes of the variables on which the process depends, and they are executed as long as variable changes are seen. For this reason, the execution of the asynchronous processes is also a fixpoint computation that terminates as soon as a fixpoint of the variables' values is found. After this, the synchronous processes are executed once to complete the simulation cycle.

As can already be seen from the above coarse description, the execution of synchronous languages like Esterel and SystemC have more in common as may have been expected if only their main paradigms were considered. Clearly, there are also many differences between these languages:

- The semantics of Esterel is given in form of a very concise structural operational semantics that can be directly used as specification of a simulator. In contrast, the semantics of SystemC is only given in terms of natural language (except for some attempts like [14, 15, 22]).
- In Esterel, most statements are reduced to a small core language for which hardware and software generation is available. No significant blow-up is obtained by this reduction (this is due to the so-called write-things-once-principle). In contrast, SystemC is an extension of C++ by constructs required to describe hardware systems like built-in concurrency, wait/interrupt mechanisms, and special data types like bitvectors. As a consequence, hardware synthesis is only available for a rather small subset of SystemC.
- Esterel offers comfortable preemption statements for aborting or suspending other statements. A first attempt towards preemption statements will be obtained by SystemC's **watching** statement, that does however not yet reach the power of Esterel's abortions.
- Esterel has special variables that model events. These variables take a default value unless they are assigned another value in the current macrostep.
- Esterel has a fully orthogonal set of statements. In particular, concurrency is an ordinary statement that can be combined with all other statements, while in SystemC programs consist of a set of processes that implement sequential code.

- SystemC offers different kinds of abstraction levels like ‘untimed functional’, ‘timed functional’, ‘bus cycle-accurate’, and ‘cycle-accurate’ modeling to support refinements from transaction levels down to register-transfer level descriptions.

Hence, there are also many differences between these languages. Some of theses difference may, however, only exist in the current versions of these languages and may disappear in later versions.

In this paper, we outline the differences and similarities of synchronous languages like Esterel and SystemC. In particular, we define classes of systems that can be easily described in both languages in a way that allows one to structurally translate these descriptions into each other. This is the result of the similarities that we have identified between the two languages. On the other hand, the differences we will outline in the following may be interesting for those who work on later versions of both languages. With this paper, we therefore hope to stimulate the discussion between the communities of SystemC and synchronous languages.

The rest of the paper is organized as follows: In the next section, we describe the languages SystemC and Esterel in more detail. In Section 1.3, we compare the execution of Esterel and SystemC programs in more detail and show that there are some correspondences. These correspondences give rise to define simple classes of programs that can be easily translated between both languages. In addition to this, we list differences between the two languages that lead to problems for the translation between the languages in Section 1.4. Finally, we conclude with a short summary in Section 1.5.

1.2 Esterel and SystemC

In this section, we give a rough overview of the main concepts and paradigms of Esterel and SystemC. Section 1.3 outlines then some similarities between the languages, while Section 1.4 outlines some major differences.

1.2.1 Esterel

Esterel [2, 4, 5, 12] is a synchronous language [1, 8] that can be used both for hardware and software synthesis. As usual for synchronous languages, the computation of an Esterel program is divided into single reactions. Within each reaction, new inputs are read and new outputs are generated for these inputs with respect to the current state of the program. Moreover, the reaction determines the next state of the program that is used in the next reaction step.

The state of the program is determined by the current values of the variables and the current set of active control flow locations of the program. Control flow locations are statements like the **pause** statement where the control flow may rest for one unit of time.

Since Esterel statements include the parallel statement $S_1 \parallel S_2$, it may be the case that the control flow may rest at several control points at the same point of time.

Besides the usual statements like assignments, conditionals, sequences and loops, Esterel provides also many statements to implement complex concurrent behaviours. In particular, there are four kinds of abortion statements that run some Esterel code while observing an abortion condition in each macro step. If the condition holds, then the code is aborted and the abortion statement terminates. Other preemptive statement are suspension statements that suspend the execution of an Esterel statement if a given condition holds in a macro step.

It is very important that variables do not change during the macro step, i.e., all microsteps are viewed to be executed in zero time. Therefore, all microsteps are executed at the same point of time with the same variable environment. As a consequence, the values of the variables are uniquely defined in each macro step.

Due to the instantaneous reaction, synchronous programs may suffer from causality conflicts [3, 18, 19]. These causality conflicts occur if an assignment modifies the value of a variable that is responsible for the execution of the assignment. Compilers check the causality of a program at compile time with algorithms that are essentially the same as those used for checking the speed independence of asynchronous circuits via ternary simulation [6]. These algorithms essentially consist of a fixpoint computation that starts with unknown values for the output variables, and successively replaces these unknown values by known ones. While this analysis is usually done at compile time, we consider this fixpoint iteration in the following as being part of the execution that is performed within a macro step. This is done to outline similarities to the execution of SystemC programs.

Several generations of compilation techniques [7, 20, 24] have been developed for Esterel that can be used to generate hardware circuits at the gate level as well as software in sequential programming languages from the same Esterel program. Moreover, some of these compilation techniques have already been formally verified [16, 17].

1.2.2 SystemC

SystemC is a language used for the simulation of complex hardware software systems. SystemC simulations may run up to 1,000 times faster than corresponding descriptions given in hardware description languages like VHDL and Verilog due to the higher level of abstraction that is used in SystemC. SystemC supports several levels of abstractions, which allows one to describe completely untimed systems down to cycle-accurate descriptions of hardware circuits at the gate level.

SystemC is not a self-contained language; instead, it is a class library for the well-known C++ programming language [23]. SystemC extends C++ by typical data types used for hardware circuits like bitvectors and arithmetic on binary numbers with a specified bit-width. Moreover, SystemC offers concurrency in a similar way as hardware description languages, i.e., SystemC programs consist of a set of concurrent processes. To this end, SystemC features three different kinds of process types:

- Methods are triggered by signal events. Methods are entirely executed in a single simulation cycle and correspond to combinatorial circuits, i.e., their execution does not take time.
- Asynchronous processes are also triggered by signal events, but they may not be entirely executed within one simulation cycle. Instead, the control may stop at wait statements and may rest there until it is triggered by a new event.
- Synchronous processes are triggered by clocks. Like asynchronous processes, synchronous processes may not be entirely executed within one simulation cycle, and the control may stop at wait statements of the process. In contrast to asynchronous processes, the execution of synchronous processes is only triggered by the next clock event.

Although SystemC shares with VHDL the discrete-event based semantics, it does not have the possibility to assign signal assignments with delay. Hence, progress of time is only driven by clocks. Between these simulation steps, the output signal updates that are due to assignments of synchronous processes are not committed immediately. Instead, they are deferred to the beginning of the next simulation step. In contrast to this, local variables can always be modified, and the effect becomes visible without delay.

1.3 Similarities Between SystemC and Esterel

From a general point of view, SystemC and synchronous languages are based on different models of computation: While SystemC has a discrete-event based semantics, synchronous languages rely on a global clock triggering the overall execution, i.e., a cycle-based semantics. However, a closer look to the features of each language reveals that there are similarities that allow us to define a common core of both languages. In particular, the integration of synchronous processes in SystemC provides some hooks to establish links between both worlds.

First of all, consider when variables change. In Esterel, there are immediate and delayed assignments that change the value of a variable immediately or only at the next macrostep. Similarly, the asynchronous processes of SystemC immediately update variable values, while the assignments of synchronous processes are committed only before the next simulation cycle.

However, synchronous languages follow the paradigm of perfect synchrony, i.e. all variable assignments are made simultaneously in a macrostep. This has the consequence that all variables can only have one value per clock cycle.

The perfect synchrony also has another consequence. Programs may not be executed in the order given by the programmer. Data dependencies of the program may require to execute the statements in a completely different order than specified by the programmer. Thus, the simulator does not simply execute the code of a synchronous program once, but it reiterates the execution and deduces from iteration to iteration the value of more signals until no further values can be deduced. As an example, consider a sequence in which the following operations are

performed: assign a a value depending on b and c, then assign b a value depending on c and finally assign c some constant value. Without reordering (which is generally not applicable), the simulator needs three iterations to compute all outputs.

Figure 1.1 compares the execution of a SystemC and an Esterel program. There are apparent similarities in the execution of both types of programs: Both of them start with the determination of the time of the next step. In SystemC, this is determined by the next changing clock signal, whereas the logical time of Esterel just requires to wait for the next clock tick. Then, both simulators enter an iteration. In SystemC, the methods and asynchronous processes are executed as long as some signals change. In Esterel, there is a similar condition. The outputs are computed in a fixpoint operation that incrementally computes all signals of the current step. Subsequently, actions with immediate effects are executed, which are followed by the updates caused by delayed actions. Both in SystemC and Esterel, these updates stem from the previous clock cycle. If the iterative part of a step is finished, the SystemC simulator executes the synchronous processes that have been scheduled in the previous step. Similarly, the Esterel compiler executes the code at the currently active control flow locations with the determined signal values. Both programs now schedule processes and produce delayed actions for the next clock cycle.

This comparison shows that Esterel and the synchronous part of SystemC basically follow the same overall execution scheme. However, as already mentioned above, the execution of the individual processes is generally different. SystemC processes are sequential and thus, they are executed as specified by the programmer, while Esterel is inherently parallel, and its execution follows the data dependencies. Hence, a synchronous program cannot be directly translated to SystemC, since causility problems must be considered.

```
function SystemCStep()
  // determine next changing clock signal
  do
    // execute activated sc_methods and sc_threads
    // update outputs of sc_methods and sc_threads
    // update outputs of previous sc_threads
  while (signals change);
  // execute scheduled sc_threads

function EsterelStep()
  // proceed to next macrostep
  do
    // execution: determining current signals
    // update immediate outputs;
    // update delayed outputs of previous step
  while (fixpoint not reached);
  // execution: prepare next macrostep
```

Fig. 1.1 Comparing the execution of SystemC and Esterel programs

Nevertheless, for most programs that appear in real-world applications, the problems are not as difficult as outlined before. With restricting to a subclass of synchronous programs that covers most important applications, a direct structural mapping is possible. Basically, the following classes can be distinguished.

- *Programs that contain only delayed action*: No problems occur if programs that solely contain delayed actions are translated. For this class of programs, the iterative part is almost redundant: Only the outputs from the previous step must be committed once. The fixpoint iteration can be completely omitted, since no actions manipulate them in the course of the current step and thus, they are all known in advance. The actual execution of the program code is done after the loop, which is equivalent to SystemC synchronous processes.
- *Programs requiring only one fixpoint iteration*: In principle, the condition for the input set of programs does not have to be as strict as described above: The only thing that must be guaranteed is that a single iteration of is enough to determine the output values. In this case, the execution scheme is again analogous and a directly translated program shows the same behavior. Hence, programs may contain immediate actions which must be however set before their usage in the step. In particular, the individual threads of a program have to be executed in the right order that respects inter-thread data dependencies.
- *All other programs*: The set of programs for a translation does not need to be restricted at all. The causality analysis of synchronous programs can be simulated in SystemC with the help of asynchronous processes. Each program fragment (i.e. either equations or the result of the compilation method presented in the next section) is wrapped in an asynchronous process that contains all used variables in its sensitivity list. Like this, its execution is triggered each time a value changes. Note that Esterel program that are not causally correct, may result in SystemC programs that have a nonterminating simulation cycle: Asynchronous processes may infinitely often trigger each other and thus, simulate an oscillating wire in the circuit design they represent.

1.4 Differences Between SystemC and Esterel

The previous section showed that synchronous processes in SystemC and Esterel programs share a common core, which can comprehend many practical systems. While most elements of SystemC can be mapped more or less directly to Esterel, some problems arise for the other way around due to the rich set of control flow statements Esterel provides.

First, problems occur due to the Esterel's orthogonal use of parallelism. Since parallel and sequential code can be arbitrarily mixed in Esterel but not in SystemC, threads in synchronous programs must be reorganized. Second, there are many preemption constructs in Esterel, which are all based on some primitive abortion

and suspension statements. As SystemC does not provide preemption, this part must be also removed before a translation to SystemC code.

Recently, we developed a new compilation scheme for our Esterel-variant Quartz, which compiles programs to an intermediate code, which represents a small synchronous programming languages without complicated control flow statements [20, 21]. The basic building block of this format is a job. Such a job $J = (x,S_x)$ is a pair, where x is a label and S_x a code fragment. These jobs resemble synchronous processes in SystemC. The overall idea of compilation is as follows: In a first step, for each control flow location ℓ of the program, a job (ℓ,S_ℓ) is computed that has to be executed if the control flow resumes the execution from location ℓ.

Definition 1. [**Job Code Statements**] The following list contains the job code statements. S, S_1, and S_2 are also job code statements, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and λ is a lock variable:

- *nothing* (empty statement)
- $y = \tau$ and $next(y) = \tau$ (assignments)
- $init(x)$ (initialize local variable)
- $schedule(\ell)$ (resumption at next reaction)
- $reset(\lambda)$ (reset a barrier variable)
- $fork(\lambda)$ (immediately fork job λ)
- $barrier(\lambda,c)$ (try to pass barrier λ)
- $if(\sigma)\ S_1\ else\ S_2$ (conditional)
- $S_1;S_2$ (sequence)

The atomic statements *nothing*, $y = \tau$, and $next(y) = \tau$ have the same meaning as in ordinary synchronous programs. The meaning of conditionals and sequences is also the same. The statement $init(x)$ replaces a local variable declaration. The $schedule(\ell)$ statement inserts the job corresponding to control flow location ℓ to the schedule of the next step. The statements $reset(\lambda)$, $fork(\lambda)$, and $barrier(\lambda, c)$ are used to implement concurrency based on *barrier synchronization*. The statement $barrier(\lambda,c)$ first increments the integer variable λ and then compares it with the constant c. If $\lambda \geq c$ holds, it immediately terminates, so that a further statement S can be executed in a sequence $barrier(\lambda,c);S$. If $\lambda < c$ holds, the execution fails, so that the code behind the barrier is not yet executed. Executing $reset(\lambda)$ simply resets $\lambda = 0$. The statement $fork(\lambda)$ immediately executes the job $\mathcal{S}_\lambda$ that is associated with λ.

As explained in detail in [20], the compilation of preemption statements first computes the normal execution that is performed when no abortion takes place. Then, as a post-processing, the potential preemption behavior is added to all jobs. To this end, each location ℓ inside the abort statement's body the corresponding job S_ℓ is protected by the abortion and suspension guards so that the statements are not executed if a preemption condition hols.

Figure 1.2 contains a small example that illustrates how **Quartz** code can be translated to SystemC. The lower left part of the figure lists the job code of the module and the right hand-side shows how it can be used for the translation to

```
module Wait(event a, b, r, &o)
  loop{
      ℓa : await(a); || ℓb : await(b);
      emit next(o);
      ℓr : await(r);
  }

ℓ0 : reset(λ1);
   schedule(ℓa);
   schedule(ℓb);
ℓa : if(¬a) schedule(ℓa) else fork(λ1);
ℓb : if(¬b) schedule(ℓb) else fork(λ1);
ℓr : if(r){
       reset(λ1);
       schedule(ℓa);
       schedule(ℓb);
   } else
       schedule(ℓr);
λ1 : barrier(λ1, 2);
   emit next(o);
   schedule(ℓr);
```

```
void Wait ::ℓ0(){
  r.write(false);ℓa.write(true);ℓb.write(true);
}
void Wait ::ℓa(){
  while(true){
    wait_until(ℓa.delayed());
    wait_until(a.delayed());
    ℓa.write(false);
  }}
void Wait ::ℓb(){
  while(true){
    wait_until(ℓb.delayed());
    wait_until(b.delayed());
    ℓb.write(false);
  }}

void Wait ::ℓr(){
  wait_until(r.delayed());
  ℓa.write(true);ℓb.write(true);
}
void Wait ::λ1(){
  wait_until(!ℓa.delayed()&&!ℓb.delayed());
  r.write(true);
  wait();
  r.write(false);
  ℓr();
}
```

Fig. 1.2 Module *Wait in Quartz, Job Code (left) and SystemC* (right)

SystemC. The fine-grained parallelism used by the threads of ℓ_a and ℓ_b is mapped to coarse-grained parallelism of SystemC.

Figure 1.3 shows another example, which extends the previous one. It illustrates how preemption statements are removed by the compilation into JobCode. The translation to SystemC is not affected by this part, as only additional conditional statement are inserted, which do not pose significant problems.

Obviously, the various kinds of preemption statements in Esterel are powerful and convenient components used to program complex concurrent behaviors. The translation as performed by the Job code compilation is a solution, but it would be better if SystemC could benefit from the same programming possibilities as imperative synchronous languages. While the watching statement provides rudimentary abortion functionality, a complete support of all abortion and suspension variants would be desirable.

Moreover, fine-grained parallelism would be a second important extension of SystemC, from which a translation of imperative synchronous programs would benefit.

```
module ABRO(event a,b,r,&o)
  loop{
    abort{
      ℓa : await(a); || ℓb : await(b);
      emit next(o);
    } when(r);
    ℓr : await(r);
  }
```

```
ℓ0 : reset(λ1)
  schedule(ℓa);
  schedule(ℓb);
ℓr : if(¬r){
    reset(λ1);
    schedule(ℓa);
    schedule(ℓb);
  } else
    schedule(ℓr);
λ1 : barrier(λ1,2);
  emit next(o);
  schedule(ℓr);
ℓa : if(r){
    reset(ℓ1);
    schedule(ℓa);
    schedule(ℓb);
  } else
    if(¬a) schedule(ℓa) else fork(ℓ1);
ℓb : if(r){
    reset(ℓ1);
    schedule(ℓa);
    schedule(ℓb);
  } else
    if(¬b) schedule(ℓb) else fork(ℓ1);
```

Fig. 1.3 Module ABRO in Quartz and Job Code

1.5 Summary

In this paper, we identified similarities of the execution of SystemC and Esterel programs. Despite their different paradigms, we identified a class of programs that can be easily translated from one language to the other. Furthermore, we investigated language features that cause problems in a transformation process: In particular, preemption and fine-grained parallelism as in Esterel programs were identified as major differences, which might be interesting extensions of SystemC.

References

1. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
2. G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge, USA 1998.
3. G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/, July 1999.
4. G. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *LNCS*, pages 389–448, Springer Pittsburgh, PA 1984.

5. G. Berry and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79:1293–1304, 1991.
6. J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, New York 1995.
7. S. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, 2003.
8. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Dordrecht, 1993.
9. IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*. New York, 2000. IEEE Std. 1076–2000.
10. IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. New York, 2001. IEEE Std. 1394–2001.
11. IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*. New York, USA, December 2005. IEEE Std. 1666–2005.
12. IEEE Computer Society. *IEEE Standard Esterel Language Reference Manual*. New York, USA, to appear 2007. IEEE Std. 1778.
13. Open SystemC Initiative. *SystemC Version 2.1 User's Guide*, 2005.
14. W. Müller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiel. The simulation semantics of SystemC. In *Design, Automation and Test in Europe (DATE)*, pages 64–70, IEEE Computer Society Munich, Germany, 2001.
15. W. Müller, J. Ruf, and W. Rosenstiel. An ASM based SystemC simulation semantics. In W. Müller, J. Ruf, and W. Rosenstiel, editors, *SystemC – Methodologies and Applications*, pages 97–126, Kluwer Dordrecht, 2003.
16. K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Springer Hampton, VA, 2002.
17. K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.
18. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, 2005.
19. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Application of Concurrency to System Design (ACSD)*, pages 106–115, IEEE Computer Society St. Malo, France, 2005.
20. K. Schneider, J. Brandt, and E. Vecchié. Efficient code generation from synchronous programs. In F. Brewer and J.C. Hoe editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 165–174, IEEE Computer Society Napa Valley, CA, 2006.
21. K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In *IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*, Springer Braga, Portugal, 2006.
22. R.K. Shyamasundar, F. Doucet, R. Gupta, and I.H. Krüger. Compositional reactive semantics of SystemC and verification in RuleBase. In *Workshop on Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 2007.
23. B. Stroustrup. *The C++ Programming Language*. Series in Computer Science. Addison-Wesley, Reading, MA, 1986.
24. J. Zeng, C. Soviani, and S.A. Edwards. Generating fast code from concurrent program dependence graphs. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–181, ACM Washington, DC, 2004.

Chapter 2
Timed Asynchronous Circuits Modeling and Validation Using SystemC

Cédric Koch-Hofer and Marc Renaudin

Abstract ASC is a SystemC library designed for modeling asynchronous circuits. In order to respect the semantic of asynchronous circuits, the synchronization primitives of ASC rely on SystemC immediate notification. In this paper we present a time model which allows us to properly trace ASC processes activity. This time model is not restricted to ASC and could be used to model asynchronous circuits using a CSP based modeling language. Moreover, this time model can be used for validating timed models of circuits mixing synchronous and asynchronous parts. This time model is therefore used for designing the tracing facilities of ASC. This paper also presents a patch of the OSCI SystemC simulator allowing to properly validate ASC models. As relevant examples, two versions of the Octagon interconnect are modeled and verified using the ASC library.

Keywords Asynchronous Circuits, SystemC, Time Model, Simulation and Validation

2.1 Introduction

With advances in digital VLSI technologies, asynchronous design styles are becoming more and more popular. The intrinsic properties of asynchronous circuits are well adapted to new interconnects paradigms like "Network on Chip" [1] (NoC). An Asynchronous circuit [2] use a local handshaking protocol to synchronize data transfers between its components. Therefore, there are no longer any problems with NoC clock management, and the integration of cores with different clock frequencies is properly managed [3]. Moreover, asynchronous NoCs take advantage of the benefits of asynchronous circuits such as low power consumption, communication robustness…

TIMA laboratory, 46 Av. Félix Viallet, 38031 Grenoble, France
Email: {cedric.koch-hofer, marc.renaudin}@imag.fr

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Today, the lack of tools for the design of asynchronous circuits are the principal inhibitors for their adoption [4]. Two families of tools are available. The first family of tools uses graphical description as input. Examples of such tools are: Petrify [5], minimalist [6], 3D [7]. These kinds of tools allow the production of very efficient small circuits; nevertheless they can not be used for designing complex systems like NoC. The second family of tools uses programming languages as input. Examples of such languages are: CHP [8], Balsa [9] and Tangram [10]. These modeling languages do not support standard CAD tools and are not adequate to model synchronous circuits. However, these facilities are required for the design of an Asynchronous NoC interconnecting the synchronous components of a "Globally Asynchronous Locally Synchronous" [11] (GALS) "System on Chip" (SoC). Moreover, the design frameworks associated with these modeling languages do not allow us to properly codesign the hardware and software part of a SoC.

In order to leverage these problems, we have developed ASC [12], an extension of the SystemC [13] language for modeling asynchronous circuits. The semantic of ASC is based on CSP [14]. Indeed, an ASC model is composed of a set of concurrent processes communicating via synchronous point-to-point channel. This SystemC library also includes a set of operators and statements for accurately modeling the basic components of an Asynchronous Network on Chip.

The standard tracing facilities defined by SystemC are based on changes of variable values between different simulation times or between two different delta-cycles [13]. By this way, it is not possible to trace several communications occurring over an ASC channel if they happen in the same delta-cycle. For example, Fig. 2.1 illustrates what happens if standard tracing facilities of SystemC are used for tracing the variable *var*. In this example the *foo::process* sends two chars to the *bar::process*. Nevertheless, only the last change of value can be recorded by the standard tracing facilities of SystemC. Indeed, the ASC channels use immediate notification to synchronize their connected processes and therefore multiple communications can be executed during a delta-cycle over the same channel. Thus, standard SystemC tracing facilities only display the last change of value and can not be used for validating ASC models.

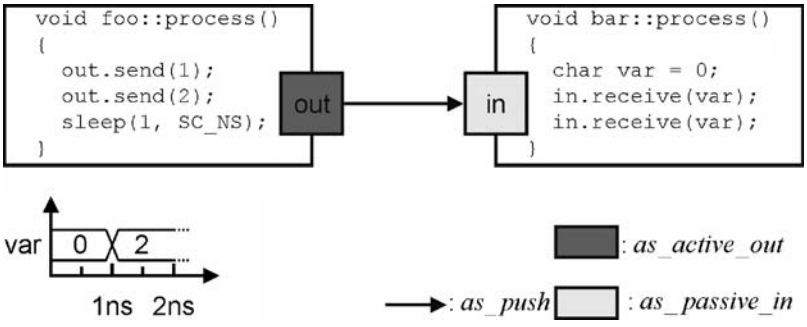


Fig. 2.1 Trace with SystemC tracing facilities

An obvious solution resolving this problem could be adding latencies in ASC channels. However, this solution adds extra dependencies on the order of execution of the processes, not allowing proper ASC processes delay insensitivity checking In fact, tracing activities of such a distributed system requires using a time model not based on a single common clock.

The "Lamport clocks" [15] is a time model commonly used for synchronizing activities of distributed systems. In this time model each process has its own local clock. The messages exchanged by the processes are used for synchronizing their local clocks. In this paper we present a time model, called AST (Asynchronous SystemC Time), based on "Lamport clocks" allowing proper tracing of ASC processes activity. More generally, this time model can be used for tracing activities of any models of asynchronous circuits specified with a modeling language based on CSP.

Previous works [16–18] on timing models for asynchronous circuits use models at the gate level. They are used to perform static analysis of latencies of the circuit components. For example, they use min-max algorithm, Monte-Carlo simulation… for checking that the delay limits are respected. Thus, these models manipulate very low level abstraction entities like signals. These models of time are therefore not suited to handle high level language constructs like processes, channels…

A SystemC framework based on "Lamport clocks" time model is presented in [19]. However, they do not use it for tracing activities of channels but for improving simulation speed. Indeed, the "Lamport clocks" time model is used in this framework to efficiently manage the execution of the SystemC processes on a distributed simulation platform. The execution of these processes is synchronized according to the time stamp of the packets received by the processes.

The ASC library enables us to model any class of asynchronous circuits (QDI [20], micro-pipeline [21]…). Thus, we want to be able to validate any kind of asynchronous circuits modeled using ASC. For properly checking the delay insensitivity of an ASC model of a Delay Insensitive (DI) asynchronous circuit, all the valid scheduling of the processes should be tested. Hopefully, the specification of the SystemC scheduler [13] is non-deterministic. However, the system has to be simulated with a particular implementation of the scheduler. For example, the SystemC reference simulator [22] is deterministic. In order to leverage this problem, we have developed a patch for this simulator allowing a non-deterministic scheduling of the processes.

This paper also presents how the AST time model was used to define the tracing facilities of ASC. To demonstrate the relevance of this approach, this paper finally presents how ASC is used to model and validate two versions of an asynchronous Octagon interconnect [23].

The organization of the paper is as follows. Section 2.2 presents the AST time model. The ASC library is introduced in Section 2.3. As illustrative examples, Section 2.4 describes the two ASC versions of Octagon interconnect. Finally, conclusions and future works on the ASC library are presented in Section 2.5.

2.2 Time Model

A model of asynchronous circuits based on CSP is a set of processes which communicate with one another by exchanging messages via synchronous point-to-point channels. In this kind of distributed system, all processes are running concurrently and it is therefore difficult (even impossible) to say that one of two events occurred first. As in [15], our goal is to adapt and extend the relation "happened before" in order to define a partial ordering of the events happening in such a system. At the end, we want to be able to assign a coherent time stamp to each event occurring in this kind of system. For example, Fig. 2.2 shows different events occurring when executing a CSP model of an asynchronous circuit composed of three processes (P_0, P_1 and P_2). Figure 2.2 also illustrates the time stamps associated to these events. The different kind of events and their relationship are described formally in Sub-section 2.2.1. The rules for computing the time stamp of these events are presented in Sub-section 2.2.2.

A nice property of this time model is that it can be easily extended. For example in Sub-section 2.2.3 we present an extension of this time model allowing interfacing these asynchronous clocks with the clock of a synchronous circuit.

2.2.1 Partial Ordering

In the AST time model, the execution of a CSP model of an asynchronous circuit is represented by a set of processes $P = \{p_0, p_1 \ldots\}$ and a set of channels $CH = \{ch_0, ch_1 \ldots\}$. A process p_i is defined by the sequence of events $p_i = (e_0, e_1 \ldots)$ occurring in this process during its execution. The first event of a process p_i is its "initialization"

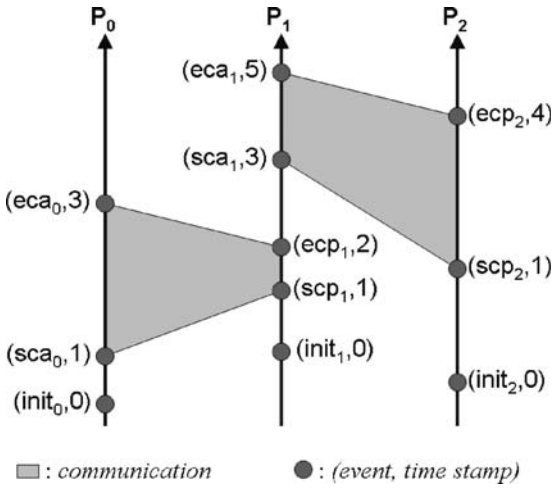


Fig. 2.2 Time stamping of CSP processes' events

$init_i$. When a process p_i terminates, its last event is its "termination" end_i. A channel ch_k is specified by a couple $ch_k = (p_i, p_j)$ where p_i and p_j are the processes using ch_k. p_i and p_j are connected to ch_k by an active port and by a passive port, respectively. It can be noticed that in this time model the direction of the data communicated through the channel is not relevant. A communication $c = \{sca_i, scp_j, ecp_j, eca_i\}$ between two processes p_i and p_j over a channel $ch_k = (p_i, p_j)$ is defined by the following four events:

- sca_i and eca_i: beginning and termination of the communication c for the process p_i
- scp_j and ecp_j: beginning and termination of the communication c for the process p_j

A process p_j connected to a channel $ch_k = (p_i, p_j)$ can probe it. The probing action is atomic and generates one, and only one, of the two following events:

- pp_j: this event, called a "positive probe", happens if the process p_i has initiated a communication on the channel ch_k.
- np_j: this event, called a "negative probe", occurs if the process p_i does not initiated a communication on the channel ch_k.

In our formalism a task $t_{i,l}$ is a sequence of instructions of a process p_i. In standard CSP, it is not possible to perform a set of tasks in parallel in the same process. In order to leverage this restriction most of the modeling language for asynchronous circuits based on CSP defines a parallel composition operator. This operator enables concurrently execution of a set of tasks $T_i = \{t_{i,0}, t_{i,1} \ldots\}$ in the same process p_i. Each task $t_{i,l}$ is concurrently executed by a sub-process p_m. The main process p_i is blocked until the termination of all these sub-processes. Execution of this composition operator is characterized by the following two events:

- cti_i: this event, called "concurrent tasks initialization", occurs when a set of concurrent tasks are triggered by process p_i.
- ctt_i: this event, called "concurrent tasks termination", happens when all the sub-processes triggered by process p_i for executing a set of concurrent tasks have terminated.

The sequence of events (e_i, e_i'…) defining a process p_i respects the order of occurrences of its events. We are assuming that two events in the same process can not happen at the same time, and therefore the sequence of events (e_i, e_i'…) respects a total ordering. However, our goal is to define an ordering relation on the set $E = \{e, e' \ldots\}$ of all the events. For this purpose, we define the "happened before" relation $\prec : E \rightarrow E$. This relation is defined by the following conditions:

(C0) If e_i and e_i' are events in the same process, and e_i occurs before e_i', then $e_i \prec e_i$'

(C1) $\forall\, e, e', e'' \in E, (e \prec e' \wedge e' \prec e'') \Rightarrow e \prec e''$

(C2) If $\{sca_i, scp_j, ecp_j, eca_i\}$ is a communication between processes p_i and p_j, then $sca_i \prec ecp_j$, $scp_j \prec ecp_j$ and $ecp_j \prec eca_i$

(C3) If $c = \{sca_i, scp_j, ecp_j, eca_i\}$ is a communication between processes p_i and p_j, and pp_j is a "positive probe" by the process p_j of the communication c, then $sca_i \prec pp_j$ and $pp_j \prec scp_j$

(C4) If np_j is a "negative probe" done by the process p_j on the channel $ch_k = (p_i, p_j)$, and $\{sca_i, scp_j, ecp_j, eca_i\}$ is a communication between processes p_i and p_j via the channel ch_k, then $ecp_j \prec np_j$ or $np_j \prec sca_i$

(C5) If $init_m$ is the initialization event of a sub-process p_m created for performing a concurrent task $t_{i,l}$, and cti_i is the "concurrent task initialization" generated by the composition operator which triggered the process p_m, then $cti_i \prec init_m$

(C6) If end_m is the termination event of a sub-process p_m created for performing a concurrent task $t_{i,l}$, and ctt_i is the "concurrent task termination" generated by the composition operator which triggered the process p_m, then $end_m \prec ctt_i$

Obviously, in this kind of system an event can not occur before itself $\forall\, e \in E, \neg(e \prec e)$. Moreover, the asymmetric property of the relation $\prec$ can be easily demonstrated. Thus, the relation $\prec$ defines a strict partial ordering of E.

2.2.2 Computing Time of Events

The AST time model associates a time stamp to each event. The value of this time stamp is defined by a function $\mathrm{clk} : E \rightarrow \mathbb{N}$ respecting the strict partial ordering $\prec$. This last function represents the logical time of the system and it is defined according to the logical local time of each process. The logical time of a process p_p is defined by a function $\mathrm{clk}_p : E_p \rightarrow \mathbb{N}$ where $E_p \subseteq E$ is the set of all the events occurring in p_p. The time stamp $\mathrm{clk}(e_p) = \mathrm{clk}_p(e_p)$ of an event $e_p \in E_p$ occurring in a process p_p is computed with the help of the following computation rules:

(R0) If $e_p = \varnothing$ is an event which has never happened, then $\mathrm{clk}(\varnothing) = 0$

(R1) If $e_p = init_i$ is the initialization of the process p_i and this process is not a sub-process, then $\mathrm{clk}_i(init_i) = 0$

(R2) If $e_p = init_m$ is the initialization of the process p_m, and this process is a sub-process triggered by the event cti_i of the process p_i, then $\mathrm{clk}_m(init_m) = \mathrm{clk}_i(cti_i) + 1$

(R3) If $e_p = end_i$ is the last event of the process p_i, and le_i is the last event occurring in p_i before end_i, then $\mathrm{clk}_i(end_i) = \mathrm{clk}_i(le_i) + 1$

(R4) If $e_p = sca_i$ is the beginning of a communication performed by a process p_i over a channel $ch_k = (p_i, p_j)$, and np_j is the last negative probe of the process p_j of the channel ch_k, and le_i is the last event occurring in p_i before sca_i, then $\mathrm{clk}_i(sca_i) = \max(clk_i(le_i), \mathrm{clk}_j(np_j)) + 1$

(R5) If $e_p = scp_j$ is the beginning of a communication performed by a process p_j over a channel (p_i, p_j), and le_j is the last event occurring in p_j before scp_j, then $\mathrm{clk}_j(scp_j) = \mathrm{clk}_j(le_j) + 1$

(R6) If $e_p = ecp_j$ is the end of a communication $\{sca_i, scp_j, ecp_j, eca_i\}$ performed by a process p_j over a channel (p_i, p_j), then $\text{clk}_j(ecp_j) = \max(\text{clk}_i(sca_i), \text{clk}_j(scp_j)) + 1$

(R7) If $e_p = eca_i$ is the end of a communication $\{sca_i, scp_j, ecp_j, eca_i\}$ performed by a process p_i over a channel (p_i, p_j), then $\text{clk}_i(eca_i) = \text{clk}_j(ecp_j) + 1$

(R8) If $e_p = pp_j$ is a positive probe of the communication $\{sca_i, scp_j, ecp_j, eca_i\}$ performed by a process p_j, and le_j is the last event occurring in p_j before pp_j, then $\text{clk}_j(pp_j) = \max(\text{clk}_j(le_j), \text{clk}_i(sca_i)) + 1$

(R9) If $e_p = np_j$ is a negative probe performed by a process p_j of a channel $ch_k = (p_i, p_j)$, and $\{sca_i, scp_j, ecp_j, eca_i\}$ is the last communication on the channel ch_k, and le_j is the last event occurring in p_j before np_j, then $\text{clk}_j(np_j) = \max(\text{clk}_j(le_j), \text{clk}_j(ecp_j)) + 1$

(R10) If $e_p = cti_i$ is the initialization of a composition operator, and le_i is the last event occurring in p_i before cti_i, then $\text{clk}_i(cti_i) = \text{clk}_i(le_i) + 1$

(R11) If $e_p = ctt_i$ is the termination of a composition operator, and $p_m, p_{m+1} \ldots$ are the sub-processes created by this composition operator, and end_m, $end_{m+1} \ldots$ are the last events occurring in these sub-processes, then $\text{clk}_i(ctt_i) = \max(\text{clk}_m(end_m), \text{clk}_{m+1}(end_{m+1})\ldots) + 1$

As explained in [15], the function $\text{clk} : E \rightarrow \mathbb{N}$ respects the strict partial ordering $\prec$ if the following condition is respected:

Clock Condition. $\forall\ e, e' \in E, (e \prec e' \Rightarrow \text{clk}(e) < \text{clk}(e'))$

The lack of space does not allow us to give details of the proof of the *clock condition*. Briefly, this proof consists of proving that all the conditions defining the relation $\prec$ are respected by the previous computation rules defining the function $\text{clk} : E \rightarrow \mathbb{N}$.

2.2.3 Interfacing with Synchronous World

One of the goals on ASC is to model circuits composed of asynchronous and synchronous components. For being able to trace activities of such system, our time model must be able to take into account its synchronous time. In order to leverage this problem, we extend the set of processes P of the AST time model with a new process $p_\Delta \in P$. This process represents the system's global clock of the synchronous parts. Indeed, at the end of each global clock cycle, an event ge_Δ occurs in the process p_Δ.

To preserve the coherency of the $\prec$ relation we extend it with the following condition:

(C7) If e_i is an event occurring in p_i and ge_Δ is an event occurring in p_Δ before e_i, then $ge_\Delta \prec e_i$

(C8) If ge_Δ is an event occurring in p_Δ and e_i is an event occurring in p_i before ge_Δ, then $e_i \prec ge_\Delta$

The computation rules of the time stamp also have to be updated. Firstly, we add the following computation rule:

(R12) If $e_p = ge_\Delta$ is the end of a clock cycle happening in the process p_Δ, and le_Δ, le_0, le_1 … are the last event happening in processes p_Δ, p_0, p_1 … of $P = \{p_\Delta, p_0, p_1 \ldots\}$ before ge_Δ, then $\text{clk}_\Delta(ge_\Delta) = \max(\text{clk}_\Delta(le_\Delta), \text{clk}_0(le_0), \text{clk}_1(le_1)\ldots) + 1$

Secondly, we update the rules (R2) to (R11) for taking into account the local time of the process p_Δ. For example, for the rule (R2), if we take the same hypothesis and if le_Δ is the last event occurring in p_Δ before $init_m$, then $\text{clk}_m(init_m) = \max(\text{clk}_i(cti_i), \text{clk}_\Delta(le_\Delta)) + 1$. The other rules (R3) to (R11) are updated in the same way.

2.3 ASC Library

An ASC model is composed of a set of ASC **modules** interconnected via predefined ASC **ports** and ASC **channels**. New methods and operators are also defined by ASC enabling **parallel communication** and **non-deterministic choice**.

The ASC tracing facilities are composed of several functions. These functions are used to trace communications and events happening in the ASC channels. The generated output trace file can not be directly used by the standard CAD tools, but it can be converted in standard VCD trace file [24] with the *ast2vcd* tool we have developed.

For being able to properly validate an ASC model, we have developed a patch of the OSCI SystemC simulator. The resulting simulator allows us testing different interleaving of the processes execution.

2.3.1 ASC Modeling Language

ASC defines two different kinds of module. The *container modules* are used to define the hierarchical structure of the system. They can contain other modules, channels and ports. The ASC *process modules* specify the behavior and the concurrent aspects of an asynchronous circuit. The behavior of a process module is defined by its *process* method.

The ports are the communication interfaces of ASC processes. An ASC port is unidirectional (*input* or *output*) and can be connected to at most one ASC channel. The emission of data through an output port is done with its *send* method. The *receive* method of the input port connected to an output port allows to get the data sent by an output port. A handshaking protocol is used to synchronize the communication between two ASC ports. They are two different kinds of port: active and passive. An *active port* initiates the handshaking protocol and a *passive port* acknowledges it. A passive port has a special method called *probe* allowing it to check if its connected active port has initiated a communication or not.

The channels are the mediums used by the ASC processes to communicate and synchronize their executions. A *pull* and a *push* channel interconnects an active input port to a passive output port and an active output port to a passive input port, respectively. These channels implement the communication and synchronization primitives offered by the ASC ports. Indeed, the previous methods of these ports (*send*, *receive* and *probe*) just forward their procedure call to the methods of their connected channels.

To synchronize its execution, an ASC process can use its *idle* methods. A first version of this method is used to wait until at least one of its passive ports is ready to communicate. A second version is used to wait that a set of parallel communications have been completed. A parallel communication is triggered with the *par_receive* or *par_send* methods of the ASC ports, and a set of parallel communications is constructed with the overloaded operator //.

The two new statements *as_choice_nd* and *as_guard* are provided by the ASC library. The *as_choice_nd* defines a non-deterministic choice over a set of guarded commands. A guard of a non-deterministic choice is specified with the statement *as_guard*.

2.3.2 Tracing Facilities

A trace file respecting the AST time model is created with the function *as_create_ast_trace_file*. This function takes as a parameter the name of the output trace file and returns a pointer on this trace file. This pointer can be used by the *as_trace* function to define the ASC channels to trace. This pointer can also be used with the *as_set_time_unit* function to set the time resolution used for performing the simulation. Finally, a trace file shall be closed by calling the function *as_close_ast_trace_file*.

An ASC channel has a template parameter defining the DATA carried out by this channel. Any kind of channel can be traced with the *as_trace* function. Currently, the value of a data transferred over a traced channel will be reported only if its type belongs to one of the following C++ types: *bool*, *char*, *short*, *int*, *long*, *long long*, *unsigned char*, *unsigned short*, *unsigned int*, *unsigned long*, *unsigned long long*, *float*, *double*. However, ASC tracing facilities can be easily extended to handle specific user data types. Indeed, the *as_trace* function can be overloaded in order to handle any kind of data.

The *ast2vcd* takes as input an ASC trace file and produces a VCD output trace file. For each traced ASC channel *ch* is defined the following VCD signals:

- *Data*: represents the data transferred during a communication.
- *sca*, *scp*, *eca*, *ecp*: represent the events defining a communication.
- *p*: the call to the probe method of the channel. The value of the channel is equal to the result of the probe.

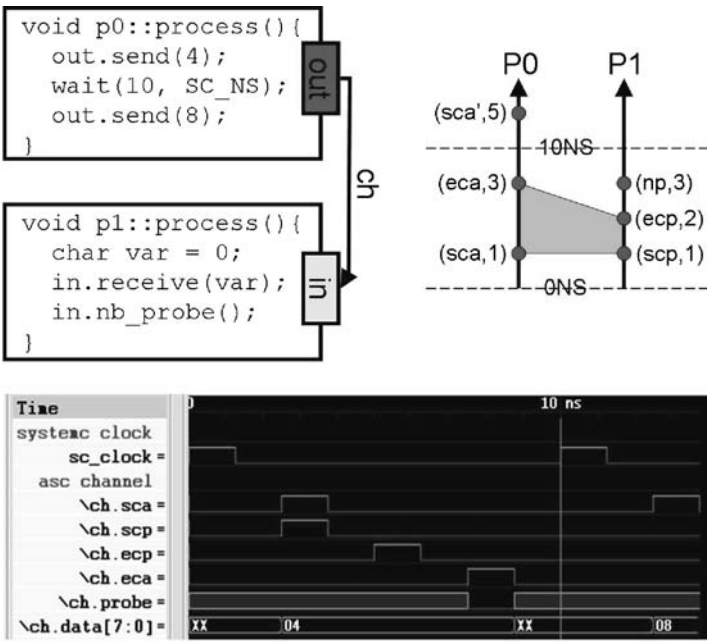


Fig. 2.3 Traces with ASC tracing facilities

Figure 2.3 shows the VCD and the AST traces generated by the simulation of the two ASC processes *p0* and *p1*. These two processes are connected via an ASC channel *ch*. All the events represented in this figure, except *sca'*, happen at the simulation time 0 nanosecond (NS). However, in the resulting VCD, these events do not happen at 0 NS. Indeed, to represent the AST time stamp and make the trace readable, the events occurring at the same SystemC simulation time, but at different AST times, are separated by ε time steps. In order to know at which SystemC simulation time an AST event occurs, the SystemC simulation clock is represented by the *sc_clock* signal. The ε value is automatically computed by ast2vcd. It takes care that each AST event occurs after its SystemC simulation time and before the next *sc_clock* signal.

2.3.3 ASC Simulator

Because DI asynchronous circuits are not sensitive to delays, the execution order of the processes modeling such circuits should not have any impact on the correctness. For checking this fundamental property of a DI asynchronous circuit, the selection of a process to execute among the set of runnable processes should be non-deterministic.

The current implementation of the SystemC kernel simulator [22] uses two pseudo-fifo lists for managing the set of runnable processes. The first one contains the runnable *sc_method* and the second one the runnable *sc_thread*. These pseudo-fifo are divided into two lists: *get_list* and *push_list*. The *get_list* is used by the scheduler for selecting the new process to execute. The *push_list* is used for inserting a new runnable process into the pseudo-fifo. During an evaluation phase, all the processes which are in the *get_list* of the *sc_method* pseudo-fifo are firstly executed. Secondly, all the processes which are in the *get_list* of the *sc_thread* pseudo-fifo are executed. Finally, if the *push_list* are not empty, they are swapped with their corresponding *get_list*. These three steps are repeated until the two *get_list* are empty at the beginning of the first step. Thus, we can see that this scheduling algorithm is deterministic and do not allow us to test different interleaving of processes execution.

As illustrated in Fig. 2.4, the patch that we have defined merges the two pseudo-fifos into one priority queue. We have also defined a new common class for the *sc_thread* and the *sc_method* defining their priority of execution. When a process is becoming runnable, a new priority is affected to this process and then it is inserted into the priority queue. The priority value is computed by a pseudo random generator. In order to be able to replay a simulation, the seed of this pseudo random generator can easily be determined. When the active process execution finished, the scheduler chooses the process in the priority queue with the lowest priority. By this way, we are able to test different interleaving of processes execution.

Another promising solution for this problem is presented in [25]. It presents a method and tools enabling to efficiently generating the different scheduling allowed by the scheduler specification. They use dynamic partial-order reduction techniques to avoid the generation of two schedulings that have the same effect on the system's behavior.

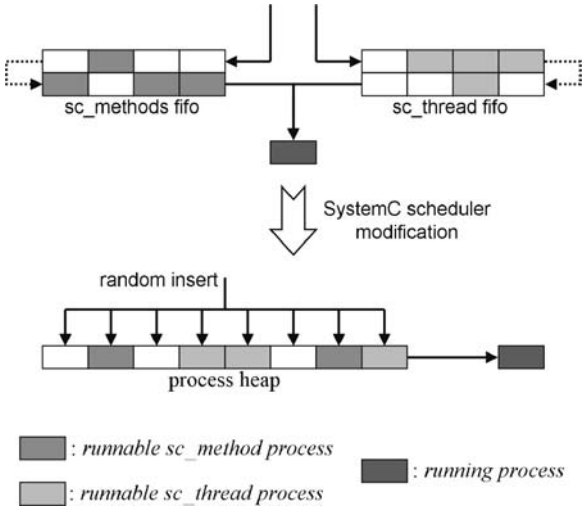


Fig. 2.4 SystemC simulator scheduler

2.4 Octagon NoC with ASC

The Octagon [23] interconnect was developed by STMicroelectronics to efficiently interconnect eight CPUs on a single chip. This interconnect is composed of 8 nodes and 20 bidirectional links. However, in our version of the Octagon, each bidirectional link is replaced by two unidirectional links. The resulting configuration of the system is illustrated in Fig. 2.5. In this figure, the integer associated to each node is the address used by a CPU for sending a packet to another CPU. Each node uses an algorithm based on the Octagon topology and on arithmetic properties to route its incoming packets to the right output.

The first ASC version of the Octagon operates in packet switching mode. Figure 2.6 exhibits the ASC code of the nodes used in this version of the Octagon. These nodes wait for a new packet on one of the four input ports. When at least one packet is available, the nodes perform a non-deterministic choice over the set of input ports ready to transmit a new packet. A packet is then received on the selected input port. Finally, this packet is forwarded to an output port according to the routing Octagon algorithm.

The second ASC version of the Octagon operates in circuit switching mode. In this version there are two different kinds of packet: *request packet* and *response packet*. The request packets are sent by a CPU which is willing to access a resource of another CPU. When a request packet is received by a CPU, it sends a response packet to the CPU which sent this request packet.

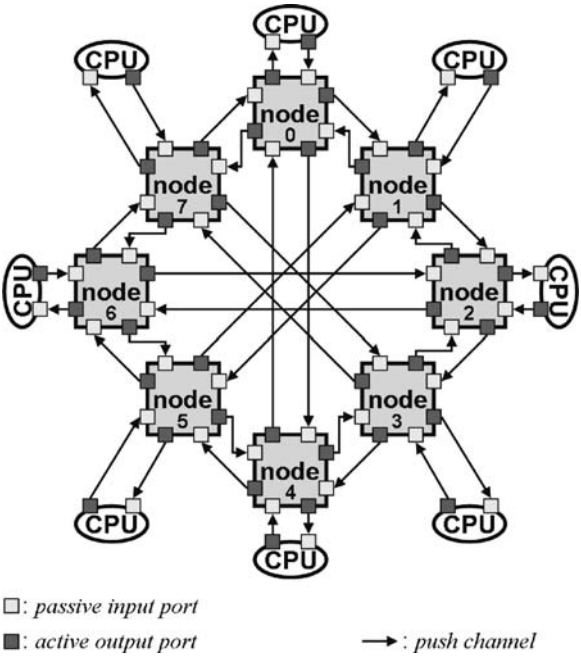


Fig. 2.5 Octagon NoC confguration

Fig. 2.6 Packet switching router

```
void node::process() {
  idle(in_ip | in_clk | in_cclk | in_frt);

  as_choice_nd(
    as_guard(in_ip.nb_probe(), IP),
    as_guard(in_clk.nb_probe(), CLK),
    as_guard(in_cclk.nb_probe(), CCLK),
    as_guard(in_frt.nb_probe(), FRT)))
  {
     case IP: in_ip.receive(pkt);
     case CLK: in_clk.receive(pkt);
     case CCLK: in_cclk.receive(pkt);
     case FRT: in_frt.receive(pkt);
  }

  switch( (pkt.adr - this->adr) mod 8 ) {
    case 0: out_ip.send(pkt); break;
    case 1:
    case 2: out_clk.send(pkt); break;
    case 6:
    case 7: out_cclk.send(pkt); break;
    default: out_frt.send(pkt); break;
}}
```

Fig. 2.7 Circuit switching router

```
void node::process() {
  receive_req(l_pkt_req, l_in_dir);
  l_out_dir = route(l_pkt_req.adr_dest);
  forward_req(l_pkt_req, l_out_dir);
  receive_rsp(l_pkt_rsp, l_out_dir);
  forward_rsp(l_pkt_rsp, l_in_dir);
}
```

The ASC code of the routers used in this version of the Octagon is summed up in Fig. 2.7. When one of these nodes receives a new request packet, it stores which input port (*l_in_dir*) transmitted the packet. As for the previous version, the packet is then transmitted through the right output port. However, this time the node does not restart to wait for a packet on all its input ports, but it waits on the input port associated to the output port (*l_out_dir*) which was used to send the packet. In this way the next packet received by this node can only be the response packet of the previous request packet. When this last response packet is received, it is forwarded through the output port corresponding to the input port which received the request packet. Thus, in this mode, the entire path between the CPU which sends the request and the CPU which receives it is reserved for the response packet.

In a first step, the ASC tracing facilities enabled us to validate the functional behavior of the two versions of the Octagon. For example, they helped us to check the behavior of the routers and to understand how dead-locks were happening in such a NoC. To this end, we have replaced the CPUs with traffic generator processes and traffic consumer processes. In a second step, we added latencies to the

different components (consumers, producers and routers) and to the ASC channels. By this way, we were able to analyze the congestions and latencies of the NoC under different pattern of traffic (uniform, hot-spot and random).

2.5 Conclusion

This paper presented a time model which can be used to validate asynchronous circuit models using a language based on CSP. This time model was used to define the tracing facilities of the ASC library. These tracing facilities produce traces of the ASC process activities over their connected channels, which can then be used to generate standard VCD. However, the VCD format is not really adapted to asynchronous circuits. Thus, we are currently investigating other trace formats like SCV. We are also evaluated the time model on complex multiple clock systems.

Finally, modeling and validating asynchronous logic with the ASC library is the first step towards the synthesis. Our final goal is to be able to synthesize these models with the TAST framework [26]. We are currently formally defining the synthesis process of ASC based models to efficiently generate gate level asynchronous circuits.

Acknowledgments The authors thank Y. Remond for initiating the research on this time model, and K. Morin-Allory for reviewing initial versions of the document, and R. Solari for reviewing final versions of the document. This work is partially supported by the French government in the MEDEA + framework, through the 2A703 NEVA project (Networks on Chips Design Driven by Video and Distributed Applications).

References

1. Jantsch A, Tenhunen H (2003) Networks on chip. Kluwer, Boston, MA
2. Sparsø J, Furber S (2001) Principles of asynchronous circuit design. Kluwer, Boston, MA
3. Nielsen SF, Sparsø J (2001) Analysis of low-power SoC interconnection networks. In: 19th Norchip, pp 77–86
4. Edwards DA, Toms WB (2004) Design, Automation and Test for Asynchronous Circuits and Systems. Technical Report IST-1999-29119, 3rd edn. Working Group on Asynchronous Circuit Design (ACiD-WG). http://www.scism.sbu.ac.uk/ccsv/ACID-WG
5. Cortadella J, Kishinevsky M, Kondratyev A, Lavagno L, Yakovlev A (1997) Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In: IEICE Trans Inf. and Syst, pp 315–325
6. Fuhrer RM, Nowick SM, Theobald M, Jha NK, Lin B, Plana L (1999) Minimalist: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines. Technical Report CUCS-020-9. Columbia University, Computer Science Department
7. Yun KY, Dill DL (1992) Automatic synthesis of 3D asynchronous state machines. In: ICCAD92, pp 576–580. Santa Clara, CA
8. Martin AJ (1990) Programming in VLSI: from communicating processes to delay-insensitive circuits. In: Developments in Concurrency and Communication, pp 1–64. Hoare CAR, UT Year Programming Series

9. Edwards D, Bardsley A (2002) Balsa: an asynchronous hardware synthesis language. In: The Computer Journal, Volume 45, Issue 1, pp 12–18
10. Berkel KV (1993) Handshake circuits – an asynchronous architecture for VLSI programming. Cambridge University Press, Cambridge
11. Quartana J, Fesquet L, Renaudin M (2005) Modular asynchronous Network-on-Chip: application to GALS system rapid prototyping. In: Very Large Scale Integration Systems (VLSI-SoC'05). Perth, Australia
12. Koch-Hofer C, Renaudin M, Thonnart Y, Vivet P (2007) ASC, a SystemC extension for modeling asynchronous systems, and its application to an asynchronous NoC. In: 1st International Symposium on Networks-on-Chip (NoC'07). Princeton, NJ
13. IEEE Std 1666–2005, SystemC Language Reference Manual (2005)
14. Hoare CAR (1978) Communicating Sequential Processes. In: Communications of the ACM, Volume 21, Issue 8, pp 666–677
15. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. In: Communications of the ACM, Volume 21, Issue 7, pp 558–565
16. Ashkinazy A, Edwards D, Fansworth C, Gendel G, Sikand S (1994) Tools for validating asynchronous digital circuits. In: 1th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'94), pp 12–21. Salt Lake City, UT
17. Chakraborty S, Dill DL, Yun KY, Chang KY (1997) Timing analysis for extended burst-mode circuits. In: 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'97), pp 101–111. Eindhoven, The Netherlands
18. Karlsen PA, Røine PT (1999) A timing verifier and timing profiler for asynchronous circuits. In: 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'99), pp 13–23. Barcelona, Spain
19. Viaud E, Pêcheux F, Greiner A (2006) An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In: Design, Automation and Test in Europe (DATE'06). Munich, Germany
20. Martin AJ (1993) Synthesis of Asynchronous VLSI Circuits. Internal Report, Caltech-CS-TR-93-28. Caltech Institute of Technology, Pasadena, CA.
21. Sutherland IE (1989) Micropipelines. In: Communication of the ACM, Volume 32, Issue 6, pp 720–738
22. Open SystemC Initiative (2007) SystemC v2.2. http://www.systemc.org/
23. Karim F, Nguyen A, Dey S, Rao R (2001) On-chip communication architecture for OC-768 network processors. In: Design Automation Conference (DAC'01). Las Vegas, NV, pp 678–683
24. IEEE Std 1364-2001, Behavioural languages – Part 4: Verilog hardware description language (2001) pp 349–374
25. Helmstetter C, Maraninchi F, Maillet-Contoz L, Moy M (2006) Automatic Generation of Schedulings for Improving the Test Coverage of System-on-a-Chip. Verimag Research Report, TR-2006-6
26. Renaudin M, Rigaud JB, Dinh Duc AV, Rezzag A, Sirianni A, Fragoso J (2002) TAST CAD Tools. TIMA Research Report. TIMA–RR-02/04/01—FR

Chapter 3
On Construction of Cycle Approximate Bus TLMs

Martin Radetzki and Rauf Salimi Khaligh

Abstract Transaction level models (TLMs) can be constructed at different levels of abstraction, denoted as untimed (UT), cycle-approximate (CX), and cycle accurate (CA) in this contribution. The choice of a level has an impact on simulation accuracy and performance and makes a level suitable for specific use cases, e.g. virtual prototyping, architectural exploration, and verification. Whereas the untimed and cycle-accurate levels have a relatively precise definition, cycle-approximate spans a wide space of modelling alternatives between UT and CA, which makes it a class of levels rather than a single level. In this contribution we review these modelling alternatives in the context of SystemC and with focus on bus models, provide quantitative measurements on major alternatives, and propose a CX modelling level that allows to obtain almost cycle accuracy and a simulation performance significantly above CA models.

Keywords Transaction-level modelling, SystemC, embedded systems

3.1 Introduction

Transaction level modelling has become a widely used technique in embedded systems and system on chip design. A variety of system design languages such as SystemC [7] and SpecC [5] can be used for modelling at transaction level. However, transactions and many other typical elements of transaction level models (TLMs) are not available as syntactic language features. The TLM creators instead have to create the transaction level abstractions themselves, using language features such as channels and interfaces. This is supported by mostly informal descriptions of the TLM methodology, e.g. [6], and by methodology-specific libraries, e.g. the SystemC TLM library [10].

Institut für Technische Informatik, Pfaffenwaldring 47, 70569 Stuttgart, Germany
Email: martin.radetzki@informatik.uni-stuttgart.de

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Methodologies and libraries leave degrees of freedom to implement TLMs in different ways. This has the positive effect that the transaction level in fact spans multiple (sub-)levels of abstraction, facilitating trade-offs between simulation accuracy and performance. However, these levels, subsequently denoted as untimed (UT), cycle-approximate (CX) and cycle accurate (CA), are not formally defined but rather characterized by model properties. The lack of a formal definition makes it difficult to describe how to systematically construct TLMs at a given level.

Despite this drawback, there exist relatively precise and consistent characterizations of UT and CA, as we will show in Section 3.2. CX models, however, can cover a wide range between UT and CA, and there appears to be no consensus on the characteristics of a favourable CX model. We will attempt the definition of such a model based on the consideration of modelling alternatives. For this purpose, we use the following non-orthogonal criteria characterizing TLMs in addition to their timing accuracy:

- The underlying communication mechanism, which can be a subprogram call with transfer of control flow (blocking) or message passing with data flow (potentially non-blocking).
- The use of concurrency in the model, namely the presence or absence of individual threads in the modelled master, slave, and bus components. A component with (without) a thread is called passive (active).
- The programming abstraction provided to the users of a bus model, including no abstraction (direct access to port/channel), procedural application programming interface (API), communication mechanisms that could be adopted from concurrent/distributed systems (e.g. RPC, CORBA).
- The bus features covered by the model, including single transfers, bursts, locked transfers, split transfers, wait states (inserted by slave), busy cycles (inserted by master), bus phases and pipelining, in-order or out-of-order completion of transfers, and arbitration policy.
- The modelling mechanism used for arbitration, in particular the use of events to trigger arbitration (no events, one event, multiple events).
- The use cases of a particular model, including verification, exploration, virtual prototyping.

In the next section, we review the related work with respect to the above criteria. Section 3.3 presents considerations and alternatives towards accurate CX models, and Section 3.4 investigates their performance.

3.2 Related Work

Donlin [4] presents the transaction level terminology used by the SystemC TLM working group. It includes a Programmer's View (PV) characterized by untimed communication and the use case of providing a functionally accurate representation of hardware subsystems to software programmers. A Programmer's View with

Time (PV + T) results from annotating a PV model with time and approximate arbitration. A Cycle Accurate (CA) view is characterized by fully bus protocol compliant arbitration and timing accurate to the level of individual cycles.

In the OCP terminology [9], three TLM layers are defined: The Transfer Layer (L-1) is characterized by cycle-true behaviour and use for verification and precise simulation. At the Transaction Layer (L-2), modelling abstracts from the details of a bus protocol but can take properties like split transactions and pipelining into account. The Messaging Layer (L-3) is untimed and enables 1:1 connections between initiators and targets, abstracting from bus address mapping.

The SpecC related taxonomy from [3] takes into account the timing accuracy of computation as orthogonal to the communication timing aspect and defines cycle-timed, approximately-timed and untimed levels for both dimensions. Considering the communication dimension only and focusing on TLM models, we can identify an untimed component-assembly model (CAM) which models communication between system components by message passing, a bus arbitration model (BAM) with arbitration policy modelling that approximates timing by one wait statement per transaction, and a cycle-timed bus-functional model (BFM).

The GreenBus approach [8] makes a significant step towards a constructive definition of transaction levels. It identifies three levels of granularity called transactions, atoms, and quarks. A transaction is a sequence of uninterruptible phases (atoms), and each atom is a collection of payload values (quarks). A PV model approximates timing at transaction boundaries, a bus accurate (BA) model at atom boundaries, and a cycle callable (CC) model must model all quark updates with cycle accuracy. An untimed model is not defined.

From these considerations, it is apparent that there still exists no unified terminology in the TLM field. Table 3.1 classifies the modelling levels described in the aforementioned approaches with respect to their bus communication timing properties.

The UT approaches have in common the primary use case of virtual system prototyping and that they result in a purely functional simulation. This limits the available choices with respect to our characterization criteria as well as the impact of the remaining choices on the simulation result. Subtle differences exist – for example, the SpecC approach features message passing and active slaves at the CAM level whereas SystemC PV uses function calls from masters into passive slaves – but these should not have impact on the functional result of simulation nor the non-existent timing (whereas an impact on simulation performance is likely). Another such difference is whether bus structure, addressing scheme, and approximate arbitration are modelled (SystemC PV) or not (point-to-point connections in OCP L-3).

Table 3.1 Overview of transaction levels

Accuracy	UT	CX	CA
SystemC TLM	PV	PV + T	CA
OCP	L-3	L-2	L-1
Cai/SpecC	CAM	BAM	BFM
GreenBus	–	PV(+T), BA	CC

A similar situation can be observed at the CA level. The primary use cases are verification reference and precise performance analysis. The property of cycle accuracy strongly restricts the modelling space. All bus features must be modelled, communication is necessarily by non-blocking data flow between concurrent components, and arbitration is typically performed in each cycle. A detailed investigation of CA model code often reveals that some interface abstraction is provided, but "under the hood" the model implements communication at the level of the signals used in the bus protocol, even if these are bundled in a TLM channel. For example, Table 3.1 in [8] shows the direct correspondence between GreenBus quarks and protocol signals. In the SystemC based AMBA cycle accurate simulation interface (CASI) [2], the CA AHB channel uses a data structure whose attributes are identical to the AHB signals. A proposal for more abstract protocol modelling based on hierarchical state machines has been made in [13].

At the CX level with the primary use case of system exploration and performance (bus throughput or latency) estimation, a much wider range of modelling alternatives exist. Within the SystemC TLM and GreenBus PV + T models, timing is approximated at the granularity of transactions, arbitration abstracts from the precise bus arbitration policy, and transactions cannot be pre-empted. Thus, features such as split transfers cannot be modelled. On the other hand, the SpecC BAM and GreenBus BA models permit pre-emption of transactions and subsequent bus re-arbitration. Thereby, more precise simulation can be obtained at the cost of lower simulation performance compared to PV + T.

An interesting approach to CX modelling is presented in [14], where transactions are simulated with the optimistic assumption of not being pre-empted. If this assumption turns out to false at a later simulation time, the transaction duration is extended by the duration of pre-empting transactions. This yields a 100% accurate simulation with respect to the authors' measure of timing accuracy. However, the data of a burst transfer are transmitted in a single operation at the beginning of the transaction modelling that transfer. This means that individual data transfers are not cycle accurate and the interleaving of data from pre-empting transfers cannot be simulated, which may affect data-dependent functionality.

In the remainder of this contribution, we investigate whether a CX model can be designed to cover a maximum of bus features and to come as close as possible to cycle accuracy, including accuracy of the data transfers. We will also investigate modelling decisions that optimize simulation performance without impacting accuracy. The resulting model can provide rather accurate estimates for the purpose of system exploration, complementing the significantly less accurate yet faster PV + T models.

3.3 Modelling Alternatives and Decisions

Since we target a SystemC model implementation, we will use the SystemC TLM terminology in the following but keep the term CX for our model.

3.3.1 Concurrency

In most PV and PV + T models, slaves are passive and masters are active components. This limits the achievable accuracy because master and slave cannot operate concurrently. For example, a master cannot prepare data for the next transaction while a slave processes the master's current transaction request. To avoid this possible deviation from detailed system timing, we choose to make slaves active components in our CX model.

Another modelling alternative pertains to the modelling of the bus as an active or passive component. This is closely related to arbitration modelling, discussed at the end of Section 3.3.

3.3.2 Communication Mechanisms

PV and PV + T models typically employ transfer of control flow (blocking subprogram calls) as a mechanism for communication between master and slave. This is in conflict with the desired concurrency of master and slaves. Therefore, we use data flow to pass messages between communicating blocks. However, for large message payloads such as burst data, we use a shared memory implementation where only a pointer to the shared data is passed as part of the message. Thereby, copying of the payload is avoided and simulation performance increased. Access conflicts on the shared memory are avoided by limiting access by the communication partners (master, slave, bus model) to disjoint phases of the transaction. The dynamic memory management is handled by the master's port, hidden from the user, to avoid memory leaks and dangling pointers. Memory is allocated upon start of a transaction and freed when the master has obtained the last data of a transaction response according to the programming model (cf. next subsection).

We have tried to avoid a suspected overhead due to repeated creation and deletion of memory blocks by reusing a pool of such blocks. This had no significant impact on simulation speed; possibly because such optimization is already implemented in the C++ runtime library's heap management.

Another modelling choice must be made between use of standard TLM channels (tlm_fifo) to connect masters and slaves with the bus model vs. direct connection to interfaces exported by the bus. The latter option is likely to be more efficient because it avoids the overhead of storing and retrieving messages in/from a tlm_fifo. Moreover, the master's interface method calls will go directly into the bus model, enabling an implementation that reduces the number of context switches during simulation. Both variants have been implemented and their resulting simulation performance is compared in Section 3.4.

3.3.3 Programming Abstraction

In most PV and PV + T models, subprogram calls serve as a well-understood programming abstraction of communication operations. However, subprogram calls

cannot be used between concurrent or distributed model objects. Remote procedure calls (RPC) are a mechanism that could be adapted from distributed programming; however, in the presence of return parameters, they would block the master which would compromise the accuracy of our model. Mechanisms like CORBA enable non-blocking communication, but they are too heavyweight for use in a fast transaction level simulation.

As a compromise, we have adapted for the purpose of TLM the *active object* design pattern known from concurrent object-oriented programming, see Fig. 3.1. A key concept of this pattern is the *future object* which is immediately returned as result of a non-blocking subprogram call. In our adaptation, the call models a non-blocking communication operation and the future object can later be used by the master to obtain results from that operation (the transaction response) at its own discretion. Beyond that, the future object is also used to allow the master to delay the supply of values that belong to the transaction request to a time after request initiation. Thereby we can accurately model that the master may supply (retrieve) bus word number *i* up to (starting from) the *i*-th cycle after the start of a burst transfer instead of providing or receiving all burst data at the beginning or at the end of a transaction.

Another active object concept is the *guard* which can be defined individually per operation at the slave side, cf. Fig. 3.1. We utilize the guard as a programming abstraction of a bus feature that allows a slave to split a transfer that cannot be served immediately, and to resume that transfer when appropriate. As an efficiency improvement over [12], we have modelled an event-based resume mechanism to avoid polling the guard in each cycle during which a transaction is split. Moreover,

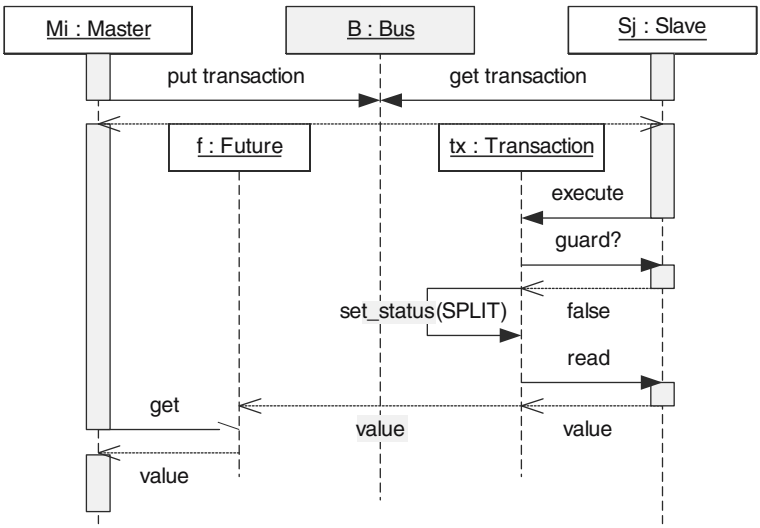


Fig. 3.1 Active object pattern for TLM

the models presented in this contribution facilitate for the first time the splitting of burst transfers at the granularity of the transaction that models the transfer rather than single bus word transfers.

3.3.4 Bus Features

The most basic bus features are single bus word read and write transfers (*single transfers*). Successive transfers to consecutive addresses can be combined into a *burst transfer*. Burst transfers may have a fixed or user-defined length. They may be pre-emptible or not (*locked*). The burst address sequence may wrap around at block boundaries (*wrapping burst*) or not. We model all these transfers and their properties in an object-oriented way as C++ transaction classes and attributes (data members). Details about this modelling style can be found in [11].

Another feature found in most high performance buses is *pipelining*. To employ pipelining, transfers are decomposed into phases, and different phases of subsequent transfers are allowed to execute in parallel. We model the phase as a state attribute of a transaction which is controlled by the bus model. Pipelining can be modelled in a cycle accurate way by introducing a number of stages into the bus model as shown in [13]. Our CX model covers pipelining within a single transaction (which is relevant for burst transactions), but neglects it at the boundary between different transactions for performance reasons.

We model *split transfers* using the guard mechanism for abstraction as presented in the previous subsection. The OCP L-2 model is the only other CX model known to have built-in split transfers. An advantage of our model is that thanks to the programming abstraction, the designer of a bus master model is relieved of taking care of the split transfer handling.

3.3.5 Arbitration Modelling

This subsection is concerned with the mechanisms employed for modelling arbitration; the discussion is largely independent of arbitration policy. In CA models, a time or clock triggered arbitration process is executed once per cycle. An efficient CX model can limit arbitration under the assumption of a time-invariant arbitration protocol because the grant decision does not change unless the state of the waiting and active transactions changes. Re-arbitration needs to be performed only in simulation cycles in which a new transaction arrives to the bus or in which the currently active transaction is finished or split (allowing a waiting transaction to be granted the bus).

Re-arbitration can be modelled with one or a combination of the following methods: If the bus model exports an interface, the interface methods, executed with the masters' processes, may perform arbitration without the need for a simulation process

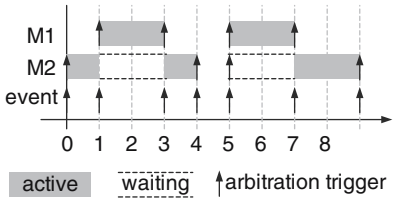


Fig. 3.2 Arbitration triggering mechanisms

context switch. This comes at the cost of multiple re-arbitration if multiple masters issue transactions in the same cycle, and it is not possible if communication is via channels (e.g. tlm_fifo). In this case, the bus model needs a process that is triggered by incoming transaction messages and performs arbitration actively (cf. M1, M2 in Fig. 3.2). Since each channel has an event of its own, this requires the overhead of creating or-event-lists to activate the arbitration process in SystemC. The number of events can be reduced to one for all incoming transactions by implementing the bus model itself as channel with interface methods that trigger an internal re-arbitration event (cf. event in Fig. 3.2). Split or finished transactions can trigger the re-arbitration event or an individual event.

3.4 Simulation Performance Results

The basic experimental setup used for performance evaluation of the bus models includes two masters of different priority and one slave. The high priority master issues transactions of increasing burst length that may be split by the slave, a RAM model. The parameters of the bus model are chosen to reflect the cycle timing of the AMBA AHB protocol [1], and priority based arbitration has been modelled. All models have been compiled with the same options and have been simulated on a computer with Pentium M 1.66 GHz.

With this setup, four different bus models have been simulated: A model CX1 at the PV + T abstraction and a cycle-accurate model CA as reference points, and two cycle-approximate versions, CX2 and CX3 using different choices of the identified alternatives. CX2 is a model using tlm_fifos as channels while CX3 implements the TLM interfaces by itself, using a single arbitration event.

3.4.1 Comparison of Different Models

Figure 3.3 shows the simulation performance, measured in the number of 32 bit bus words whose transmission is simulated per second of CPU time, for the four models and for bursts of different size. No transactions have been split in this simulation. All models exhibit a performance that increases with the burst size due to less simulation

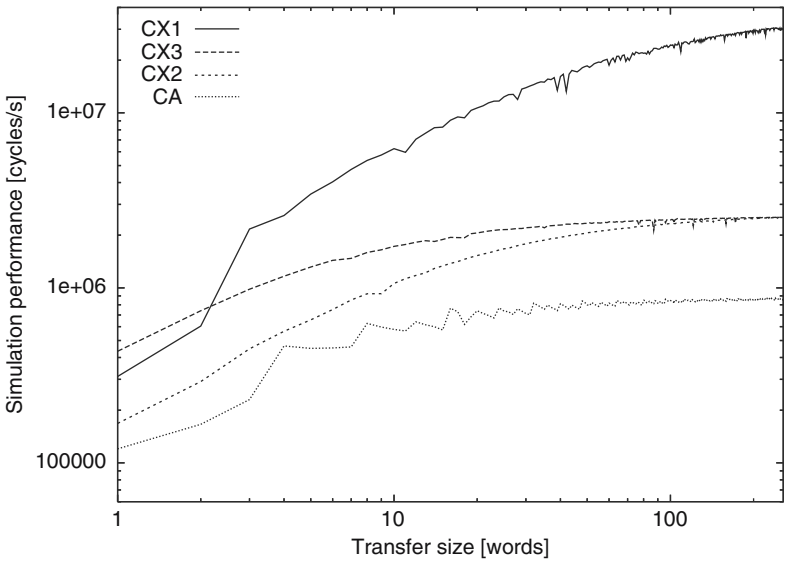


Fig. 3.3 Performance of models at different levels

overhead for arbitration and switching between transactions per transmitted bus word. We can see that the performance of the models CX2 and CX3 is consistently higher (by an average factor of about 5) than CA, and that CX1 (PV + T) exceeds CX2 and CX3 performance by an average factor of about 10. Only at very short burst length CX3 performance exceeds CX1; the reason is that CX1 lacks some of the optimizations that have been made in CX3.

At short burst length, model CX3 has a significant advantage over CX2, which diminishes towards larger bursts. The reason for this model behaviour is that the CX3 optimization of avoiding tlm_fifos and using just a single event is more significant when simulating short bursts requiring a higher rate of channel accesses and events.

3.4.2 Pre-emption Dependency

Different from PV + T, models CX2 and CX3 can simulate the pre-emption of transactions. To measure the effect of pre-emption on simulation performance, we have parameterized the slave model so that it randomly splits transactions. The percentage of bus word transfers which are split (i.e., multiple splits of a single transaction are possible) has been varied from 0% to 50%. Figure 3.4 shows the resulting simulation performance for model CX2. Performance degrades with increasing pre-emption ratio. It is reduced by a factor of up to 10 for long bursts and 50% pre-emption, compared to the non-preemptive case. Performance degradation

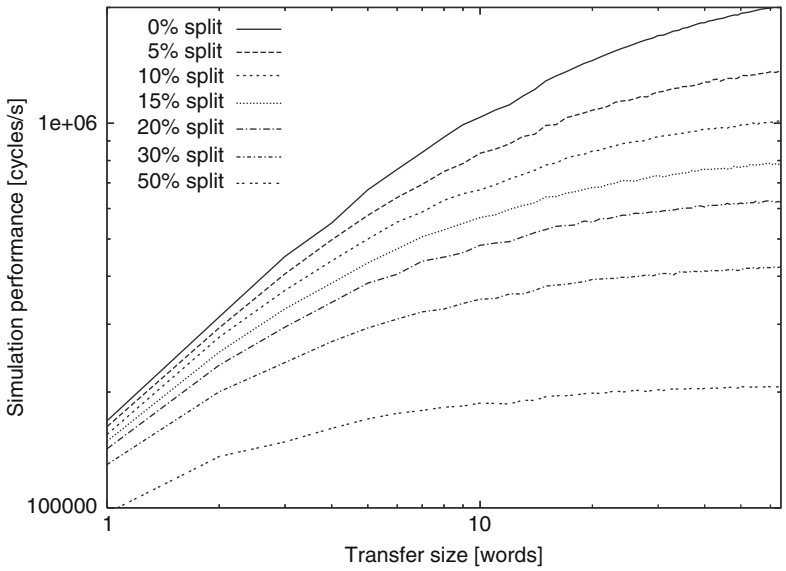


Fig. 3.4 CX2 performance with pre-emption

becomes less as transfer size decreases because re-arbitration due to a transfer split more often coincides with re-arbitration due to a request by the other master. Since the latter has to be simulated anyway, the split does not cause a simulation overhead in this case.

The same measurement has been performed using model CX3, with results shown in Fig. 3.5. Performance is generally higher compared to CX2, and the degradation factor due to pre-emption of bursts is down to a maximum of about 3. This is again due to the optimized implementation of model CX3, which also reduces the overhead of performing re-arbitration in the case of transaction pre-emption and completion.

3.4.3 Bus Component and Congestion Dependency

In order to evaluate simulation performance in the presence of more than two masters, model CX3 has been simulated in a setup with a number of masters varying from 1 to 16. The number of slaves also varies; in each of the simulations performed it corresponds to the number of masters so that n masters are simulated together with n slaves and the bus model. The masters have different static priorities. In the simulated scenario, the masters are synchronized so that for each simulated transfer size in the range of 1 to 64 words, all masters can complete their transfers of a given transfer size and then together move on to the next transfer size.

Figure 3.6 depicts the model's simulation performance under the constraint that the slaves do not split transfers. Generally, simulation performance decreases as the

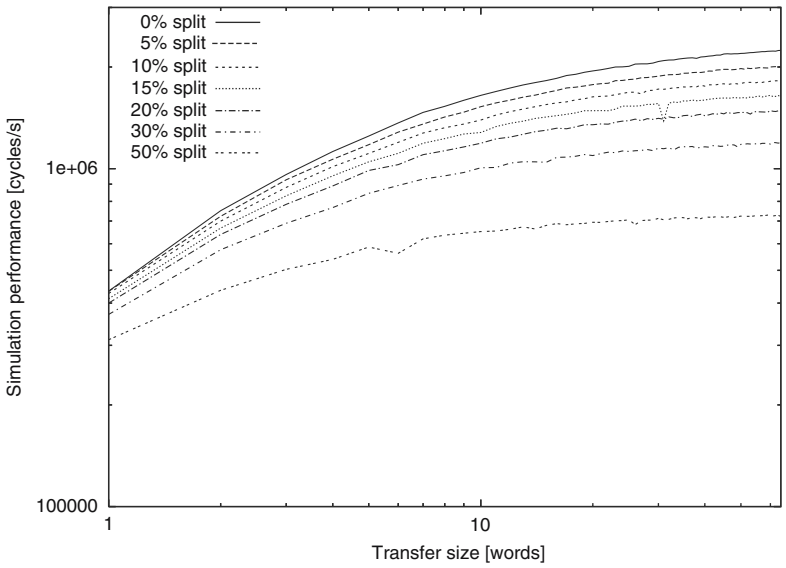


Fig. 3.5 CX3 performance with pre-emption

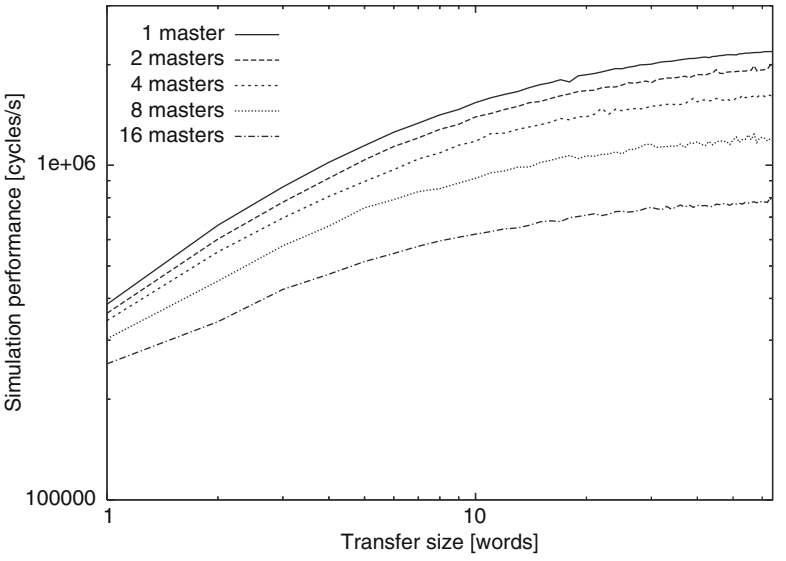


Fig. 3.6 CX3 performance for different numbers of masters (no split transfers)

number of masters increases. This is due to an increased average overhead for arbitration and due to the fact that in the presence of more masters, transfers of lower-priority masters tend to be pre-empted more often by the higher priority masters. The spread between the curves for 1 master and 16 masters is by a factor of about 2.

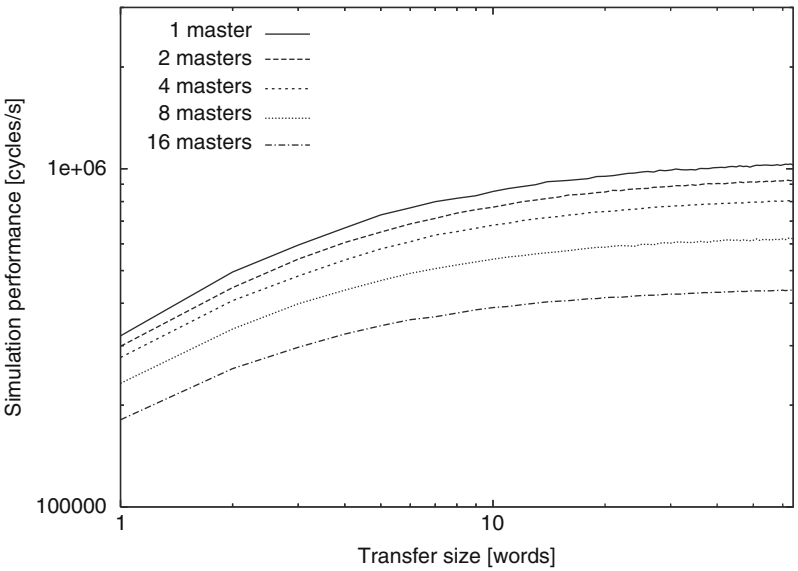


Fig. 3.7 CX3 performance for different numbers of masters (25% split transfers)

Figure 3.7 shows simulation performance for a similar scenario but with 25% of all transfers being split by a slave.

3.5 Conclusions

We have shown the design of a cycle-approximate model that covers all bus features and represents bus transfers by abstract transactions in an almost cycle-accurate way. The simulation performance of this model is between the performance of a cycle-accurate model and the performance of a PV + T model that does not cover transaction pre-emption. We argue that modelling at an accuracy level between PV + T and CA is useful for architectural exploration because it permits significantly more precise estimation than PV + T. Therefore, a CX abstraction level should complement the other levels instead of being dropped, which appears to have happened in SystemC TLM standardization.

References

1. ARM Ltd.: AMBA Specification (Revision 2.0). Document ID: ARM IHI 011A, www.arm.com/products/solutions/AMBA_Spec.html, accessed 7.11.2006.
2. ARM Ltd.: Cycle Accurate Simulation Interface (CASI). www.arm.com/products/DevTools/Real_ViewESLAPIs.html, accessed 11.10.2006.

3. L. Cai, D. Gajski: Transaction Level Modeling: An Overview. Proc. CODES + ISSS, 2003.
4. A. Donlin: Transaction Level Modeling: Flows and Use Models. Proc. CODES + ISSS, 2004.
5. R. Dömer, A. Gerstlauer, D. Gaijski: SpecC Language Reference Manual (Version 2.0). University of California, Irvine, CA, www.ics.uci.edu/specc/reference/SpecC-LRM_20.pdf, accessed 7.11.2006.
6. F. Ghenassia (Ed.): Transaction-Level Modeling with SystemC – TLM Concepts and Applications for Embedded Systems. Springer, Dordrecht, 2005.
7. IEEE Standard 1666-2005: SystemC 2.1 Language Reference Manual. IEEE, 2005.
8. W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, M. Burton: GreenBus – A Generic Interconnect Fabric for Transaction Level Modelling. Proc. 43rd Design Automation Conference (DAC). San Francisco, CA, 2006.
9. OCP International Partnership: Open Core Protocol Specification (Release 2.1). www.ocpip.org, 2006.
10. Open SystemC Initiative: TLM 1.0 API and Library. www.systemc.org, 2005.
11. M. Radetzki: Object-Oriented Transaction Level Modelling. In S. Huss (Ed.): Advances in Design and Specification Languages for Embedded Systems. Springer, Dordrecht, 2007.
12. M. Radetzki: Modellierung mit Guarded Transactions zum robusten Entwurf von Hardware-Software-Systemen in SystemC. Proc. 1. GMM/GI/ITG Fachtagung Zuverlässigkeit und Entwurf, München, 2007.
13. R. Salimi Khaligh, M. Radetzki: Efficient and Extensible Transaction Level Modeling Based on an Object-Oriented Model of Bus Transactions. Proc. Int'l Embedded Systems Symposium (IESS). Irvine, CA, 2007.
14. G. Schirner, R. Dömer: Fast and Accurate Transaction Level Models using Result Oriented Modeling. Proc. Int'l Conference on Computer Aided Design (ICCAD). San Jose, CA, 2006.

Chapter 4
Combinatorial Dependencies in Transaction Level Models

Robert Guenzel[1], Wolfgang Klingauf[1], and James Aldis[2]

Abstract Transaction-level modeling (TLM) allows for the design of virtual prototypes, providing considerably faster simulation speed than RTL models. But combinatorial dependencies are often inexactly modeled in terms of cycle accuracy, leading to imprecise simulation results. If, however, precise results are desired, additional coding and simulation effort is required. As a result, simulation performance drops down. This paper surveys the existing techniques to model combinatorial dependencies in TLM and presents a novel approach based on synchronization layers. Experimental results with SystemC prove our technique to enable higher simulation speed than the surveyed approaches, without inheriting their disadvantages.

Keywords Transaction-level modeling, SystemC, Combinatorial dependencies, Cycle accuracy

4.1 Introduction

Transaction-level modeling (TLM) enables designers to raise the abstraction level of system models, narrowing the productivity gap significantly [3, 5, 6]. With TLM, hardware and software can be described in a variety of ways, ranging from untimed models to cycle accurate models with the interfaces being just as abstract as the model requires [4].

The scope of this paper is cycle accurate TLM (CATLM), which promises busses and networks on chip to be simulated magnitudes faster than with RTL models, while achieving the same accuracy of simulation results. To this end, it is vital to fully take combinatorial dependencies into account, when extracting more abstract CATLM models from RTL models.

[1] Technical University of Braunschweig, Department E.I.S.

[2] Texas Instruments, France

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Most of the recent languages or language extensions supporting CATLM are based on discrete event simulators (DES), like SystemVerilog, SpecC or SystemC. DES use the concept of delta-cycles as infinitesimally small amounts of time. A single simulation time step can consist of many delta-cycles, whose number depends on the quantity of consecutive signal updates and event notifications during a simulation time step.

A major problem in CATLM is combinatorial calculation, such as combinatorial arbitration in busses.

In CATLM, modules are connected via channels as an abstraction of RTL wires. Processes that read values from such TLM channels are not aware of the process execution order, so that they cannot know whether modules that write to the channel are already executed at the current simulation time step. Thus, the reading module cannot identify the value to be valid.

Figure 4.1a shows an example of combinatorial arbitration (IBM CoreConnect OPB [7]) in RTL. All masters issue their requests at the same point of simulated time but each in another delta-cycle, denoted as Δ. In RTL the grant signals get re-evaluated with every change of one of the request signals and thus produce false intermediate results.

A poor CATLM implementation of the OPB arbiter would equally grant each incoming request, as it does not know whether higher priority request will arrive during the same cycle (Fig. 4.1b). If one does, the new grant to the higher priority master implies the removal of the grant to the lower priority one, which complicates the code, rendering it less abstract as necessary. Ideally a grant call should only appear once per cycle to obtain maximum simulation speed. To this end, the simulation process that reads all the input channels, calculates the result of the arbitration, and does the grant call, should only execute after all input channels carry a stable request value for the recent cycle. So this process has to be synchronized with all the input channels. In other words it must not be executed before all processes that might request access to the bus have been executed (Fig.4.1c).

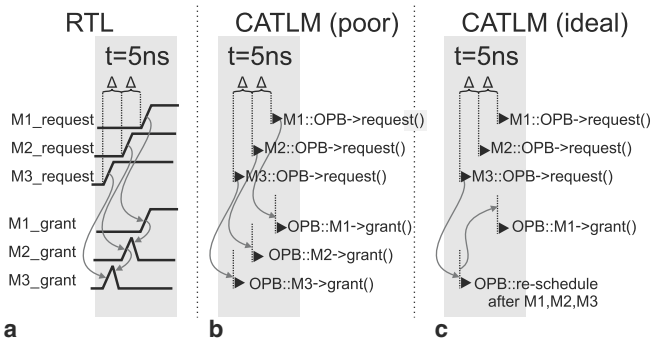


Fig. 4.1 OPB combinatorial arbitration

In today's system-level design languages, however, this kind of manually controlled partial process execution order is not supported.

Section 4.2 will show how the problem is tackled throughout academia and industry. In Section 4.3 we will introduce a novel solution for the synchronization problem and finally Section 4.4 compares all the approaches and we conclude in Section 4.5.

4.2 Known Solutions

In this section already existing solutions to the synchronization problem are described and their advantages and disadvantages are discussed. Three important terms that will be used throughout this section are *alteration calls* (AC), *readout calls* (RC) and the *length of a combinatorial chain*. In CATLM interface method calls (IMCs) can be classified as either calls that alter the state of the connected channel (AC) or that read the state of the connected channel (RC).

A combinatorial chain is considered a sequence of combinatorial dependencies. For example if a signal c is combinatorially calculated out of signals a and b, the length of the chain from a to c is one. If there is also a signal d which gets calculated out of a signal e and the aforementioned signal c, the length of the chain from a to d is two.

To simplify matters, we assume that there are events that get notified if a CATLM channel changed its internal state, so the connected modules can react to these changes. This assumption is true for many recent TLM frameworks [8, 9, 11].

4.2.1 Explicit Retraction

The most naive way solving the synchronization problem is to implement a retraction just like in RTL simulations: Processes will always assume that RCs return stable values and perform the corresponding ACs. Afterwards they will listen to an event indicating a change of the channel state, and in case it occurs will redo the AC with updated content or have to explicitly retract their previous AC (e.g. an OPB arbiter would have to retract a grant call). Since ACs have an immediate influence on the module(s) connected to the channel, the modules have to be implemented expecting multiple transfers over a channel during a single clock cycle. This introduces a certain amount of implementation overhead and in case of large or branching combinatorial chains the retraction can consume a severe amount of simulation time, as it leads to many recalculations. Furthermore, explicit retraction limits the way in which the CATLM module internal behavior can be more abstract than the RTL module behavior, since they have to be able to handle glitch-like communication, just as RTL modules do.

4.2.2 Negative Edge Exploitation

As introduced in [6] the synchronization between combinatorially dependent modules can be done using the negative edge of the clock. Using this methodology, RCs that are supposed to return stable values should be executed at the negative edge of the clock used, since all connected channels will get updated at the rising edge of the clock. This scheme works well in small systems and introduces close to no implementation or simulation overhead, but absolutely fails when combinatorial chains exceed the length of one, because the designer simply runs out of negative edges.

4.2.3 Delta-Cycle Waiting

Another approach to synchronization is waiting until the value to be read by an RC is known to be stable. Here we assume that all modules know for each of their input ports the number of delta-cycle (after the clock edge has been seen) until the input can be considered valid.

If this information is available as a number ranging from zero (the delta-cycle of the clock edge) to infinity, each module can determine the maximum and minimum of these numbers namely n_{min} and n_{max}. Since in DES a single process cannot determine from which event it was started, a module process supposed to perform the RCs and the resulting AC will start due to the occurrence of any one of the state change events from one of its inputs and then wait for $n_{max}-n_{min} + 1$ delta-cycles, thereby ensuring that all the inputs are stable regardless which one started the process. As a consequence, the delta-cycle in which the module will perform the AC will occur $n_{max} + 1$ or $n_{max}-n_{min} + 1 + n_{max}$ delta-cycles after the clock edge or in between those two values depending whether the process was started by the earliest, the latest or some other event. The uncertainty of the delta-cycle in which the AC occurs is called delta-cycle jitter.

4.2.4 Time Waiting

These drawbacks of delta-cycle waiting were also identified by the OCP-IP SLDWG [9] and were overcome by waiting for time instead of delta-cycles.

Here a module does not need to be aware of the delta-cycles after the clock edge after which the inputs are stable, but the time at which inputs are stable. Thus, inputs that are stable an arbitrary number of delta-cycles after the clock edge are treated to be stable a small fraction of simulated time after the clock edge. That means that ACs that originate in combinatorial modules occur a measurable time after the clock edge. So again n_{min} and n_{max} can be identified and the time to wait

after the occurrence of any input channel change event can be calculated as (n_{max} – n_{min} + 1)*(*period fraction*). As a result, the delta-cycle wait loops of the former method can be replaced by single timed waits, reducing the simulation overhead significantly.

The major drawback of this approach is, that now a delay that is not a multiple of the clock period is introduced, which has no equivalent in the RTL model or even silicon. These additionally added latencies complicate the comparison between RTL and CATLM traces.

It is important to note that both the delta-cycle and the time waiting technique rely on an information distribution mechanism that allows modules to receive and send information when channels get stable values.

4.2.5 Always Transmitting

A fourth way of synchronizing is used by the cycle accurate simulation interface (CASI) of ARM's RealView ESL API [1]. In this approach *every* module performs all its ACs during a clock cycle, either altering the state of the target channel or indicating that nothing is to be changed. As a consequence, combinatorial modules can simply wait for all input channels to be updated before issuing ACs themselves. Of course this introduces a significant simulation overhead, especially when there are only infrequent real updates to channels and therefore many 'no-change'-calls.

4.2.6 Cycle Based Simulation

Cycle based RTL or gate level hardware simulators are able to reorder event and process executions due to the known process execution dependencies based on signal sensitivities [10]. Thereby all simulation processes are executed at most once per cycle. In other words all processes are synchronized to each other. However, in CATLM the simulator cannot create such a static process execution order because of the fact that a CATLM process can read channels without being directly or indirectly sensitive to any of the channel's events. Hence, the simulator does not know which AC on a channel might affect a certain process without executing it.

4.2.7 Comparison

In conclusion, explicit retraction should be avoided as it prevents the designer to raise the abstraction of the internal behavior sufficiently above RTL. Negative edge exploitation is not an adequate generic approach as it limits combinatorial chains to length one. The delta-cycle waiting approach produces correct results by a fair

amount of code overhead but introduces an unacceptable simulation overhead, while the time waiting approach requires only small code and simulation overhead but produces undesirable delays. Finally the always transmitting technique provides accurate simulation results and is well suited for designs in which each module communicates intensively, but leads to significant simulation performance losses if modules communicate only infrequently (see Section 4.4 for experimental results).

Section 4.2.6 showed that knowledge about process execution dependencies may also help solving the synchronization problem.

4.3 A Novel Synchronization Approach

The comparison of combinatorial calculation techniques in transaction-level modeling points out, that all examined approaches either lack simulation performance or introduce a considerable overhead in terms of development effort.

The most appealing approaches are the delta-cycle waiting and time waiting techniques, as they introduced only a small implementation overhead. Both achieve synchronization by moving the call of the RC to a delta-cycle in which it is known that the RC will return a stable value. While delta-cycle waiting creates exact simulation results, its major disadvantage is that the simulation overhead quickly becomes significant.

An ideal solution should both provide the accuracy of the delta-cycle wait and perform as fast as the timed wait.

In the following our approach based on *synchronization layers* is presented, which meets these requirements and is based on process execution reordering similar to cycle based simulation.

4.3.1 Basic Definitions

Before we describe our approach in detail, some definitions are needed:

As stated in Section 4.2 for each channel *ch* in a CATLM system there is a set of ACs denoted as *AC(ch)* and a set of RCs denoted as *RC(ch)*. For each given channel in a CATLM model applies that a channel can be read and written:

$$AC(ch) \neq \emptyset \;\&\; RC(ch) \neq \emptyset. \tag{4.1}$$

Furthermore for each $c \in AC(ch)$ there is a set of RCs whose return values get altered by calling c, which is denoted as $\alpha(c, ch)$. There may be IMCs that are AC as well as RC, which is only allowed if the call does not alter the value it returns, in other words:

$$c \in AC(ch) \cap RC(ch) \Rightarrow c \notin \alpha(c, ch) \tag{4.2}$$

For each module m in a given CATLM model there is a set of ports that are owned by this module, denoted as $\pi(m)$.

Each $p \in \pi(m)$ is bound to exactly one channel ch, so that each IMC on p affects ch. Function $con(p)$ returns the channel which p is connected to, function $par(p)$ returns the module that owns the port p.

For each IMC c and channel ch, $\pi(c, ch)$ returns all ports which are connected to ch and may issue c. Because IMCs in CATLM have a mapping to RTL signals, and RTL signals may not have more than one driver,

$$c \in AC(ch) \Rightarrow \pi(c, ch) = 1. \tag{4.3}$$

A combinatorial dependency between an RC and an AC in a module can be defined as a pair of triples $((m,p,c),(m,q,d))$ where m is a module, $p \in \pi(m)$ and $c \in RC(con(p))$ and c has to return a stable value before $d \in AC(con(q))$ with $q \in \pi(m)$ can be called.

In the following m, p, c, d and q are always defined as before if not stated otherwise.

For each such triple $t = (m,p,c)$ let $\tau(t)$ be the set of all triples $(m,q1,d1)\ldots(m,qn,dn)$ that fulfill the aforementioned property.

So τ is basically the set of ACs that will be directly executed after the RC c on port p of module m has been called.

The set of combinatorial modules in a given CATLM model mod is called $\kappa(mod)$, and for each $m \in \kappa(mod)$ there is at least one triple $t = (m,p,c)$ such that: $\tau(t) \notin \varnothing$.

4.3.2 Synchronization Layers

The novel synchronization approach will be based on synchronzation layers that can be defined as a property of a triple (m,p,c) with $m \in \kappa(mod)$, $p \in \pi(m)$ and $c \in AC(con(p)) \cup RC(con(p))$. This property can be assigned to a triple t by $SL(t, newsl)$, where $newsl$ is a non-negative integer and can be read from a triple t by $SL(t)$. If a synchronization layer of a triple is not assigned yet, $SL(t)$ will return 0.

Figure 4.2 shows a simple system with two combinatorial arbiters. Masters issue requests on their channels and arbiters combinatorially forward the higher priority request to their outputs. The target will accept requests and signals this acceptance within the same cycle. The target is implemented in a way that it only expects a single request per cycle, which enables a high simulation performance as internal housekeeping can be kept small. We assume that the masters and arbiters use an AC named startReq to put a request on a channel. The target and the arbiters use an RC named getReq to get a request from a channel. The numbers annotated on channels and ports in Fig. 4.2 represent the desired execution sequence of those ACs and

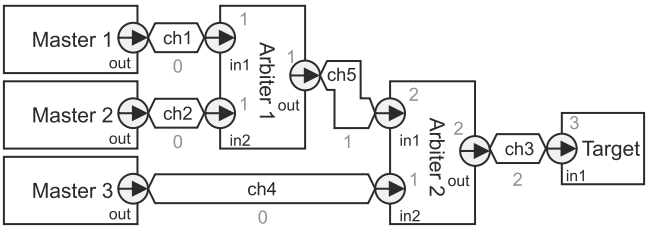


Fig. 4.2 Synchronization layer determination

RCs. The numbers of the channels and output ports relate to ACs, while the numbers of the input ports relate to the RCs. The semantic is that a call with number x has to be called before a call of number $y > x$, thereby ensuring that ACs on a channel are always called before RCs. We denote these numbers as *synchronization layers*.

With the definitions from Section 4.3.1 the following functions can be defined:

```
function setSLp(p, c, newSL);
  p is port; c is RC; newSL is integer;
  setSLm (par(p), p, c, newSL+1) ;
end;
function setSLm(m, p, c, newSL);
  m is module; p is port; c is RC; newSL is integer;
  if (SL(m,p,c)<newSL)
    SL( (m,p,c), newSL);
  for each (m,q,d) ∈ τ(m,p,c) do
    if (SL(m,q,d)<newSL)
      setSLch(con(q), d, newSL);
      SL( (m,q,d), newSL);
end;
function setSLch(ch, c, newSL);
  ch is channel; c is AC; newSL is integer;
  for each d ∈ α(c, ch) do
    for each p ∈ π(d,ch) do
      setSLp(p, d, newSL);
end;
```

The function setSLch informs all ports that are connected to a channel at which synchronization layer the return values of RCs are updated, while the function setSLp forwards this information from a port to the module that owns the port. The function setSLm stores the provided synchronization layer information for a triple *(m, p, c)* with $m \in \kappa(mod)$, $p \in \pi(m)$ and $c \in RC(p)$ and assigns the synchronization layer to all triples $(m, q, d) \in \tau((m,p,c))$ and also informs the channel that is connected to q about the synchronization layer at which d will be called. Note that a synchronization layer of a triple is only updated when the new value is larger than the old value.

Given the functions setSLp, setSLm and setSLch, all synchronization layers can be determined by calling setSLch(ch,c,0) for each AC *c* of each channel *ch*.

This will inform all channels that their ACs will occur on synchronization layer 0. In fact, this is only correct for channels whose ACs are not called due to combinatorial dependencies. So if the function selects a channel whose inputs get set due to combinatorial dependencies first, false SL information will be distributed. However, this false information will then be overridden by the correct information as soon as the channel whose ACs are not called due to combinatorial dependencies gets initialized with setSLch(ch,c,0). It is important to notice that the if-clause in function setSLm prevents correct information to be overridden.

Thus, the numbers shown in Fig. 4.2 will be determined by applying setSLch(ch,c,0) to each AC *c* of each channel *ch* in the model, provided that the sets τ and α are set up correctly in the arbiters and channels.

4.3.3 Use of Synchronization Layers

Now we can use the synchronization layer information as follows:

At start of simulation a global synchronization layer is set to zero. Whenever a simulation process reaches a point at which it is about to do RCs followed by ACs without simulation time passing in between (i.e. there exists a combinatorial dependency between the RCs and the ACs), the process will check whether the global synchronization layer is equal to or larger than the highest synchronization layer of the RCs the module wants to perform. If the check fails, the process will be suspended and the simulation continues with another runnable process. Now an arbitrary number of delta-cycles may pass, in which other processes may become runnable and will be executed. During this time other processes may also be suspended because their synchronization layer check fails. When there are no more processes ready to run (which is normally the point of time at which the simulation time is increased), the global synchronization layer gets incremented. All suspended processes related to the new synchronization layer number are now started again. Due to these wake ups, other processes may get started, suspended due to the synchronization layer checks, and again an arbitrary number of delta-cycles may pass. When there are no more runnable processes, the synchronization layer gets incremented again, which should wake up the processes that want to execute on this new synchronization layer. This sequence is repeated until there are no more runnable processes and no more synchronization layer related suspended processes. Then, and only then the synchronization layer is reset to zero and the simulation time is advanced.

For example if the process of Arbiter2 in Fig. 4.2 wakes up because of the event from ch4, which happens at synchronization layer zero, it does not know which event started it. So it will check the global synchronization layer and find it zero. If it wants to make sure that the value on channel 4 is stable, it will suspend and reawake at synchronization layer one. But since it knows it needs both a stable value

from ch5 and ch4 to arbitrate correctly, it will not do this, but suspend and re-awake at synchronization layer 2.

4.3.4 Incorporation into DES

To examine the concept of synchronization layers, it has to be used within a DES. The following explanations refer to SystemC but can also be mapped onto other DES.

SystemC and C++ offer simple means by which π, *con* and *par* (in all their variations) can be determined and so we implemented a set of small base classes for ports, channels and modules from which the designer can derive its own modules, ports and channels. Thereby the synchronization layer information distribution as described in Section 4.3.2 gets enabled automatically. The information that must be added by the designer is the definition of α and τ and the synchronization layer checks, but the base classes for the channels and modules offer simple APIs for that.

The SystemC kernel execution is shown in Fig. 4.3. The dark shaded boxes, arrows and texts show the standard kernel execution according to [2], while the light shaded box shows the necessary additions to use the synchronization layers. We added those changes using a small kernel extension that only needs one additional line of code in the standard SystemC kernel, while the rest of the extension

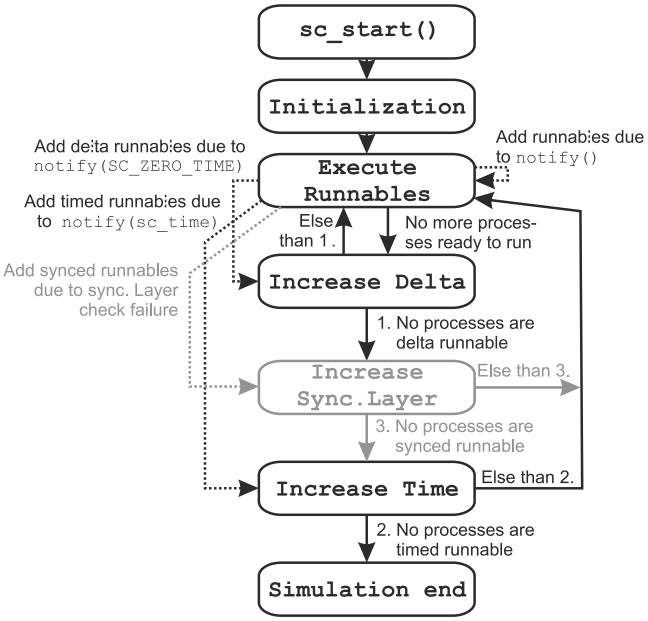


Fig. 4.3 Extended SystemC kernel execution

is kernel external. Note that the transition marked as 1. ends in Increase Time in the standard execution.

Before sc_start() the synchronization layer information is distributed as described in Section 4.3.2. During simulation and before simulation end the behavior of the modified SystemC kernel matches the one described in Section 4.3.3.

4.4 Experimental Results

To verify the performance valuation mentioned in Section 4.2, we implemented a simple, scalable test scenario. The test system comprises of an adjustable number of arbiters, which combinatorially arbitrate between two masters, and are connected as shown in Fig. 4.2. To increase the number of arbiters in the system of Fig. 4.2, the output of arbiter 2 is connected to the first input of the first additional arbiter, while its second input will be driven by a new master. The outputs of the new arbiters are connected to further new arbiters or to the target module, thus extending the arbiter chain shown in Fig. 4.2. This simple design is a worst case scenario for synchronization, as the first input of each arbiter is always driven through the longest possible combinatorial chain, while the other input is always driven directly by a clocked process. The system was implemented using all but the negative edge exploitation approach, because it does not support chain lengths longer than one.

Each master issues a request, waits for the acceptance of the request and then waits a randomized number of clock cycles. This sequence is repeated until each master has successfully sent out 100,000 requests.

So besides the adjustable number of arbiters and therefore an adjustable combinatorial chain length, the other test parameters where the size of the data within a request and the average number of clock cycles to wait between consecutive requests (denoted as the *break* in the following). The complete test comprised of about 3,000 different configurations, but due to space restriction we will show only the most important ones here.

Figure 4.4 shows how simulation time changes with increasing length of the combinatorial chain from master 1 to the target. For all measurements the average latency between consecutive requests was 7.5 clock cycles and the data size was 64 bit. As stated in Section 4.2 the delta-cycle waiting approach leads to large simulation times, when chain lengths exceed three stages.

So with short chains only, the difference between the approaches is minimal, but gets significant as soon as there is at least a chain of length 4 or many parallel combinatorial modules in the system.

Figure 4.5 shows how simulation time depends on the length of the break between consecutive requests. It can be seen, that the always transmitting technique is strongly affected by that, while the other approaches are not. The reason is that with increasing break length, the number of ‘no-change’ transfers increases, stressing the simulation execution. On the other hand, in systems where each module

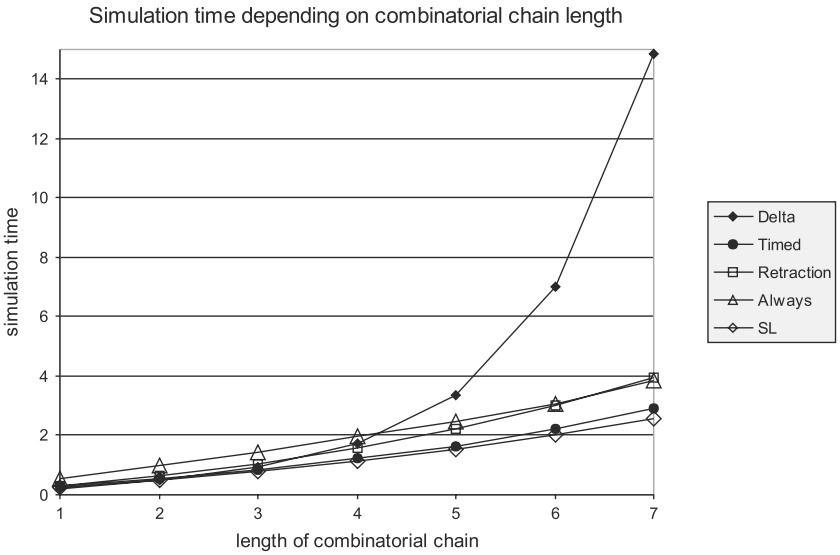


Fig. 4.4 Experiment: simulation time depending on combinatorial chain length

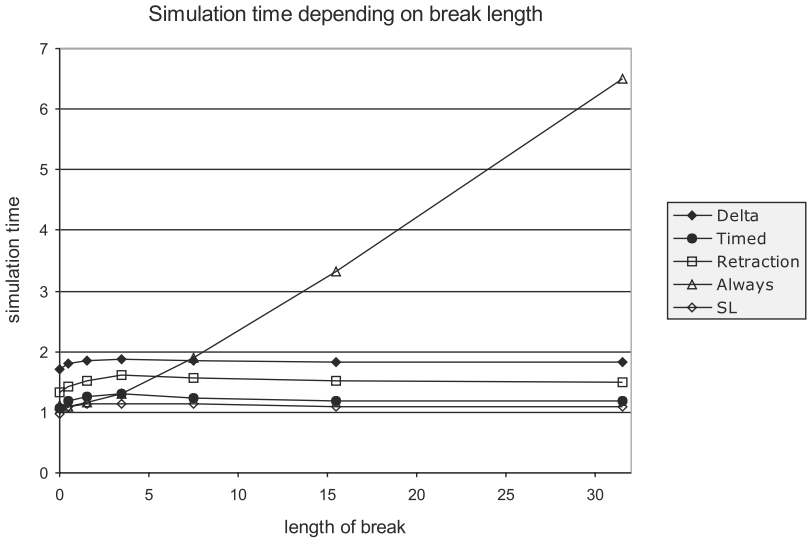


Fig. 4.5 Experiment: simulation time depending on break length

communicates at nearly every clock edge always transmitting is an adequate alternative to timed waiting or synchronization layers.

Figure 4.6 shows the number of lines of code which were necessary to implement the behavioral parts of the modules and channels. By that we mean just the code inside interface method calls and simulation processes, the rest of the code is neglected. Explicit retraction introduces by far the most code overhead, while the

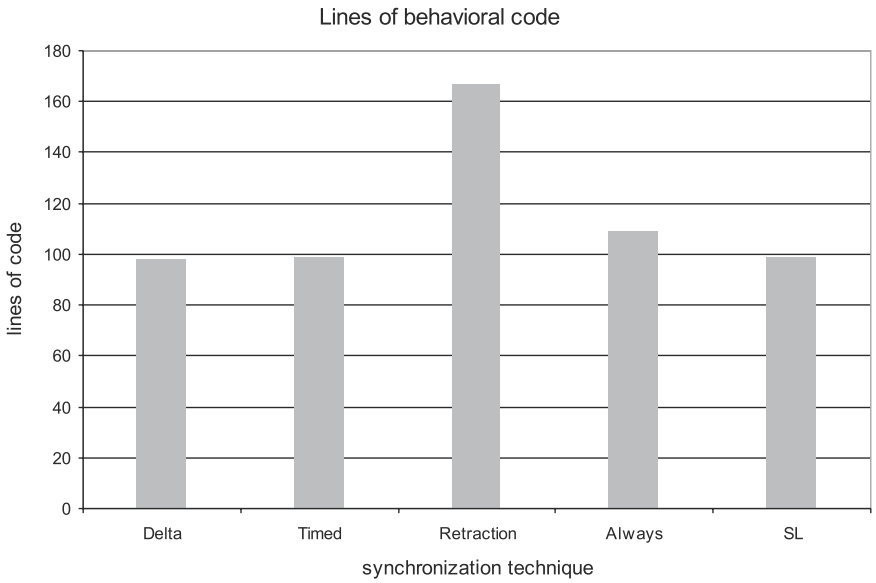


Fig. 4.6 Analysis: lines of behavioral code

other approaches all share about the same amount of code. The reason is that with explicit retraction, a lot of effort has to be spend on being able to compensate temporal mis-arbitration due to communication 'glitches'.

In summary, always transmitting fails when module communication happens only infrequently, delta cycle waiting fails with long combinatorial chains and explicit retraction introduces a significant code overhead. The two techniques that provided best performance, that scale best when combining chain length and break lengths and that introduce only a small code overhead are therefore timed waiting (as expected in Section 4.2) and our synchronization layer approach.

But since the timed waiting technique introduces undesirable simulated time delays, we favor the use of synchronization layers, in case the system comprises of many combinatorial modules.

4.5 Conclusion

In this paper we discussed the problem of modeling combinatorial dependencies accurately at the transaction level. We described various solutions that have been proposed and compared them to each other.

Out of this comparison came the idea for a novel approach, which we presented and evaluated as the *synchronization layer* approach. Experiments showed that the

novel approach can compete with the best performing already existing solutions, while avoiding their disadvantages.

References

1. ARM Limited (2007) RealView ESL API. http://www.arm.com/products/DevTools/RealViewESLAPIs.html. Accessed 02 February 2007
2. Black DC, Donovan J (2004) SystemC: From the Ground Up. Kluwer, Dordrecht, The Netherlands
3. Burton M, Morawiec A (2006) Platform Based Design at the Electronic System Level. Industry Perspectives and Experiences. Springer, Dordrecht, The Netherlands
4. Cai L, Gajski D (2003) Transaction Level Modeling: An Overview. In: International Conference on Hardware/Software Codesign and System Synthesis. Wiley-IEEE, Hoboken, NJ
5. Ghenassia F (2005) Transaction-Level Modelling with SystemC. Springer, Dordrecht, The Netherlands
6. Groetker T, Liao T, Martin S(2002) System Design with SystemC. Kluwer, Dordrecht, The Netherlands
7. IBM (2001) The CoreConnect Bus Architecture. http://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture. Accessed 21 December 2006
8. Klingauf W, Guenzel R, Bringmann O, Parfuntseu P, Burton M (2006) GreenBus – A Generic Interconnect Fabric for TLM. In: Design Automation Conference, 2006 43rd ACM/IEEE. ACM, New York
9. OCP-IP (2006) A SystemC OCP Transaction Level Communication Channel r2.1.3. http://www.ocpip.org/socket/systemc/. Accessed 03 December 2006
10. Palnitkar S, Parham D (1995) Cycle Simulation Techniques. In: Verilog HDL Conference, 1995. Proceedings. Wiley/IEEE Press, Hoboken, NJ
11. OSCI TLM WG (2007) TLM Transaction Level Modeling Library, Release 2.0 Draft 2. http://www.systemc.org/downloads/drafts_review. Accessed 05 December 2007

Abbreviations

AC	Alteration call
CASI	Cycle accurate simulation interface
CATLM	Cycle accurate transaction level modeling
DES	Discrete event simulator(s)
IMC	Interface method call
OCP-IP	Open core protocol international partnership
OPB	On-chip peripheral bus
RC	Readout call
RTL	Register transfer level
TLM	Transaction level modeling
SLDWG	System level design working group

Chapter 5
An Integrated SystemC Debugging Environment*

Frank Rogin[1], Christian Genz[2], Rolf Drechsler[2], and Steffen Rülke[1]

Abstract Since its first release the system level language SystemC had a significant impact on various areas in VLSI-CAD. One remarkable benefit of SystemC lies in the support of abstraction levels beyond RTL. But being able to implement complex System-on-Chip (SoC) designs in SystemC raises the necessity of new techniques to support debugging, system exploration, and verification.

We present an integrated debugging environment that facilitates designers in simulating, debugging, and visualizing their SystemC models combining high-level debugging with visualization features. Our work mainly focuses on developing an easy to handle interface which supports debugging and system exploration of SystemC designs.

Keywords High-level Debugging, SystemC, Graphical Debugging Environment, System Level Design, System Exploration and Visualization

5.1 Introduction

SystemC is a C++ based system level description language that facilitates system architects to specify their designs using a broader spectrum of abstraction levels than traditional hardware description languages (HDL), like VHDL or

[1] Fraunhofer Institute for Integrated Circuits, Division Design Automation, Zeunerstraße 38, 01069 Dresden, Germany; Email: {frank.rogin, steffen.ruelke}@eas.iis.fraunhofer.de

[2] University of Bremen, Institute for Computer Science, Bibliothekstraße 1, 28359 Bremen, Germany; Email: {genz, drechsle}@informatik.uni-bremen.de

* Partial funding provided by SAB-10563/1559 and European Regional Development Fund (ERDF).

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Verilog, do. Equivalently to HDLs, cycle accurate operations as well as word and bit level data types are supported. But also untimed algorithmic descriptions can be included into a model raising the abstraction level, e.g. to transaction level modelling (TLM). Thus, pure functional and even object-oriented code can be used for specifications where the compiled model can be executed with higher performance than an HDL simulation can do. All these features make SystemC an excellent approach for modelling SoCs and allow implementing HW/SW co-designs at various abstraction levels. For more details concerning SystemC see [14].

Currently, the SystemC standard does not define a sophisticated debugging interface, nor it provides any visualization support. Even though the simulation kernel offers an interface to access signal values and interconnection structure, a direct communication with the kernel requires additional C++ code in the model. This forces a designer to gain advanced knowledge of many details regarding the system and SystemC itself. Another point is that with growing integration of SW components in HW designs, also size and complexity of the considered system tend to increase. Thus it becomes less obvious where to start and which blocks to observe in a debugging process. Furthermore, language features such as multi-threading and event-based communication increase the program complexity and introduce nondeterminism in the system behavior. Consequently, many of the features mentioned above potentially complicate debugging SystemC models.

In this paper we introduce an integrated debugging environment (IDE) for SystemC. Besides simulation control and data hiding our approach extends the data introspection capabilities of SystemC. It is non-intrusive and does not alter the simulated model, nor the simulation kernel, or additional libraries (C++ STL, SCV). Our solution supports SystemC aware debugging [15] with visualization capabilities [9]. The user debugs and visualizes a design at arbitrary levels of abstraction working at the functional level (e.g. finite-state machines, algorithms, data-flow graphs) or the system level that means at the level of SystemC concepts (e.g. signals, ports, events, processes, modules). The debugger kernel is based on the Open Source debugger GDB [10] while the visualization makes use of the visualization engine from Concept Engineering [4]. The visualization engine generates different views of the model, supporting cross probing and annotation of the visualized context. During a debug session the user has various possibilities to explore dynamic and static debugging information, and to control the simulation. Thus, he gets a fast and concise insight into the observed SystemC model which accelerates and eases defect (also colloquial bug) detection, understanding, localization and correction.

The rest of this chapter is organized as follows. Section 5.2 discusses related approaches and tools which allow debugging SystemC designs. In Section 5.3 the general architecture of our IDE is described in more detail while Section 5.4 considers the provided debugging interface and the graphical frontend and its debugging support. In Section 5.5 we illustrate some IDE features exemplarily and

demonstrate their feasibility using a short example. Finally, Section 5.6 concludes the paper and gives a perspective on future work.

5.2 Related Work

Debugging SystemC models requires hybrid techniques that grant access to design components quickly but also allow evaluating ordinary C++ code. Unfortunately, C++ fragments cannot be reached by using SystemC data introspection techniques. And even though there are commercial and academical tools, supporting SystemC debugging, only few of them offer an advanced visual interface to the designer that has features like data hiding and cross probing to the source code level.

RealView Debugger Suite [1] comprises a complete integrated development suite that allows to implement, to simulate, to debug, and to analyze SystemC/C++ designs. It addresses architectural analysis as well as SystemC component debugging at low level and at transactional level where especially the debugging of embedded applications (running on remote targets such as ARM processors) is supported. *Platform Architect* [5] targets system-level design and verification based on the Eclipse development framework [6]. It utilizes a native simulation environment which is specially adopted to fit SystemC needs. The integrated debugger offers specific commands supporting source-level and simulation breakpoints and QThread debugging. Additionally, the user can initiate a graphical transaction tracing of SystemC events, threads, and interface method calls activations. Contrary to our approach both commercial solutions come with their own vendor-specific SystemC kernel which prevents the easy integration into an already existing design flow.

The GRACE++ system [16] uses SystemC simulation results to create Message Sequence Charts in order to visualize and analyze inter-process communication. Various filters help to reduce information complexity. The approach presented in [3] applies the observer pattern [8] to connect external software to the SystemC simulation kernel. This general method facilitates loose coupling but requires possibly undesired modifications of the kernel.

One of the first approaches accomplishing SystemC design visualization has been introduced in [11]. The implementation uses the SystemC kernel to analyze models during execution. An interactive graphical backend facilitates the design visualization. Even though models can be specified using C++ features, but analysis and visualization are limited to SystemC objects. Only the data flow can be viewed, no behavioral information is available. Since this approach has to execute the model without further information of declarations, it is not aware of detailed positional information regarding the objects. Hence, cross probing facilities are very restricted.

Another approach that facilitates designers in visualizing SystemC models is [7]. Since it is based on data introspection too, it shares many restrictions with [11].

One major difference to [11] is the usage of an own graphical user interface that has been especially designed for this approach but does not support features like cross probing or path fragment navigation.

Contrary to the works described above, SystemCXML [2] and LusSy [12] do not use data introspection for the purpose of analysis. While the extraction of the hierarchy in SystemCXML is done via Doxygen, LusSy uses PINAPA [13]. The visualization is realized as graph structures. But while LusSy generates a graphical output showing the control flow graph of processes only, SystemCXML limits the visualization to data flow graphs.

None of the listed tools and approaches includes the following set of features:

- Work with the OSCI SystemC kernel
- Support high-level debugging
- Provide a highly developed visualization of SystemC designs

From this a small set of requirements can be derived, to support high-level SystemC debugging:

- Non-intrusiveness to prevent the model, the SystemC kernel and additional libraries from being altered
- Advanced commands implementing a high-level debugging interface
- Visualization that allows for abstraction, with direct linkage to all lower abstraction levels defined in the design

All mentioned works do not meet the requirements in terms of non-intrusive debugging and visualization facilities.

5.3 Debugging Environment

Our IDE consists of three components. Each of these components realizes a particular task. As sketched in Fig. 5.1 our debugging flow starts at the original system description which is being compiled to an executable.

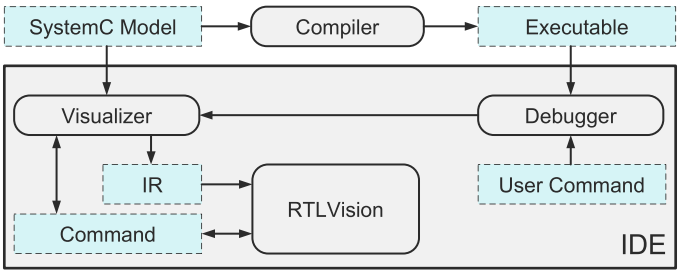


Fig. 5.1 Architecture of the IDE

The executable can be run in the debugger. In parallel the system description is statically analyzed by the visualizer. The intermediate representation (IR) that is generated after analysis can be used to render the model inside the graphical back-end. RTLVision from Concept Engineering is used for this purpose. After passing the SystemC elaboration phase successfully the debugger waits for user commands. Those commands can be used to show or to hide details inside the visualization back-end, as well as to control the simulation of the executed model. All commands that influence the graphical view are directly propagated to the visualizer. Being aware of the model structure the visualizer assembles commands and maps SystemC components to the appropriate graphical symbols. Thus, RTLVision can be instructed to switch to specific parts of the design and to update signal values during execution.

The communication between the visualizer of our environment and RTLVision is realized using TCP/IP. Thus a system engineer has a comfortable and secure way sharing his knowledge with other colleagues far away. The exchange of data among the visualizer and the debugger kernel is done using a protocol based on socket communication.

5.4 Debugging Features

This section introduces the features our IDE offers for debugging SystemC applications. First, the debugging capabilities provided at system level are summarized. Second, the visualization interface is detailed.

5.4.1 Debugging Interface

System level debugging requires various kinds of high-level information that should be fast and easy retrievable. There, defects occur at different abstraction levels that influence the appropriate debugging procedure and the used tools.

At **functional level** the defect is located at the source code level that means mainly in low-level program details such as an erroneous implemented algorithm or a faulty memory management. Because of SystemC C++ conformance due to a class library, each standard C++ debugger can be applied at this level. For that reason, our debugger kernel is based on the Open Source debugger GDB. GDB provides various features which include for example stopping and continuing the simulation, or examining the actual program stack, local variables, the memory, or source files.

At the more abstract **system level** the architecture and/or the interaction between the different parts of a SystemC design are responsible for defects such as a wrong communication between components (e.g. a specific protocol) or the faulty integration of an (third-party) IP block. C++ debugging features are not sufficient to retrieve such defects quickly. Hence, the IDE enables the user to debug a SystemC design at

system level. Here, high-level breakpoints (e.g. breakpoints on events or processes), the retrieval of static and dynamic simulation information (e.g. signal paths, or state of scheduling queues), and the graphical design representation provide comprehensive debugging support. A number of commands allow to interactively control the visualization of a SystemC design and its simulation state. This additional abstraction further simplifies and thus accelerates debugging. To explore the static system structure as well as the dynamic behavior, the IDE offers two command types:

- **Examining commands**. These commands allow getting a fast insight into the parts of a design relevant for the actual debug session while non-relevant data are explicitly excluded.
- **Monitoring commands**. Commands of this type support the user in obtaining different data about the simulation state (such as signal values, or process activations) logged over a specified simulation time.

Examining and monitoring commands do not only have a direct impact on the execution of the model. They also alter the visualization of the design. The given set of commands can be used to follow critical paths being observed for incorrect behavior. But since these commands do not rely on the stimuli generated by a certain test bench, they can be used for system exploration as well. Table 5.1 assembles a list of visualized high-level debugging commands.

An important requirement for all monitoring commands is a fast tracing of requested values where the impact on the simulation performance should be minimized. Retrieving current values directly by patching several SystemC kernel methods would be the fastest, easiest, and most obvious approach. But to meet the requirement of a non-intrusive solution, we use library interposition and preload a shared library (**libscpatch.so** in Fig. 5.2). This library overwrites the corresponding kernel methods with methods using callbacks to forward needed debugging information. To activate preloading the LD_PRELOAD environment variable has to be set. Thus, the dynamic linker is instructed to search our library first, thus using the patched methods.

Table 5.1 Visualized debugging commands

Examining commands	
vlsb	Visualize the specified channel and all connected modules.
vlsio_rx	Highlight I/O ports matching the given regular expression of the specified module.
vlsm	Highlight all SystemC modules in the given hierarchy.
vzp	Visualize the given process and all its driving and driven signals.
Monitoring commands	
vlsv	Label the specified signal or port with the current value that it holds at a specific time stamp.
vrmv	Remove the label of the specified signal or port.
vtrace	Trace the given signal or port and record its value at each simulation time step until the specified time is reached, then tracked values are attached as label.
vtrace_at	Trace the given signal or port and record its value at the specified simulation time, then the tracked value is attached as label.
vpt	Visualize the trigger events for the given process.

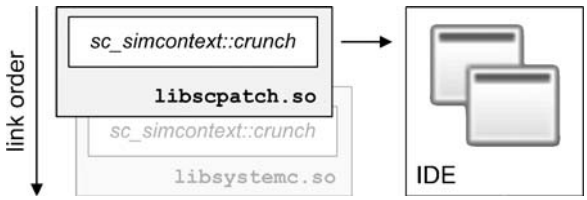


Fig. 5.2 Preloading kernel methods

5.4.2 Graphical Interface

The graphical interface for what RTLVision is used, bridges different abstraction levels. Since our approach bases on the GDB debugger, text return messages proposing changes regarding the system state can be very detailed. The graphical interface bypasses this problem by rendering the structure of the simulated model to three different views, as can be seen in Fig. 5.3. The schematic view shows modules as functional blocks that can be collapsed and signals as interconnecting wires. The cone view limits the set of currently displayed objects to a critical path. Both views are bidirectionally connected to a source code view. The advantages of these visualization features in our approach are:

- Annotation of SystemC names and declaration names
- Hierarchical visualization
- Cross probing
- Path fragment navigation
- Module exploration

All these features are controlled by the IDE observing the simulator that proposes each state change to RTLVision. A state change alters the current display by:

- Highlighting signals, modules or ports
- Expanding or collapsing module hierarchies
- Annotating values to signals and ports

5.5 Practical Application

This section shows the practical application of our proposed debugging features. Some provided features are highlighted in the first part of this section while the second part demonstrates the successful and efficient debugging of a faulty RISC-CPU design.

5.5.1 Feature Illustration

To illustrate the utilization of our IDE we used the RISC-CPU design that is provided with the OSCI SystemC v2.0.1 library package [14]. Fig. 5.3 shows an example debug session simulating this design. The different views allow exploring

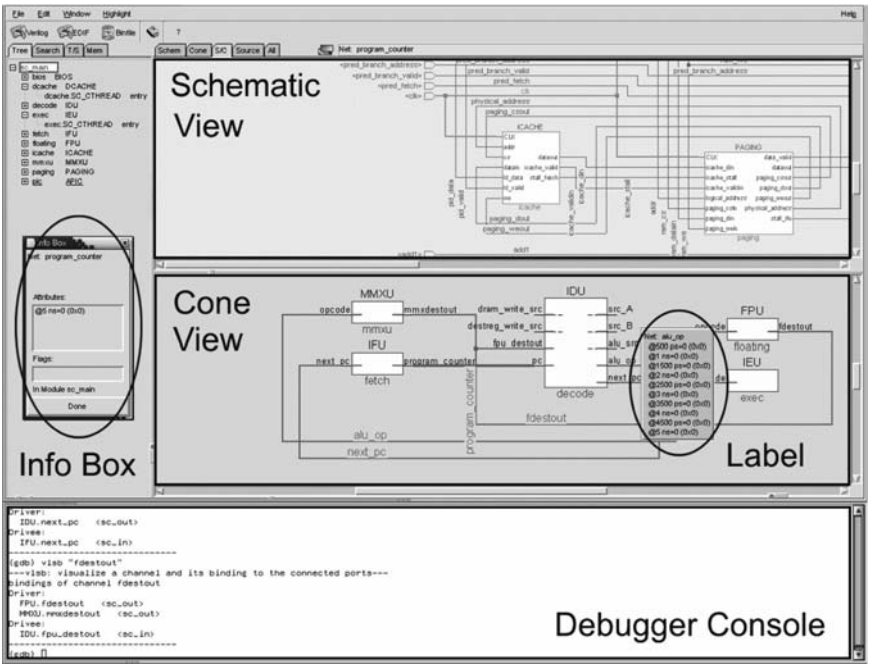


Fig. 5.3 Example debug session

the RISC-CPU design at various abstraction levels. Static and dynamic debugging information are presented by different colorings, info boxes, labels, and dedicated displays in the GUI, and as text output in the debugger console. Thus, the developer gets a quick and concise insight into the overall CPU design structure and its behavior.

The following two commands illustrate the provided visualized debugging functionality exemplarily.

The **vlsb** command (Table 5.1) visualizes the specified channel and all connected modules in the cone view of RTLVision. In case of a failure related to a specific signal the user gets a quick overview about all its connections. Thus, architects can focus on error search to the relevant modules only which helps accelerating debugging. Fig. 5.4 sketches the visualization output after calling **vlsb** with two signals of the RISC-CPU design in order to check their bindings to the right ports:

```
(gdb) vlsb "ram_cs"
(gdb) vlsb "next_pc"
```

The **vtrace_at** command (Table 5.1) is a typical representative of the monitoring command type. It traces the given signal or port and records the actu al value at the specified simulation time stamp. The logged value is attached as label text

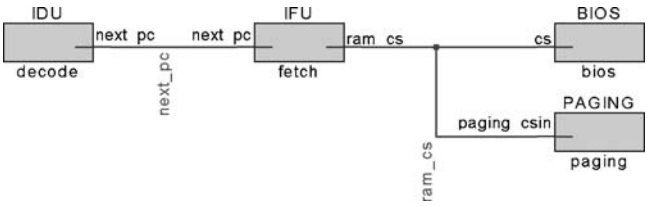


Fig. 5.4 Debug command vlsb

in RTLVision and can be displayed in an info box additionally. Monitoring dedicated signal values during simulation is very helpful when the user does not exactly know what is going wrong and when the defect infection occurs. Fig. 5.5 illustrates the visualized tracing of the top-level signal **addr** in the RISC-CPU design at different time stamps to check whether the right addresses are forwarded to the RAM:

```
(gdb) vtrace_at "addr" 42000
(gdb) vtrace_at "addr" 46000
(gdb) vtrace_at "addr" 50000
(gdb) c
...
(gdb) vlsb "addr"
```

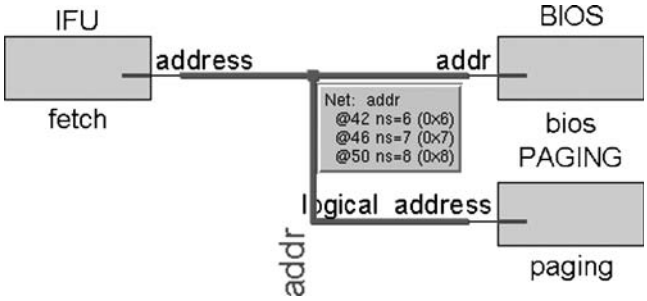


Fig. 5.5 Debug command vtrace_at

Table 5.2 underlines the efficiency of our non-intrusive, patch-free approach using library interposition (Section 5.4.1) while illustrating the performance of the **vtrace** command (Table 5.1). So, the observation of 750,000 data sets over 125 signals leads to a slow down of factor 4 compared to a trace-free simulation while the tracing of 50 signals increases the simulation time about 80%.

Table 5.2 Exemplary performance slow down

	Slowdown over simulation time (# observed data sets)		
# of traced signals	1,000 ns	2,000 ns	3,000 ns
0	1.0	1.0	1.0
5	1.3 (10,000)	1.3 (20,000)	1.4 (30,000)
50	1.8 (100,000)	1.8 (200,000)	1.8 (300,000)
75	2.3 (150,000)	2.6 (300,000)	3.0 (450,000)
100	3.0 (200,000)	3.2 (400,000)	3.6 (600,000)
125	3.2 (250,000)	3.9 (500,000)	4.0 (750,000)

5.5.2 Example Debug Session

To show the efficiency and feasibility of our solution we want to investigate why a small program works faulty on the RISC-CPU design. For this purpose, we use several exploration and visualization features (Section 5.4.1) to locate the defect quickly. First, the following program is simulated on the RISC-CPU which indicates its incorrect processing.

```
1:    ldpid  0
2:    movi   R5, 10
3:    movi   R6, 6
4:    movi   R7, 2
5:    add    R4, R5, R6
6:    mul    R4, R7, R4
```

After the initialization statement the three registers R5, R6, and R7 are loaded with the integer values 10, 6, and 2, respectively. Then, R5 and R6 register contents are added and the result is multiplied with the register content of R7, subsequently. Thus, after program execution the value 32 has to be stored in register R4. Instead, the register dump shows that R4 contains the value 8:

```
REG DUMP ============================
         R4(0x00000008) R5(0x0000000a)
         R6(0x00000006) R7(0x00000002)
```

We start a debug session to find the failure cause. For simplification reasons we suppose that the ALU works correctly. Furthermore, the right integer values seem to be loaded into the registers, as seen in the register dump above. So, we assume that the defect has to be searched in the controlling of the ALU where the ALU is implemented by the module instance **IEU**. To get the right control signal the

vlsio_rx command (Table 5.1) is applied at first. We suppose that the name of the attached control port includes the string **code**:

```
(gdb) vlsio_rx "IEU" "code"
```

Using the path fragment navigation feature in RTLVision subsequently shows that the only port reported by the **vlsio_rx** command is connected to the signal **alu_op** (Fig. 5.6).

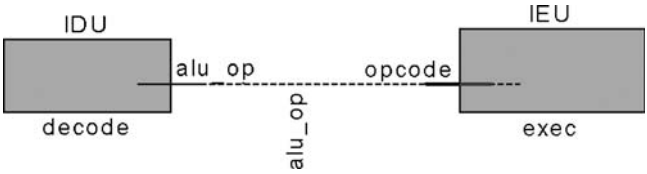


Fig. 5.6 Tracking down the op-code signal

Besides, we should trace the program counter represented by the signal **program_counter** to observe the program execution. Consequently, we initiate a monitoring of the two interesting signals using the **vtrace** command (Table 5.1) and continue simulation:

```
(gdb) vtrace "program_counter" 110000
(gdb) vtrace "alu_op" 110000
(gdb) c
```

After simulation has stopped we investigate the traced behavior. To focus the error search onto the relevant design parts only, the **vlsb** command (Table 5.1) is applied (Fig. 5.7):

```
(gdb) vlsb "program_counter"
(gdb) vlsb "alu_op"
```

Knowing that the reset phase ends after 30 ns the first operation code of interest is transferred from the decoder unit (module instance **IDU**) to the ALU at 35.5 ns. The reported value **0x0** corresponds to the **ldpid** command in our example program. From 49.5 ns till 91 ns the operation code holds **0x3**. The traced values of the program counter indicate that this code corresponds to the three **movi** commands (line 2 to 4) loading registers R5, R6, and R7 with integer values. The next operation code **0x4** is transferred at 91.5 ns which should notify the **add** command. But as we know from the processor specification the operation code for additions has to be indicated by **0x3**. Looking into the source code of the instruction decoder using the source code view in RTLVision shows the wrong operation code in line 161 causing the error:

```
153  case 0x01:       // add R1, R2, R3
...
161  alu_op.write(4); // WRONG CODE!
```

Fixing this statement and a subsequent simulation reports the correct result in register R4.

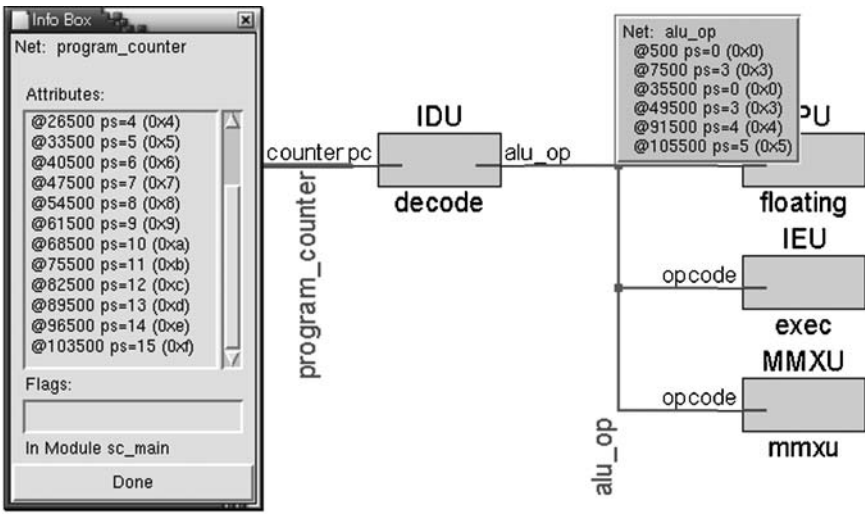


Fig. 5.7 Exploring traced signals

A conventional debug procedure would set several breakpoints on the right positions into the instruction decoder and the ALU. On any stop of these breakpoints we then had to print out the transferred operation code and the actual program counter. This can turn out to be a time consuming task where the printed values are split over and merged with the usual trace output in the debugger console. Thus, a fast and simple observation of interesting program details is made very difficult which complicates debugging.

5.6 Conclusion

In this work we introduced an integrated debugging environment (IDE) where the debugger kernel is based on the Open Source debugger GDB and the visual interface utilizes an available visualization tool. The special feature of our environment is its non-intrusive usability that means it does not alter any code (SystemC kernel, existing models, additional libraries) to enable using arbitrary SystemC designs in the IDE. We demonstrated the advantages of our debugging features applying them to the RISC-CPU design of the SystemC library.

Future work will improve the provided debugging and exploration functionality especially regarding an explicit TLM support. One of the main goals is to fit the debugging environment to the specific needs of the application being developed (e.g. CPU design).

Acknowledgments We would like to thank Lothar Linhard and Gerhard Angst from Concept Engineering, who supported this work.

References

1. ARM Ltd. MaxSim Developer. Home page: www.arm.com
2. D. Berner, H. Patel, D. Mathaikutty, J.-P. Talpin, S. Shukla: SystemCXML: An extensible SystemC front end using XML. Technical Report 06, FERMAT@Virginia Tech, Apr. 2005
3. L. Charest, M. Reid, E. Aboulhamid, G. Bois: A methodology for interfacing open source SystemC with a third party software. In Design, Automation and Test in Europe, Munich, Germany, pp. 16–20, 2001
4. Concept Engineering. Home page: www.concept.de
5. CoWare Platform Architect. Home page: www.coware.com
6. Eclipse Foundation. Project home page: www.eclipse.org
7. C. Eibl, C. Albrecht, R. Hagenau: gSysC: A graphical front end for SystemC. In European Conference on Modelling and Simulation, Riga, Latvia, pp. 257–262, 2005. Source available at www.iti. uni-luebeck.de/albrecht/gSysC
8. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design pattern – elements of reusable object-oriented software. In Addison-Wesley Professional Computing Series, 1999
9. C. Genz, R. Drechsler, G. Angst, L. Linhard: Visualization of SystemC designs. In IEEE International Symposium on Circuits and Systems, New Orleans, USA, pp. 413–416, 2007
10. GNU debugger. Home page: www.gnu.org/software/gdb
11. D. Große, R. Drechsler, L. Linhard, G. Angst: Efficient automatic visualization of SystemC designs. In Forum on Specification and Design Languages, Frankfurt, Germany, pp. 646–657, 2003
12. M. Moy, F. Maraninchi, L. Maillet-Contoz : LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In Fifth International Conference on Application of Concurrency to System Design, St. Malo, Frankreich, pp. 26–35, 2005
13. M. Moy, F. Maraninchi, L. Maillet-Contoz : PINAPA: An extraction tool for SystemC descriptions of systemson-a-chip. In ACM International Conference on Embedded Software (EMSOFT'05), Jersey City, USA, pp. 317–324, 2005
14. OSCI. SystemC. Home page: www.systemc.org
15. F. Rogin, E. Fehlauer, S. Rülke, S. Ohnewald, T. Berndt: Non-intrusive high-level SystemC debugging. In Advances in Design and Specification Languages for Embedded Systems. Springer Netherlands, pp. 131–144, July 2007
16. A. Wieferink, M. Doerper, T. Kogel, G. Braun, A. Nohl, R. Leupers, G. Ascheid, H. Meyr: A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In IEE Proceedings: Computers & Digital Techniques, volume 152, pp. 3–11, Jan. 2005

Chapter 6
Measuring the Quality of a SystemC Testbench by Using Code Coverage Techniques

Daniel Große[1], Hernan Peraza[1], Wolfgang Klingauf[2], and Rolf Drechsler[1]

Abstract The system description language SystemC enables to quickly create executable specifications at adequate levels of abstraction for both hardware/software integration and fast design space exploration. Besides the modelling of a system, verification has become a dominant factor in circuit and system design. Since SystemC is a versatile language based on C++, testbenches at different abstraction levels can easily be built. But the fault coverage of a manually developed testbench is hard to quantify. In this paper, an approach for measuring the quality of SystemC testbenches is presented. The approach is based on dedicated code coverage techniques and identifies all the parts of a SystemC model that have not been tested. Experimental results show the applicability of our methodology.

Keywords SystemC, Testbench Quaility, Coverage

6.1 Introduction

To cope with the design complexity of hardware/software systems that consist of up to one billion transistors, raising the level of abstraction in modelling has been exercised during the past years in the computer aided design community. In this context, C/C++-based languages have found entrance into industry. Here, the system description language SystemC is the de facto standard and was standardized by the IEEE [13]. Additionally to the inherent SystemC feature of specifying hardware and software in one language the concept of *Transaction Level Modeling* (TLM) [2] is supported by SystemC. TLM allows describing the communication in a system in terms of abstract operations (transactions).

[1] Institute for Computer Science, University of Bremen, 28359 Bremen, Germany;
Email: {grosse, drechsle}@informatik.uni-bremen.de

[2] Department E.I.S., Technical University of Braunschweig, 38106 Braunschweig, Germany;
Email: klingauf@eis.cs.tu-bs.de

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Besides the modelling aspect the verification – i.e. ensuring the correct functional behaviour – is the most challenging problem. Since complete formal verification methods are only applicable to medium sized designs, simulation-based techniques are used most frequently [6, 17]. Here system level languages like SystemC already offer some features for verification and are therefore superior to traditional *Hardware Description Languages* (HDLs). For example, in SystemC the testbench can easily be integrated as part of the model and all features of C++ can be used for the generation of tests. Also the result analyzer that is typically build to check the response of the *Device Under Verification* (DUV) is a SystemC module. As an add-on for SystemC the *SystemC Verification* (SCV) library has been introduced [15]. Besides advanced verification features like data introspection and transaction recording the SCV library enables constraint-based randomization.

However, all these verification features do not include a measure how thorough the design was executed during the simulation. As the size of the testbench grows the designer needs a reliable feedback about its quality.

In this paper, an approach for SystemC to measure the quality of the testbench is presented. Our analysis is based on dedicated code coverage techniques that we have developed for SystemC models. By exploiting automated code instrumentation based on a SystemC parser, for each test run a coverage report is generated that presents the user all statements in the model that have not been executed during simulation. The report is based on the analysis of the exercised control flow statements. It includes exact source code references to unexecuted code blocks in combination with SystemC specific information like process context and hierarchy information.

The rest of this paper is structured as follows. Related work is described in the next section. In Section 6.3 we present our approach. We start with the overall flow and continue with a detailed description of the three phases of our approach. Along the way we provide an example to show the effects of each phase. Case studies for two SystemC designs are presented in Section 6.4. The first design is a RISC CPU and the second design is a TLM-based video processor. Finally, in Section 6.5 the paper is summarized.

6.2 Related Work

In software testing code coverage techniques have been used to measure the fraction of code that has been exercised by a test case [1]. From this domain coverage methods have been derived and extended for HDLs. For Verilog or VHDL several approaches and tools exist (for an overview see e.g. [16]). However, to the best of our knowledge no code coverage method to measure the quality of a SystemC testbench has been proposed. Note, that approaches based on standard C++ coverage tools (like e.g. the GNU COVerage tool gcov [7]) have several drawbacks. On the one hand the SystemC kernel is also included in the coverage analysis. On the other

hand SystemC specific data like e.g. context information or hierarchy information is only implicitly available and has to be extracted manually.

In the following we present an approach to overcome such limits.

6.3 Measuring the Quality of a SystemC Testbench

In this section the code coverage-based approach for measuring the quality of the testbench is introduced. Our approach consists of three phases: SystemC analysis, code instrumentation and coverage analysis. Before the details on the three phases are given the overall flow is presented. Throughout the description of the phases a simple example is used to demonstrate the effects of each phase.

6.3.1 Overall Flow

The overall flow of our approach is depicted in Fig. 6.1. In the analysis phase the SystemC code of the DUV is parsed, analyzed and transformed into an *Abstract Syntax Tree* (AST) representation. This AST is traversed in the consecutive code instrumentation phase. During the traversal the original SystemC DUV is augmented with SystemC specific code that enables the collection of coverage information during simulation. Then, the rewritten SystemC DUV, the coverage library of our approach and the SystemC libraries are compiled into an executable. By running this executable simulation is performed and the data structures available through our coverage library are filled.

Finally, in the coverage analysis phase the collected data is interpreted and the coverage report is generated. By the report the verification engineer is informed which statements have not been executed due to the tests defined in the testbench. This information is presented with exact source code references to unexecuted

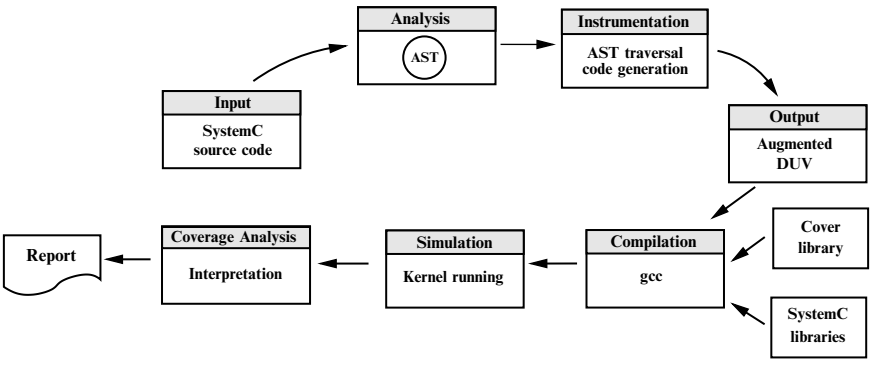


Fig. 6.1 Overall flow of our approach

blocks in the original SystemC DUV including hierarchy. Furthermore the frequency of the execution of statement blocks can be given for further analysis.

In the following we describe the three phases in more detail.

6.3.2 SystemC Analysis

For the transformation of the SystemC DUV into an AST the front-end from [5, 8] is used that is part of the design environment SyCE [3]. The parser of the front-end was build with PCCTS (Purdue Compiler Construction Tool Set) [14]. PCCTS enables the description of the SystemC syntax in form of a grammar, provides facilities for AST construction and finally generates a parser. Note that the front-end has an exact source code reference including character positions of each token. Therefore, a special C++ pre-processor has been implemented to allow for identification of the SystemC macros before they are expanded. The correct source code information annotated to each node in the AST is very important for our approach. Without this information only a non-reliable feedback for the verification engineer would be possible. In the following the analysis phase is demonstrated by an example.

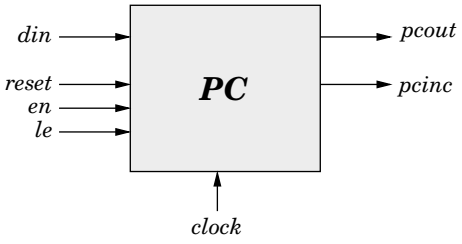


Fig. 6.2 Program counter

Example 1 *Since we use a program counter of a RISC CPU also as example for the other phases we give some details on this module. Fig. 6.2 shows the program counter (PC) with all its inputs and outputs. In order to address the 2,048 entries of the program memory, the PC has an 11 bit register which holds the address of the current instruction. Output* pcout *holds this address.* pcinc *outputs the address increased by one. An address can be loaded into the PC via the input* din, *if the input* le *(load enable) is set to 1. Using the* reset *signal, the PC can be set to 0. On every positive edge of the* clock *signal the current address is increased if the input* en *(enable) is set to 1. In Fig. 6.3 the method that computes the* next_state *of the PC is shown. This method is sensitive to the positive clock.* pc *is the internal register of the PC module. Figure 6.4 depicts a sample of the AST of this method, which has been generated by our tool. Please note that for each AST node only a fragment of the available information is shown. The second number in each line corresponds to the line number of the parsed element.*

```
 9  void prog_count::next_state() {
10   if (reset.read()) {
11    pc = 0; //reset to adress 0
12   } else {
13      if (en.read()) {
14       if (le.read()) {
15        pc = din; //load address
16       } else {
17         // increase counter
18         pc = pc.read() + 1;
19       }
20      } else {
21        pc = pc.read();
22      }
23   }
24  }
```

Fig. 6.3 Parts of original SystemC DUV

```
 1  10 IF
 2  10  LPAREN
 3  10  ID == "reset"
 4  10  DOT
 5  10     ID == "read"
 6  10  LPAREN
 7  10     RPAREN
 8  10  RPAREN
 9  10  LCURLY
10  11     ASSIGNEQUAL
11  11       ID == "pc"
12  11       OCTALINT
13  11     SEMICOLON
14  12     RCURLY
15  12  ELSE
16  12  LCURLY
17  13   IF
18  13    LPAREN
19  13      ID == "en"
20  ...
```

Fig. 6.4 AST of next_state method

As can be seen, the structure of the SystemC program is reflected and this representation is well suited for code instrumentation.

6.3.3 Code Instrumentation

In the code instrumentation phase the SystemC DUV is augmented with according instructions to allow for coverage analysis. The main steps in this phase are described in the following.

6.3.3.1 Coverage Library

First, the global variable *cov* is defined that holds an instance of our coverage class *COVER*. This class provides data structures like hash tables for coverage statistics as well as wrapper functions to take care of the control flow inside the methods of the DUV. Furthermore, the class has methods to analyze the collected coverage data and to generate the report for the user.

6.3.3.2 AST Traversal and Code Instrumentation

While traversing the AST, first the member functions that belong to a SystemC module are identified. Then, in each function the conditions of the control flow statements are substituted with wrapper functions. The idea is to perform a call-back during the simulation and thereby notifying the coverage class which control branch has been taken. The following control statements are distinguished: IF, IF/ELSE, SWITCH-CASE, FOR loop, WHILE loop. Next, the wrapper functions are explained.

6.3.3.3 Wrapper Functions

For the IF, IF-ELSE, FOR loop and WHILE loop the condition of the control statement is replaced by a wrapper function call. The arguments of the wrapper functions are:

1. The condition of the control statement (as Boolean and string).
2. The type of control statement.
3. Start position and end position of the block(s) that are executed if the control condition evaluates to true/false.
4. File name of the current method.
5. Class name if available.
6. Current method name.
7. *This* pointer, in case of a member function. The this pointer is used to distinguish between several instances of the same module.

The following example demonstrates the application of a wrapper function for an IF-ELSE control statement.

Example 2 *Consider again the program counter in Fig. 6.3 and focus on the if statement in line 10 and the corresponding else-branch starting in line 12. The condition of the if statement is the expression* reset.read(). *This expression is replaced by the wrapper function* wrapperStatement(…). *The instrumented code is depicted in Fig. 6.5. The first and second arguments of this function hold the condition as a Boolean and as a string, respectively. The third argument reflects the type of the condition statement – here* tIFELSE. *Then, the next four numbers mark the if-block, i.e. the if-block starts in line* 10 *at the absolute character position* 125 *and ends in line* 12 *at character position* 203. *The next two numbers give the same*

```
 1 #include "cover.h"
 2 #include "label.h"
 3 extern COVER *cov;
 4
 5 #include "prog_count.h"
 6 ...
 7 void prog_count::next_state(){
 8    if (cov->WrapperStatement(reset.read(),
      "reset.read()",tIFELSE,10,125,12,203,22,419,
      "prog_count.cc", "prog_count", "next_state",
      this)){
 9   pc = 0;
10 } else {
11 ...
```

Fig. 6.5 Instrumented code of the next state method

information for the else-block, but only the end position of the else-block is used; the else-block ends in line 22 *at character position* 419. *Then, the file name where the method is implemented (*prog_count.cc*), the class name (*prog_count*), the method name (*next_state*) and the* this *pointer are given.*

In a SWITCH-CASE statement at the beginning of each CASE-block we instrument a wrapper function that has as additional argument the value of the current case. After a SWITCH-CASE statement a wrapper function call is instrumented that enables the propagation of all possible CASE values. Note that the approach is able to handle also nested variants of all types of control statements.

In the next section the coverage analysis phase is explained.

6.3.4 Coverage Analysis

After the compilation of the instrumented SystemC code the coverage analysis is executed during simulation. Based on the instrumented wrapper functions the instance of the cover class collects all the coverage data. The main data structures in the cover class are based on *Standard Template Library* (STL) maps. As unique keys the arguments of the wrapper functions are transformed into a string representation. To each coverage point we associate two counters to track the frequency of the evaluation of the corresponding condition to true or false. For case statements obviously only one counter is needed. Finally, in the coverage report that is started by a call from *sc_main* after the end of the simulation, the coverage data is analyzed. For IF, IF/ELSE a warning is generated if the condition was always true/false and thus a block was never executed. In case of FOR loops or WHILE loops we inform the user if the condition was false all the time and therefore the loop body was skipped. For SWITCH-CASE statements each case is identified that was never activated. In total this allows to argue about the quality of the tests defined by the testbench. If blocks have been identified that have been never executed these blocks are dead code or the testbench has to be improved.

In the following example the results of the coverage analysis are shown for the program counter.

Example 3 *A testbench has been written for the program counter shown in Fig. 6.3. The testbench includes three tests. We applied our approach for this example. The automatically generated coverage report is shown in Fig. 6.6. As can be seen the scenario to load a value into the program counter by setting load enable to one was not executed. We added another test for this behaviour and thereby closed this gap.*

```
<< COVERAGE REPORT >>

IF-ELSE Statement: *IF-BLOCK NOT EXECUTED*
  File name: prog_count.cc
  Class: prog_count
  Instance: pc
  Func. Member: next_state
  Condition: le.read()
  IF start: line 14 pos 246
  IF end:   line 16 pos 322
  count total: 87
  count TRUE: 0 count FALSE: 87
```

Fig. 6.6 Coverage report for program counter

6.4 Case Studies

In this section we apply the approach to two examples. The first example is a hardware oriented model, a RISC CPU is considered. The second example is a system for colour region recognition in video data.

6.4.1 Hardware Model: RISC CPU

Before we apply our method to the RISC CPU the basic data of the CPU is briefly reviewed (see [9] for more details).

6.4.1.1 Specification

In Fig. 6.7 the components of the RISC CPU are shown. The CPU has been designed as Harvard architecture. The data width of the program memory and the data memory is 16 bit. The size of the program memory is 4 kB and the size of the data memory is 128 kB. The length of an instruction is 16 bit. We briefly describe the five different classes of instructions in the following: six load/store instructions, eight arithmetic instructions, eight logic instructions, five jump instructions and five other instructions. For the RISC CPU a compiler has been implemented which

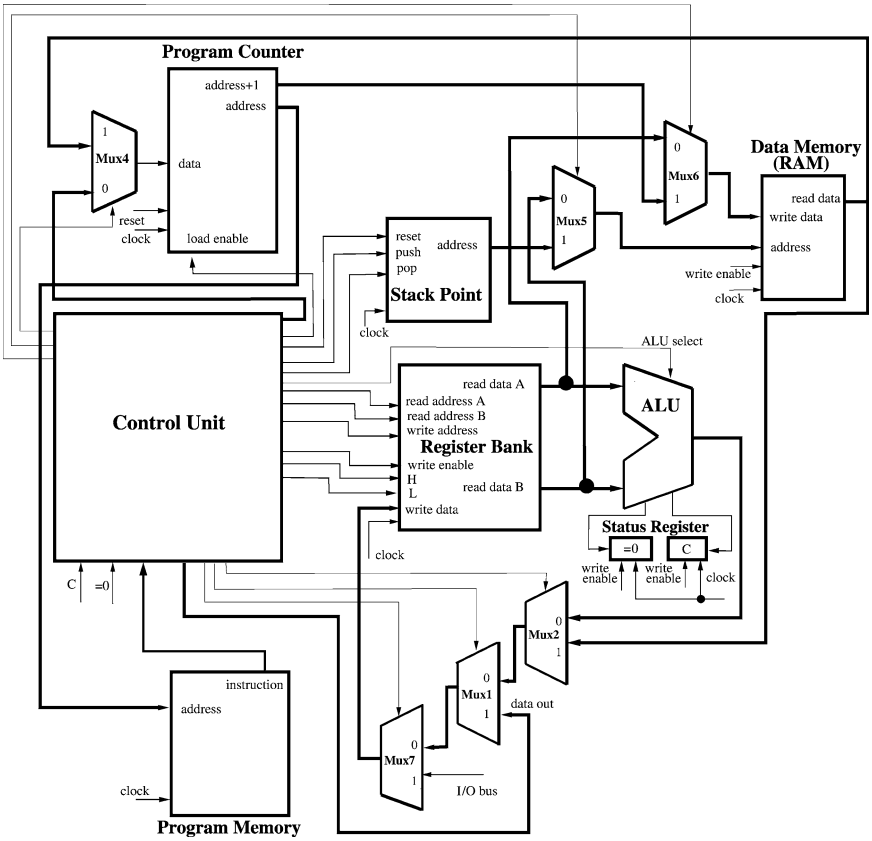


Fig. 6.7 RISC CPU including data and program memory

generates object code from an assembler program. This object code runs on the SystemC model, i.e. the model of the CPU executes an assembler program.

6.4.1.2 Testbench Quality

Based on successful simulation of each component the designer starts with the simulation at the system level. For this purpose usually a high-level testbench is created that enables a black-box test of the design. For the CPU such a testbench corresponds to the execution of a set of assembler programs including the analysis of the simulation results. In the following we describe how the high-level testbench was created and how this process was improved by our approach. The SystemC model of the RISC CPU was automatically instrumented with code to analyze coverage. The following non-trivial assembler program was formulated to test the CPU.

Example 4 *The assembler program shown in Fig. 6.8 converts a set of numbers into gray-code. The gray code encodes numbers such that in the binary encoding adjacent numbers have a hamming distance of 1. The number* n *of elements to be converted is given in the data memory at address 0. After clearing the register* R[6] *and* R[2], n *is loaded into register* R[3]. *Then, in the loop each single number is converted. The idea is to invert each bit if the next higher bit of the input value (read from the data memory into register* R[4]*) is set to one. Therefore the input is shifted by one and a bitwise XOR operation is performed. The result* R[6] *of the conversion is stored in the data memory to the same position as the input.*

```
 1        LDL R[6], 0
 2        LDH R[6], 0
 3        LDL R[2], 0
 4        LDH R[2], 0
 5        LDD R[3],R[2]
 6  loop1:
 7        ADD R[2],R[2],R[1]
 8        LDD R[4],R[2]
 9        ADD R[5],R[4],R[0]
10        SHR R[5],R[5]
11        XOR R[6],R[4],R[5]
12        STO R[2],R[6]
13        SUB R[3],R[3],R[1]
14        JNZ loop1
15        HLT
```

Fig. 6.8 Assembler program for gray code

After simulation of the gray code program on the CPU our approach reported unexecuted code fragments in the following modules: *stack_point*, *mux4*, *mux5*, *mux6*, *mux7* and *alu*. The handling for the cases of push and pop operations in the *stack_point* module was not tested, since the inputs from the control unit to this module have been zero during the complete simulation. To test this behaviour another program that uses push and pop instructions has to be added.

For the multiplexor modules we found that in the method *do_select* which describes the functionality of a multiplexor only the ELSE-block for the select condition was simulated. For the CPU this observation corresponds to the fact that the select inputs of the multiplexors have been zero all the time and thus only one data input was routed to the multiplexor output. As can be seen in Fig. 6.7 all multiplexors belong to the data path of the CPU. To also test the effects on the CPU in case of data coming through the other input, a different data path has to be activated. The multiplexor *mux5* is part of the stack pointer data path and thus was tested by using stack pointer operations (see above). For *mux4* and *mux6* the alternative data path is activated by adding a program that uses sub-routine calls. For *mux7* we set the select input to one by an additional program that uses I/O instructions.

In case of the ALU several CASE statements of the main SWITCH statement have not been executed since not all operations of the ALU are activated by the

considered assembler programs. Therefore we created another program to check to remaining arithmetic operations.

In total by adding additional assembler programs to the testbench the quality of the testbench was improved. Here our approach supported the verification engineer by directly pointing to untested functionality of the RISC CPU.

6.4.2 High-Level Model: Colour Region Recognition

In the second example we applied our approach to a high-level SystemC model of a video processor System-on-Chip. In contrast to the RISC CPU (which has been implemented as an RTL design), this model resides at the transaction-level of abstraction.

6.4.2.1 Specification

The configurable model *EmVid* consists of a set of SystemC cores that can be integrated to build a video processor. For video input and output, abstract TLM channels are used. The video processing IP cores use the *SystemC High-level Interface Protocol* (SHIP) [10] for data exchange over these channels. Communication with the main memory (DDR RAM) is established by ST's TAC protocol [12]. In the following, we consider a System-on-Chip for colour region recognition that is based on EmViD cores. The system processes video frames in real-time and draws rectangles around detected regions. A high-level schematic of the system is shown in Fig. 6.9. The system has been configured as a pipelined architecture and for the connection

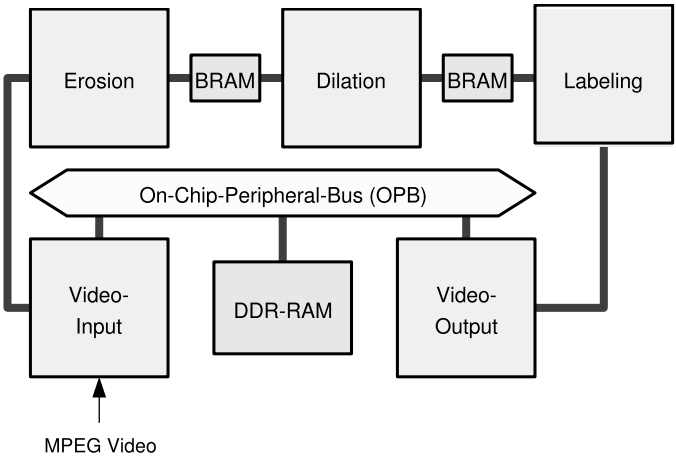


Fig. 6.9 Colour region recognition schematic

of the DDR RAM an IBM CoreConnect *On-Chip Peripheral Bus* (OPB) is used. The complete transaction-level interconnect (including an OPB simulation model) is set up using the GreenBus TLM fabric [11]. EmViD can be found on [4].

The video processing starts by reading in an MPEG video as video input. Then, dilation and erosion is performed. In the labelling stage the regions are recognized and the rectangles are added. Afterwards the core outputs the image to a display.

6.4.2.2 Testbench Quality

As a concrete application we decided to detect skins in the video data. We set the colour range for the recognition accordingly. The system segmentates the processed video data in the labelling phase. Therefore adjacent pixels are analyzed and the image is partitioned into a set of regions using the defined colour information.

In the overall video processor system the high-level testbench consists of the video data (coming from video files or a camera). We applied our approach to the system. We simulated the system with different video files and observed that depending on the video data different parts of the system have not been executed. For example, in the *morph_segm* module (labelling phase) the segmentation algorithm checks the minimum region size with an IF-condition. For video data that contains no skins or very small areas no regions are detected. Here, our approach presents directly the SystemC file with the exact source code position of the never executed block(s). Note that this improves the debugging during the development of such high-level models significantly.

Moreover, analyzing the results of nested control structures, our approach helps the verification engineer to test the design thoroughly. To give an example, the segmentation algorithm is realized as a state machine with 47 states, which are traversed in different (partial) execution orders depending on the video input data. With the output of the coverage analysis, untaken control paths can be discovered and the stimulus video material can be adjusted accordingly.

6.4.2.3 Further Design Analysis

During the analysis of the video processor model, we also experimented with different communication architecture configurations for the design. As one might expect, some architectures are better suited than others to meet efficiency requirements such as a given frame rate. In particular, when connecting all components to a shared bus with fixed-priority scheduling (here, the OPB), the overall video processing performance highly depends on the priority allocation.

We utilized the ability of our coverage analysis to count the number of executions for the various processes in the model in order to identify the location of communication bottlenecks in design configurations with poor frame rates. Table 6.1 presents some results of the experiments.

Table 6.1 Video processor execution traces

Config	#ex video	#ex detect.	FPS video	FPS detect.	Comment
Bus only model 1	500	500	24.98	24.98	Ascending priority
Bus only model 2	451	872	22.55	43.60	Higher detection priority
Mixed bus/pipeline model 1	500	500	24.98	24.98	Lower pipeline priority
Mixed bus/pipeline model 2	500	999	24.98	49.90	Higher pipeline priority

The column "#ex video" shows the total number of video frames successfully sent from the video input component (mpeg decoder) to the video output component (display controller). The column "#ex detect." shows the total number of video frames processed by the region detection. From these numbers the overall frame rates have been calculated (columns "FPS video" and "FPS detect.").

Row 1 and row 2 show the frame rates we got with a bus-only model. While in row 1, the bus access priorities were assigned in ascending order according to the sequence of video processing stages in the model, in row 2 we assigned a higher priority to the region detection components than to the video display data path. As expected, the frames per second processed for region detection goes up, but as an unintentional side effect due to higher bus workload, the number of video frames displayed per second drops down. Rows 3 and 4 show the results we achieved with a mixed bus/pipeline model as depicted in Fig. 6.9. Here, we could considerably increase the video display frame rate by just swapping the bus access priorities of two components. With this setup, ~25 frames per second full resolution live video display is achieved while the region detection runs at the high rate of ~50 frames per second.

6.5 Conclusions

In this paper, we have presented an approach to measure the quality of a testbench for a SystemC design. The approach is based on dedicated code coverage techniques using a SystemC front-end. Thus, a reliable feedback for untested parts of the design is presented to the user. This data includes exact source code information in combination with SystemC specific information, like process context and module hierarchy. In summary, our approach helps to create a high quality testbench. The experiments showed that our approach is suitable for both RTL and TLM designs. Moreover, the TLM example revealed that our analysis methodology also can support design space exploration.

References

1. B. Beizer. Software Testing Techniques. Wiley, New York, 1990.
2. L. Cai and D. Gajski. Transaction level modeling: an overview. In CODES+ ISSS'03, pp. 19–24, 2003.

3. R. Drechsler, G. Fey, C. Genz, and D. Große. SyCE: An integrated environment for system design in SystemC. In IEEE International Workshop on Rapid System Prototyping, pp. 258–260, 2005.
4. EmViD: Embedded Video Detection. http://www.greensocs.com/GreenBench/EmViD.
5. G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. ParSyC: An efficient SystemC parser. In Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), pp. 148–154, 2004.
6. R. S. French, M. S. Lam, J. R. Levitt, and K. Olukotun. A general method for compiling event-driven simulations. In Design Automation Conference, pp. 151–156, 1995.
7. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
8. C. Genz and R. Drechsler. System exploration of SystemC designs. In IEEE Annual Symposium on VLSI, pp. 335–340, 2006.
9. D. Große, U. Kühne, and R. Drechsler. Hw/sw coverification of embedded systems using bounded model checking. In Great Lakes Symp. VLSI, pp. 43–48, 2006.
10. W. Klingauf. Systematic transaction level modeling of embedded systems with SystemC. In Design, Automation and Test in Europe, pp. 566–567, 2005.
11. W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton. Greenbus: A generic interconnect fabric for transaction level modelling. In Design Automation Conference, pp. 905–910, 2006.
12. S. Microelectronics. TAC: Transaction Accurate Communication. http://www.greensocs.com/TACPackage, 2005.
13. Open SystemC Initiative, http://www.systemc.org. SystemC 2.1 Language Reference Manual, 2005.
14. T. Parr. Language Translation using PCCTS and C++ : A Reference Guide. Automata Publishing, San Jose, CA, 1997.
15. SystemC Verification Working Group, http://www.systemc.org. SystemC Verification Standard Specification Version 1.0e.
16. S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. In IEEE Design and Test of Computers, 18(4), pp. 36–45, 2001.
17. J. Yuan, C. Pixley, and A. Aziz. Constraint-based Verification. Springer, New York, 2006.

Chapter 7
SystemC-Based Simulation of the MICAS Architecture

Dragos Truscan[1], Kim Sandström[2], Johan Lilius[1], and Ivan Porres[1]

Abstract We present our approach in using SystemC for simulating a custom configurable architecture, MICAS. However, there are certain aspects of the architecture, like configuration specific information or programming interface, which cannot be directly represented using SystemC concepts. Thus, we define a C++-based specification language for MICAS that allows us to specify additional properties of the architecture at simulation level and furthermore, to combine these properties with the SystemC executable specification.

Keywords System-on-Chip (SoC), Service Oriented Architecture (SOA), SystemC simulation, executable specification

7.1 Introduction

Due to the increasing complexity of system specifications simulation has become a necessary tool for system designers. Simulation enables the evaluation of system specifications against requirements, at early stages of the development, before proceeding to hardware implementation. The approach eliminates costs and shortens the design life cycle of new products. According to Moretti [1], most of the integrated circuits developed today require at least one return to early phases of the development, due to errors.

In recent years, SystemC [3] has become one of the most popular languages for system-level modeling and simulation. SystemC is an extension of C++, in which the hardware components (i.e., modules) of the architecture are specified as C++ classes. A given architectural configuration is represented, at simulation level, as

[1]Åbo Akademi University, Joukahaisenkatu 35, FIN20500, Turku, Finland
Email: {Dragos.Truscan, Johan.Lilius, Ivan.Porres}@abo.fi

[2]Nokia Research Center, P.O. Box 407, FIN00045 Nokia Group, Finland;
Email: Kim.G.Sandstrom@nokia.com

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

module instances interconnected at port level. SystemC advocates reuse at code and component level, allowing the reuse of the developed components from one design to another. The approach facilitates the use of component libraries for rapid creation of new designs.

In our work, we have employed SystemC to provide executable specifications of a configurable architecture, namely MICAS [4]. MICAS is configurable not only because different hardware configurations can be built by adding new components, but also because the programming interface of a given configuration can be customized, at design time, to facilitate the application mapping on the architecture. The MICAS design flow uses a component library from which SystemC specifications of MICAS resources can be instantiated at simulation time. However, there are certain aspects of MICAS that cannot be directly modeled in SystemC. For instance, upon instantiation different modules added to a given MICAS configuration have to be adorned with configuration specific information, like address spaces, IRQ numbers, etc. Such information is not typically stored in a component library in order to increase the reusability of component specifications. Similarly, the programming interface that is designed for a specific configuration cannot be stored in the component library, but rather has to be generated for each configuration in part.

In order to integrate configuration specific information with the SystemC executable specification of a given configuration, we define a C++-based specification language for MICAS. This language enables us to express additional properties of the architecture in an executable form, easy to integrate within the MICAS simulation framework.

We proceed, in Section 7.2, with a general overview of the MICAS architecture and of its design process. Then, we introduce, in Section 7.3, a C++-based specification language for the MICAS architecture. In Section 7.4 we show how this language is used to describe MICAS configurations and how the resulting specification is integrated with the SystemC simulation framework of MICAS. We also discuss the customizations applied to the MICAS Simulation library such that the simulation model of a given configuration can be automatically generated. We conclude with final remarks.

7.2 The MICAS Architecture

Microcode Architecture For a System On a Chip (SoC) (MICAS) [4] is a novel concept developed at Nokia Research Center, Helsinki, Finland, which proposes both a SoC architecture for sequential data streaming processing systems (e.g., multimedia applications, personal video recorders, media streaming applications, etc.) and a method for controlling the hardware accelerators of such architectures. Several goals are pursued in MICAS:

- Separation of the data- and control-flows of the architecture, by using dedicated hardware units (***HW processes***) to assist data processing tasks and ***controllers*** to drive the activity of these units.

- Decentralization of the control communication from the "main processor" of the system, typically running a ***real-time operating system*** (RTOS), and the distribution of this communication to dedicated controllers, which only control "local" resources.
- The use of ***microcode*** (i.e., software running on controllers) to control the functionality of the HW processes and of the data streaming between them. The microcode provides a ***hardware abstraction layer*** (HAL) of the architecture, which allows one to create data streams between HW processes and to invoke the functionality of a given HW process using a standard interface.

7.2.1 Hardware Architecture

An overview of the MICAS architecture is given in Fig. 7.1. A MICAS configuration comprises several ***domains***. A ***domain*** represents a collection of hardware processing elements situated on the same physical silicon chip and controlled by the same ***controller***. Domains provide fast processing speed for dedicated tasks. They are interconnected by off-chip external networks using for instance, serial, Bluetooth or WLAN technology.

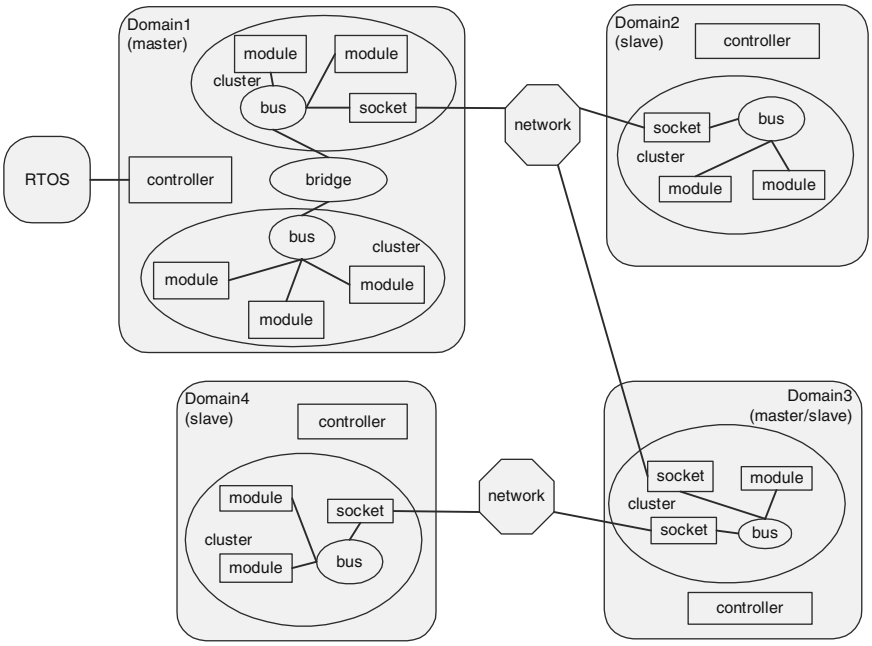


Fig. 7.1 Generic view of the MICAS architecture

The organization of domains is hierarchical, following a master-slave relationship. Typically, one domain of a given MICAS configuration is connected to a general purpose processor running an RTOS, like Symbian [2]. Such a domain is called ***master domain***. Domains connected to a master domain are regarded as ***slave domains***.

Each MICAS domain may contain several programmable hardware components, ***HW processes***, which implement dedicated tasks in hardware. HW processes are universally interconnected via ***buses*** and may be grouped into ***clusters***. There may be three types of HW processes inside a MICAS domain: ***bridges***, ***sockets*** and ***modules***. Buses belonging to different clusters may be connected to each other through ***bridges.*** ***Sockets*** mediate and transform the on-chip communication into off-chip communication, whereas ***modules*** implement dedicated processing tasks over streams of data.

7.2.2 Programming Interface

The MICAS programming interface defines a set of ***services*** that can be used to invoke complex functionality of a given MICAS configuration. The ***services*** are defined per domain and are implemented as a consistent combination of (data) ***streams*** between HW processes. Domain controllers serve as a control interface to any external entity (i.e., MICAS domain or RTOS). Any request for a service from the external environment is handled by the controller, which implements the streams of a given service by dispatching the corresponding microcommands to the appropriate HW processes. The concept of ***subservice*** of a service is used in MICAS to depict a service from a remote domain that is used by a service in a given domain.

7.2.3 The MICAS Design Process

An overview of the MICAS design process is given in Fig. 7.2. Starting from the ***Application Requirements*** one identifies the services (i.e., ***Service List***) that a given MICAS configuration has to provide. These services represent the programming interface of that particular configuration.

In the ***Service Specification*** phase, each service is specified in terms of data streams. In turn, each stream is implemented as a combination of microcommands used to program the corresponding HW processes. Based on the microcommands required to implement the streams, HW processes are added to the domain under design in the ***Hw Configuration*** phase.

Three artifacts are produced after the completion of the previous phases:

- ***Service Description*** – specification of the services of a given configuration, and their implementation in terms of streams and microcommands;
- ***Structural Configuration*** – the hardware components of the configuration and their interconnections;
- ***Functional Configuration*** – configuration specific properties (address spaces, IRQs, etc.) of the selected hardware components.

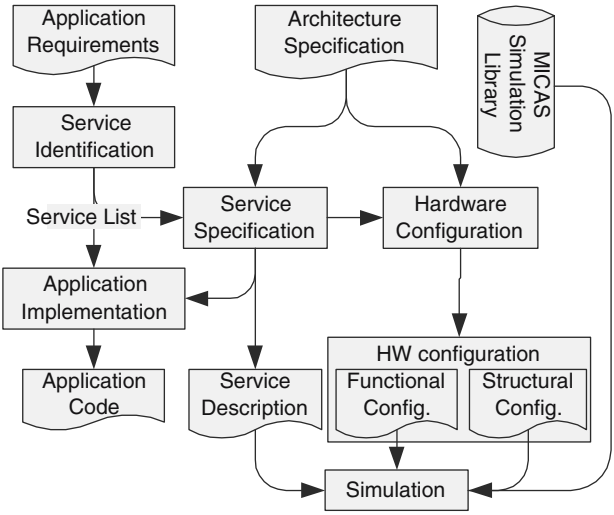


Fig. 7.2 MICAS design process

All these three artifacts serve as input to the ***Simulation*** phase along with the executable specifications of the selected hardware components. However, out of these artifacts, only the ***Structural Configuration*** of the architecture can be specified using SystemC, more specifically via the `main.cpp` file. A different approach has to be employed for the remaining two artifacts in order to specify them in an executable form, easy to integrate with the simulation environment. To address this issue we define a C++-based specification language for the MICAS architecture.

7.3 A C++-Based Specification Language for MICAS

The MICAS C++-based specification language models MICAS resources using type definitions. A `struct` data type is used to define these resources, while the fields of each `struct` type are used to represent their properties. The approach allows the use of MICAS resources as properties of other MICAS resources.

7.3.1 Specifying the Functional Configuration

At the highest level, a MICAS configuration is composed of several domains. Thus, a `Domain` data type is defined as follows:

```
struct Domain {
  std::string name;
  unsigned int domain_id;
  Bus* busList[10];
  int b_no;
  Process* processList[10];
  int m_no;
  struct Process* master_domain_socket;
  struct Process* master_domain_ctrl_socket;
  struct Process* slave_domain_socket;
  struct Process* slave_domain_ctrl_socket;
  unsigned int DPRAM_int;
  unsigned int int_ctrl_reg_addr; };
```

A `name` and a numeric `domain_id` are used to identify the domain during the simulation. The domain contains a number (`m_no`) of processes stored in the `processList` array, each of them being characterized in turn by specific information. In addition, a number (`b_no`) of buses are present in each domain, and they are similarly stored using a `busList` array. The external connections of the domain are modeled directly by the sockets present in that domain, specifying whether these sockets are connected to a master or to a slave domain. The socket description may be seen as a "routing table" for the inter-domain communication. In our current MICAS implementation, we have assumed that a domain may have at most two sockets communicating with its master or slave domains respectively, but a more general approach may be followed. We recall that domains have a hierarchical relationship to each other, being possible for each domain to have a master domain and, at the same time, being itself master to another (slave) domain. Two sets of pointers are modeling this information. The `master_domain_socket` and the `master_domain_ctrl_socket` are used to indicate to the controller the socket through which the data and respectively, the control communication with the master domain has to be directed. Similarly, a pair `slave_domain_socket – slave_domain_ctrl_socket` indicates to the controller the sockets through which the data and the control communication, respectively, with the slave domain has to be forwarded. A `null` pointer in one of these fields indicates that no master and respectively, no slave domain are connected to the domain in question.

The processor running the RTOS is connected to the MICAS master domain through a DPRAM module using an interrupt-based mechanism. We model it as a separate entry (`DPRAM_int`), not only because this is a high-priority interrupt, but also to allow specifying explicitly if an RTOS is connected to a domain. A non-valid value assigned to this field indicates that no RTOS is connected to the domain in question.

Finally, each controller has an interrupt controller, through which it communicates via a control bus. When an interrupt is raised by any of the HW processes in the domain, the corresponding interrupt number is passed to the controller via a control register, whose address is modeled by the `int_ctrl_reg_addr` field.

Each HW process included in the processList of a given domain is characterized by its own set of properties, as shown in the following type definition:

```
struct Process{
  std::string name;
  enum micas_process_type type;
  unsigned int ctrl_reg_addr;
  unsigned int master_reg_addr;
  unsigned int slave_reg_addr;
  unsigned int irq;
  unsigned int slave_data_buffer;
  unsigned int master_data_buffer; };
```

The name is used during the simulation for debugging purposes, whereas a type property specifies whether the HW process is a module, a bridge or a socket, based on the definition of the micas_process_type enumeration, which we omit here. Based on its placement relative to the other elements in a domain, a HW process is characterized by other types of information, like address spaces used for communication purposes. HW processes are controlled by the controller via a control bus, to which the HW process is connected by a control register. To be able to uniquely identify each HW process on the bus, each control register is assigned a unique identifier, the control_register_address. When the controller issues a command to a given HW process, in fact it writes the command identifier to the address of the control register.

The communication on the data bus between different HW processes is handled in a similar fashion. Each HW process is connected to the bus through a master or slave interface, and in addition, it has an unique identifier with respect to that bus. Thus, two such identifiers are defined master_reg_addr and slave_reg_addr, respectively. The communication between HW processes and the controller is done via an interrupt-based mechanism. The domain controller uses an interrupt controller for receiving interrupt signals from HW processes. Each module has a unique identifier (i.e., irq) corresponding to the interrupt signal to which it is assigned.

Similarly to HW processes, buses are stored in a busList, containing elements with the following structure:

```
struct Bus {
  std::string name;
  unsigned int maxCap;
  unsigned int avCap; };
```

Beside the name, the total capacity of the bus (maxCap) and the available capacity at a given moment (avCap) are included as properties.

One decision that we took was to group all the generated information in a single file, rather than create separate files for each domain description. Therefore, at

simulation time, the controllers of different domains will share this information from the same file. We do not consider this to be an impediment, since the generated information is read only, and thus, it does not pose the problem of arbitrating the access to it. As such, all the domain descriptions included in a given MICAS configuration are grouped in a `domainList` array.

```
Domain* DomainList[];
```

7.3.2 Specifying Service Description

A service represents an atomic piece of functionality provided by a given MICAS domain. Each domain provides its own service list (i.e., `service-Table`), in which a number (`s_no`) of services are stored.

```
struct Domain {
  Service* serviceList[10]
  unsigned int s_no; };
```

The `Service` type is characterized by a `name`, a list of `CompositeStreams` (i.e., consistent combinations of streams), a pointer to a `subservice` from a remote domain, and an `allocated` flag to be used at run-time for keeping track if the service is enabled at a given moment in time. The definition of the `Service` is shown below.

```
struct Service {
  std::string name;
  struct Subservice *subservice;
  CompositeStream* compositeStreams[10];
  unsigned int allocated; };
```

In turn, the `Subservice` is characterized by the identifier (`remote_domain_id`) of the remote domain from which it can be accessed and the service identifier (`remote_service_id`) in that remote domain. In addition, pointers to the local control sockets (`local_ctrl_socket`) are provided to indicate to the controller where to "route" the commands for using a given subservice, and from or to what socket (`local_socket`) it can access or send the data provided by the service. The `Subservice` definition is given in the following.

```
struct Subservice {
  unsigned int remote_domain_id;
  unsigned int remote_service_id;
  Process* local_socket;
  Process* local_ctrl_socket; };
```

A service is supported by one or many composite streams depicting the data-flow perspective of that service. In turn, each composite stream is implemented by a number (b_no) of basic streams, included in the basicStreams array.

```
struct CompositeStream {
  std::string name;
  struct BasicStream* basicStreams[10];
  int b_no; };
```

A ***basic stream*** (i.e., a data-flow between two HW processes) provides an intrinsic perspective on the associated control-flow needed to setup these HW processes. Thus, there is a need for thoroughly characterizing the properties of each basic stream. As such, a basic stream is specified by a name, a category and a capacity. In addition, each stream transfers data over a physical bus, between a source HW process (src_process) and a destination HW process (dst_process). The latter are represented as pointers to the corresponding elements. From a control perspective, a basic stream is equivalent to one or many HW process commands that trigger the data transfers over the bus. These commands are gathered in the microcommands array and executed every time the basic stream is triggered.

```
struct BasicStream {
  std::string name;
  enum category cat;
  unsigned int Capacity;
  struct Bus *bus;//pointer to the bus //transporting the
   stream
  struct Process *src_process;
  struct Process *dst_process;
  Microcommand* microcommands[10];
  unsigned int m_no; };
```

The Microcommand is characterized by a name and an implementation (impl). In turn, the implementation consists of a command identifier (command), which is a numeric value to be written by the controller to the control register (master_address) of the master HW process. The microcommand will also specify the address (slave_address) of the slave HW process to which the master process is to communicate over the bus.

```
struct Microcommand {
  std::string name;
  struct impl {
    unsigned int command;
    Module* slave_address;
    Module* master_address;
    } impl; };
```

We mention that these type definitions and their data structures are independent of the specific configurations that can be created in MICAS. They are intended only to provide a common framework to specify the MICAS architecture in C++. These type definitions may be regarded as a textual language for specifying MICAS configurations at simulation level.

7.4 Generating the Simulation Model

A graphical specification language [5] is used to create MICAS configurations and to design the services provided by a given configuration. For instance, Fig. 7.3 presents the hardware configuration of an ***Audio*** domain, while Fig. 7.4 depicts the stream definition of an ***encodeAudio*** service provided by this domain.

From the graphical specifications of MICAS configurations, the simulation code is generated automatically using the C++-based specification language of MICAS. See [5] for more details.

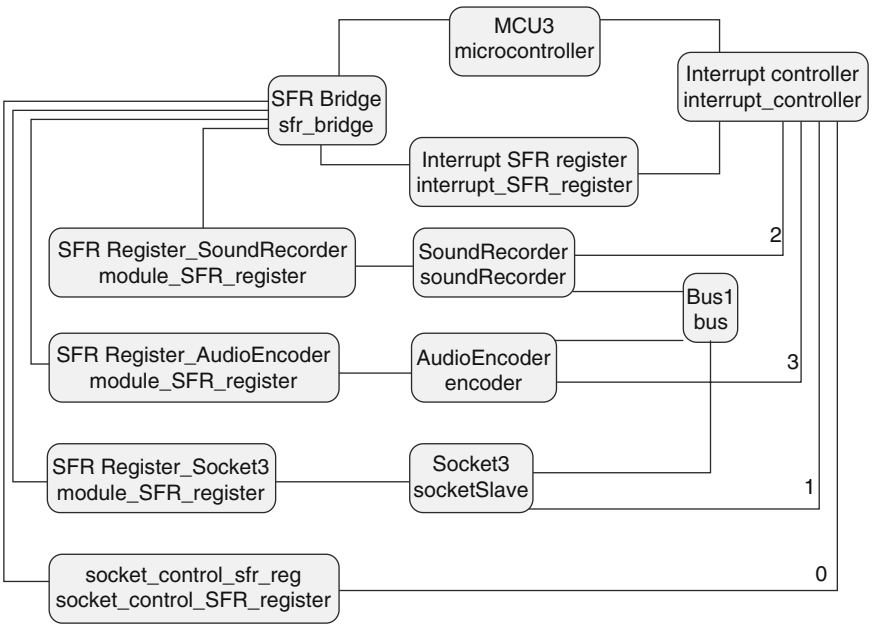


Fig. 7.3 MICAS domain model example

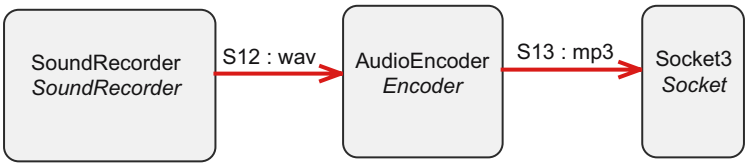


Fig. 7.4 Streams of the ***encodeAudio*** service

7.4.1 Specifying Functional Configuration and Service Description

The generated code has two parts, declaration and initialization, similar to a C++ program. In the ***declaration*** part, the MICAS components of a given configuration are instantiated using the C++ data types defined previously. The following code represents the declaration of the domain shown in Fig. 7.3.

```
Domain Audio;
Module Audio_Socket3;
Module Audio_socket_control_sfr_reg;
Module Audio_AudioEncoder;
BasicStream Audio_S11;
Microcommand mc_S11_Audio_record_sound_wav;
Microcommand mc_S11_Audio_transmit_data_to_domain;
BasicStream Audio_S13;
Microcommand mc_S13_Audio_encode_wav_2_mp3;
Microcommand mc_S13_Audio_transmit_data_to_domain;
CompositeStream Audio_encodedAudio;
Module Audio_SoundRecorder;
Bus Audio_Bus1;
BasicStream Audio_S12;
Microcommand mc_S12_Audio_record_sound_wav;
CompositeStream Audio_unencodedAudio;
Service Audio_encodeAudio;
Service Audio_plainAudio;
```

In the ***initialization*** part, the properties of each instantiated component are initialized with data extracted from the MICAS models. Due to the large size of the generated code, we only show the properties of the ***Socket3*** process, of the service ***encodeAudio***, and of the ***S13*** basic stream.

```
Audio_Socket3.name = "Socket3";
Audio_Socket3.master_ctrl_reg_addr = 10;
Audio_Socket3.slave_reg_addr = 1;
Audio_Socket3.slave_data_buffer = 32;
```

```
Audio_Socket3.master_data_buffer = 32;
Audio_Socket3.irq = 2;
Audio_Socket3.type = SOCKET;
……
Audio_encodedAudio.name = "encodedAudio";
Audio_encodedAudio.basicStreams[0] = &Audio_S12;
Audio_encodedAudio.basicStreams[1] = &Audio_S13;
Audio_encodedAudio.b_no = 2;
……
Audio_S13.name = "S13";
Audio_S13.bus = &Audio_Bus1;
Audio_S13.src_module = &Audio_AudioEncoder;
Audio_S13.dst_module = &Audio_Socket3;
Audio_S13.Capacity = 100;
Audio_S13.cat = MP3;
Audio_S13.microcommands[0] =
  &mc_S13_Audio_encode_wav_2_mp3;
Audio_S13.microcommands[1] =
  &mc_S13_Audio_transmit_data_to_domain;
Audio_S13.m_no = 2;
```

7.4.2 Specifying the Structural Configuration

As previously mentioned, the structural perspective of a given MICAS configuration is modeled at simulation-level using the SystemC language. The ***MICAS Simulation library*** is used for providing ready-built SystemC specifications of MICAS resources. Following this approach, only the top-level configuration file of the SystemC model has to be generated in order to obtain the hardware simulation model of a given MICAS configuration.

In order to integrate, at run-time, the structural and functional information of the configuration, the SystemC module specifications stored in the library have been customized to also take into account, at initialization time, the functional configuration and the service description of a given MICAS configuration.

7.4.2.1 Providing Reusable Module Specifications

SystemC promotes reuse of module specifications allowing the instantiation of the same module for implementing (simulating) several hardware components in the same configuration. However, each module instance has to be made "aware" of its configuration settings in terms of assigned address spaces, IRQ numbers, parameters, etc. This information has to be passed to instances at instantiation time, or following the SystemC terminology, at elaboration time.

Using the information provided by the functional configuration and by the service description, respectively, we configure, at elaboration time, SystemC module instances with specific information. The approach enables the reuse of the same SystemC module specification in several architectural configurations. For instance, if two MICAS modules ***VideoEncoder*** and ***AudioEncoder*** are used in a configuration, each of them as part of a different MICAS domain, a generic SystemC ***encoder*** module can be used to simulate both components. Thus, the encoder has to be instantiated once for each of the two MICAS modules, and the functional information specific to each MICAS module has to be passed to its corresponding instance.

A couple of customizations have been applied to the components of the ***MICAS Simulation library***. Firstly, we have defined a mechanism that enables us to pass the configuration properties to module instances at elaboration time. Secondly, the SystemC processes implementing the module behavior have been customized to take the functional configuration and the service description into account during simulation.

Passing information to module instances. As mentioned previously, each SystemC module specification is basically a C++ class. As such, beside the SystemC specific constructs, one can define additional properties of that class, like attributes and methods.

In the previous section, we have declared several C++ data types (e.g., `Domain`, `Process`, `Bus`, etc.), each of them specifying the functional properties of a specific type of MICAS hardware resource. We integrate each such data type with the corresponding SystemC module by declaring a `mymodule` attribute of the module class. For instance, classes specifying MICAS processes contain a `Process mymodule` attribute, whereas classes specifying bus modules have a `Bus mymodule` attribute. An example is given in the following:

```
SC_MODULE (socketMaster){
  SC_CTOR (socketMaster){
  ......
  }
  public:
  Process mymodule; };
```

During the elaboration phase, when modules are instantiated in the `main.cpp` file, the information is passed to a given module instance in the following way:

```
socketMaster Socket1("Master_Socket1");
Socket1.mymodule = *this_domain->moduleList[Socket1_id];
```

We have followed a similar approach in case of the modules implementing MICAS controllers, with the difference that the entire functional configuration of a domain is passed to the controller (`MCU1.mymodule = *this_domain;`) as an attribute. We have employed this approach since the domain controller manages the resources of the entire MICAS domain and therefore, it requires access to the properties of all domain resources.

Customizing module behavior. Having configuration information passed to SystemC modules also requires the customization of the SystemC processes that are modeling the behavior of each module, such that they take into account the fields of the `mymodule` data structure.

For instance, a `socketMaster` module uses two methods (`transfer_over_socket()` and `transfer_to_slave()`) to specify its internal processes, as shown below:

```
void transfer_to_slave();
void transfer_over_socket ();
SC_CTOR (socketMaster){
  SC_METHOD (transfer_over_socket);
  sensitive_pos (Clk);
  SC_CTHREAD (transfer_to_slave, Clk.pos ()); }
```

The `transfer_over_socket()` method manages the data transfer from the socket over the external socket network, while the `transfer_to_slave()` method handles data transfers on the local bus.

Each process has a corresponding implementation, situated in the `.cpp` file of the module specification. For the sake of example, an excerpt of the code implementing the `transfer_to_slave()` process is shown below. The presented code reads the control register address of a MICAS component (`mymodule.master_ctrl_reg_addr`) and writes it to the `M_MData` port of the `socketMaster` instance.

```
void socketMaster::transfer_to_slave() {….
  M_MData.write(mymodule.master_ctrl_reg_addr);
  …. }
```

Therefore, using the functional properties of the module as variables, instead of having them hardcoded in the process specification, enables us to reuse the same process in a generic manner.

7.4.3 The SystemC Top-Level File

Based on the previous customizations of the ***MICAS Simulation library***, the process of generating the SystemC top-level file for a given configuration is fully automated. The SystemC code corresponding to the MICAS domain presented in Fig. 7.3 is shown below:

```
//main.cpp
#include "microcontroller.h"
#include "interrupt_SFR_register.h"
#include "AHB_bus.h"
#include "bus.h"
#include "soundRecorder.h"
```

```
  #include "socketSlave.h"
  #include "encoder.h"
  #include "inverter.h"
  #include "interrupt_controller.h"
  #include "sfr_bridge.h"
  #include "module_SFR_register.h"
  #include "socket_control_SFR_register.h"
  #include "config1.h"

  namespace MicasSystem {
    namespace Audio {
      microcontroller MCU3("Audio_MCU3");
      socket_control_SFR_register
   socket_control_sfr_reg("Audio_socket_control_sfr_reg");
      socketSlave Socket3("Audio_Socket3");
      interrupt_controller Interrupt_controller(
          "Audio_Interrupt_controller");
      encoder AudioEncoder("Audio_AudioEncoder");
      soundRecorder SoundRecorder("Audio_SoundRecorder");
      sfr_bridge SFR_Bridge("Audio_SFR_Bridge");
      module_SFR_register SFR_Register_Socket3(
          "Audio_SFR_Register_Socket3");
      module_SFR_register SFR_Register_SoundRecorder(
          "Audio_SFR_Register_SoundRecorder");
      bus Bus1("Audio_Bus1");
      module_SFR_register SFR_Register_AudioEncoder(
          "Audio_SFR_Register_AudioEncoder");
      interrupt_SFR_register Interrupt_SFR_register(
          "Audio_Interrupt_SFR_register");
  } // Audio namespace end
} // MicasSystem namespace end

  int sc_main(int argc, char* argv[]) {
    sc_clock TestClk ("TestClock", 10, SC_NS, 0.5);
    initialize();
    { using namespace MicasSystem::Audio;
      MCU3.Clk(TestClk);
      socket_control_sfr_reg.Clk(TestClk);
      Socket3.Clk(TestClk);
      Interrupt_controller.Clk(TestClk);
      AudioEncoder.Clk(TestClk);
      SoundRecorder.Clk(TestClk);
      SFR_Bridge.Clk(TestClk);
      SFR_Register_Socket3.Clk(TestClk);
      SFR_Register_SoundRecorder.Clk(TestClk);
```

```
    Bus1.Clk(TestClk);
    SFR_Register_AudioEncoder.Clk(TestClk);
    Interrupt_SFR_register.Clk(TestClk);
    Domain* this_domain = domain_list[Audio_id];
    MCU3.mydomain = *this_domain;
    socket_control_sfr_reg.socket_ctrl_register_addr =
      this_domain->
      moduleList[socket_control_sfr_reg_id]->master_ctrl_reg_addr;
    Socket3.mymodule = *this_domain->moduleList
      [Socket3_id];
    SFR_Register_Socket3.module_sfr_register_addr =
      this_domain->moduleList[Socket3_id]->master_ctrl_reg_addr;
    AudioEncoder.mymodule =
      *this_domain->moduleList[AudioEncoder_id];
    SFR_Register_AudioEncoder.module_sfr_register_addr =
      this_domain->moduleList[AudioEncoder_id]
      ->master_ctrl_reg_addr;
    SoundRecorder.mymodule =
      *this_domain->moduleList[SoundRecorder_id];
    SFR_Register_SoundRecorder.module_sfr_register_addr =
      this_domain->moduleList[SoundRecorder_id]
      ->master_ctrl_reg_addr;
    Interrupt_SFR_register.int_ctrl_SFR_
    register_addr =
      this_domain->int_ctrl_reg_addr;
    //connect ports
    …… }
{//connect domains
……
}//end ELABORATION PHASE
int n = 600000000;
if( argc > 1 ) std::stringstream(argv[1], std::
stringstream::in) >> n;
sc_start (n); //START SIMULATION
return 0; } // sc_main end
```

7.5 Conclusions

We have presented a C++-based specification language for the MICAS architecture that is used to integrate, at simulation time, the configuration related properties with the SystemC-based specification of the MICAS hardware. The approach favors the use of simulation libraries and enhances support for automation.

We have shown how the defined language can be used to model various characteristics of the MICAS configurations and how it can be integrated with the SystemC specification of a given configuration. In addition, we have discussed the customizations applied to the components of the ***MICAS Simulation library***, such that the simulation model of a given MICAS configuration can be automatically generated.

We mention that although the process of upgrading the library required some additional effort, the benefit of the approach is twofold: (a) it enables for different instances of the same module specification not only to be instantiated in several architectural settings, but also to reuse the same module for implementing different MICAS components; (b) it facilitates the automated generation of the simulation model.

References

1. G. Moretti. The search for the perfect language. EDN, Feb. 2004.
   Online at ***http://www.edn.com/article/CA376625.html.*** *Last checked 10/12/2007.*
2. Symbian OS. At ***http://www.symbian.com***.
3. Open SystemC Initiative. SystemC Specification. At ***http://www.systemc.org***.
4. K. Sandström. Microcode Architecture For A System On A Chip (SoC). Nokia Corporation Patent NC37341, 872.0167.P1(US) (Filing Date: 07.10.2003), Oct. 2002.
5. D. Truscan. Model Driven Development of Programmable Architectures. Ph.D. thesis, Åbo Akademi University, March 2007.

Chapter 8
Heterogeneous Specification with HetSC and SystemC-AMS: Widening the Support of MoCs in SystemC

F. Herrera[1], E. Villar[1], C. Grimm[2], M. Damm[2], and J. Haase[2]

Abstract This chapter provides a first general approach to the cooperation of SystemC-AMS and HetSC (*He*terogeneous *S*ystem*C*) heterogeneous specification methodologies. Their joint usage enables the development of SystemC specifications supporting a wide range of Models of Computation (MoCs). This is becoming more and more necessary for building complete specifications of embedded systems, which are increasingly heterogeneous (they include the software control part, digital hardware accelerators, the analog front-end, etc.). This chapter identifies the syntactical and semantical issues involved in the specifications which include facilities from both, SystemC-AMS and HetSC methodologies. This work, which is an extension of the paper presented in FDL'07 [7], considers the availability and suitability of the MoC interface facilities provided by both methodologies, especially those of SystemC-AMS, which will be proposed for future standardization. Some practical aspects, such as the current set of MoCs covered by the methodologies and the compatibility on the installation of their associated libraries are also covered by this chapter. A complete illustrative example is used to show HetSC and SystemC-AMS cooperation.

Keywords Heterogeneity, Models of Computation, System-Level Design, SystemC.

8.1 Introduction

Support for heterogeneity has become an important feature for specification methodologies that aim to cope with the current complexity of embedded systems. In this context, heterogeneity is the ability of the specification methodology to enable the building of models with parts specified under different MoCs [1].

[1]University of Cantabria, Spain

[2]Technical University of Vienna, Austria

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Each design domain adopts a specification methodology which usually corresponds to a specific model of computation (MoC). One of the most characteristic points associated with the MoC is the handling of time. For instance, analog models (Continuous Time (CT) models [2] handle strict-time information, that is, specification events have an associated time tag representing physical time and fixing strict order relationships among them. In contrast, concurrent software models often neglect such detail in the time domain and consider only partial order relationships among the events associated to the code.

The development of a system-level heterogeneous specification methodology is, to a great extent, a unification work. Some works developed interfaces between different languages, i.e., to connect hardware description languages (HDLs) with high-level programming languages [3]. This enabled certain decoupling between different design teams, which can fix the connection points and work separately. However, a system-level specification methodology has to enable the generation, understanding, edition and simulation of the specification of the whole system. This is a unification work which involves finding common points for the specification and simulation methodologies handled by the different design communities.

An effort to develop a common specification and simulation framework was done. Relevant examples are Metropolis [4] and Ptolemy II [5]. These frameworks enable specification under different MoCs, approaching the separation of computation and communication in different ways. Both provide support for graphical specification, while Java adopts the role of underlying implementation language.

Up to now, the focus of this unifying work has tended to be the language itself. The lack of a unified system specification language has been identified as one of the main obstacles bedevilling SoC designers [6]. A common specification language is a major aid in generating a specification methodology which aims to combine and achieve coherence among traditionally different and separated design approaches. SystemC has started to play a role as unifying system-level language for embedded system design. Becoming an IEEE standard is a symptom of its acceptance and of a stated syntax and unambiguous semantic for the language constructs which are used by SystemC-based methodologies.

In this context, several proposals have appeared for building heterogeneous specifications in SystemC. This chapter shows how two of them, HetSC and SystemC-AMS can be jointly used to enable models based on the SystemC language and comprising a wide spectrum of MoCs. This work is based on [7], which is improved and extended here. After this introduction, Section 8.2 reviews previous work on heterogeneous specification in SystemC. The main focus is on the HetSC and SystemC-AMS specification methodologies. Section 8.3 deals with general issues about the interoperability of these methodologies. First, some practical issues concerning the installation and the scope of the libraries are discussed. Then, how the SystemC-AMS and HetSC constructs are mixed in the same specification is explained. Reviewing and understanding how the existing facilities provided by the two methodologies for MoC connection can be used and combined serves later to propose improved connections. Section 8.4, provides an illustrative example of the previous concepts. Last section ends with the main conclusions and advances further steps of the research on this topic.

8.2 Heterogeneous Specification in SystemC

Although the SystemC core language supports hardware specification (RTL and Behavioural) and a generic Discrete Event (DE) modelling, there is a set of MoCs which are not sufficiently supported by the core language. Such support must include new specification facilities, MoC rule checkers, report tools, etc. Several works have attempted to cover such deficiencies. In the following subsections these works are overviewed. Most of these methodologies are supported by an associated library; however, they extend SystemC in different ways.

8.2.1 SystemC-AMS

SystemC-AMS [8] is a specification methodology developed by the OSCI SystemC-AMS working group which provides support for analog and mixed-signal specification. This involves supporting the Synchronous Dataflow (SDF), discrete-time (DT) and continuous time (CT) MoCs. Among the CT MoCs, it is possible to specify linear behavioral models by means of transfer functions (TF). Currently, two views are supported for TFs: the numerator-denominator (ND) view and the zero-pole (ZP) view. In addition, the specification of linear electrical networks (LEN), which enable a circuit level description, is also supported.

SystemC-AMS is extensible by other models of computation through a synchronization layer. Solvers for the MoCs supported are layered over the synchronization layer. The design of the synchronization layer of SystemC-AMS and the MoCs provided are oriented to a system-level modelling where simulation speed is a more important factor than a very fine simulation accuracy.

The synchronization layer supports directed communication and only a simple synchronization; on user specified events or in fixed time steps. In this way, the simulation of linear networks with SystemC-AMS can be orders of magnitudes faster than the more general numerical integration for non-linear networks [9]. From the specification point of view, SystemC-AMS offers a new set of facilities, such as new kinds of modules (*SCA_SDF_MODULE*), ports (*sca_sdf_in*, *sca_sdf_out*, etc.), channels (*sca_sdf_signal*), and other MoC specific facilities, such as the *sca_elec_node*, *sca_elec_port*, etc. Linear behavioural models are embedded in SDF modules, while LENs are enclosed in SystemC modules. SystemC-AMS provides converter ports and facilities to enable different MoCs to communicate (i.e. DE with SDF, SDF with LEN, etc).

8.2.2 HetSC

HetSC [10] is a methodology for enabling heterogeneous specifications of complex embedded systems in SystemC. MoCs supported include untimed MoCs, such as Kahn Process Networks (KPN), its bounded fifo version (BKPN), Communicating

Sequential Processes (CSP) and Synchronous Dataflow (SDF). Synchronous MoCs, such as Synchronous Reactive (SR) and Clocked Synchronous (CS) and the timed MoCs already supported in SystemC are also included. HetSC aims at a complete system-level HW/SW codesign flow. Indeed, the methodology has been checked in terms of system-level profiling and software generation [11].

The HetSC methodology defines a set of specification rules and coding guidelines for each specific MoC, which makes the designer task more systematic. The support of some specific MoC requires new specification facilities providing the specific semantic content and abstraction level required by the corresponding MoCs. The HetSC library, associated with the HetSC methodology, provides this set of facilities to cover the deficiencies of the SystemC core language for heterogeneous specification. In addition, some facilities of the HetSC library help to detect and locate MoC rule violations and assist the debugging of concurrent specifications. One of the main contributions of HetSC is its efficient support of abstract MoCs (untimed and synchronous). This is because they are directly supported over the underlying discrete event (DE) strict-time simulation kernel of SystemC. New abstract MoCs do not require additional solvers since the new MoC semantic is embedded in the implementation of the new specification facilities (usually channels) related to the abstract MoC. When the new MoC can be abstracted from the DE strict-time MoC, then, it is possible to find a mapping of internal events of the new specification facility, i.e., a channel, over the strict-time axis of the DE base MoC. This makes it feasible to write the implementation of such a channel by using SystemC primitives, such as SystemC events, which control when things happen within the channel and, therefore, in the processes related by the channel.

8.2.3 SystemC-H

SystemC-H [12] is a methodology that proposes a general extension of the SystemC kernel for the support of different MoCs. This methodology proposes the extension of the SystemC kernel by including a solver for each MoC. The current scope of the SystemC-H library covers the SDF and CSP untimed MoCs. For instance, SystemC-H provides a solver for static scheduling of SDF graphs which enables schedulability analysis and provides a 75% speed-up respect to DE [12]. However, this extension is not always worthwhile. Indeed, the speed-up for some abstract MoCs can be negligible [13]. In addition, the effects of Amdahl law can make simulation speed-ups vanish. For instance, in [12] the speed-up of a mixed DE-SDF example decreases to 13%. This suggests that providing a specific solver for each MoC can be not always worthy. In cases like these, it can be more efficient to let several MoCs to share the same simulation kernel. This is the approach of HetSC, where similar speed-ups to those of [12] were reported for the dynamic approach to SDF for large-grain SDF specifications [14]. Another problem of this approach is that the way the extension is proposed requires modifying the standard kernel of the library. In contrast, SystemC-AMS and HetSC methodological libraries rely on the SystemC standard library, which remains untouched.

8.2.4 SysteMoC

SysteMoC [15] focuses on providing a methodology with the ability to extract and analyze the MoC employed in the SystemC design. This is understood to be a prerequisite for the rest of the design activities. In order to achieve this, the SysteMoC library provides support for a basic MoC called *Funstate*. Specifications written under this MoC express their communication behaviour under the finite state machine (FSM) MoC. This enables the automatic extraction and analysis of the MoC employed, only by analyzing communication FSMs together with the topology of the specification.

8.3 HetSC/SystemC-AMS Interoperability

8.3.1 Installation and Scope

Figure 8.1 describes the installation requirements of the SystemC user. Apart from the SystemC core library, the SystemC-AMS and HetSC libraries have to be installed on top of the SystemC core library. There is flexibility with respect to the development platform (i.e., Linux, Unix and Windows-Cygwin are supported).

There is no compatibility problem in the installation of HetSC and SystemC-AMS libraries. In this work, the HetSC library is extended with some specific facilities for enabling an easier connection of HetSC and SystemC-AMS parts. These HetSC facilities use some SystemC-AMS facilities through forward declarations. This prevents obliging an installation order between HetSC and SystemC-AMS libraries, making the installation procedure easier. Once such an installation has been done, the development system is ready for compiling and executing SystemC specifications written under a wide range of MoCs. The user only has to include the SystemC-AMS and HetSC libraries in the source code of the heterogeneous specification.

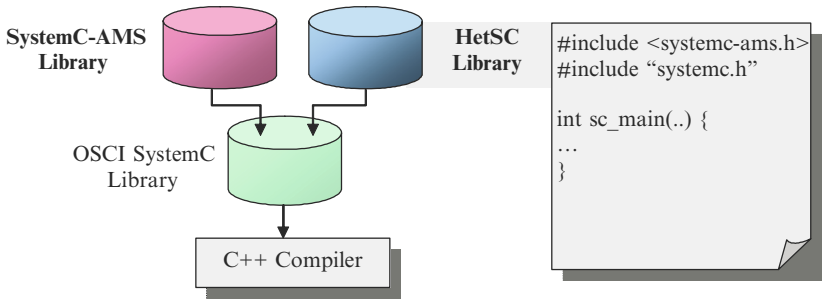


Fig. 8.1 SystemC-AMS and HetSC libraries are installed over the SystemC library

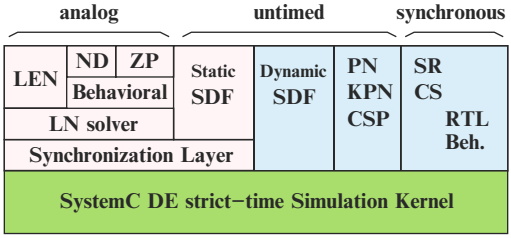


Fig. 8.2 MoCs spectrum provided by the cooperation of SystemC-AMS and HetSC

Figure 8.2 shows the supported MoCs. The cooperation of SystemC-AMS and HetSC provides a complementary MoC support. While SystemC-AMS provides support for analog MoCs and static synchronous data flow, untimed and synchronous MoCs are supported by HetSC and SystemC core facilities.

This is also an efficient configuration for the support of a wide spectrum of MoCs. The reason is that specific solvers are provided only for a set of MoCs where the simulation speed up is significant. This set corresponds to analog MoCs, where the simulation speed ups can be of orders of magnitude. Bearing in mind the limited speed-ups reported in [10, 12, 13], untimed and synchronous MoCs can be satisfactorily supported directly over the SystemC kernel. The exception would be fine grain SDF specifications, where the speed up of a static SDF compared to a dynamic SDF could be significant. Specifications without CT parts but with synchronous hardware (RTL or behavioural) could also justify a cycle-accurate simulator. However, the study of these exceptions is not in the scope of this work.

8.3.2 Syntactical and Semantical Issues

There are some basic issues to consider in a general discussion of the connection between HetSC and SystemC-AMS. In terms of the resulting structure, two parts can be distinguished in the specification. One corresponds to the AMS part, while the other corresponds to the HetSC part.

From the syntactical point of view, the SystemC-AMS part will be identified by *SCA_SDF_MODULEs* and/or SCA hierarchical modules. This part presents a hierarchical heterogeneity where the underlying MoC is the static synchronous dataflow (SDF) MoC. The HetSC part is characterized, in general, by an amorphous heterogeneity. This means that the HetSC specification permits mixing MoC facilities in a flat hierarchy. Nevertheless, the HetSC specifier will often make use of module hierarchy for separating parts of the system under different MoCs. Thus, in many cases, module partition will correspond with MoC boundaries.

From the semantical point of view, there is a basic consideration. While HetSC directly relies on the DE strict-time simulation kernel, SystemC-AMS relies on a synchronization layer, which provides support for the solvers. In SystemC-AMS, CT descriptions are always embedded in dataflow clusters [8]. That is, the most

important solver is the SDF one which, from the point of view of time semantics, is the basis for the analog MoCs. The time axis in SystemC-AMS is actually sliced by each SDF cluster in strict-time delays called cluster periods ($T_{cluster}$), which depends on the sample period (T) and rates of the cluster SDF graph. Thus, with respect to the premises of [14], the SDF approach of SystemC-AMS is not an untimed SDF. Internally, modules of the cluster can be viewed as a strict-time timed approach to the SDF MoC (denoted as T-SDF here), which enables a static execution of the AMS processes at each cluster period. More important for the purpose of this work, from an external point of view, the cluster can be conceived as a timed-clocked synchronous (CS) block which triggers at each cluster period. Thus, the cluster period must be taken into account to synchronize the DE part with the SystemC-AMS part.

Since every MoC supported by HetSC is abstracted over the DE strict-time simulation kernel and every SystemC-AMS MoC is clustered in the T-SDF MoC, the problem is reduced to providing a SystemC/SystemC-AMS connection, which is basically a DE/T-SDF connection. In SystemC-AMS, this connection is done by means of SystemC signals (*sc_signal* channels) and a set of SystemC-AMS connection facilities (*sca_scsdf_in*, *sca_scsdf_out*, *sca_sc2v*, *sca_sc2r*, etc.). Each of these connection facilities are based on the sampling and/or update of a SystemC signal at each cluster time. Therefore, an immediate conclusion is that these elements can be directly employed to combine HetSC and SystemC-AMS.

Such direct usage of the *sc_signal* and the DE/SystemC-AMS connection facilities is immediate in some HetSC/SystemC-AMS connections. On the left hand side of Fig. 8.3, a HetSC part under a synchronous reactive MoC (SR MoC) is represented. Both in Figs. 8.3 and 8.5 the graphical representation used in HetSC methodology is employed. There is a simple reactive chain composed of a generator process (GP) which triggers a reactive process (RP). This RP is also a border process (BP), since it writes to a SystemC signal channel (*sc_signal*), which is connected to a SystemC-AMS part. Its connection with the SystemC-AMS part by means of a signal channel is syntactically and semantically compatible with the SR MoC rules. These rules and, specifically, perfect synchrony, are respected since the write access to the SystemC signal is non-blocking. This enables the reactive chain to be computed consuming one or more simulation delta cycles, but without requiring a SystemC time advance. This is the way in which perfect synchrony is implemented in HetSC.

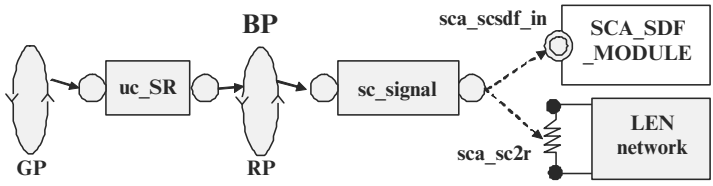


Fig. 8.3 Connection of HetSC and SystemC-AMS parts by means of a border process

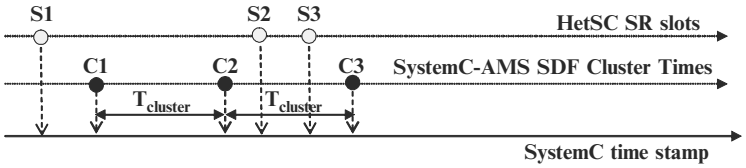


Fig. 8.4 Strict time information in the HetSC/SystemC-AMS connection

From the SystemC-AMS part, the connection is coherent too. In the connection to a *SCA_SDF_MODULE*, the value of the *sc_signal* is read at each cluster time. This value can be read as many times as necessary, as the consumption rate of the *sca_scsdf_in* port determines. Another possibility is the connection to a linear electrical network (LEN MoC of SystemC-AMS) by means of a converter facility, for instance, a *sca_sc2r* in Fig. 8.3. This facility enables the update of its associated resistance value whenever the *sc_signal* channel is written. Then, this updated resistance value is employed by the LEN solver in the following cluster times to solve the differential equation corresponding to the electrical network the converter facility belongs to.

The time stamp information of the HetSC SR slot is irrelevant to the effect of the HetSC SR MoC itself (the only necessary condition is that each slot has to happen at different time stamps). However, it is important to the effect of the (HetSC) SR MoC/(SystemC-AMS) LEN MoC connection, since it tells when the differential equation system is updated, before or after a given cluster time. For instance, in Fig. 8.4, the time stamps of the SystemC-AMS cluster computations are represented as black dots. Their time stamps are equally spaced. The time stamps of the SR time slots are represented as white dots. As mentioned, there is still consistency in the SR part if slot time stamps are not equally separated. However, actual time stamps of SR slots affect the relationship of the SR part with the timed SystemC-AMS part. For example, the first two cluster computations, C1 and C2, use the signal value updated in the slot S1, while the cluster computation C3 uses the signal value updated in the slot S3. If the S2 slots moves to a time stamp before C2, then, although this is no relevant to the effects of the SR part, it affects the SystemC-AMS part, since C2 takes the value updated in S2.

The set of MoCs abstracted from the DE MoC and supported by HetSC is rich enough to consider specific connections which cannot be directly handled by only a SystemC signal plus SystemC-AMS connection facility. For instance, the connection of a KPN MoC with a LEN MoC involves fifo channel semantics on one side and electrical nodes on the other side. It would be convenient to count on some connection facility which enables such direct connection, without the explicit intermediation of the SystemC signal (*sc_signal*).

In order to get such a direct connection, both in syntactical and semantical terms, the SystemC signal-SystemC-AMS connection facility can be conveniently complemented and wrapped by one of the basic concepts employed in HetSC for the connection of MoCs, the border process. Externally, the connection facility can take

the shape of a HetSC border channel (BC). Figure 8.5 shows a border channel (*uc_inf_fifo_sca_sdf*), which enables a direct connection between a KPN MoC and a T-SDF MoC. It is built as a hierarchical channel which on the one hand exports the write interface of an *uc_inf_fifo* channel, while on the other hand offers a T-SDF port (*sca_sdf_out*) port. Internally, it uses a border process which consumes fifo tokens, whose values are used to update the internal SystemC signal. The signal is connected to a converter port (*sca_scsdf_in*) of an inner SystemC-AMS module. In addition, BCs provide a scalable way to construct these direct connections since it does not require the SystemC-AMS kernel to be changed.

BCs provides a semantical solution for the untimed/timed connection which arises when untimed MoCs of HetSC are connected to SystemC-AMS MoCs. In the (HetSC) SR-(SystemC-AMS) T-SDF example the solution was based on sampling (read) and updating (write) signals and considering the relationship of the actual time stamps of HetSC SR slots and the cluster period of the SystemC-AMS part. The connection of SystemC-AMS with untimed HetSC MoCs is more complex because of the differences in terms or communication semantics. HetSC untimed MoCs handle a different behaviour in terms of the destructive and non-destructive semantics of the write and read accesses.

For instance, a KPN part, expects that writing to a (fifo) channel provokes the accumulation of tokens within the channel in case they cannot be immediately transferred, thus they are never lost. This is a non-destructive write semantic. It also expects to consume instead of peeking or sampling the data present in the channel, that is, a destructive read. This communication semantic, typical from untimed MoCs has to be coherently connected with the T-SDF part, which writes and reads at a fixed pace (determined by the cluster period and port rates) with a non-accumulative (destructive) write and sampling (non-destructive) read semantic. Then, some kind of adaptation has to be introduced to convert consumption in sampling (and vice versa) and production in writing (and vice versa). Actually, this type of adaptation is not comprised by any of the SystemC-AMS connection facilities. Such adaptation can be explicitly written, i.e. as a HetSC border process. The BC enables the packaging of such adaptation in a specification primitive. For instance, in the *uc_inf_fifo_sca_sdf* is a BC. In this BC, when to consume fifo tokens is defined by means of a sampling period, which, in general, can be different from the cluster period. The BC can also raise an error if the internal fifo gets empty when a new sampling is given.

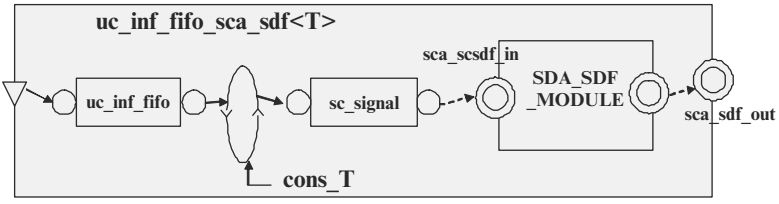


Fig. 8.5 Structure of a uc_inf_fifo_sca_sdf channel

8.4 Example

In order to demonstrate the previous general concepts, an example has been developed. This example is available in [16]. It consists of a soundboard, which is shown in Fig. 8.6. The system has an audio input and an audio output. The audio input undergoes three stage filtering. The first filter is a noise filter to remove any signal component over 22 kHz. The second one is a 10-channel equalizer. The last one is an integrator, which, at the same time, controls the general volume and filters the DC component of the audio output. The system has other inputs, as well as the audio input. A dial enables selection of the equalizer channel, while another dial tunes the gain of the selected channel in dBs (in a [-10 dB, 10 dB] range). A state display shows the current state of the equalization, while an edition display shows the currently selected channel and the currently edited equalization profile. This profile is not applied till the *set* button is pressed. Then, the state display changes to reflect this equalization profile. If the *cancel* button is pressed instead, then the edition display and the edition equalization profile return to the initial state (0 dB for every channels). Another dial controls the general gain of the system (also in a [-10 dB, 10 dB] range). It does not depend on the *set* button. That is, its change immediately updates the system gain.

Figure 8.7 depicts how this has been solved using HetSC and SystemC-AMS together. The system is enclosed in a SystemC module (*soundboard*). This top module contains another SystemC module (*panel_control*), which contains the

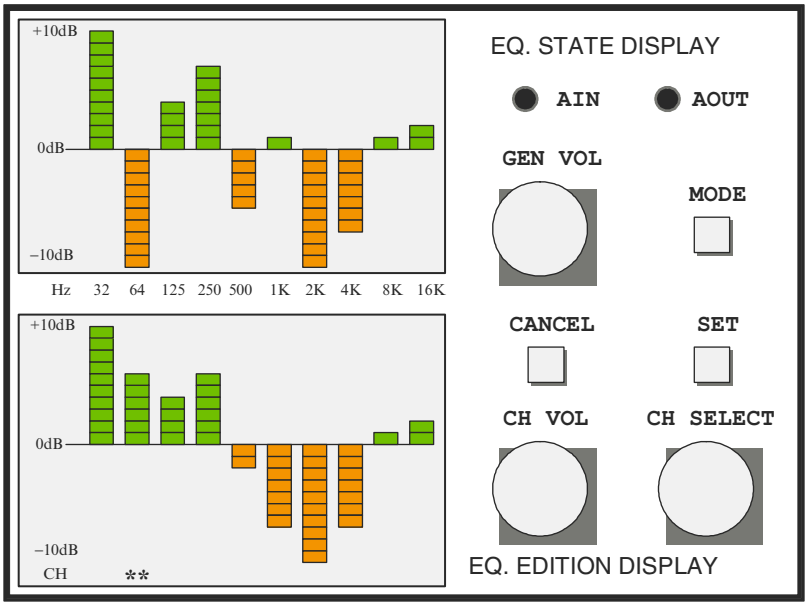


Fig. 8.6 Soundboard system

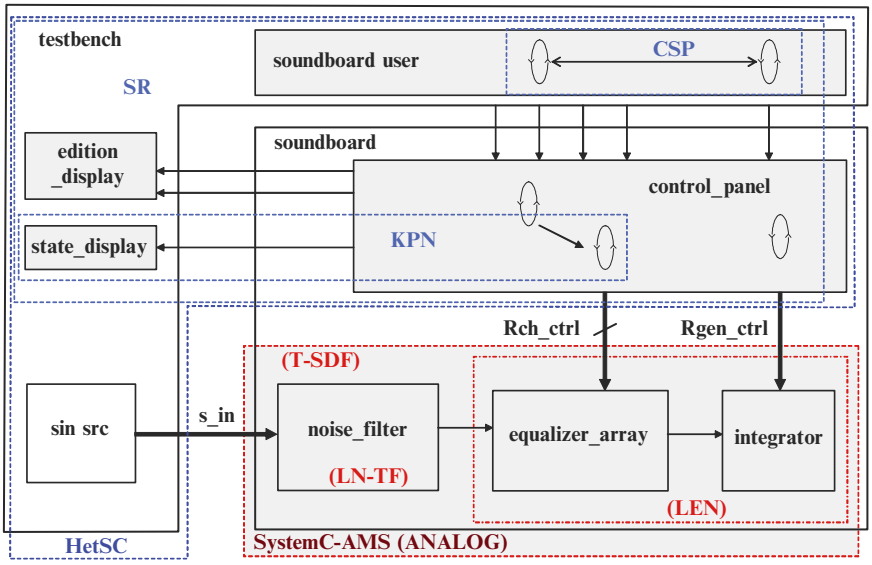


Fig. 8.7 SystemC-AMS-HetSC specification

HetSC part of the system and uses the HetSC library specification facilities. The *soundboard* module also contains three modules which use SystemC-AMS facilities. In this case, the testbench model (*testbench* module) is composed of four modules which only use HetSC facilities. In another version, part of the testbench (the audio input) was specified using SystemC-AMS specification facilities). In this sense, several combinations were possible leading to the same result.

In Fig. 8.7, the correspondence with the MoCs employed is depicted with dashed lines. In the testbench module, two processes (*left_hand* and *right_hand*) model the handling of dials and buttons of the soundboard. The two processes are synchronized through a rendezvous channel, to ensure the left hand edits the equalizer profile before the right hand pushes the *set* button and raises the general volume. Because of this, this part is a CSP network. In addition, each of the processes is an autonomous process generating a SR reactive chain. Dial turn and button press are modelled as writes to *uc_SR* channels. The reactive chain which controls the general volume is pure in that it is composed only of generator and reactive SR processes. The reactive process converts the dial events (turning left or right), which mean plus 1 dB or minus 1 dB, considering the bounds of the [-10 dB, 10 dB] range, in a control SystemC signal which affects the value of a resistor composing the integrator module. A similar thing happens with the channel equalization control. However, here there is not a pure reactive chain, since the two reactive processes are border processes, as they also write to infinite fifo HetSC channels (*uc_inf_fifo*), proper of the KPN MoC. For instance, one is used to pass the new equalization profile to the state display when the *set* button is pressed.

In the analog part, the noise filter is modelled through a SystemC-AMS SDF module (*noise_filter*). This module has an input port, to read the *s_in* external signal which provides the audio samples. It is designed as a second order Butterworth low pass filter with a cut frequency of 22 kHz, which is modelled under the LN-TF MoC of SystemC-AMS, using the ND view. The other two blocks are modelled at a circuit level, under the LEN MoC. The *equalizer_array* module encloses an array of ten equalizer cells. Each of them is an active band pass filter centred at the channel frequency. This filter is described as a circuit with three resistors, two capacitors and a model of operational amplifier (OA) which considers the gain, the input and output resistance. Each equalizer cell is instantiated taking the capacitor values as the parameter for centring each filter at the channel frequency (32 Hz for channel 0, 64 Hz for channel 1 and so on till 16 kHz for channel 9). The output of each equalizer cell is connected to a resistor instance of type *sca_sc2r*, controlled by one of the signals of the *Rch_ctrl* signal array. These resistors are connected to the same electrical output node, where the contribution of each equalizer cell is added. This node is used as input to the *integrator* module. This module is also described as a circuit which also instantiates the previously mentioned OA model, a capacitor, and a resistor controlled by the *Rgen_ctrl* signal, to control the gain of the integrator and, thus, of the whole system.

In both, the HetSC and SystemC-AMS parts, elements are employed to connect MoCs. For instance, BPs connect KPN and SR MoCs in the HetSC part, and a *sca_sdf2v* instance connects the noise filter to the equalizer array. In Fig. 8.7, the connections between the HetSC and the SystemC-AMS part have been highlighted with thicker arrows. Specifically, the audio input samples are transferred to the *soundboard* module through an instance of the *uc_inf_fifo_sca_sdf* channel introduced in the previous section. This border channel enables a direct connection between the untimed part, which generates the samples, and the SystemC-AMS input converter port of the noise filter. The connection of the SR reactive chains to the LEN part of the model is placed between the lower part of the *control_panel* module and the *equalizer_array* and *integrator* analog modules. For instance, the reactive process triggered by the turn events of the general volume dial is indeed a border process which writes the *Rgen_ctrl* SystemC signal. A similar thing happens with the non pure reactive chain, which drives an array of ten signals (each one for its corresponding equalizer channel). Each of these signals controls the value of a SystemC-AMS *sca_sc2r* primitive.

A time domain simulation and two frequency analyses have been performed. The time domain simulation is dumped to data and waveform files. The first frequency analysis is done in the middle of the time domain simulation. At this time, the soundboard response corresponds to that of the initial state (0 dB gain for every channel and for the general volume). The second frequency analysis is done at the end of the time domain simulation, once a manual configuration has been performed and the *set* button pressed. Additional results of the simulation are two data files, with the frequency response of the equalizer (thus, the equalization profile) at different points of the simulation time. The result has been post-processed with *Octave specification execution*, just to reflect the change on the equalization profile (Fig. 8.8). Other outputs of the system are two log files which reflect the activity of the displays.

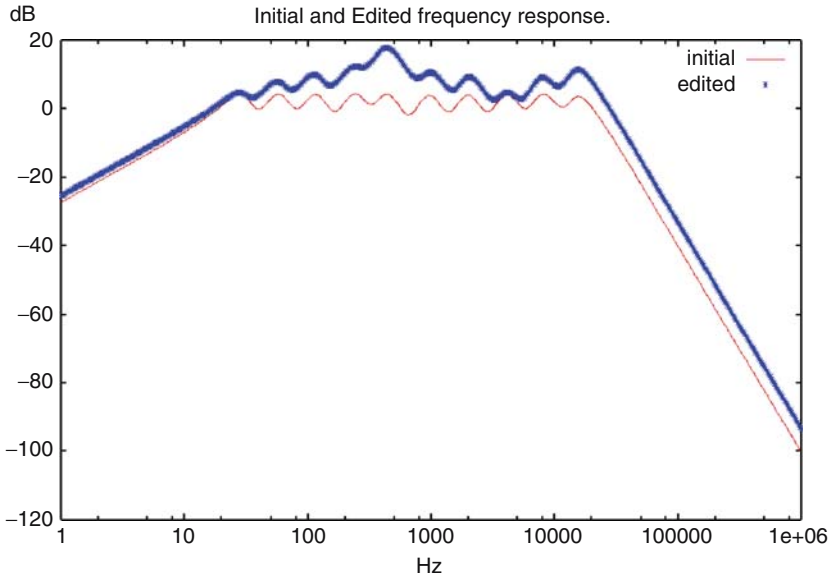


Fig. 8.8 Initial and edited frequency spectrum

Table 8.1 Host configurations where the example has been compiled and executed

OS	GCC	SystemC	SystemC-AMS	HetSC
Linux 2.6.3/32 bits	3.3.2	2.1v1	0.15RC1/RC2	1.2
Linux 2.6.3/32 bits	4.0.0	2.2.0	0.15RC4	1.2
Linux 2.6.3.2/64 bits	4.1.2	2.2.0	0.15RC4	1.2

This specification took around 2,500 SystemC code lines including test bench modules and around 30 man-hours (ignoring learning time). The simulation time was less than 53 s in an Intel PIV 2.8 GHz, Linux 2.6.3 development platform. This illustrates how fast the system-level specification and analysis of such heterogeneous system can be done using HetSC and SystemC-AMS. The example has been checked for three configurations of development platforms, reflected in Table 8.1.

8.5 Conclusions and Future Work

This work addresses how the HetSC and SystemC-AMS specification methodologies can be used together. With their cooperation, a wide range of MoCs, from untimed to analog ones, are efficiently covered. This is a key feature in enabling the early system-level specification of embedded systems. The installation and compatibility of the HetSC and SystemC-AMS libraries has been checked. Furthermore,

the syntactical and semantical issues related to the connection have been discussed. SystemC-AMS is based on a timed SDF MoC, where AMS clusters can be conceptually seen as timed-clocked synchronous blocks from the DE part. SystemC-AMS provides facilities for this AMS/DE connection which are based on the sampling and update of the SystemC signal. Since HetSC MoCs are abstracted from the underlying DE strict-time MoC, the connection of any HetSC MoC with any System-AMS MoC can be reduced to a SystemC DE/SystemC-AMS connection. Thus, SystemC-AMS facilities for the DE/AMS connection can be used. Moreover, the HetSC border channel can be conveniently used to provide direct connections among specific untimed and synchronous (HetSC) MoCs and analog (SystemC-AMS) MoCs, hiding the intermediation of DE signals in the connection of MoCs that do not employ such specification primitives and encapsulating the detection of error situations which consider the cluster period, the time conditions of HetSC part, etc. The immediate evolution of this work can be found [17], where converter channels are introduced. These channels incorporate concepts of polymorphic signals [18], releasing from any manual engagement in the system refinement. As well as adaptations on the time and communication domain, converter channels also introduce adaptations at the data type domain. Finally, this work implicitly states the need for a formal environment in order to obtain a common understanding of the interoperation of this kind of methodologies.

Acknowledgments Work supported by the FP6-2005-IST-5 European project.

References

1. E.A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(12), December 1998.
2. A. Jantsch. Modelling Embedded Systems and SoCs. Morgan Kaufmann, San Francisco, CA, June 2003. Morgan Kaufmann Publishers An imprint of Elsevier Science 340 Pine Street, Sixth Floor San Francisco, California 94104-3205 www.mkp.com
3 R. Gupta. HDL/C Interface Exploration. Tech. Report, ICS Dpt., University of California, California, 2002.
4. A. Davare et al. A Next-Generation Design Framework for Platform-Based Design. In DVCon 2007, San Jose, CA, USA, February 2007.
5. C. Brooks et al. Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java. Tech. Report, University of California, Berkeley, CA, July 2005.
6. L. Geppert. Electronic Design Automation. IEEE Spectrum, 37(1), January 2000.
7. F. Herrera, E. Villar, C. Grimm, M. Damm and J. Haase. A General Approach to the Interoperability of HetSC and SystemC-AMS. In Proceedings of the Forum of Design Languages 2007. FDL'07, Barcelona, Spain, 2007.
8. A. Vachoux, C. Grimm, and K. Einwich. Towards Analog and Mixed-Signal SoC Design with SystemC-AMS. In IEEE DELTA'04, Perth, Australia, 2004.
9. A. Herrholz et al. ANDRES – Analysis and Design of Runtime Reconfigurable Heterogeneous Systems. In Proceedings of DATE'07, Nice, France, April 2007.

10. F. Herrera and E. Villar. A Framework for Embedded System Specification Under Different Models of Computation in SystemC. In Proceedings of DAC'06, San Francisco, CA, July 2006.
11. H. Posadas, F. Herrera, V. Fernandez, P. Sanchez, and E. Villar. Single Source Design Environment for Embedded Systems Based on SystemC. Transactions on Design Automation of Electronic Embedded Systems, 9(4):293–312, December 2004.
12. H.D. Patel and S.K. Shukla. SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling. Springer, July 2004.
13. H.D. Patel, D. Mathaikutty, and S.K. Shukla. Implementing Multi-Moc Extensions for SystemC: Adding CSP and FSM Kernels for Heterogeneous Modelling. Tech. Report, FERMAT, Virginia Tech., June 2004.
14. E.A. Lee and D.G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Trans. on Computers, C-36(1):24–35, 1987.
15. J. Falk, C. Haubelt, and J. Teich. Efficient Representation and Simulation of Model Based Designs in SystemC. In Proceedings of FDL'06, Darmstad, Germany, September 2006.
16. http://www.teisa.unican.es/HetSC/downloads.html
17. J. Haase, M. Damm, C. Grimm, F. Herrera, E. Villar. Using Converter Channels within a Top-Down Design Flow in SystemC. The 15th Austrian Workhop on Microelectronics, Graz, Austria, October, 2007.
18. R. Schroll, C. Grimm, and Waldschmidt K. Verfeinerung von Mixed-Signal Systemen Mit Polymorphen Signalen. In Analog'05. VDE-Verlag, Berlin, Germany, 2005.

Chapter 9
An Extension to VHDL-AMS for AMS Systems with Partial Differential Equations

Leran Wang, Chenxu Zhao, and Tom J. Kazmierski

Abstract This paper proposes VHDL-AMS syntax extensions that enable descriptions of AMS systems with partial differential equations. We named the extended language VHDL-AMSP. An important specific need for such extensions arises from the well known MEMS modelling difficulties where complex digital and analogue electronics interfaces with distributed mechanical systems. The new syntax allows descriptions of new VHDL-AMS objects, such as partial quantities, spatial coordinates and boundary conditions. Pending the development of a new standard, a suitable pre-processor has been developed to convert VHDL-AMSP into the existing VHDL-AMS 1076.1 standard automatically. The pre-processor allows development of models with partial differential equations using currently available simulators. As an example, a VHDL-AMSP description for the sensing element of a MEMS accelerometer is presented, converted to VHDL-AMS 1076.1 and simulated in SystemVision.

Keywords Hardware description language, VHDL-AMS, mixed-technology modelling, partial differential equations, MEMS

9.1 Introduction

VHDL-AMS is a hardware description language designed to support modelling at various abstraction levels in mixed, electrical and non-electrical physical domains as well as mixed, digital and analogue components [1]. These features make it straightforward for VHDL-AMS to be used as the modelling language in MEMS design. Since MEMS systems are combinations of subsystems from both the electrical and mechanical domains, the field of MEMS design is interdisciplinary in nature. Several VHDL-AMS based MEMS models have already been reported in literature, such as a yaw rate sensor [2] and a vibration sensor array [3].

School of Electronics and Computer Science, University of Southampton, UK
Email: {lw04r, cz05r, tjk}@ecs.soton.ac.uk

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Although VHDL-AMS is a very powerful and flexible mixed physical domain modelling tool, it faces a challenge in MEMS related applications. The current VHDL-AMS (IEEE 1076.1) can only describe the continuous parts of a system by using differential and algebraic equations (DAEs). Support for partial differential equations (PDEs) was intentionally left out in the development of VHDL-AMS standard due to the complexity [4]. This limits accurate modelling of system blocks that include distributed physical effects [5]. However, simulation of single-domain characteristics of micro devices is usually performed by solving PDEs with geometry-related boundary conditions [6]. Such blocks are currently modelled in VHDL-AMS mainly by reduced-order models (ROMs) [2, 3]. Because of the size of a MEMS device, distributed effects are not negligible and may even play vital roles, for which reduced-order MEMS models are often not accurate enough. Thus an implementation of PDEs in VHDL-AMS is in demand. Suggestions have been made to extend other AMS-HDLs, such as Modelica [7] and Verilog-AMS [8], to add PDE support.

Some attempts have already been made to implement PDEs within the existing limits of VHDL-AMS. A transmission line example [5] and a system with electro-thermal coupling [9] are modelled using VHDL-AMS 1076.1. The way is to discretize the equations with respect to spatial variables and leave the time derivatives to be handled by VHDL-AMS [5]. The problem with this approach is that the discretization is done manually. When some modifications are made to the system, a series of equations have to be rewritten which makes the modelling very inefficient. New language extensions for PDE support have also been raised [5, 9] but currently no simulator can handle the new operators.

The work presented in this paper implements PDEs in VHDL-AMS in such a way that pending the development of a corresponding standard, PDEs can be written directly but no new simulators are needed. Necessary language constructs have been adopted from previous work [5, 9] and some improvements have been made. A translation pre-processor has been developed to convert the extended language (VHDL-AMSP) into VHDL-AMS 1076.1 automatically so that models with PDEs can be simulated using currently available simulators. Using this new method VHDL-AMS models that describe systems with distributed physical effects can now be built and simulated more efficiently.

The proposed methodology is expected to have particular advantages in mixed mechanical-electrical systems with tight control feedback loops, of which the MEMS block is an integral part. For example, the work presented in a recent paper [10] intends to develop new and innovative control and interface systems, technologies and circuits for MEMS physical sensors. The primary methodology is based on the incorporation of micro-mechanical sensing elements (e.g. for accelerometers and gyroscopes) in high-order $\Sigma\Delta$ modulator (SDM) loops. The loop filter consists of mechanical and electronic integrators; the former is constituted by the micromachined sensing element which is, to a first order approximation, a second order transfer function. The tools currently used for simulating such a complex and highly coupled system are primarily system level tools, such as Matlab/Simulink. The lumped parameter model of the sensing element captures only the first mechanical mode.

However, when designing higher-order electro-mechanical SDM loops, higher order mechanical modes may well be of considerable significance for the stability and performance of the control loop. Consequently, having a distributed mechanical model using partial differential equations would be a significant breakthrough for the design of such devices. To demonstrate the efficiency of our approach, the sensing element of such a MEMS accelerometer in SDM loop has been modelled in VHDL-AMSP, translated to VHDL-AMS 1076.1 and simulated. Simulation results show that high-order behaviour of the cantilever beam has been captured, which is not possible in conventional methodologies.

9.2 VHDL-AMS Extensions for PDE Support

The extensions outlined below support equations that may contain high-order partial derivatives describing systems in a multidimensional space.

9.2.1 Partial Quantity

With the keyword **partial**, a partial quantity is defined as a physical variable which has a continuous value not only over a period of time but also over a hypercube in a multidimensional space. It is declared as:

```
partial quantity q : real;
```

The corresponding BNF (Backus-Naur Form) notation is:

```
partial_quantity_declaration ::=
   partial quantity identifier_list : subtype_indication;
```

Partial quantities may act as interfaces between entities as well as appear in architecture bodies.

9.2.2 Spatial Coordinate

With the keyword **coordinate**, spatial coordinate is declared over which a partial quantity is distributed. Multiple coordinate declarations will form a hypercube in space. The declaration can define a range in space and the discretization step size.

The range is obligatory as it defines the hypercube, but the step size is optional. It is up to the designer to decide whether to use default step size or to give a fixed value. The following is an example of a spatial coordinate declaration:

```
coordinate x : real range 0.0 to 10.0 step 0.1;
```

Two new grammar productions have been added to the language BNF:

coordinate_declaration ::= **coordinate** identifier_list : subtype_indication;
step_size ::= **step** simple_expression

The existing *range* construct is extended by the new *step* construction as:

range ::= range_attribute_name [step_size]
|simple_expression direction simple_expression [step_size]

9.2.3 Partial Derivatives

As suggested in the papers by Nikitin et al. [5, 9], a new language attribute name is introduced as *′dot(x)*. If *q* is a partial quantity and *x* is a coordinate, *q′dot(x)* represents the derivative of *q* with respect to *x*. Unlike the example given in the paper [9] where a high-order derivative is represented by multiple ticks, e.g. *q″ dot(x)* for the second order, VHDL-AMSP uses the same notation as VHDL-AMS, namely *q′dot(x)′dot(x)*. This kind of representation is in the spirit of the existing VHDL-AMS 1076.1 standard and *q′dot(x)* as a whole is still a partial quantity. A partial quantity can also have a derivative with respect to time, using the attribute *′dot*, so items like *q′dot(x)′dot* are valid. Multidimensional derivatives are supported, such as *q′dot(x)′dot(y)* where *x* and *y* are two coordinates. Since there is no predefined attribute name and attribute designator in VHDL-AMS, this extension does not affect the language BNF.

9.2.4 Simultaneous Statement with Partial Derivatives

A simple example is:

```
q'dot(x) == A * q'dot;
```

which represents $\frac{\partial q}{\partial x} = A\frac{\partial q}{\partial t}$.

Partial differential equations can also appear in simultaneous *if* or *case* statements. High-order derivatives or derivatives of more than one spatial coordinate can also be described in a simultaneous statement.

9.2.5 Boundary Conditions

A boundary condition is defined as a special simultaneous statement as shown below. The expression after the keyword **at** specifies the spatial boundary where the conditions should apply. Conditions are written in the form of simultaneous statements. An example is:

```
boundary x at 0.0 is
begin
q == 0.0;
q'dot(x) == 0.0;
end BOUNDARY;
```

The corresponding production in the language BNF is:

simultaneous_boundary_statement ::=
  [boundary_label:]
  boundary coordinate_name **at** simple_expression **is begin**
  simultaneous_statement {simultaneous_statement}
  end boundary [boundary_label];

9.3 Translation to VHDL-AMS 1076.1

We have developed a translation pre-processor to automatically convert VHDL-AMSP models into VHDL-AMS 1076.1. The pre-processor can be used as a tentative measure to implement PDEs in VHDL-AMS pending the development of an appropriate standard. The translation pre-processor uses a modified version of a VHDL-AMS parser [11] where the modifications incorporate the new syntax into the parser and allow syntax analysis by recursive scanning of the parse tree. During the scanning, new language constructs can be identified and replaced by necessary VHDL-AMS 1076.1 constructs. How the new constructs are converted into existing constructs is demonstrated below, using the examples from Section 9.2.

In the declaration part of the model, a partial quantity is converted into a quantity vector by the same name. The vector size is determined by the coordinate's range and step, i.e. *range/step*. The coordinate won't appear in the output file but a differential coefficient (*dx* in the example) will be declared as a constant, which has the value of the step size. The declaration part will therefore contain:

```
quantity q : real_vector (0 to 100);
constant dx : real:=0.1
```

In the architecture part, a PDE will be replaced by a series of DAEs. Finite difference approach [12] is used as the discretization method. Note that the discretization only applies to the middle part of a hypercube space while the borders will be described by boundary conditions. The PDE in Section 9.2.4 will be discretized as:

```
(q(2)-q(1))/dx == A*q(1)'dot;
(q(3)-q(2))/dx == A*q(2)'dot;
(q(4)-q(3))/dx == A*q(3)'dot;
…
```

The boundary statements in Section 9.2.5 are translated into simple simultaneous statements:

```
q(0) == 0.0;
(q(1)-q(0))/dx == 0.0;
```

These DAEs are solvable by a VHDL-AMS 1076.1 simulator.

9.4 MEMS Accelerometer in a ΣΔ Control Loop

Figure 9.1 shows the block diagram of a MEMS accelerometer in fifth-order SDM control loop [10]. Like most conventional modelling approaches, the micromechanical sensing element is modelled as a second-order spring damping system:

$$M\ddot{z}(t) + C\dot{z}(t) + Kz(t) = F(t) \tag{9.1}$$

where M is the proof mass, C and K are effective damping and spring factor respectively, $z(t)$ is the relative displacement and $F(t)$ is the feedback force. The frequency response of the lumped model is shown in Fig. 9.2.

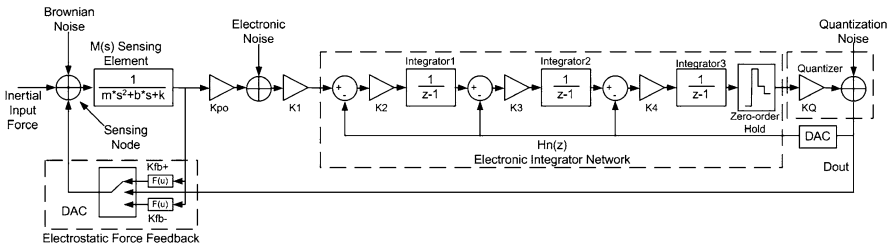


Fig. 9.1 MEMS accelerometer in SDM loop

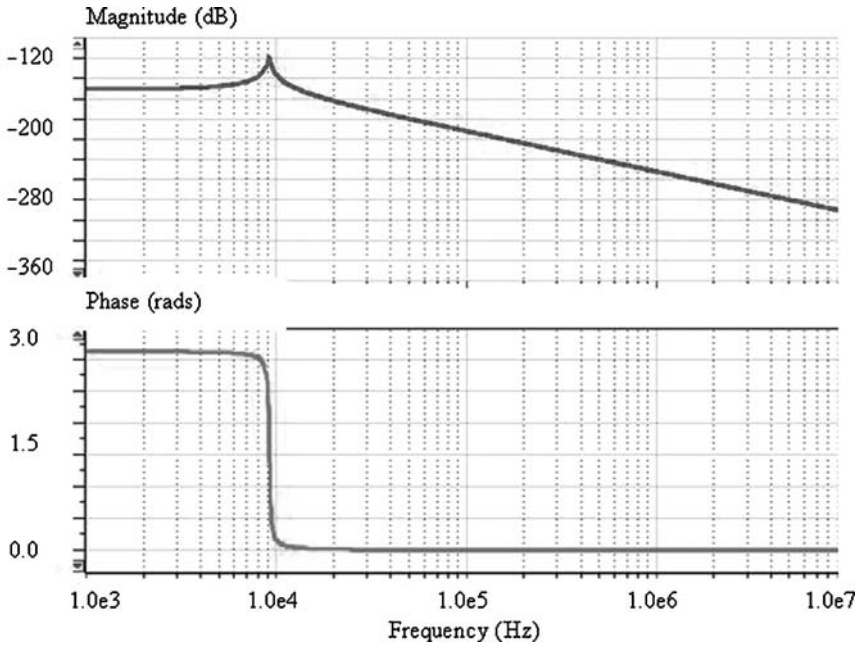


Fig. 9.2 Frequency response of the lumped model

The proof-mass displacement is converted to electronic signal by differential capacitive position sensing. The electronic signal is then passed through a third-order low-pass filter, which is implemented with distributed feedback structure. The filtered signal is digitized by a 1-bit quantizer and the output is the digital signal. The electrostatic feedback force is generated by a DAC. Such a SDM control loop has the advantages of increased dynamic range, linearity and bandwidth [10] thus it has attracted great research interests.

In actual situation, the sensing element consists of a MEMS cantilever beam located between two plate electrodes (Fig. 9.3). Instead of moving as a lumped mass, the cantilever beam itself vibrates and has higher frequency modes. It has

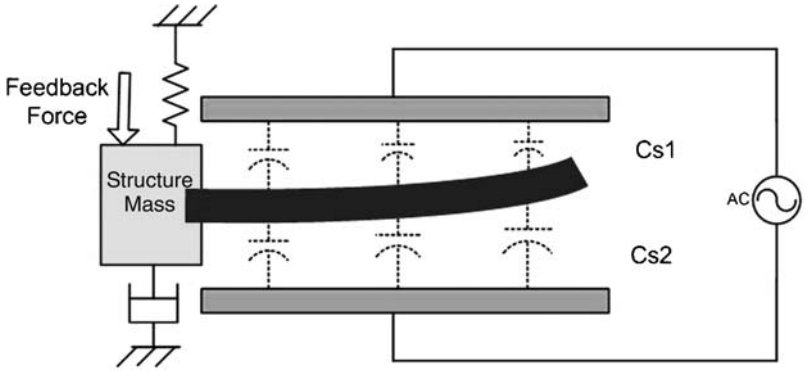


Fig. 9.3 MEMS cantilever beam as the sensing element

been proved that higher-order resonant frequencies can affect the performance of an SDM loop [13]. However, as shown in Fig. 9.2, such behaviour cannot be captured by the conventional lumped model.

9.5 VHDL-AMSP Model of the Sensing Element

9.5.1 Model Description

Figure 9.3 shows the sensing element of an accelerometer in SDM control loops. The feedback force is acting on the base of the cantilever (non-collocated dynamics) [13] and the cantilever beam is only deformed by distributed electrostatic force. The governing equation of this model is:

$$EI\frac{\partial^4 y(x,t)}{\partial x^4}+c_D\frac{\partial^5 y(x,t)}{\partial x^4\partial t}+\rho S\frac{\partial^2 y(x,t)}{\partial t^2}=F_e(x,t) \tag{9.2}$$

where $y(x,t)$ is the relative displacement at position x and time t, E is the Young's modulus, I is the moment of inertia, c_D is the damping factor, ρ is the material's density, S is the cross sectional area and $F_e(x,t)$ is the electrostatic force.

The boundary conditions at the clamped end and the free end are shown in Eqs. 9.3 and 9.4 respectively [14],

$$\begin{aligned} y(0,t)&=z(t)\\ \theta&=\frac{\partial y(0,t)}{\partial x}=0 \end{aligned} \tag{9.3}$$

$$M = -EI\frac{\partial^2 y(L,t)}{\partial x^2} = 0$$
$$Q = -EI\frac{\partial^3 y(L,t)}{\partial x^3} = 0 \tag{9.4}$$

where θ, M and Q denote the slope angle, the bending moment and the shear force respectively, L is the length of the beam.

The initial condition is simply:

$$y(x,0)=0 \tag{9.5}$$

The electrostatic force $F_e(x, t)$ is given by:

$$F_e(x,t) = \frac{1}{2}\varepsilon A\left[\frac{V_0^2}{(d_0 - y(x,t))^2} - \frac{V_0^2}{(d_0 + y(x,t))^2}\right] \tag{9.6}$$

where ε is the permittivity of the gap, A is the area of the electrode, d_0 is the spacing between the beam and the electrode and V_0 is the amplitude of the applied AC voltage.

The distributed capacitance between the cantilever and the electrode is given by:

$$C_{s1} = \frac{\varepsilon A}{d_0 - y(x,t)}, C_{s2} = \frac{\varepsilon A}{d_0 + y(x,t)} \tag{9.7}$$

The output voltage can be calculated as:

$$V_{out}(t) = \frac{C_{s1} - C_{s2}}{C_{s1} + C_{s2}} V_0 \sin(\omega t) \tag{9.8}$$

For small displacement cases, it can be assumed that $y^2 << d_0^2$. The above equation could be simplified as:

$$V_{out}(t) = -\frac{\overline{y}(t)}{d_0} V_0 \sin(\omega t) \tag{9.9}$$

where $\overline{y}(t)$ is the average beam position.

9.5.2 VHDL-AMSP Code

The VHDL-AMSP model of the cantilever beam presented below provides an example of how the elements discussed in Section 9.2 are implemented. *y* is the partial quantity which represents the deflection of the beam and *FE* is also a partial quantity which represents the electrostatic force. *x* is the spatial coordinate.

Boundary conditions have been applied and typical values are used for the constants.

```
library IEEE;
use IEEE.ENERGY_SYSTEMS.all;
use IEEE.ELECTRICAL_SYSTEMS.all;
use IEEE.MECHANICAL_SYSTEMS.all;
use IEEE.MATH_REAL.all;
entity COMB_DRIVE is
  generic(E:real; --Young's modulus
    I:real; --moment of inertia
    rou:real; --densigy
    L:real; --length of beam
    d0:real; --gap spacing
    K:STIFFNESS; --effective spring stiffness
    D:DAMPING; --effective damping
    S:real; --cross sectional area
    C:real; --cantilever damping
    A:real; --electrode area
    ep0:real; --permittivity
    M:MASS);
  port(terminal PROOF_MASS:TRANSLATIONAL);
end entity COMB_DRIVE;
architecture BCR of COMB DRIVE is
  constant N:real:=5.0;
  partial quantity y:real;
  partial quantity FE:real;
  coordinate x:real range 0.0 to L step L/N;
  quantity z across F0 through PROOF_MASS to TRANS-
LATIONAL_REF;
begin
    M*z'DOT'DOT+D*z'DOT+K*z==F0;
    --movement of proof mass
    E*I*y'dot(x)'dot(x)'dot(x)'dot(x)+ROU*S*y'dot'dot
        +C*y'dot(x)'dot(x)'dot(x)'dot(x)'dot==FE;
    --dynamics of cantilever
    FE==0.5*ep0*A*(1.0/((d0-y)**2)-1.0/((d0+y)**2));
    --electrostatic force
    BOUNDARY x at 0.0 is
    begin
      y==z;
      y'dot(x)==0.0;
    end BOUNDARY;
```

```
    --boundary condition at clamped end
    BOUNDARY x at L is
    begin
      y'dot(x)'dot(x)==0.0;
      y'dot(x)'dot(x)'dot(x)==0.0;
    end BOUNDARY;
    --boundary condition at free end
end architecture BCR;
```

9.5.3 Output from the Translation Pre-Processor –VHDL-AMS 1076.1 Code

In the output from the translator shown below, partial quantity *y* and *FE* each has been replaced by a quantity vector. The beam is discretized into five sections where the number of sections is calculated as *range/step*. The differential coefficient *dx* represents the step size. From the PDE and the boundary conditions, two sets of six DAEs are created to describe the distributed behaviour of the beam. The comments in the code below were added manually for clarity.

```
library IEEE;
use IEEE.ENERGY_SYSTEMS.all;
use IEEE.ELECTRICAL_SYSTEMS.all;
use IEEE.MECHANICAL_SYSTEMS.all;
use IEEE.MATH_REAL.all;
entity COMB_DRIVE is
    generic(…);
    port(terminal PROOF_MASS:TRANSLATIONAL);
end entity COMB_DRIVE;
architecture BCR of COMB DRIVE is
  constant N:real:=5.0;
  constant dx:real:=L/N;
  quantity y:real vector(0 to 5):=(others=>0.0);
  quantity FE:real vector(0 to 5):=(others=>0.0);
  quantity z across F0 through PROOF_MASS to TRANS-
LATIONAL_REF;
begin
    M*z'DOT'DOT+D*z'DOT+K*z==F0;
    --movement of proof mass
    FE(0)==0.5*ep0*A*(1.0/((d0-y(0))**2)-
1.0/((d0+y(0))**2));
```

```
    --electrostatic force for clamped end
    FE(1)==0.5*ep0*A*(1.0/((d0-y(1))**2)
-1.0/((d0+y(1))**2));
    --electrostatic force for section 1
    FE(2)==0.5*ep0*A*(1.0/((d0-y(2))**2)
-1.0/((d0+y(2))**2));
    --electrostatic force for section 2
    FE(3)==0.5*ep0*A*(1.0/((d0-y(3))**2)
-1.0/((d0+y(3))**2));
    --electrostatic force for section 3
    FE(4)==0.5*ep0*A*(1.0/((d0-y(4))**2)
-1.0/((d0+y(4))**2));
    --electrostatic force for section 4
    FE(5)==0.5*ep0*A*(1.0/((d0-y(5))**2)
-1.0/((d0+y(5))**2));
    --electrostatic force for section 5
    y(0)==z;
    --dynamics of clamped end
    E*I*(y(3)-4.0*y(2)+6.0*y(1)-3.0*y(0))/dx**4
      +ROU*S*y(1)'DOT'DOT+C*(y(3)'DOT-4.0*y(2)'DOT
      +6.0*y(1)'DOT-3.0*y(0)'DOT)/dx**4==FE(1);
    --dynamics of section 1
    E*I*(y(4)-4.0*y(3)+6.0*y(2)-4.0*y(1)+y(0))/dx**4
      +ROU*S*y(2)'DOT'DOT+C*(y(4)'DOT-4.0*y(3)'DOT
      +6.0*y(2)'DOT-4.0*y(1)'DOT+y(0)'DOT)/
dx**4==FE(2);
    --dynamics of section 2
    E*I*(y(5)-4.0*y(4)+6.0*y(3)-4.0*y(2)+y(1))/dx**4
      +ROU*S*y(3)'DOT'DOT+C*(y(5)'DOT-4.0*y(4)'DOT
      +6.0*y(3)'DOT-4.0*y(2)'DOT+y(1)'DOT)/
dx**4==FE(3);
    --dynamics of section 3
    E*I*(-2.0*y(5)+5.0*y(4)-4.0*y(3)+y(2))/dx**4
      +ROU*S*y(4)'DOT'DOT+C*(-2.0*y(5)'DOT+5.0*y(4)
'DOT
      -4.0*y(3)'DOT+y(2)'DOT)/dx**4==FE(4);
    --dynamics of section 4
    E*I*(y(5)-2.0*y(4)+y(3))/dx**4+ROU*S*y(5)'DOT'DOT
      +C*(y(5)'DOT-2.0*y(4)'DOT+y(3)'DOT)/
dx**4==FE(5);
    --dynamics of section 5
end architecture BCR;
```

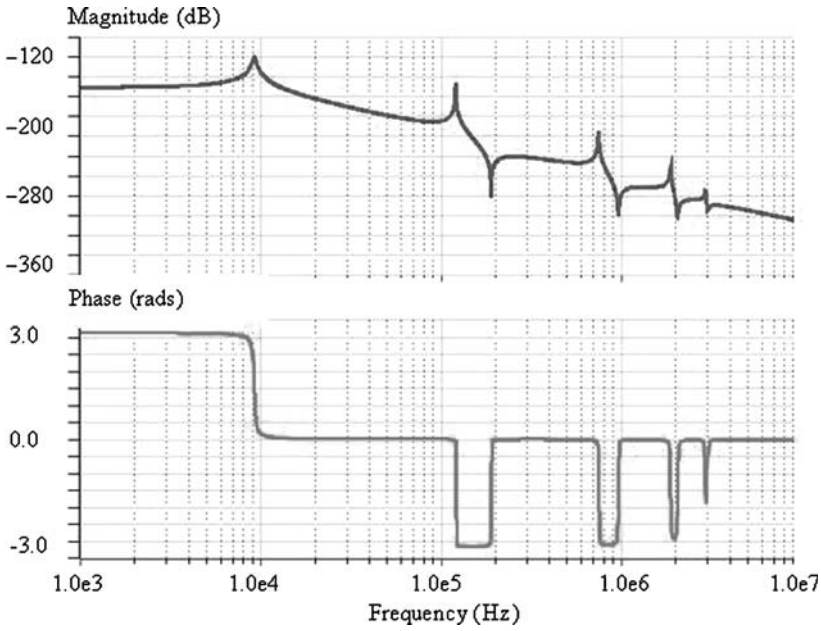


Fig. 9.4 Frequency response of the distributed beam model

9.5.4 Simulation Results

The VHDL-AMS 1076.1 description generated by the translation pre-processor has been simulated by SystemVision from Mentor Graphics [15] and simulation results showing the frequency response of average beam position are presented in Fig. 9.4. It is clear that higher-order resonant modes have been captured.

9.6 Conclusion

This paper proposes extensions to efficiently implement general partial differential equations in VHDL-AMS. The current version of VHDL-AMS (IEEE 1076.1) can only support ordinary derivatives with respect to time and faces difficulties when applied to the modelling of distributed systems. In the proposed VHDL-AMSP language, new constructs are introduced to describe PDEs in a direct form. A translation pre-processor has been developed to convert VHDL-AMSP models into VHDL-AMS 1076.1 automatically, such that models with PDEs can be simulated using currently available simulators. The added PDE support enhances the ability

of VHDL-AMS to model MEMS systems where distributed behaviour is essential. The efficiency of this new approach has been investigated by VHDL-AMSP based modelling and simulation of the sensing element of a MEMS accelerometer in high-order SDM loop. Simulation results show that VHDL-AMSP model could describe the distributed behaviour of a system which is not possible in current VHDL-AMS 1076.1 language.

References

1. Christen E and Bakalar K (1999) VHDL-AMS–a hardware description language for analog and mixed signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(10):1263–1272
2. Mahne T, Kehr K, Franke A, Hauer J, and Schmidt B (2005) Creating virtual prototypes of complex micro-electro-mechanical transducers using reduced order modelling methods and VHDL-AMS. In *Forum on Specification and Design Languages, Proceedings*, pages 27–30
3. Schlegel M, Bennini F, Mehner JE, Herrmann G, Muller D, and Dotzel W (2005) Analyzing and simulation of MEMS in VHDL-AMS based on reduced-order FE models. *Sensors Journal, IEEE*, 5(5):1019–1026
4. Shi C-J and Vachoux A (1995) VHDL-AMS design objectives and rationale. *Current Issues in Electronic Modeling, Kluwer Academic Publishers*, 2:1–30
5. Nikitin PV, Shi CR, and Wan B (2003) Modeling partial differential equations in VHDL-AMS. In *Systems-on-Chip Conference, 2003. Proceedings. IEEE International*, pages 345–348
6. Bushyager N, Tentzeris MM, Gatewood L, and DeNatale J (2001) A novel adaptive approach to modeling MEMS tunable capacitors using MRTD and FDTD techniques. In *Microwave Symposium Digest, 2001 IEEE MTT-S International*, volume 3, pages 2003–2006
7. Saldamli L, Fritzson P, and Bachmann B (2002) Extending Modelica for partial differential equations. In *2nd International Modelica Conference, proceedings*, pages 307–314
8. Proposed Verilog-A language extensions for compact modeling (2004) *http://www.eda.org/verilogams/htmlpages/compact.html*
9. Nikitin PV, Normark E, and Shi C-JR (2003) Distributed electrothermal modeling in VHDL-AMS. In *Behavioral Modeling and Simulation, 2003. BMAS 2003. Proceedings of the 2003 International Workshop on*, pages 128–133
10. Dong Y, Kraft M, Gollasch C, and Redman-White W (2005) A high-performance accelerometer with a fifth-order sigma-delta modulator. *Journal of Micromechanics and Microengineering*, 15:1–8
11. Southampton VHDL-AMS Validation Suite (2007) *http://www.syssim.ecs.soton.ac.uk/*
12. Evans G, Blackledge J, and Yardley P (1999) *Numerical methods for partial differential equations*. Springer, London
13. Seeger JI, Xuesong J, Kraft M, and Boser BE (2000) Sense finger dynamics in a SD force feedback gyroscope. In *Tech. Digest of Solid State Sensor and Actuator Workshop*, pages 296–299
14. Liu Y, Liew KM, Hon YC, and Zhang X (2005) Numerical simulation and analysis of an electroactuated beam using a radial basis function. *Smart Materials and Structures*, 14(6):1163–1171
15. Mentor Graphics Corporation (2004) *SystemVision User's Manual*. Version 3.2, Release 2004.3

Chapter 10
Mixed-Level Modeling Using Configurable MOS Transistor Models

Jürgen Weber[1], Andreas Lemke[1], Andreas Lehmler[1], Mario Anton[1], and Sorin A. Huss[2]

Abstract This contribution presents an approach to mixed-level modeling using configurable MOS transistor models as part of a behavioral model. All effects of the complete MOS transistor model can be specifically enabled or disabled in the configurable model. By activating only the effects required for the behavioral model, simulation times can be reduced significantly with very little effort. The new method is demonstrated by partitioning the MOS level-1 transistor model according to effects and implementing a configurable MOS level-1 transistor model in Verilog-A. Several examples of use will show the reduction in simulation time. The proposed approach can be used with any type of transistor model and is easily integrated in circuit simulators such as SPICE.

Keywords mixed-level modeling, Verilog-A, behavioral model, configurable, MOS transistor, virtual test

10.1 Introduction

The generation of behavioral models [1] is becoming more and more significant in the development of integrated circuits. In modern mixed-signal system design flows, a top-down design methodology followed by bottom-up verification [6, 3] is used. In the bottom-up method, the specific transistor level components of the entire design will be realized first, after that the components will be connected to larger units, and finally verified by simulation. In integrated circuit design, behavioral models are needed in different applications. Since no unified modeling strategies that cover all application ranges (executable specifications, top-down and bottom-up methodology, customer models and virtual tests [7, 5]) have been established yet, custom-designed solutions with the largest coverage of the different requirements must be used. The component based mixed-level modeling approach [8] is an efficient

[1] Atmel Germany GmbH, Heilbronn

[2] Integrated Circuits and Systems, Department of Computer Science, TU Darmstadt

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

modeling method. It applies the concept of mixed-level simulations also at the component level. Thus it is possible to describe the circuit behavior at required points with the highest precision but with the additional advantage of substantially decreasing the simulation time, with specific simplifications. In transistor models, which are used in this method, properties which are redundant for behavioral models are included. These are realized with BSIM, EKV, or other higher SPICE or SPECTRE level models, which use different regions of operation that are described using equations within the model. For example, in virtual tests the full description of the primitives is not necessary in most cases.

In this article, a method will be introduced which describes how transistor models can be partitioned and characteristics can be activated or deactivated with the goal of reducing the number of equations used, thus achieving better performance. Furthermore, MOSFET-HDL models which can be universally implemented in a multitude of modelling applications, for example in the development process (top-down methodology) or in the virtual test, can be generated.

To demonstrate this method, a boost converter, which is realized with mixed-level modelling, is used. In practical applications, low-level MOSFET models are rarely used. In the majority of cases BSIM or EKV are established here. The demonstrator, which is introduced in this article, is based on EKV models. Because of its complexity, level-1 models are used to demonstrate the new method instead. This MOSFET behavioural model is realized in Verilog-A and then simulated using CADENCE SPECTRE. The basics of a level-1 model are defined Section 10.2. In Section 10.3, the realisation of the MOSFET behavioral models in Verilog-A is introduced, with a demonstration of this method in Section 10.4.

10.2 MOSFET Level-1 Model

In the MOSFET level-1 model three regions of operation are defined according to the voltage differences between the gate, source, and drain terminals. These regions are listed in Table 10.1.

If the gate-source voltage V_{GS} is less than the threshold voltage V_{th}, no conducting channel can exist. In that case the transistor is in the cut-off region, where for the drain current $I_D \approx 0$ holds independent of the drain-source voltage V_{DS}. If V_{GS} exceeds the threshold voltage V_{th}, a channel is formed and current can flow. The resulting current I_D is approximately proportional to V_{DS} (for small V_{DS}). Thus, this region is called the linear region. If V_{DS} is increased beyond $V_{DS,sat} = V_{DS} - V_{th}$, the channel is pinched off at the drain side and I_D rises only slowly. The transistor is in the saturation region. Assuming an ideal charge distribution in the channel, the drain current can be approximated using Sah's Model [2].

Table 10.1 Regions of operation in the MOSFET level-1 model

Cut-off region	$V_{GS} < V_{th}$
Linear region	$V_{GS} \geq V_{th} \wedge 0 \leq V_{DS} < V_{DS,sat}$
Saturation region	$V_{GS} \geq V_{th} \wedge V_{DS} \geq V_{DS,sat}$

The slow rising of I_{DS} in the saturation region is caused by channel length modulation. This is described in the Shichman-Hodges model [2]. The equations for an NMOSFET in its different regions of operation can now be stated as follows:

$$I_D = \begin{cases} 0 & \left| \begin{array}{l} V_{GS} < V_{th} \\ V_{GS} \geq V_{th} \wedge \\ 0 \leq V_{DS} < V_{DS,Sat} \end{array} \right. \\ \frac{KnW}{L} V_{DS}(V_{GS} - V_{th} - \frac{V_{DS}}{2})(1 + \lambda V_{DS}) & \left| \begin{array}{l} V_{GS} \geq V_{th} \wedge \\ V_{DS} \geq V_{DS,Sat} \end{array} \right. \\ \frac{KnW}{2L}(V_{GS} - V_{th})^2 (1 + \lambda V_{DS}) & \end{cases} \tag{10.1}$$

Figure 10.1 shows the equivalent circuit of the MOSFET level-1 model.

In addition to the drain current source I_D according to Eq. 10.1, series resistances, capacitances and bulk diodes are included. The threshold voltage is a function of the source-bulk voltage V_{SB}. This behavior is called body effect and described by the following equation in the level-1 model:

$$V_{th} = VT0 + \gamma(\sqrt{|\Phi - V_{BS}|} - \sqrt{\Phi}) \tag{10.2}$$

In modern technologies the short channel effect shifts the threshold voltage of MOS transistors with short channel lengths [2]. This effect is not included in the original MOSFET level-1 model. For the model presented in Section 10.3 the short channel effect was added to Eq. 10.1 using the factor

$$\ldots * (1 + \lambda(\frac{1 + \Phi * L_{eff}}{\Phi * L_{eff}}) * V_{DS} \tag{10.3}$$

as proposed in [2].

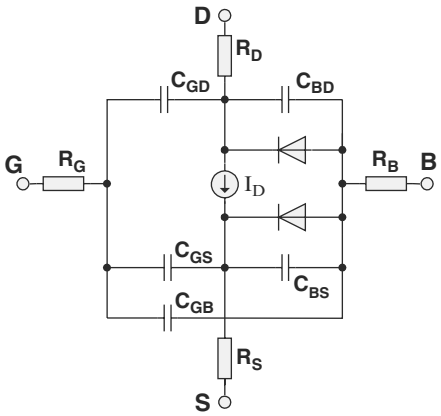


Fig. 10.1 MOSFET level-1 model equivalent circuit

10.3 Variable MOSFET HDL Model

In this section a configurable Verilog-A transistor model is developed from the level-1 MOSFET model. In a configurable model the various effects of the model can be enabled as needed with all others being disabled. Thus, the size of the Jacobian matrix is minimized and the computational effort reduced. Based on the level-1 model, the calculation of the drain current, the body effect, and the presence of the capacitors, resistors, and diodes of Fig. 10.1 are made configurable. The variants for the drain current calculation are listed in Table 10.2.

Furthermore, the components of the MOSFET behavioral model given in Table 10.3 can be selected.

In the following, three versions of the configurable level-1 model are presented: a static model using preprocessor statements that disable parts of the source code before compilation, a static model using parameters, and a dynamic model that can be reconfigured during runtime.

10.3.1 Static MOSFET Model Using Preprocessor Statements

Preprocessor statements such as *ifdef* [9] are used to select sections of the source code of the model. The advantage of this is having only these selected sections in the compiled model. However, the selection has to be made before compilation and it is global for all instances of the transistor. The following source code illustrates this approach:

'ifdef res_nodal
- With series resistances

Table 10.2 Drain current calculation in the Verilog-A model

Variant	Description
0	Voltage controlled resistance
1	Without channel length
2	Modulation
3	With channel length modulation
	With short channel effect

Table 10.3 Selectable components of the MOSFET Verilog-A model

Component	Description
res_nodal	Series resistances
cap_gate	Gate capacitances
cap_sub	Junction capacitances
dio_sub	Substrate diodes
threshold	Body effect

```
'else
   - No series resistances
'endif
```

An additional advantage of this approach is the changing of the circuit topology by adding or removing internal nodes of the transistor. An example of this is the series resistances component of the model. The series resistances require introducing internal nodes into the model. Only with preprocessor statements is it possible to have their declaration electrical *Bi*, *Di*, *Si*, *Gi*; optional. In the remaining source code the internal nodes (*Si*, *Di*, etc.) and the pins (*D*, *S*, etc.) are used, respectively.

```
'ifdef res_nodal
   vds = V(Di, Si);
'else
   vds = V(D, S);
'endif
```

10.3.2 Regions of Operation

To switch between the different regions of operation of the transistor, the variable *region* is introduced.

```
if ((vgs <= vtho) || (vds <= 0))
   region = 1;
else if ((vds < (vgs-vtho)) && (vgs>vtho))
   region = 2;
else
   region = 3;
```

10.3.3 Calculation of the Drain Current

In the behavioral model, four methods of calculating the drain current (cf. Table 10.2) are available. The most simple variant (var = 0) assumes that the transistor operates in the linear region only. The drain current is calculated as $I_D \approx \beta * ((V_{GS}-V_{th}) * V_{DS})$. This corresponds to a voltage controlled resistor with

$$R_{DS} = \frac{1}{\beta * (V_{GS} - V_{th})}, \beta = \frac{K_n * \omega}{L} \tag{10.4}$$

Another variant (var = 1) is the calculation based on the assumption of an ideal charge distribution in the channel (Sah's Model). Channel length modulation is taken into account with var = 2 and calculated according to Eq. 10.1 (Shichman-Hodge model).

This effect is controlled by the parameter λ. The final variant (var = 3) adds the short channel effect as given in Eq. 10.3. The following source code shows the implementation of the variants of the drain current calculation in Verilog-A:

```
if (var == 0) begin
   case(region)
      1: id = 'ids;
      2: id = beta * (vgs - vtho) *vds;
      3: id = beta * (vgs - vtho) *vds;
      default: id = 'ids;
   endcase
end
if (var == 1) begin
   case(region)
      1: id = 'ids;
      2: id = beta*( (vgs-vtho)-(vds/2) )*vds;
      3: id = (beta/2)*(pow( (vgs-vtho),2) );
      default: id ='ids;
   endcase
end
if (var == 2) begin
early_effect = 1 + lambda * vds;
   case(region)
      1: id = 'ids;
      2: id = beta*( (vgs-vtho-(vds/2) )*vds*early_effect;
      3: id = (beta/2)*(pow( (vgs-vtho),2) )*early_effect;
      default: id = 'ids;
   endcase
end
if (var == 3) begin
   case(region)
      1: id = 'ids;
      2: id = beta *( (vgs-vtho)-(vds/2) )*vds *(1+lambda*( (1/(2e5*Leff) )+1)*vds);
      3: id = (beta / 2)*(pow( (vgs-vtho), 2) ) *(1+lambda*( (1/(2e5*Leff) )+1)*vds);
      default: id = 'ids;
   endcase
end
```

10.3.4 Drain Current Assignment

If the series resistances component of the model is used, the drain current is assigned to the internal nodes *Di* and *Si*. Otherwise it is assigned to the external nodes *D* and *S*.

```
‘ifdef res_nodal
   I(Di,Si)<+ id;
‘else
   I(D,S)<+ id;
‘endif
```

10.3.5 Series Resistances

The series resistances connect the external nodes to the internal nodes. The model uses the effective resistances of the drain R_D and the source R_S. The resistances of the bulk and the gate are not taken into account.

```
‘ifdef res_nodal
   V(S, Si) <+ I(S, Si) *RS;
   V(D, Di) <+ I(D, Di) *RD;
‘endif
```

10.3.6 Gate Capacitances

Gate capacitances are calculated in the three regions of operation depending on C_{ox} and the terminal voltages as described in [2]. Overlap capacitances are included as well. Switching the capacitances between the regions of operation can cause convergence difficulty in the simulation. This problem is solved using the *transition* statement that provides smooth switching but also increases the computational effort.

```
‘ifdef cap_gate
   if (region == 1) begin
      cgsk=0;       cgdk=0; cgbk=cox;
   end
   if (region == 2) begin
      cgsk=( (2*cox)/3)*(1-pow( ( (vgs-vtho-vds) / (2* (vgs-vtho)-vds) ),2) );
      cgdk=( (2*cox)/3)*(1-pow( ( (vgs-vtho) / (2* (vgs-vtho)-vds) ),2) );
      cgbk=0;
   end
   if (region == 3) begin
      cgsk=(2*cox)/3;        cgdk=0; cgbk=0;
   end
qgs = (transition(cgsk)+ ‘cgso*W)* vgs;
qgd = (transition(cgdk)+ ‘cgdo*W)* vgd;
qgb = (transition(cgbk)+ ‘cgbo*L)* vgb;
   ‘ifdef res_nodal
```

```
      I(Gi,Di) <+ ddt(qgd);
      I(Gi,Si) <+ ddt(qgs);
      I(Gi,Bi) <+ ddt(qgb);
   `else
      I(G,D) <+ ddt(qgd);
      I(G,S) <+ ddt(qgs);
      I(G,B) <+ ddt(qgb);
   `endif
`endif
```

10.3.7 Junction Capacitances

Junction capacitances arise from the pn junctions at the interfaces from source and drain to substrate. These capacitances are voltage dependent, cf. [2].

```
`ifdef cap_sub
   fbp =`FC*`mj;
   if (vbd <= fbp)
      cbd=`cj*`Abd *(1-(vbd*1/`PB) );
   else
         cbd=( (`cj*`Abd)/pow( (1-`FC),1+`mj) ) *(1-(1+`mj)*`FC+`mj*vbd/`PB);
   if (vbs <= fbp)
         cbs =`cj*`Abs*(1-(vbs*1/`PB) );
   else
         cbs=( (`cj*`Abs)/pow( (1-`FC),1+`mj) )*(1-(1+`mj)*`FC+`mj* vbs/`PB);
      `ifdef res_nodal
            I(Bi,Si) <+ ddt(cbs * vbs);
            I(Bi,Di) <+ ddt(cbs * vbs);
   `else
         I(B,S) <+ ddt(cbs * vbs);
            I(B,D) <+ ddt(cbs * vbs);
      `endif
`endif
```

10.3.8 Substrate Diodes

Substrate diodes are located between the internal nodes of bulk and drain and source, respectively. By using *$vt*, the temperature voltage calculated by the simulator is accessed. The reverse saturation current *`is* is used for both $I_{S,S}$ and $I_{S,D}$.

```
`ifdef dio_sub
   ibd = `is*(limexp(vbd /$vt)-1.0);
   ibs = `is*(limexp(vbs /$vt)-1.0);
```

```
  ‘ifdef res_nodal
    I(Bi,Di) <+ ibd ;
    I(Bi,Si) <+ ibs ;
  ‘else
    I(B,D) <+ ibd ;
    I(B,S) <+ ibs ;
  ‘endif
‘endif
```

10.3.9 Body Effect

The threshold voltage is mainly dependent on the bulk-source voltage (body effect). As a model parameter the zero-bias threshold voltage *vt0* is passed.

```
‘ifdef threshold
  vtho = vt0+(gamma*( (sqrt(abs(phi-vbs) ))-(sqrt(phi) ) ));
‘else
  vtho = vt0;
‘endif
```

10.3.10 Static MOSFET Model Using Instance Parameters

In the static MOSFET model using instance parameters, the *if-else* statement is used instead of preprocessor statements. The individual functions of the model are enabled and disabled by instance parameters so that each transistor instance is configured individually. This is an advantage of this method as opposed to preprocessor statements. As a disadvantage, the computational effort increases slightly in the simulation.

```
parameter integer res_nodal = 0;
if (res_nodal == 1)
  - With series resistances
else
  - No series resistances
end
```

10.3.11 Dynamic MOSFET Model

In some applications such as the virtual test, several tests are grouped and must be simulated in one simulation run. Thus, a MOSFET model is required that can be reconfigured dynamically during runtime. This is implemented by replacing the parameters of the static model with variables.

```
integer res_nodal;
if (res_nodal == 1)
   - With series resistances
else
   - No series resistances
end
```

These variables can be switched individually for each instance by the test bench. The following source code shows an example of a test bench that activates the series resistances of an instance after 10,000 units of time.

```
   I_top.I_inv.I_NMOS1.res_nodal = 0;
   # 10000
   I_top.I_inv.I_NMOS1.res_nodal = 1;
end
```

Currently, AMS-Designer and Spectre do not support analog statements in Verilog-A that use variables inside of *if* clauses. Therefore, these statements, e.g. to calculate currents of capacitances using *ddt* or currents of diodes using *limexp*, have to be kept outside of *if* clauses. Thus, the maximum possible reduction in simulation time cannot be achieved with the dynamic models. For that reason, in the following section the static model using instance parameters is used.

10.4 Results

In this section, the use of configurable MOSFET behavioral models within mixed-level modeling is demonstrated on a boost converter and the results are discussed.

10.4.1 DC Behaviour with a Configurable MOSFET Model

Figure 10.2 shows the simulated output characteristic of the NMOS with varied settings for the configurable MOSFET behavioral model.

The capacitances (gate and junction capacitances) do not influence the DC behaviour. The same applies to the substrate effect, because bulk and source are connected together.

10.4.2 Implementation of the Configurable MOSFET Model

The following aspects have to be considered during the configuration of the configurable MOSFET behavioral model:

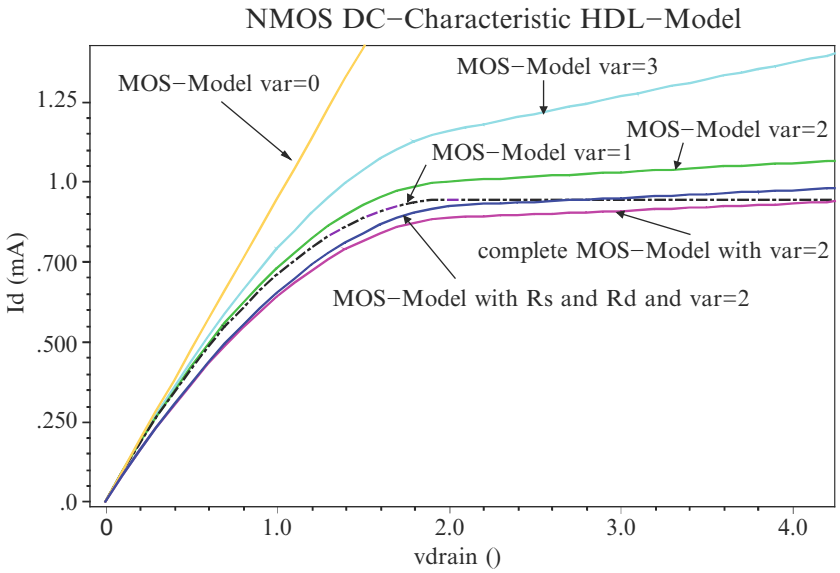


Fig. 10.2 MOSFET characteristic of varied settings for the variable MOSFET HDL model

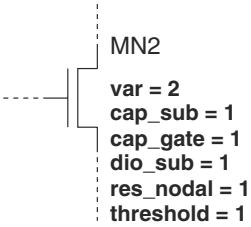


Fig. 10.3 Configuration of the behavioural model at the symbol

Application of the circuit (testbench), operation point of the circuit and the transistor, respectively, simulation type (DC, transient, etc.), required accuracy of the simulation, purpose of the simulation (virtual test, development phase, system simulation, etc.). To choose the correct settings, circuitry knowledge is necessary. The settings are made directly at the transistor symbol in the schematic editor, as shown in Fig. 10.3.

The following examples show the proceeding for various simulation tasks.

10.4.3 Boost Converter

In the following example, a boost converter is introduced. The circuit was taken from an antenna driver IC for passive-entry-go systems. This type of circuit presents a problem for system simulations because of its complexity it requires a

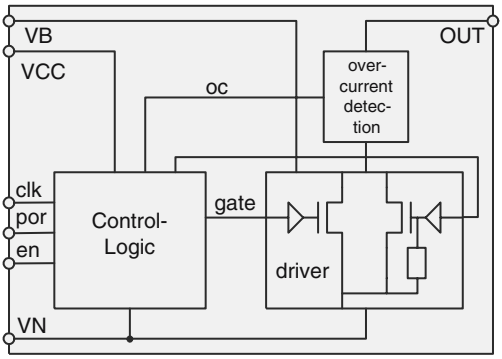


Fig. 10.4 Block diagram simplified model

lot of processing and thus, substantially increases the simulation time. The behavioral description of the converter was generated for two different abstraction levels: firstly, a simplified model created by applying meet-in-the-middle design methodology [4] and, secondly, a complex HDL model created using bottom-up strategies. Both behavioral models include a driver stage, which is implemented at transistor level with configurable MOSFET HDL models. In Fig. 10.4, a block diagram of the simplified HDL model of the boost converter is shown.

A simplified model for the gate control of the output driver with an over current detection, a voltage divider, control logic and the driving stage are implemented in this block diagram. Conversely, in the complex HDL model all sub-blocks of the transistor circuit are included. This includes, for example, the error amplifier, the compensation stage, the ramp generator, the over current detection, the voltage divider, the control logic, and the driver stages. The block diagram is shown in Fig. 10.5.

This type of model is optimal for the development of transistor models, since the complete control loop is mapped in the model. Analysis of stability and compensation of the control loop, respectively, are now possible.

The driver stage consists of four buffers, which are used for the gate control of the four output transistors and a sensor transistor, which is responsible for the over current detection (see Fig. 10.6). The performance benefit and the model difference of the HDL models in comparison to the original transistor circuit are shown in Table 10.4.

Here the run-up, as can be seen in the simulation results in Fig. 10.7, was tested.

In the driving stages all functions of the variable MOSFET HDL models were enabled during simulation. The model difference was calculated using the Euclidean distance. Within the circuit design, the complex behavioral model was used which allows the user, for example, to optimize/stabilize the compensations of the circuit. With suitable settings of the MOSFET HDL model functions, only a limited performance benefit is achieved. This is because the simulation activities occur mostly in the control loop instead of the driving stage.

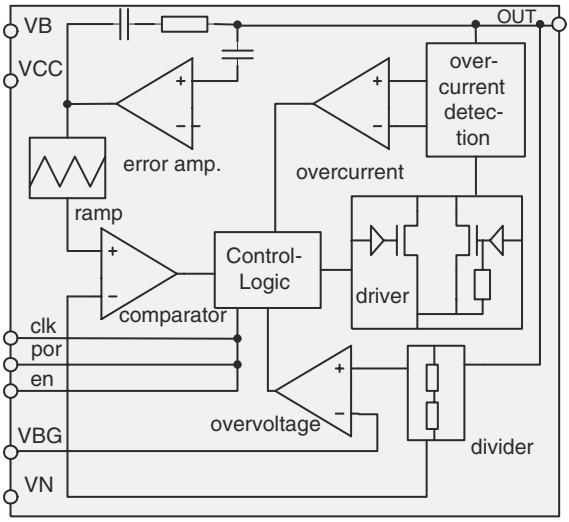


Fig. 10.5 Block diagram complex model

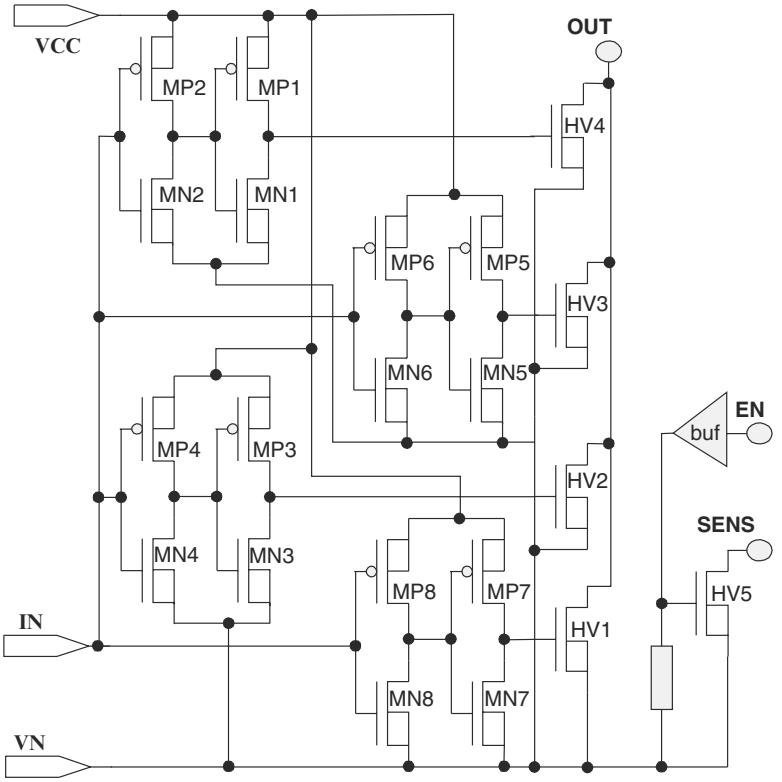


Fig. 10.6 Block diagram boost driver

Table 10.4 Performance and accuracy of the boost converter models

Model level	Performance	Difference
Complex model	92 ×	7.2%
Simplified model	181 ×	13.4%

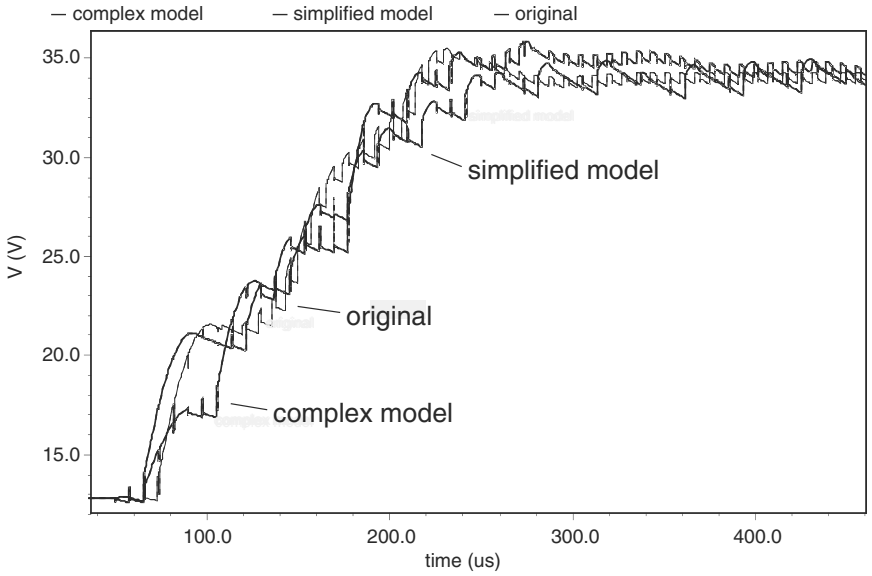


Fig. 10.7 Simulation results boost converter (simplified, complex, original)

Table 10.5 Results of the complex boost converter models with suitable settings of the MOSFET-HDL model function

Model level	Performance	Difference
Complex model	123 ×	9.4%

The components of the control loop, such as the error amplifier, are partly described with Laplace functions in Verilog-A. Table 10.5 shows the results in comparison to the original transistor circuit. During a system simulation and a test simulation, respectively, the simple behavioral models which include the primary functions like current and voltage switch-off as well as all digital control functions to switch on and off the stage are often used. In the following example, a simulation task checks the current switch-off of the converter. Here the converter must be regulated first to the steady-state voltage and then the threshold of the current switch-off can be determined by allowing the load current to rise slowly.

Here, as opposed to the complex behavioral models, a higher performance benefit is achieved since most of the simulation activities occur in the driver stage (see Table 10.6)

Table 10.6 MOSFET-HDL models configurations

Variante	M × 1,3,5,7	M × 2,4,6,8	HV1,2,3,4,5
var	1	1	2
res_nodal	No	No	Yes
cap_gate	No	No	Yes
cap_sub	No	No	Yes
dio_sub	No	No	Yes
threshold	No	No	Yes

10.5 Conclusions

In this article, a method was discussed detailing how to partition MOSFET models. It also demonstrated how to activate or deactivate their characteristics, with the goal of achieving an improved performance. This method was demonstrated using MOSFET behavioral models which were implemented on the basis of level-1 MOSFET calculations. Additionally, the models have to offer selectable options such as short channel effect or simplifications which allow use of the transistor as a voltage-controlled resistor. The model characteristics, which can be activated or deactivated, were described individually in the source code. The model was created using Verilog-A and simulated with SPECTRE (CADENCE). Three different scenarios, and the pros and cons found for each, were used to demonstrate the correct selection of the MOSFET model characteristics. In conclusion, the configurable MOSFET HDL models were applied in a simulation example. For this example, the simulation time and the model error rate were determined using various simulation tasks. An improved simulation time by a factor of 719 and an error rate of 16.1% was achieved. A disadvantage was detected with convergence problems appearing several times. However, this problem was corrected by choosing suitable simulator settings.

It is a well known fact that using original simulator transistors (e.g. EKV, BSIM) is faster than adopting the most complex stage of expansion for MOSFET HDL models. To counteract this, CADENCE implemented a C-Compiler for Verilog-A in its latest simulator version. However, it is seen that improved simulation time occurs using the optimal MOSFET HDL models rather the original EKV transistors. To achieve improved performance it is desired to include the models, which are described in this article, in the simulator. In this article, a MOS HDL model based on the level-1 model was realized, but this just serves to demonstrate the method. It would make sense to use such a method in all MOSFET model types. Circuitry knowledge is required of the modeler to be able to determine the optimal settings needed for the model characteristics.

Acknowledgments This work has been carried out within the BMBF project “Verification of analog circuits” (VeronA).

References

1. Abidi A A (2001) Behavioral Modeling of Analog and Mixed Signal IC'sv. IEEE International Conference, 06–09 May 2001, Pages 443–450
2. Chen W (1999) The VLSI Handbook. CRC Press LLC, Boca Raton, FL, December 1999
3. Enright D, Mack R J, Massara R E (2003) Mixed-Level hierarchical analogue modelling. Circuits, Devices and Systems; IEE Proceedings, Volume 150; February 2003, Pages 78–84
4. Eschermann B, Dai W M, Kuh E S, Pedram M (1988) Hierarchical Placement for Macromodels: A Meet-In-The-Middle Approach. IEEE International Conference, 07–10 November 1988, Pages 460–463
5. Miegler M, Wolz W (1996) Development of Test Programs in a Virtual Test Environment. IEEE, VLSI Test Symposium, Pages 99–102
6. Sommer R, Rugen-Herzig I et al. (2002) From System Specification to Layout: Seamless Topdown Design Methods for Analog and Mixed-Signal Applications. DATE 02, Paris, March 4–8, ISBN 0-7695-1471-5
7. Weber J, Anton M, Huss S A (2003) Verhaltensmodellierung von Ein- und Ausgangsstufen für den Virtuellen Test von Mixed-Signal Automotive Schaltkreisen. Analog 2003, Heilbronn, 10–12 September, Pages 91–96
8. Weber J, Anton M, Huss S A (2005) Effiziente Mixed-Level Modellierung integrierter Mixed-Signal Automotive Schaltkreise. Analog 2005, Hannover, 16–18 March, Pages 217–222
9. Accellera Verilog Analog Mixed-Signal Group (2004) Verilog-AMS Language Reference Manual. Version 2.2, November 2004.

Chapter 11
Modeling AADL Data Communications with UML MARTE

Charles André, Frédéric Mallet, and Robert de Simone

Abstract The emerging OMG UML Profile for Modeling and Analysis of Real-Time Embedded systems (MARTE) aims, amongst other things, at providing a referential Time Model subprofile where semantic issues can be explicitly and formally described. As a full-size exercise we deal here with the modeling of immediate and delayed data communications in AADL. It actually reflects an important issue in RT/E model semantics: a propagation of immediate communications may result in a combinatorial loop, with ill-defined behavior; introduction of delays may introduce races, which have to be controlled. We describe here the abilities of the MARTE time model in this respect.

Keywords MARTE, UML, AADL, Timed MoCC

11.1 Introduction

The modeling phase in Real-Time Embedded design is increasingly required to support various types of timing analysis prior to final code production and testing. AADL [7] and MARTE [5] are two such modeling formalisms, in part similar in their objectives. They both provide independent descriptions of the functional applications and the execution platforms, and the possible allocation of the former onto the latter. They also support the description of both the structural organization of systems, and to some extent of their dynamic behaviors.

Our belief here is that AADL relies on a number of assumptions that make the definition of dynamic behaviors visibly simple, but largely implicit and informal – with the risk of ambiguity or misdesign, which various analysis tools then try to spot and identify. Conversely, MARTE explicit Time model with powerful ***logical time***

I3S, Université de Nice-Sophia Antipolis, CNRS, F-06903 Sophia Antipolis, Inria, F-06902
Email: {candre,fmallet,rs}@sophia.inria.fr

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

constraints allows precise specification of the scheduling aspects of application elements. Multiform logical time supported by MARTE, is inspired from the theory of tag systems [1]. Time relations and constraints between various "clocks" can be stated so as to represent the time activations of concurrent tasks. Clock constraints can thus be viewed in a way similar to the Object Constraint Language [6], as providing fancy particular constructions of Timed Models of Computations and Communications (MoCC). These MoCC are to be defined by a model architect and should be transparently used by the end-user of the modeling framework. Synchronous, time-triggered or purely asynchronous formalisms are simple – and extreme – examples of that.

In this paper, we use MARTE to make explicit part of the MoCC underlying AADL. AADL applications comprise threads, often of periodic nature – with distinct periods – connected through event or data ports. As can be seen here, the same model provides structural information – the thread connections – together with a crude abstraction of behaviors usually needed for schedulability analysis – the relative speeds of threads. AADL thread modeling thus requires the conjunct of two MARTE models – one behavioral and one structural – with the relevant logical clocks defining the relative ordering of dispatch events for the threads according to the desired semantics.

Data communications can be ***immediate*** or ***delayed***. Delayed communications are needed in particular to break down cycle propagation of data. They implicitly impose a partial order on how various threads – and their containing processes – can be executed/simulated in a simultaneous step. The issues of priority inversion involved here are dealt with in [4].

When the flow contains data-port, the communication essentially amounts to sampled production/consumption of a data value shared between two tasks. Operations are performed at the pace of the – often periodic – tasks, and the scheme is ***event-less***. In particular, data can be written or read several times if ever the relative speeds of the tasks demand it. Such a communication pattern is not readily present in UML – and thus MARTE. Modeling AADL data-port communications in MARTE is the prime goal of this paper. The operational semantics is made explicit, and the various protocols – immediate/delayed – can be constructed in a formal way. The hope is that such construction can then allow, by analytic techniques, to prevent non-determinism and pathological priority inversions to occur, in a way that is predicted and guaranteed rather than monitored by non-exhaustive model simulations.

11.2 Background

11.2.1 Time in MARTE

The metamodel for time and time-related concepts is described in the "Time modeling" chapter of the UML profile for MARTE, available at the OMG site. The time chapter is briefly described in another paper [2].

In MARTE, Time can be ***physical***, and considered as ***continuous*** or ***discretized***, but it can also be ***logical***, and related to user-defined clocks. Time may even be ***multiform***, allowing different times to progress in a non-uniform fashion, and possibly independently to any (direct) reference to physical time. The ***time structure*** is defined by a set of ***clocks*** and ***relations*** on these clocks. Here clock is not a device used to measure the progress of physical time. It is rather a mathematical object lending itself to formal processing. A clock that refers to physical time is called a ***chronometric*** clock. A distinguished chronometric clock called idealClk is provided in the MARTE time library. This clock represents the "ideal" physical time used, for instance, in physical and mechanics laws. At the design level most of the clocks are ***logical*** ones.

The mathematical model for a clock is a 5-tuple $(\mathcal{I},\preccurlyeq,\mathcal{D},\lambda,u)$ where $\mathcal{I}$ is a set of instants, $\preccurlyeq$ is an order relation on $\mathcal{I}$, $\mathcal{D}$ is a set of labels, $\lambda:\mathcal{I}\rightarrow\mathcal{D}$ is a labeling function, ***u*** is a symbol, standing for a ***unit***. For a chronometric clock, the unit can be the SI time unit `s` (second) or one of its derived units (`ms, µs …`). The usual unit for logical clocks is `tick`, but `clockCycle`, `executionStep` may be chosen as well. Since instants of a clock are fully ordered, $(\mathcal{I}, \prec)$ is an ordered set.

Clock are ***a priori*** independent. They become dependent when their instants are linked by ***instant relations*** imposing either ***coincidence*** between instants (coincidence relation $\equiv$) or ***precedence*** (precedence relation $\preccurlyeq$). ***Clock relations*** are a convenient way to impose many – often infinitely many – instant relations. Examples of clock relations are given in Section 11.3.2.

A ***Time Structure*** is a 4-tuple $(\mathcal{C}, \mathcal{R}, \mathcal{D}, \lambda)$ where $\mathcal{C}$ is a set of clocks, $\mathcal{R}$ is a relation on $\cup_{a,\, b\in\mathcal{C},\, a\neq b}(\mathcal{I}_a\times\mathcal{I}_b)$, $\mathcal{D}$ is a set of labels, $\lambda:\mathcal{I}_{\mathcal{C}}\rightarrow\mathcal{D}$ is a labeling function. $\mathcal{I}_{\mathcal{C}}$ is the set of the instants of a time structure. $\mathcal{I}_{\mathcal{C}}$ is not simply the union of the sets of instants of all the clocks; it is the quotient of this set by the coincidence relation induced by the time structure relations represented by $\mathcal{R}$. A time structure specifies a poset $(\mathcal{I}_{\mathcal{C}}, \preccurlyeq_{\mathcal{C}})$.

During a design we introduce several (logical) clocks that are progressively constrained. This causes strengthening of the ordering relation of the application time structure.

11.2.2 AADL Inter-Thread Communications

As a demonstration of the expressiveness of MARTE, we take as an example the inter-thread data communication semantics of AADL.

In AADL, the communications can be ***immediate*** (Fig. 11.1a) or ***delayed*** (Fig. 11.1b). The threads are concurrent schedulable units of sequential executions. Several properties can be assigned to threads; the one of concern here is the ***dispatch protocol***. We actually consider only periodic threads, associated with a period and a deadline, specified as chronometric time expressions (**e.g.,** period = 50 ms or frequency = 20 Hz). By default, when the deadline is not specified it equals the period.

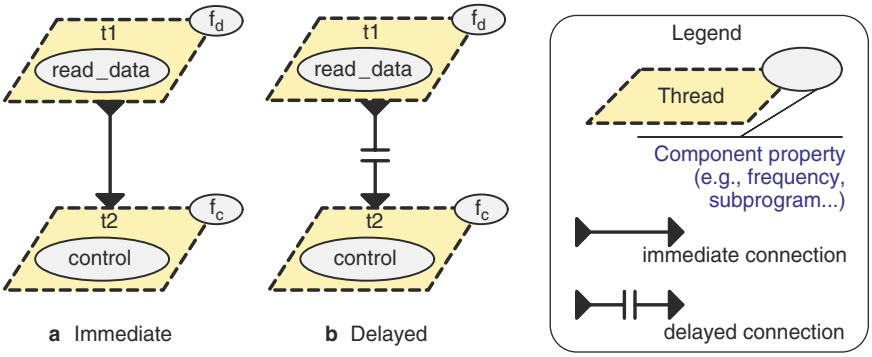


Fig. 11.1 AADL inter-thread data communication

11.3 The Explicit Modeling of AADL Communication Aspects

11.3.1 Application and Clock Refinement

A first difference with AADL is that MARTE differentiates the algorithm from the underlying structure. The algorithm is represented as an activity diagram (Fig. 11.2, left-most part). The structure is modeled as a composite structure diagram (Fig. 11.2, right-most part). Each part has its own causality constraints. MARTE refinement mechanism, and its associated clock constraints, allows for making explicit relations amongst the clocks of both parts. In MARTE, activation conditions of all application model elements are represented by clocks identified with the appropriate stereotypes, for instance `TimedProcessing`. As a starting point, we consider the clocks of each element as independent, and then the context (dependencies and refinement) constrains these clocks. Finally, a timing analysis tool may resolve the constraints to determine a (family of) possible schedule. We strive to avoid over-specification and keep the model as generic as possible, adding only required constraints. From the algorithmic point of view, the actions `read_data` and control are `CallBehaviorAction` that execute a given behavior repetitively according to their activation condition (clocks `^d` and `^c` respectively).

11.3.2 Introducing Clock Constraints

From the structural point of view, the threads `t1` and `t2` are also associated with clocks (`^t1` and `^t2` respectively). These purely logical clocks represent the dispatches of the threads. In AADL, the period of a thread is expressed as a chronometric time expression and therefore, at some point, we need to establish relations between these clocks and chronometric clocks. This aspect is addressed in Section 11.3.5, but we need to set up some causality relations first.

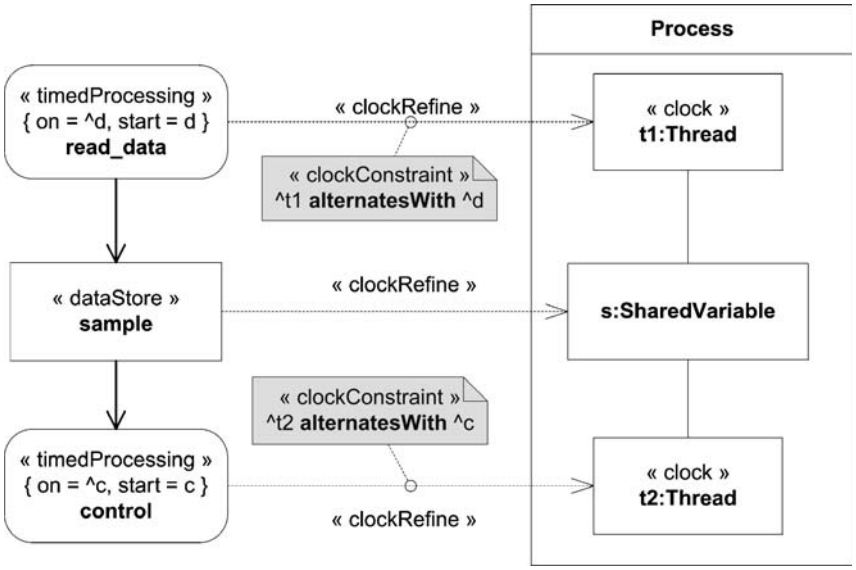


Fig. 11.2 Application/execution platform in MARTE

Deciding that a given behavior (**e.g.,** `read_data`) is executed by a periodic thread (**e.g.,** `t1`) implies that each thread dispatch (modeled by clock `^t1`) causes and therefore precedes a new execution of subprogram `read_data`, and that this execution must complete before the deadline (the next dispatch by default). In MARTE, we differentiate atomic behaviors, for which the execution time is considered negligible as compared to the period, from non-atomic ones. If we consider the behaviors as atomic, the association of a behavior with a thread is simply expressed with the constraint given by Eq. 11.1. Note that this constraint is not symmetrical since t1 may cause d, but not the converse.

$$\hat{}t1 \text{ alternatesWith } \hat{}d \tag{11.1}$$

If the execution time is not negligible, each action can be represented by two events, the start (**e.g.,** `ds` for `d`, `cs` for `c`) and the finish (**e.g.,** `df` for `d`, `cf` for `c`), and a duration. In this case, we need three constraints to express that the behavior `read_data` is repetitively executed on thread `t1` (Eqs. 11.2–11.4).

$$\hat{}t1 \text{ alternatesWith } \hat{}ds \tag{11.2}$$

$$\hat{}t1 \text{ alternatesWith } \hat{}df \tag{11.3}$$

$$\hat{}ds \text{ isFasterThan } \hat{}df \tag{11.4}$$

The first two constraints express that the behavior starts and finishes between two consecutive dispatches of thread `t1`. The last constraint, which reads clock `^ds` is

faster than clock ^df, specifies that the action read_data starts before it finishes; it is sufficient to impose that it finishes within the same cycle of execution.

The next constraint comes from the communication itself. We use a UML data store to mean that the action read_data can overwrite the existing value (in the object node) without generating a new token and this very same value can be read several times by the action control (non depleting read). In UML, there must be at least one writing before any reading (Eq. 11.5).

$$\text{^d[1] precedes ^c[1]} \tag{11.5}$$

Let ^wr be the (logical) clock for ***significant writings*** in the data store. There could be several consecutive writings in the datastore before one reading. In that case, only the last one is considered significant. Let ^rd be the corresponding (logical) clock for ***significant readings*** from the data store. When the same value is read several times, only the first reading is considered to be significant. Furthermore, AADL assumes that communicating threads must have common dispatches. A simple way to achieve that is if all threads start their execution at the same time (they are in phase). The AADL standard considers three cases: ***synchronous*** threads with the same period, ***oversampling*** (the period of control is evenly divided by the period of read_data), ***undersampling*** (the period of read_data is evenly divided by the period of control). Let q1 and q2 be natural numbers such that $f_d/f_c = q1/q2$. They represent the relative periods of read_data and control. Section 11.3.6 discusses how to compute q1 and q2 in the general case. When the threads are synchronous (Eq. 11.6), q1 = q2 = 1. When oversampling (Eq. 11.7), q1 = 1 and q2 > 1. When undersampling (Eq. 11.8), q1 > 1 and q2 = 1. max(q1,q2) is called the hyper-period. In Eq. 11.7 (resp. Eq. 11.8), the binary word [3] following the keyword filteredBy expresses that each instant of ^t1 (resp. ^t2) is synchronous with every $q2^{th}$ (resp. $q1^{th}$) instant of ^t2 (resp. ^t1).

$$\text{^t1} \equiv \text{^t2} \tag{11.6}$$

$$\text{^t1} \equiv \text{^t2 filteredBy } (1.0^{q2-1}) \tag{11.7}$$

$$\text{^t2} \equiv \text{^t1 filteredBy } (1.0^{q1-1}) \tag{11.8}$$

Selecting the significant writings and readings consists in choosing one every $q1^{th}$ instant of ^d (Eq. 11.9) and one every $q2^{th}$ instant of ^c (Eq. 11.10).

Additionally, Eq. 11.11 states that each significant writing must precede its related significant reading.

$$\text{^wr isPeriodicOn ^d period q1} \tag{11.9}$$

$$\text{^rd isPeriodicOn ^c period q2} \tag{11.10}$$

$$\text{^wr alternatesWith ^rd} \tag{11.11}$$

We restrict our comparison to the three cases considered by the AADL standard. However, in Subsection 11.3.6 we elaborate on the general case.

We have defined all general constraints. In particular, note that contrary to Eqs. 11.7–11.10 do not specify which instant is chosen as a significant writing or reading. The actual instant depends on the semantics of the communication. The following two subsections study the three different cases (synchronous, oversampling, undersampling) with both an immediate and a delayed communication, each subsection gives stronger constraints compatible with Eqs. 11.9–11.11.

11.3.3 Immediate Communication

An immediate communication means that the result of the sending thread (here `read_data`) is immediately available to the receiving thread (here `control`). When threads are synchronous (Fig. 11.3a), this is denoted by $\hat{}\texttt{wr} \equiv \hat{}\texttt{d}$ and $\hat{}\texttt{rd} \equiv \hat{}\texttt{c}$, or more precisely by $\hat{}\texttt{wr} \equiv \hat{}\texttt{df}$ and $\hat{}\texttt{rd} \equiv \hat{}\texttt{cs}$. In case of oversampling (Fig. 11.3b), the result of the action `read_data` must be written in the object node early enough so that the ***first*** (for each `q2`-long hyper-cycle) execution of the action `control` can use it. This is denoted by $\hat{}\texttt{wr} \equiv \hat{}\texttt{d}$ and $\hat{}\texttt{rd} \equiv \hat{}\texttt{c}\ \texttt{filteredBy}\ (1.0^{\texttt{q2}-1})$. The latter constraint is stronger than Eq. 11.10, it implies it. In case of undersampling (Fig. 11.3c), AADL specifies that the execution of the ***first*** (for each `q1`-long hyper-cycle) execution of the action `read_data` must complete before the execution of the action control. This is stated by $\hat{}\texttt{rd} \equiv \hat{}\texttt{c}$ and $\hat{}\texttt{wr} \equiv \hat{}\texttt{d}\ \texttt{filteredBy}\ (1.0^{\texttt{q1}-1})$.

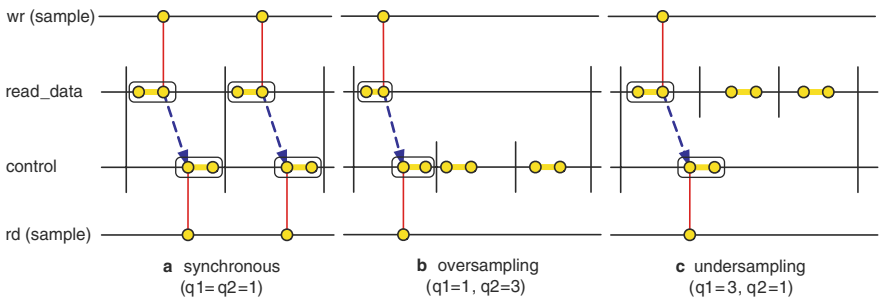


Fig. 11.3 Immediate communications

11.3.4 Delayed Communication

A delayed communication means the result of the sending thread is made available only at its ***next*** dispatch while the receiving thread only reads ***after*** its own dispatch

and ***ultimately*** when the data is required. The dispatches of the sending and the receiving threads are not necessarily all synchronous, even if there must synchronize at some point. When the thread are synchronous (Fig. 11.4a), the constraint is denoted by Eqs. 11.12, 11.13. Note that `δ4` offers the possibility to delay the actual execution of `read_data`. The thread `t1` can either be idle or be executing another action before starting to execute `read_data`. Eq. 11.12 states that $(\exists \delta 4 \in \mathbb{N})(\forall k \in \mathbb{N}^*)(\hat{}wr[k] \equiv \hat{}t1[\delta 4 + k])$.

$$(\exists \delta 4 \in \mathbb{N}))(\hat{}wr \equiv \hat{}\text{t1 filteredBy } 0^{\delta 4}\ (1))) \tag{11.12}$$

$$\hat{}\text{rd} \equiv \hat{}\text{c} \tag{11.13}$$

For oversampling (Fig. 11.4b), the result is available for the ***first*** execution of the action control of the ***next*** `q2`-long hyper-cycle. This leaves lots of freedom to schedule the action `read_data` anywhere within the current hyper-cycle. We keep the relation Eq. 11.12 while Eq. 11.13 is replaced by Eq. 11.14.

$$\hat{}\text{rd} \equiv \hat{}\text{c filteredBy } (1.0^{q2\text{-}1}) \tag{11.14}$$

For undersampling (Fig. 11.4c), the result of the ***last*** execution (for each `q1`-long hyper-cycle) of the action `read_data` is available for the action control at the ***next*** hyper-cycle. This is denoted by combining Eq. 11.15 with Eq. 11.13.

$$(\exists \delta 4 \in \mathbb{N})(\hat{}wr \equiv \hat{}\text{t1 filteredBy } 0^{\delta 4}\ (1.0^{q1\text{-}1})) \tag{11.15}$$

Note that the relations are not fully symmetrical. This is due to the AADL semantics that changes the rule depending on the kind of communication.

Up to here, we have only defined logical constraints. In some cases, these constraints are strong enough to get a total order, and thus a possible schedule, on all instants belonging to the defined clocks. For instance, in the delayed synchronous case, whenever the first execution of `read_data` occurs, the first significant writing occurs at the very next dispatch. However, in some other cases, we need additional

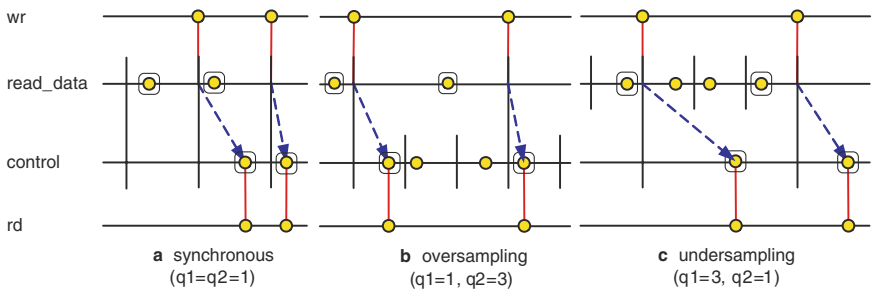


Fig. 11.4 Delayed communications

stronger constraints to get a schedule. These constraints reflect additional choices that are mainly implicit in the AADL semantics. Depending on these choices we get different deterministic schedules. These cases are studied in the next section.

11.3.5 Getting a Schedule

Figure 11.3 shows that for immediate communications, the constraints given define a total order between instants of `^d` and `^c` in both the synchronous and the oversampling cases. Combining our constraints we get the same result analytically. One question remains whether or not both executions (`read_data` and `control`) can be performed within the period of thread `t2`. If not, there is no possible schedule, otherwise, the schedule is given by Fig. 11.5, assuming both threads are executed on the same process.

For delayed communications, additional constraints are required to get a deterministic schedule. Several criteria can be considered, for instance, the size of the buffer used for the communication, or applying a well-known scheduling policy, like Earliest Deadline First (EDF).

An apparent easy way to force a total order is to project the logical clocks onto chronometric clocks. Logical clocks only give an order amongst instants (sometimes partial), while chronometric clocks give an absolute position in time. The use of chronometric clocks is implied in AADL because of the units used to describe either the frequency (`Hz`) or the period (`s`). In MARTE, we create models of chronometric clocks by discretizing `idealClk` (Section 11.2.1).

For instance, we create three chronometric clocks c_{100}, c_{10} and c_{30} of respective frequency `100`, `10` and `30 Hz` (Eqs. 11.16–11.18). Note that these are relations, whence the definition of the `30 Hz`-clock from c_{10}.

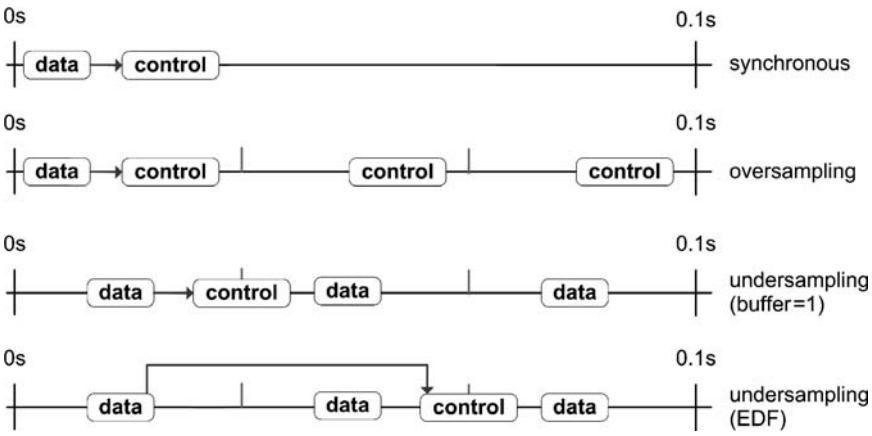


Fig. 11.5 Schedules with immediate communications

Now, we replace the three equations (Eqs. 11.6–11.8) by the three following constraints. ^t1 ≡ ^t2 ≡ c_{10} (synchronous), ^t1 ≡ c_{10} and ^t2 ≡ c_{30} (oversampling), ^t1 ≡ c_{30} and ^t2 ≡ c_{10} (undersampling). The only additional information we have here is the distance (expressed in seconds) between two consecutive dispatches. This information is useful for comparing the duration of executions with the period of the threads; however it does not change in any way the causality relations expressed.

$$c_{100} \equiv \text{idealClk discretizedBy } 0.01 \tag{11.16}$$

$$c_{10} \equiv c_{100} \text{ filteredBy } (1.0^9) \tag{11.17}$$

$$c_{10} \equiv c_{30} \text{ filteredBy } (1.0^2) \tag{11.18}$$

For the immediate undersampling, we can infer from the specified constraints that, for each hyper-cycle, the first execution of `read_data` must complete before the execution of `control`. However, we cannot decide when to execute `control` relatively to other executions of `read_data`. We need another criterion. For instance, we choose to minimize the actual size of the buffer used for the communication. To get this buffer as small as possible (size = 1), we have to schedule `control` before the second execution of `read_data`. Were we to schedule according to an EDF policy we would get another schedule, see Fig. 11.5.

For a delayed communication, we just have partial orders and we need additional criteria. For synchronous threads, the use of an EDF policy is of no help. However, reducing the size of the communication buffer gives a schedule (top-most part of Fig. 11.6). For oversampling, both criteria are compatible and we get the second schedule on Fig. 11.6. For undersampling, we get two different schedules depending on whether we apply an EDF policy or we attempt to reduce the buffer size.

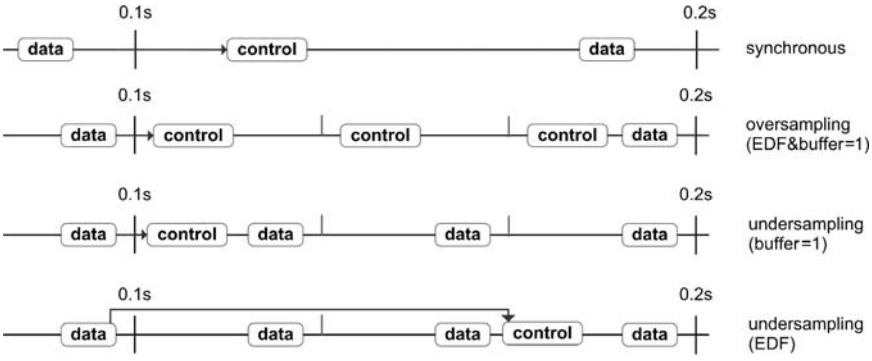


Fig. 11.6 Schedules with delayed communications

11.3.6 Generalization

We can generalize the constraints to get only two sets of constraints, one for the immediate communication and one for the delayed communication.

In this section we do not restrict to the three special cases addressed in the AADL standard. This generalization does not assume that the frequencies of the threads are natural numbers; it just assumes that they are rational numbers. It also assumes that in the notation of our binary words $Y.x^0 = Y$, for any binary word Y and any bit x.

Let $fd = n_r/d_r$ and $f_c = n_c/d_c$, $f_d/f_c = (n_r * d_c)/(n_c * d_r)$ with $n_r, n_c, d_r, d_c \in \mathbb{N}^*$. Let $r_1 = n_r * d_c$ and $r_2 = n_c * d_r$. We choose $q1$ and $q2$ such as $q1 = r_1/\gcd(r_1, r_2)$ and $q_2 = r_2/\gcd(r_1, r_2)$. Note, that we still have $f_d/f_c = q1/q2$ and that the constraints given by Eqs. 11.14 and 11.15 are general. However, Eqs. 11.6–11.8 are replaced by a single one, Eq. 11.19.

$$\hat{}t1 \text{ filteredBy } (1.0^{q1-1}) \equiv \hat{}t2 \text{ filteredBy } (1.0^{q2-1}) \tag{11.19}$$

Again, these constraints are purely logical. In the general case, these constraints are not strong enough to identify deterministically the significant writings and readings. If we take for instance, the case where $q1 = 2$ and $q2 = 5$ (Fig. 11.7). If we apply the AADL semantics, we can only say that, within an hyper-cycle (of period $\text{lcm}(q1, q2)$), the first execution of `read_data` produces the sample for the first `control`, but we cannot know what sample is used by other executions of `control`. In particular, there is no relation between `t1[2*n+1]` and `t2[5*n+2]`.

To get a deterministic behavior, we need to give more constraints. For instance we can project our clock to chronometric clocks and we model as an example the case where $f_d = 10\,\text{Hz}$ and $f_c = 25\,\text{Hz}$. We proceed by using the clock c_{100} defined in Eq. 11.16 and we add two new constraints (Eq. 11.20-11.21).

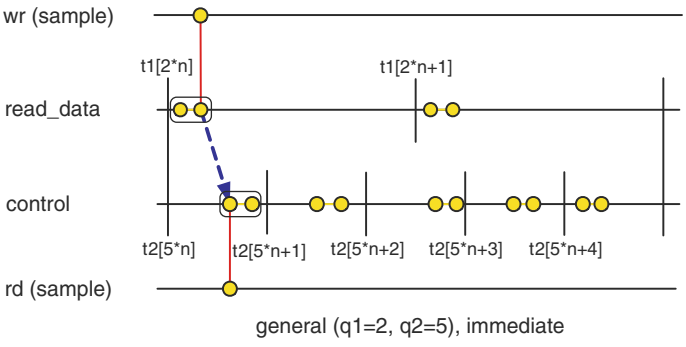


Fig. 11.7 Immediate communications and purely logical clocks (q1 = 2, q2 = 5)

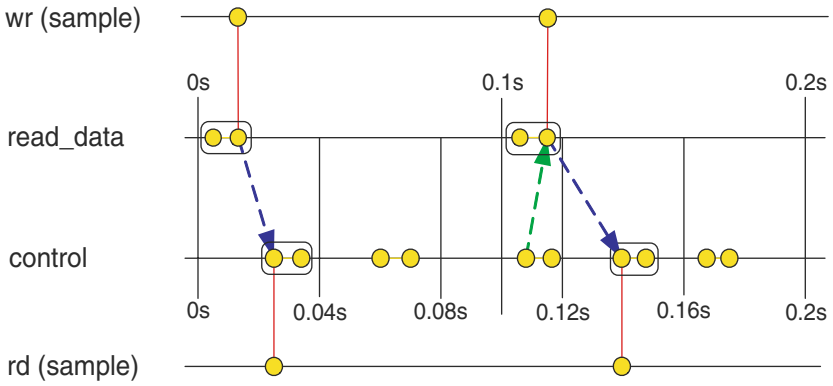


Fig. 11.8 Immediate communications and chronometric clocks (q1 = 2, q2 = 5)

$$\hat{}t1 \equiv c_{10} \tag{11.20}$$

$$\hat{}t2 \equiv c_{100} \text{ filteredBy } (1.0^3) \tag{11.21}$$

With such constraints, we get a total order (Fig. 11.8) and then there are two possible cases.

The first case appears when `duration(read_data)+duration(control)≥0.02s`. Then, we exactly get the result presented in Fig. 11.8, where, within a hyper-cycle, the third execution of control uses the sample computed by the first execution of `read_data` and the fourth execution of `control` uses the sample computed by the second execution of `read_data`.

In the second case, if `duration(read_data)+duration(control)<0.02s`, the third execution of control should use the sample computed by the second execution of `read_data`. However, note that such systems that very much depend on the exact duration of tasks are not very robust.

If we now take a look at the situation with a delayed communication (Fig. 11.9), there are several possible interpretations of a generalized AADL semantics. The simplest interpretation is that the data is made available (written in the object node) at the first dispatch (of the sending thread) following the execution of the behavior that has produced it (`read_data`). And the data is read at the first dispatch of the receiving thread following the writing (see Fig. 11.10).

A second interpretation (see Fig. 11.11) could be that the data is read at the first dispatch of the receiving thread following the actual production of the data (not waiting for the following dispatch of the sending thread). This interpretation leads to make the second significant reading synchronous with the third instant of `control` (for each hyper-cycle) instead of the fourth as in Fig. 11.10. These cases are studied in detail in [2].

Note these two interpretations can all be valid and deterministic. It is just a matter of making explicit the semantics. The first interpretation is very simple to implement and the second one requires being able to control very tightly the communication times.

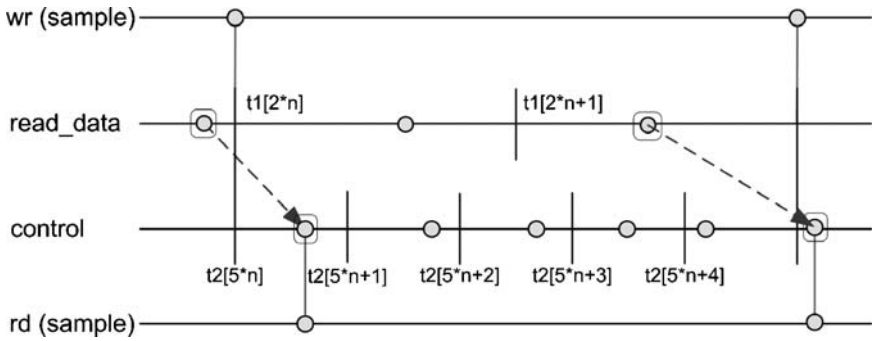


Fig. 11.9 Logical clocks (q1 = 2, q2 = 5)

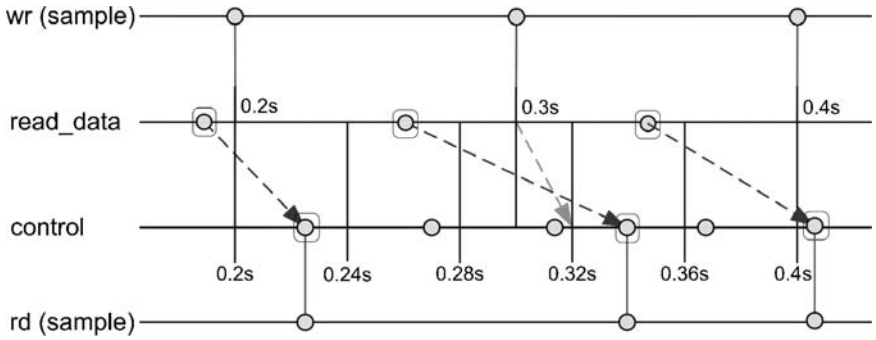


Fig. 11.10 First interpretation with delayed communications (q1 = 2, q2 = 5)

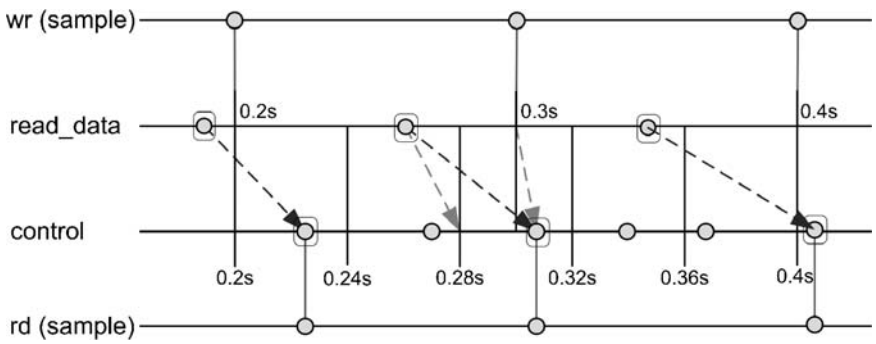


Fig. 11.11 Second interpretation with delayed communications (q1 = 2, q2 = 5)

A UML object node has two interesting attributes: it has an upper bound, possibly unlimited, and it can order events, by default according to a FIFO policy. Thus, there is no reason to assume that the threads are in phase, the sending thread writes (and possibly overwrites) tokens in the object node, while the receiving thread reads them when required. Our definition of the significant writings and readings

helps defining when the token is the same – the content must be overwritten – and when the token is different, which implies that a new token must be created. Actually, the occurrence of `^wr` should create a new token.

11.4 Conclusion

We have briefly introduced the Time model of MARTE and we have illustrated its use on an example taken from AADL. We think that our clock constraint language could be used to make formal the semantics of UML-like graphical representations that is often partially implicit. In this language, we borrowed some notations on binary words from the N-synchronous approach but in our case we do not limit ourselves to synchronous relations. We have implemented a constraint parser that has been made available with the XMI of the Time subprofile on the OMG website. This parser can be used to parse constraints extracted from UML models. Some analytic tools should reduce the constraints or compute new ones and put them back in the models. For now, all these formal computations are manual but we intend to transform our constraints into languages amenable to clock computations (time automata or synchronous languages like Signal or Esterel). Ultimately, our constraint language could be used to drive a UML simulator, in a constructive way, according to the model time semantics rather than an untimed event-driven semantics.

References

1. Lee E.A., Sangiovanni-Vincentelli A.L. (1998): A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(12):1217–1229.
2. André C., Mallet F., de Simone R. (2007): Modeling Time(s). Springer LNCS 4735:559–573.
3. Cohen A., Duranton M., Eisenbeis C., Pagetti C., Plateau F., Pouzet M. (2006): N-synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-time Systems. Conference Record of the ACM Symposium on Principles of Programming Languages, pp.180–193.
4. Feiler P.H., Gluch D.P., Hudak J.J., Lewis B.A. (2004): Embedded System Architecture Analysis Using SAE AADL. Carnegie Mellon University, Technical Note CMU/SEI-2004-TN-005, June 2004. http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tn005.pdf.
5. OMG: UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), beta 1, August 2007, Document ptc/07-08-04. http://www.omg.org/docs/ptc/07-08-04.pdf
6. OMG: Object Constraint Language (OCL), OMG Available Specification, Version 2, May 2006, Document formal/06-05-01. http://www.omg.org/docs/formal/06-05-01.pdf
7. SAE: Architecture Analysis and Design Language (AADL). June 2006, Document AS5506/1. http://www.sae.org/technical/standards/AS5506/1.

Chapter 12
Software Real-Time Resource Modeling

Frédéric Thomas[1], Sébastien Gérard[1], Jérôme Delatour[2], and François Terrier[1]

Abstract Setting up truly flexible design processes becomes an important challenge to face with the increasing complexity, the shorter time to market constraints and the constant evolution of Real-Time Embedded (RTE) software requirements. One promised solution is the model driven development (MDD) based on the principle of separating the application description from its platform specific implementation. Nowadays, this is often done through dedicated model transformations which implicitly represent the platform model. Specific transformations have shown their limits as soon as we want to optimize the implementation. In this context, a good compromise could be to make explicit a platform model. This is one of the challenges addressed by the Object Management Group (OMG) through the definition of the standard profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). In particular, the capabilities to model software real-time embedded resources will allow describing explicitly the RTE software multitasking platform characteristics. It will ease their integration in a flexible design process (both to produce implementation and to perform accurate scheduling of performance analysis).

Keywords Platform modeling, MARTE, UML profile, software modeling, multitasking

12.1 Introduction

Real-time embedded (RTE) application design methodologies are classified under two headings: sequential-based design (also called loop design) and multitasking-based design. The first category calls for designing applications as a set of ordered

[1]CEA LIST, Boîte 94, Gif sur Yvette 91191, France
Email: {frederic.thomas, sebastien.gerard, francois.terrier}@cea.fr

[2]ESEO TRAME, 49000 Angers, France
Email: jerome.delatour@eseo.fr

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

sequential actions, whereas the second category aims at designing applications as a set of units that execute concurrently and interact (i.e. communicate and synchronize) via specific mechanisms such as semaphores and messages. Our work falls under the second heading. Most multitasking-based approaches rely on a specific multitasking execution platform, called a real-time operating system (RTOS). This latter runs on top of a hardware platform and offers to designers the well-suited constructs needed to support both features, concurrency (e.g. task, thread and process) and interactions (e.g. mailbox, shared memory and semaphore).

Like the software/system engineers, real-time embedded system (RTES) engineers are faced with the challenge of developing more and more complex systems, achieving higher quality at a lower cost, and in a shorter time. Within this context, reusability, maintainability and portability become major issues in RTES design processes. The usage of RTOS platforms was an initial, "architectural" response to these problems by enabling the development of applications independently of their hardware computing platforms. The RTES design community has made efforts for standardizing application programming interfaces (APIs) of RTOS, as for example POSIX Std 1003.1 [1], OSEK/VDX-OS [2] and ARINC-653 [3]. Usage of such standards for designing multitasking-based applications has fostered reuse of applications in different software contexts. Nevertheless, they cannot answer all portability problems. Platform is a great concern for RTE system designers since their performances are passed directly on to the applications. A unique, standard and universal implementation is thus a dream. Few RTOS APIs are actually conformant to a given standard. Moreover, standard APIs provide intentional degrees of freedom for the implementation. Hence, systematic, standalone and syntax transformations (e.g. code generation) based on standard APIs fail to deal with engineer needs.

For some years, the IT community has proposed a new development approach said to be model-driven. This initiative places the model paradigm and the use of model transformations at the center of the development process. One promised solution is to separate the application description from its platform specific implementation. The most mature formulation of this vision at present is the Model-Driven Architecture (MDA) approach [4]. This latter is promoted by the Object Management Group (OMG). MDA involves a Y-chart design process in which a platform-independent model (PIM) of the software is transformed into a platform specific model (PSM); given a platform description model (PDM). All these models are described in the Unified Modelling Language (UML) [5]. We propose to investigate the MDA approach to design RTE systems. Thus, we want to model RTE multitasking execution platform with UML.

Due to its general purpose, UML lacks certain key native artifacts for describing concrete and precise RTE multitasking concepts such as task, semaphore, and mailbox. This lack has been full by a new OMG standard dedicated to modeling and analysis of real-time and embedded systems, MARTE [6]. In that context, this paper presents the Software Resource Model (SRM) UML profile dedicated to characterize RTE multitasking execution platform.

After a quick tour around related work for software execution platform modeling with UML, we show how to achieve this goal thanks to MARTE and how it can

help application design in a model-driven style. Finally, we will give some conclusions and will elaborate on some possible future work.

12.2 Related Work

Much work has already been done on platform modeling. This section is therefore divided into two sub-sections: one dedicated to related research on characterizing execution platforms and the other to model such platforms with UML.

12.2.1 Characterizing RTE Software Platforms

The MDA guide [4] provides the following generic definition of the platform concept: "*A platform is a set of subsystems and technologies that provide a coherent set of functionalities through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented*". Although this is a very broad and high-level definition which leaves a large scope for interpretation, it does make clear that the MDA guide considers a platform as a support for the execution of software applications. This correlate well with the industrial intuitive definition of "platform" which refers to machines or systems such as frameworks, middleware, virtual machines and RTOS, which are built to support an execution process.

According to B.Selic in [7], this "*enabling execution*" concept consists of providing resources (i.e., mechanisms) and services (i.e., functionalities) to be used by one or more software applications. Resources are structural entities offering services that may be qualified by non-functional properties (e.g., latency, worst case execution and pool size). These properties reflect the offered execution characteristics and the platform performances.

A. Sangiovanni-Vincentelli emphasizes in [8] that resources and services are provided by application programming interfaces (APIs). An API should provide a complete and accurate description of the platform, so that any application that is consistent with this interface is guaranteed to be processable via that platform. Hence, the API may be considered as a representation of this "*enabling execution*" concept.

We can thus summarize previous discussion on to characterizing execution platforms as follows: an execution platform is "*an abstraction layer in the design flow which interfaces through its API a set of resources (i.e., types and instances) composed of a set of services and a set of usage patterns, either with other platform resources or with other client systems called applications*". The language used to model such execution platforms must therefore allow modeling of specific RTE types, along with predefined instances and usage patterns.

12.2.2 Execution Platform Modeling with UML

UML 2.0 [5] is now widely used for software development and is emerging as a possible solution for enhancing RTE system development [9]. UML provides linguistic concepts to describe types (i.e., resources), instance specifications (i.e., resource instances), and behavioral features (i.e., services). Moreover, UML provides means to describe usage patterns as "collaborations" and "collaboration uses" within composite diagrams. Since explicit platform models can have an arbitrarily complex structure, we can also use UML 2.0's composite structures to break down a complex design into smaller parts. In such a view, the concepts of connector and port may be useful to describe the binding of applications with platforms [7]. Finally, state-machine and activity diagrams may be associated with encapsulated classifiers to define their behaviours.

UML native concepts nevertheless need to be extended to cover the semantics of RTE concepts. For that precise purpose, UML provides a lightweight extension mechanism called profile (see Section 12.18 of [5]). A UML profile consists of "stereotypes" and "constraints". Stereotypes may have properties called "tags" and are used to define extensions to existing UML language constructs (metaclasses). They likewise enable use of platform/domain-specific terminology and notation. Constraints are used to restrict or to specify the usage of the stereotypes within the context of a UML model. When they are written in Object Constraint Language (OCL) [5], constraints can be checked automatically on UML models applying a profile. They then provide support for checking static semantic rules.

A large number of UML extensions for real-time and embedded designs have already been proposed. In [10], the UML Profile for Schedulability, Performance and Time (SPT), standardized by the OMG, proposes mainly concepts for two kinds of analysis: RMA-based schedulability analysis, and performance analysis based on layered queuing theory. For platform modeling, SPT provides only high-level concepts. This lack has been one of the OMG motivations for a new RTE profile, MARTE.

In [11], P. Kukkala proposes a model-driven methodology based on both UML and a specific profile to describe applications and platforms. This methodology does not only allow the description of platform structures but also the binding of applications with platforms. The proposed profile does not, however, take into account the operating system as a platform.

In [12], R. Chen proposes a UML profile for specification of embedded system platforms. This profile provides domain-specific classifiers and relationships to supplement the SPT approach. However, it does not include means to describe platform services, and essentially enables to annotate resources. Such an approach may not allow automating completely the binding of the application with the platform.

In [7], B. Selic describes a straightforward but relatively general UML profile for platform modeling and deployment of relationships between platforms and applications. Although this model enables a systematic approach to factor crucial platform characteristics, the proposed profile does not provide specific concepts to model RTE execution resources such as tasks and semaphores.

Related work has resulted in UML extensions that make notation and semantics more suitable for highly abstract real-time concept modeling with UML. But, all this work does not provide enough detailed artifacts to describe both resources and services provided by software execution platforms. A complete explicit model will facilitate and automate binding of an application with its RTOS execution platform. We have consequently proposed a new UML 2.0 profile for that purpose, the UML profile for Software Resource Modeling (SRM). This latter is part of the MARTE standard.

The MARTE specification consists of three main packages described in figure 12.1. The first package defines the foundational concepts of MARTE. It provides basic model constructs for non-functional properties, time and time-related concepts, allocation mechanisms and generic resources, including concurrent resources. These foundational concepts are then refined in both other packages to respectively comply with modelling and analyzing concerns of real-time embedded systems.

The second package provides a generic basis for different quantitative analysis subdomains. This Generic Quantitative Analysis Modeling package is further generalized into two packages: one for schedulability analysis, to predict whether a set of software tasks meets its timing constraints; and another for performance analysis, to determine if a system with non-deterministic behaviour can provide adequate performance.

The third, "Real-Time Embedded Design modelling" package provides support for modelling high-level model constructs to depict real-time embedded features of applications, but also for enabling the description of detailed software and hardware execution platforms. The SRM profile deals with the software execution part.

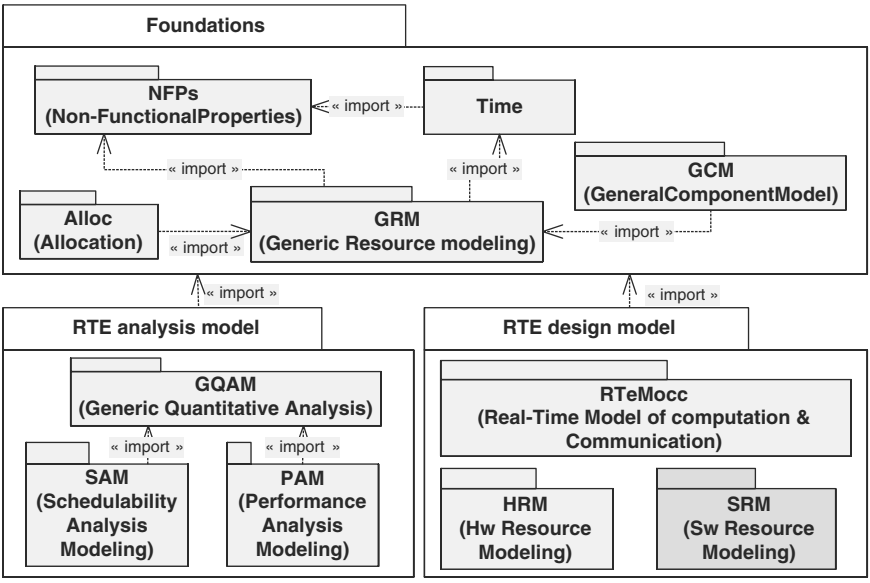


Fig. 12.1 Overview of the UML MARTE profile

12.3 Software Resource Modeling

There are currently no formal ways for designing UML profiles. We have designed our profile in two stages. The first stage aims at defining all concepts required to cover a specific domain. The output of this stage is called the "domain model" of the profile. It is considered as a specification of the domain-specific language. The second stage then consists in designing the previous language specification in terms of UML extensions, i.e., defining UML stereotypes, their properties and additional constraints. Thus, that section is organized as follows: the first subsection is an outline of the SRM domain view, the second is an overview of the SRM UML profile and the last two present some SRM Profile usage examples.

12.3.1 Outline of the SRM Domain View

The SRM profile is based on the "resource-service" modeling pattern proposed for platform modelling in [7] and [10]. That pattern allows describing resources which own properties and provide services. Some properties and services play roles. Such roles are modelled as resources attributes. Figure 12.2 illustrates that pattern on a software resource. A software resource owns some attributes. Among those attributes some are used to identify the resource. Those are referenced by the "identifierElements" meta-property. A software resource provides also services. Some may be used either to create or to delete the resource. Those are respectively referenced either by the "createServices" or by the "deleteServices" meta-properties.

The whole domain model has been built on the basis of a detailed analysis of main RTOS API standards [1–3], and certain industrial standards (e.g. [13, 14]). An overview of domain resources is shown in Tables 12.1–12.3. Real-time embedded software concepts may be classified according to following concerns:

- Concurrent execution (i.e., parallel execution) contexts such as an interrupt and a task
- Interactions between concurrent contexts for either communication or synchronization purposes (e.g., mailbox and semaphore mechanisms)

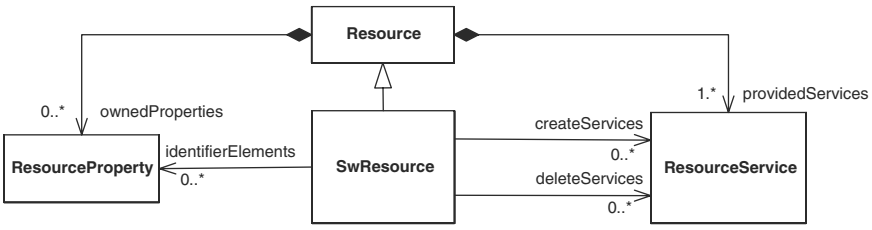


Fig. 12.2 An example of the "resource-service" modeling pattern

Table 12.1 Concurrency resources

Resource	Semantics
SchedulableResource	Encapsulated sequences of actions which execute concurrently.
MemoryPartition	Virtual address space.
InterruptResource	A computing context to execute user-delivered routines (i.e., entry point) connected to asynchronous signals.
Alarm	An executing context for a user routine, which must be connected to a timer.

Table 12.2 Interaction resources

Resource	Semantics
MessageComResource	Communication resource used to exchange messages.
SharedDataResource	Resource used to share the same area of memory among concurrent resources.
NotificationResource	Resource supporting control flow by notifying the occurrences of conditions to awaiting concurrent resources.
MutualExclusionResource	Resource that synchronizes access to shared variables.

Table 12.3 Brokering resources

Resource	Semantics
MemoryBroker	Resource that manages memory allocation, memory protection and memory access.
Scheduler	Resource that orchestrates the execution of multiple schedulable resources.
DeviceBroker	Resource that enables interfacing of hardware peripheral devices with the software execution support.

- Hardware and software resource brokering concepts, such as driver or memory management

12.3.2 SRM Profile Overview

Figure 12.3 provides on overview of the profile architecture resulting from the design of the previous SRM domain view in terms of UML extensions.

The SRM profile provides a broad range of modeling capabilities covering all RTOS concerns and with a low-level of details to enable generative approaches where models are used to generate parts of the application. Due to space limitations of this paper, it is out of the scope of the paper to describe in very details the SRM profile. Both next sections are therefore respectively dedicated to an overview of its typical modeling capabilities and its main expected use cases.

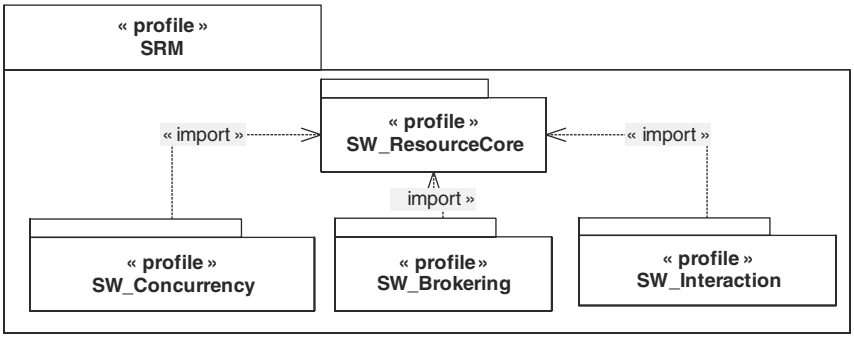


Fig. 12.3 SRM profile overview

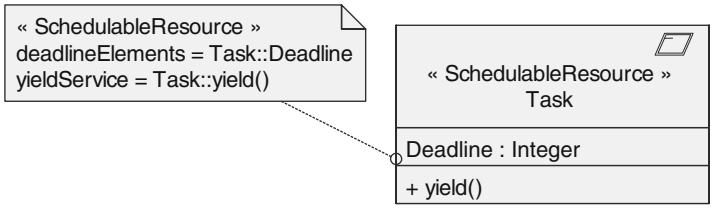


Fig. 12.4 Example of class extension

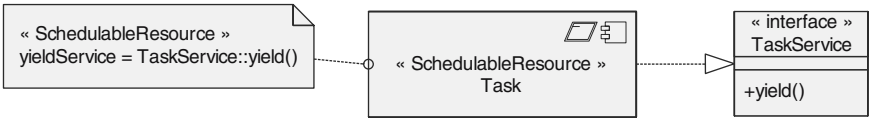


Fig. 12.5 Example of component extension

12.3.3 Modeling Examples

SRM concepts mainly extend the Classifier metaclass of UML (Fig. 12.7 shows a typical extension). Any UML "Classifier" submetaclass can thus be extended by these stereotypes (e.g., "Class", "Interface", "Component" and "AssociationClass"). Figures 12.4 and 12.5 illustrate the usage of "Class" and "Component" extensions. Figure 12.6 illustrates the use of an "AssociationClass" to describe interactions among concurrent resources. Since the "InteractionResource" stereotype extends the UML Classifier metaclass, an UML "AssocationClass" may be stereotyped as any "InteractionResource" substereotype (e.g., "NotificationResource", "Message ComResource", and "MutualExclusionResource"). In this example, the execution support provides concurrency resources ("Alarm" and "Task") to compute instructions.

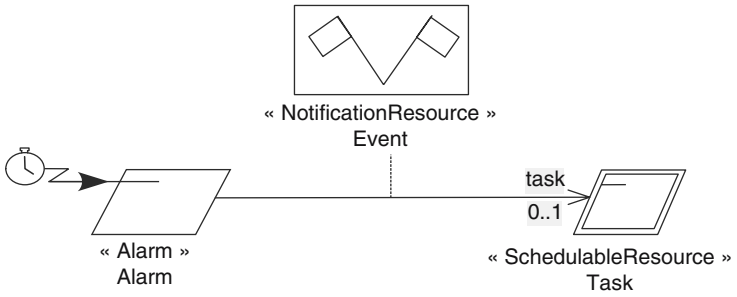


Fig. 12.6 Example of an association class

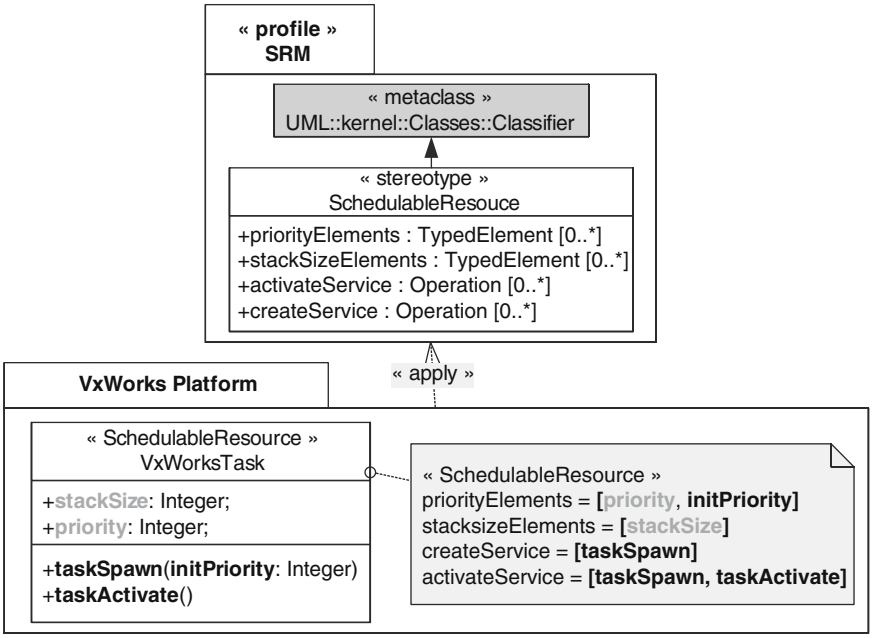


Fig. 12.7 Examples of tagged values

These resources are described as UML classes and respectively stereotyped as "Alarm" and "SchedulableRessource". In this example, an "Alarm" resource may interact with a "SchedulableResource" by mean of an event mechanism stereotyped "NotificationResource".

Since predefined instances are associated with one or more classifiers in the UML metamodel, platform providers must first define their classifiers. These classifiers should be stereotyped. This means that an extension of the UML "InstanceSpecification" metaclass is not mandatory.

Stereotype tags allow users to specify resource feature taxonomies. For example in Fig. 12.4, the "Deadline" property is referenced by the "DeadlineElements" stereotype property to clarify its taxonomy. It shows explicitly in the model that one of the attributes of this class plays the role of a deadline. This is the attribute named "Deadline". Such a modeling approach allows tools to distinguish properties and to permit automatic model transformations (e.g., code generation).

Note that there are no constraints on the tagged value owner. In the second part of Fig. 12.5, the "TaskService" interface owns a "yield" operation. This operation is tagged as a "yieldServices" via the "SchedulableResource" stereotype. But this stereotype is not applied to the interface, which means that, within the context of a "task", the service to call to release the computing resource is the "yield" operation of the "TaskService" interface.

Note also that both multiple tagged values for the same tag and multiple tags for the same feature are allowed. With this approach, the user can formally express multiple semantics for the same feature through multiple tags. Figure 12.6 describes a "taskSpawn()" service as both task creating and task activating services. In the same way, to activate a task, the user can either call the "taskSpawn()" service or the "taskActivate()" service. This also allows users to express the same semantics for multiple features through use of the same tag (see "priorityElements" in Fig. 12.7).

12.3.4 Main SRM Use Cases

Figure 12.8 shows the main use cases in which the SRM profile is likely to be involved. Potential key users of this description include "software designers" engaged in defining real-time system software architectures, "platform providers" who develop and sell real-time operating systems, and "methodology providers" who specify processes where platform modelling is important (e.g., an MDA Y-chart).

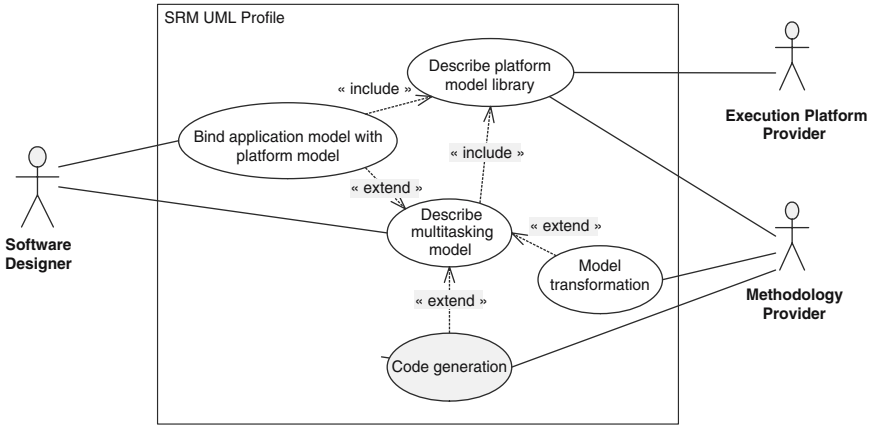


Fig. 12.8 Typical SRM use cases diagram

Figure 12.9 depicts a specific example of a robotic application built to run on the OSEK/VDX-OS RTOS [2]. This design is a basic robot motion controller. It is typical of the processes involved in RTES design. The example used here does not refer to a specific methodology, but is intended to illustrate the previously described SRM use cases.

In our example, the software designer describes the logical "RobotController" model. This model does not use the SRM profile in order to be independent of the target platform. Platform independence refers to the fact that a given design can be ported without change, from one platform to another.

The SRM profile is normally used to describe the platform model library, as is usually done by the platform provider. The platform model library includes resources and resource instances provided by the execution platform. For instance, the platform provider indicates that the "OSEK/VDX" RTOS provides a specific, structured type named "BasicTask". A UML class is used to show that the platform provider stereotypes that class as a "SchedulableResource" to indicate that this "BasicTask" concept owns the semantics of a concurrent execution resource managed by a specific scheduler. The platform provider also specifies that the integer attribute named "priority" is the priority property of that schedulable resource.

Finally, the application may be bound with the execution platform to produce a multitasking model. To do that, the application design may import the previous defined platform model library to instantiate predefined types and use predefined instances. Binding is described via a UML 2.0 dependency stereotyped with a specific stereotype. In the case of a schedulable resource, the stereotype "entryPoint"

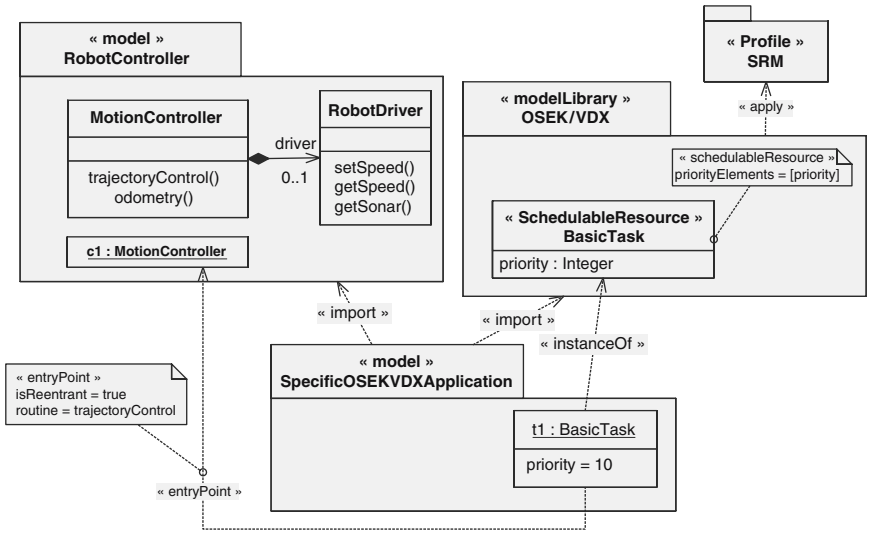


Fig. 12.9 An OSEK/VDX-OS design

is used to specify that the "trajectoryControl" operation body is the code which has to be executed in the context of that schedulable resource.

Based on this description, the methodology provider can define tools to automatically generate the OSEK/VDX configuration file described in OIL language [15]. For each UML "InstanceSpecification" whose classifier is stereotyped "SchedulableResource", an OIL Task declaration block can be generated. As the priority is explicitly referenced, tools are able to give the task priority value. Figure 12.10 is an example of the possible generated code.

Moreover, a software designer may wish to reuse and port a part of an application description to run on top of a new RTOS. Figure 12.11 illustrates such a use case with an ARINC-653 RTOS [3]. In this example, methodology provider tools must transform each UML "InstanceSpecification" of the OSEK/VDX "BasicTask" into a UML "InstanceSpecification" of an ARINC-653 process. The transformation

```
01.   OIL VERSION = "2.5" : "RobotController";
02.   …
03.   CPU cpu{
04.           TASK t1_trajectoryControl{
05.                   PRIORITY = 10;
06.           };
07.   }
08.   …
```

Fig. 12.10 Extract from an OSEK/VDX configuration file

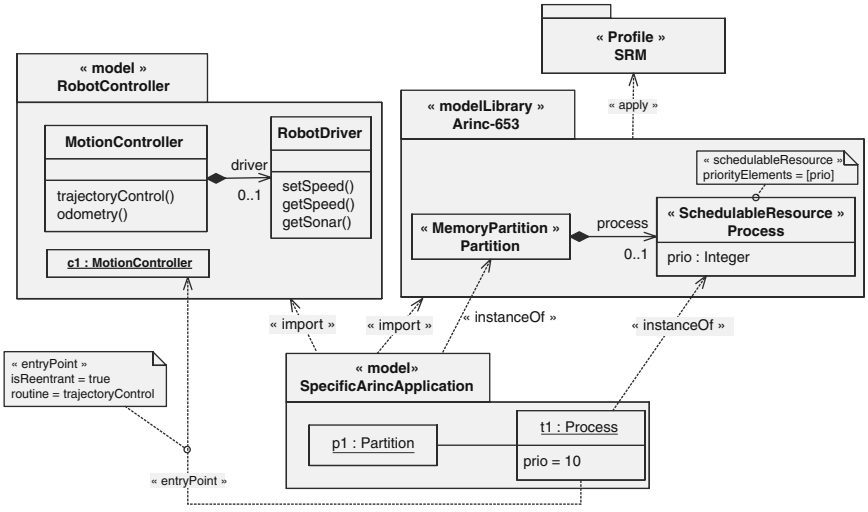


Fig. 12.11 An ARINC-653 design

pattern is generic, since both entities are described with the same "SchedulableResource" stereotype. The methodology provider can thus write a generic transformation rule to port an application from one platform to another: Each UML "InstanceSpecification" whose classifier is stereotyped "SchedulableResource" in the source RTOS must be transformed into a UML "InstanceSpecification" whose classifier is stereotyped "SchedulableResource" in the target RTOS. Such a rule can be easily written in any language for model transformation as for example ATL [16]. Tagged attributes and tagged operations can be transformed in the same way.

12.4 Conclusion and Future Work

In this paper, we have proposed a means for modeling software execution platforms with UML. This is done within the scope of providing model-driven processes that afford reusability, portability and maintainability of RTE applications. We have focused on an application built on an RTOS. We have thus sought to provide modeling artifacts for modeling existing standardized RTOS APIs, in order to be able to automatically produce code for interfacing the application with these APIs.

In this paper, we firstly defined the platform concept and investigated the state of the art related to UML-based platform modeling. This revealed that UML was lacking in certain means for describing efficiently and precisely the software execution platforms. We therefore concluded that more concrete concepts were required to enable automatic model transformations (e.g., through code generation). For this reason, we have proposed within MARTE a UML profile, called Software Resource Modeling (SRM). This profile provides both fine details and a broad range of modeling capabilities. It also provides artifacts that can be used to write generic model transformations. Such transformations can be used to generate code and assist for porting RTE applications to several multitasking platforms.

The main advantage of using the SRM profile is that this is not a new RTE API but instead a standard framework for modeling existing execution platform APIs. While the execution platforms discussed in this paper work are mainly RTOS, the SRM profile can also be used to describe APIs of other execution platforms such as RTE frameworks or virtual machines.

The SRM profile is part of the new UML profile for Modeling and Analysis of Real-Time Embedded systems (MARTE) adopted by the OMG consortium. Thus, the SRM profile is a standardized framework for describing RTE execution platforms.

Future research will deal with the transformations using the SRM profile. Efforts will thus focus on usage pattern description and behavior modeling for the purpose of obtaining an accurate description of the execution platform.

Acknowledgments The authors do thank J.P. Babau for its valuable contribution to the SRM profile.

References

1. The Open Group Base Specifications, Portable Operating System Interface (POSIX), ANSI/IEEE Std 1003.1, 2004
2. The OSEK/VDX Group, OSEK/VDX OS specification, Version 2.2.3, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, 2005
3. The Airlines electronic engineering committee, Avionics Application Software Standard Interface, ARINC Specification 653–1, Aeronautical radio, Inc., Annapolis, MD, October 2003
4. The Object Management Group, MDA guide version 1.1, http://www.omg.org/mda/, June 2003
5. The Object Management Group, UML 2.1.1 OCL 2nd revised submission, 2007, OMG document: ad/2007-02-03
6. The Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), RFP 2005, OMG document: realtime/05-02-06
7. B. Selic (2005), On Software Platforms, Their Modeling with UML2, and Platform-Independent Design, Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), IEEE Computer Society, Washington, DC, pp.15–21
8. A. Sangiovanni-Vincentelli, G. Martin (2001), Platform-based design and software design methodology for embedded systems, Design & Test of Computers, Volume 18, Number 6, November–December, IEEE Computer Society, Los Alamitos, CA, USA, pp.23–33
9. Y. Tanguy, S. Gérard, A. Radermacher, F. Terrier (2006), Model Driven Engineering for Real Time Embedded Systems, In 3rd European Congress Embedded Real Time Software (ERTS), Toulouse, France
10. The Object Management Group, UML Profile for Schedulability, Performance, and Time, Version 1.1, 2005. OMG document: formal/05-01-02
11. P. Kukkala, J. Riihimâki, M. Hämäläinen, K. Kronlöf (2005), UML 2.0 Profile for Embedded System Design, Automation and Test in Europe Conference (DATE 2005), pp.710–715
12. R. Chen, M. Sgroi, L. Lavagno, Grant Martin, A. Sangiovanni-Vincentelli, J. Rabaey (2003), UML and Platform-based Design, UML for Real: Design of Embedded Real-Time Systems, Kluwer, Norwell, MA, USA, pp.107–126
13. VxWorks 5.5 Documentation Page, http://www.windriver.com
14. RTAI 3.1 Documentation Page, http://www.rtai.org/
15. The OSEK/VDX Group, OIL specification Version 2.5, http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, 2004
16. F. Jouault and I. Kurtev (2005), Transforming Models with ATL, Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica

Chapter 13
Model Transformations from a Data Parallel Formalism Towards Synchronous Languages

Huafeng Yu[1], Abdoulaye Gamatié[2], Eric Rutten[3], and Jean-Luc Dekeyser[4]

Abstract The increasing complexity of embedded system designs calls for highlevel specification formalisms and for automated transformations towards lowerlevel descriptions. In this paper, a metamodel and a transformation chain are defined from a high-level modeling framework, Gaspard, for data-parallel systems towards a formalism of synchronous equations. These equations are translated in synchronous data-flow languages, such as Lustre, which provide designers with formal techniques and tools for validation. In order to benefit from the methodological advantages of re-usability and platform-independence, a Model-Driven Engineering approach is applied.

Keywords MDE, model transformation, Gaspard, synchronous languages, embedded system

13.1 Context and Motivation

13.1.1 MDE and Data-Parallel Applications

Data-parallel applications, such as mobile multimedia processing, high-definition TV and radar/sonar signal processing, play an increasingly important role in embedded systems. These applications generally concern computations on multidimensional data structures. But these applications become more and more complex. As a result, their design and validation turn out to be dramatically complicated. Furthermore, the productivity problem is a great constraint for the development of these applications. More efficient modeling and design methods are highly needed.

[1]INRIA Futurs Lille-LIFL, France Email: huafeng.yu@inria.fr

[2]LIFL-CNRS (UMR 8022), France Email: abdoulaye.gamatie@lifl.fr

[3]INRIA Rhône-Alpes, France Email: eric.rutten@inrialpes.fr

[4]LIFL-USTL (UMR 8022), France Email: jean-luc.dekeyser@lifl.fr

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Nowadays, among intensive research activities to address such problems, Model-Driven Engineering (MDE) [17] based methods can be mentioned. Well-defined modeling specifications lead to rapid design as well as concise and clear documentation, and their automated transformations enable to generate Platform-Specific Models (PSM) and even executable code conveniently. The re-usability and modularity of their models, Intellectual Properties (IPs) and the hierarchical modeling make the production of these applications more efficient and rapid.

13.1.2 The GASPARD Methodology for Data-Parallel Computing

GASPARD [15] is a MDE based development environment and methodology for data-parallel applications. It proposes concepts, which feature high level data parallelism, data flow and control flow mixing, hierarchical and repetitive application and architecture modeling, etc. The inherent data-parallel formalism of GASPARD is adopted by MARTE (Modeling and Analysis of Real-Time and Embedded systems) [16], which is an OMG (Object Management Group) standard for the modeling and analysis of real-time embedded systems. GASPARD concerns software/hardware co-modeling and model transformations. More precisely, it enables to model *software applications*, *hardware architectures*, their *association* and IP *deployment* through a predefined metamodel in a unique modeling environment. This modeling stays at a high abstraction level and is platform independent. GASPARD enables as well transformations from these models to PSM models.

GASPARD metamodel is partially based on the concept of the Y-chart (see Fig. 13.1 and [15]). Models for application and hardware architecture are defined separately. Then, application models can be mapped on architecture models. The obtained models are associated with software or hardware IPs during the deployment. All these models are platform-independent, and in general they are not associated with particular technologies, but they can still be associated with an execution, simulation, validation or synthesis technology. Model transformations are performed from deployed models to specific languages (synchronous languages and others, which are not detailed here and are shadowed in the Fig. 13.1, such as FORTRAN, SYSTEMC and VHDL). These characteristics of GASPARD help to reduce the system design complexity.

In the following, we briefly present main features of the high-level metamodel of GASPARD.

- **Application** focuses on the description of data dependencies between the application components. These components and dependencies completely describe the functional behaviour with potential data-parallelism.
- **Architecture** specifies the hardware architecture at a high abstraction level. It enables to dimension hardware resources in the same way as in application.
- **Association** allows one to express how the application is projected on the hardware architecture with the consideration of task and data parallelism.

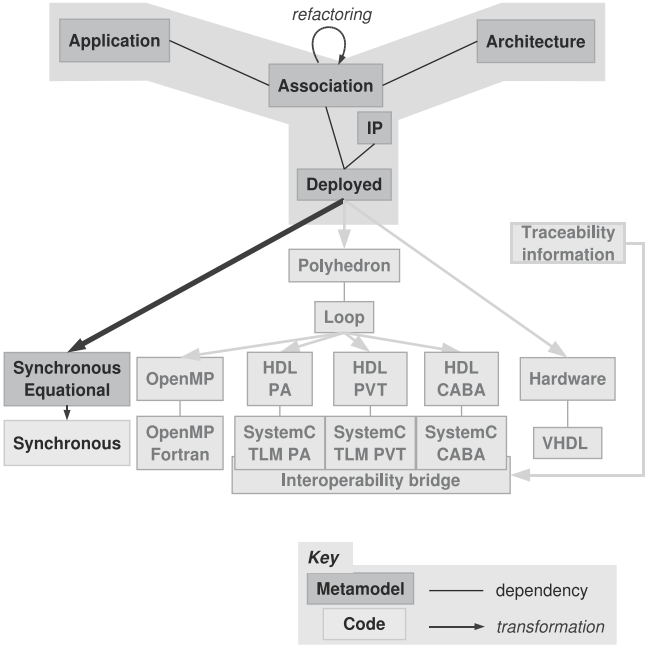


Fig. 13.1 Y-chart according to GASPARD

- **Deployment** (represented by the box tagged as "Deployed" in Fig. 13.1) enables to chose a specific target platform for code generation from GASPARD models. This is achieved by importing IPs.

13.1.3 Motivation: Connecting GASPARD to Validation Tools

The GASPARD methodology, dedicated to the data-parallel application design, adopts a top-down approach, which goes from the high abstraction level to low implementation levels. Moreover, the correctness of the design and implementation is highly required as well. However, GASPARD UML models are limited by formal semantics, which is necessary for the formal validation of the system design. Hence a map from these models on formal methods is needed. Synchronous languages are well known for their formal aspects and their richness in terms of tools for validation, verification and automatic code generation. Therefore, the connection of these two technologies is encouraged because it offers the opportunity to benefit from the capability of GASPARD in the specification of data-parallelism and also from the power of formal aspects of synchronous languages. This paper presents how MDE transformations contribute to bridge these different abstraction levels from GASPARD

to synchronous languages (LUSTRE [9] is considered here for illustration). Furthermore, the automated transformation reduces potential error occurrences caused by the manual translation. Previous works ([1, 5]) have exploited the simulation of GASPARD specifications in PTOLEMY II and also their projections into Kahn process network for the distributed execution, but they were not implemented with the MDE approach and did not aim at the formal validation and verification.

13.2 Data Parallelism and Synchronous Approach

13.2.1 Data-Parallel Application Design: GASPARD

This paper only addresses software application modeling and its deployment. Basic application models of GASPARD [3] can be summarized by the following grammar:

Task	*::= < Interface, Body >*	*(r1)*
Interface	*::= < in : {Port}, out : {Port} >*	*(r2)*
Port	*::= < type, size >*	*(r3)*
Body	*::= $Task^h$ | $Task^r$ | $Task^e$*	*(r4)*
$Task^e$	*::= < some function call >*	*(r5)*
$Task^r$	*::= < {Tiler}, (r, Task), {Tiler} >*	*(r6)*
Tiler	*::= < F, o, P >*	*(r7)*
$Task^h$	*::= < {Task}, {(Task, array, Task)} >*	*(r8)*

All GASPARD tasks share a few common features (rule (*r*1)): an *interface* (rule (*r*2) where { } denotes a set) that specifies input/output ports (typed by `in or out` in rule (*r*2) and defined in rule (*r*3)) from which each task respectively receives and produces multidimensional arrays; and a *body* (rule (*r*4)), which depends on the type of task as follows:

- *Elementary task* (rule (*r*5)). The body corresponds to an atomic computation block. Typically, it consists of a function or an IP.
- *Repetitive task* (rule (*r*6)). It expresses the data-parallelism in a task. The *instances* of the associated repeated task are assumed to be independent and schedulable following any order, even in parallel. The attribute **r** (in the rule (*r*6)) denotes the *repetition space*, which indicates the number of repetitions. It is defined itself as a multidimensional array with a shape. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops [3]. In addition, each task instance consumes and produces sub-arrays with the same shape. These sub-arrays are referred to as *patterns*. The way patterns are constructed is defined via *tilers* (rule (*r*7)), which are associated with each array. A tiler extracts (resp. stores) patterns

from (resp. in) an array based on certain information it contains: ***F***: a *fitting* matrix (how array elements fill patterns); **o**: the *origin* of the *reference pattern*; and ***P***: a *paving* matrix (how patterns cover arrays).
- *Hierarchical task* (rule (*r*8)). It is represented by a hierarchical acyclic graph in which each node consists of a task, and edges are labeled by arrays exchanged between task nodes.

An *application* is a hierarchical task in which the top-level of the hierarchy is composed of a single task, which plays a similar role to "main" in a C program.

The GASPARD application metamodel is defined according to the above basic concepts. The whole software application is modeled as an *ApplicationModel*, in which *ApplicationComponent*s model hierarchical tasks (see detailed examples in [18]). Instances of other *ApplicationComponent*, called *ApplicationComponentInstance*, can be composed in it. These instances have *PortInstance*s. *Connector*s are used to connect *PortInstance*s and/or *Port*s. Internal structures, such as *Elementary*, *Compound* and *Repetitive*, are defined in an *ApplicationComponent* according to its inside component instances.

- *Elementary* points out that the *ApplicationComponent* is an elementary task, which is a black box in Gaspard.
- *Repetitive* indicates that the *ApplicationComponent* is a repetitive task. The *Connector*s which connect *ApplicationComponent*'s ports and *PortInstance*s of its internal instance are *Tiler*s.
- *Compound* corresponds to a hierarchical task and expresses task parallelism. All the *ComponentInstance*s inside this component run in parallel.

13.2.2 The Synchronous Approach

The synchronous approach [2] proposes formal concepts that favor the trusted design of embedded systems. Its basic assumption is that computation and communication are instantaneous, referred to as *synchrony hypothesis*. There are different synchronous languages, which have strong mathematical foundations, such as LUSTRE, LUCID SYNCHRONE and SIGNAL. These languages are well-adapted for data-flow-oriented applications. All these languages are associated with tool-sets that have been successfully used in several critical domains (e.g. avionics, automotive, nuclear power plants).

In this paper, LUSTRE is taken as the example (see a segment of LUSTRE code in Fig. 13.2) for the introduction of some basic concepts in synchronous languages. A *node* is a basic functionality unit in LUSTRE. Each node gives the same results with the same inputs thanks to its determinism. Nodes have modular declarations that enable their reuse. Each node has an *interface* (input at line `(l1)` and output at `(l2)`), local definition `(l3)`, and *equation*s (line `(l5)` and `(l6)`). Variables are called *signal*s in LUSTRE. Equations are signal assignments. Furthermore only unique assignments are allowed for signals. In these equations, there are possibly node invocations `(l5)` that are declared outside this node. Obviously, in LUSTRE,

```
   node node_name (A1:int^4)          (l1)
returns(A3:int^4);                    (l2)
var A2:int^4;                         (l3)
let                                   (l4)
 A2 = a_function(A1);                 (l5)
 A3 = A1+A2;                          (l6)
tel                                   (l7)
```

Fig. 13.2 An example of LUSTRE code

modularity and hierarchy are inbuilt. The composition of these equations, denoted by ";", means their parallel execution w.r.t. data-dependencies. The node has the same meaning independent of the equation order.

13.3 A Synchronous Equational Metamodel

The metamodel proposed here aims at three synchronous data-flow languages at the same time. These languages have considerable common aspects, which enable their code generation with the help of only one metamodel. In addition, because of the obvious differences between GASPARD and synchronous languages, an intermediate model is necessary to bridge the gap between them as well. A synchronous model is therefore proposed, which follows the synchronous modeling of data-intensive applications [8]. It aims to be generic enough to target the synchronous data-flow languages mentioned earlier and to be adequate to express data-parallel applications. So, it is not intended to have exactly the same expressivity as these languages. But this is not the case of the SIGNALMETA metamodel [4], which is specifically dedicated to the SIGNAL language. This metamodel completely defines all programming concepts in SIGNAL. It has been specified in the *Generic Modeling Environment* (GME), developed at Vanderbilt University.

13.3.1 Signal

In the proposed metamodel, all input, output or local variables are called `Signals` (see Fig. 13.3). Each `Signal` is associated with a `SignalDeclaration`, which declares the name and type of the `Signal`. It is associated with at least one `SignalUsage`. The latter represents one operation on `Signal`. If the `Signal` is an array, a `SignalUsage` can be an operation on a pattern of this array. Hence, if the array has several patterns, the `Signal` is associated to the same number of `SignalUsage` correspondingly. Each of these `SignalUsages` has an `IndexValueSet`, which is a set of `IndexValue` of the associated `Signal`. A `SignalUsage` is associated with at least one `Argument` of equations, which indicates their inputs/outputs.

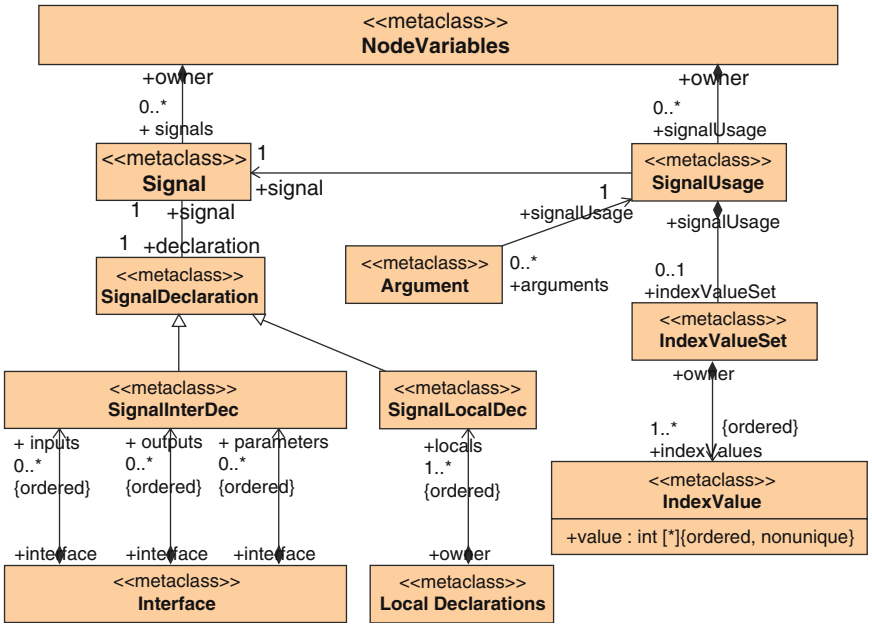


Fig. 13.3 Extract of the synchronous metamodel: `Signal`

13.3.2 Equation

`Equations` (Fig. 13.4) indicate the relations between their inputs and outputs, which are called `Arguments` here. An `Equation` has an `EquationRightPart` and at most one `EquationLeftPart`. The latter has `Arguments` as `Equation` outputs. `EquationRightPart` is either an `Array Assignment` or an `Invocation`. Array Assignment has `Arguments` and indicates that the `Equation` is an array assignment. `Invocation` is a call to another `Node` (see the following subsection `Node`). In an `Invocation`, `Function-Identifier` indicates the called function.

13.3.3 Node

Synchronous functionalities are modeled as `Nodes` (see Fig. 13.4). A `Node` has no more than one `Interface`, `LocalDeclaration`, `NodeVariables`, an `EquationSystem` and some `Implementations` and `CodeFiles`. `NodeVariables` is the container of `Signals` and `SignalUsages`. Each input/output `Signal` is associated with a `SignalDeclaration`, which belongs to the `Interface`, while local `Signals' SignalDeclarations` belong to

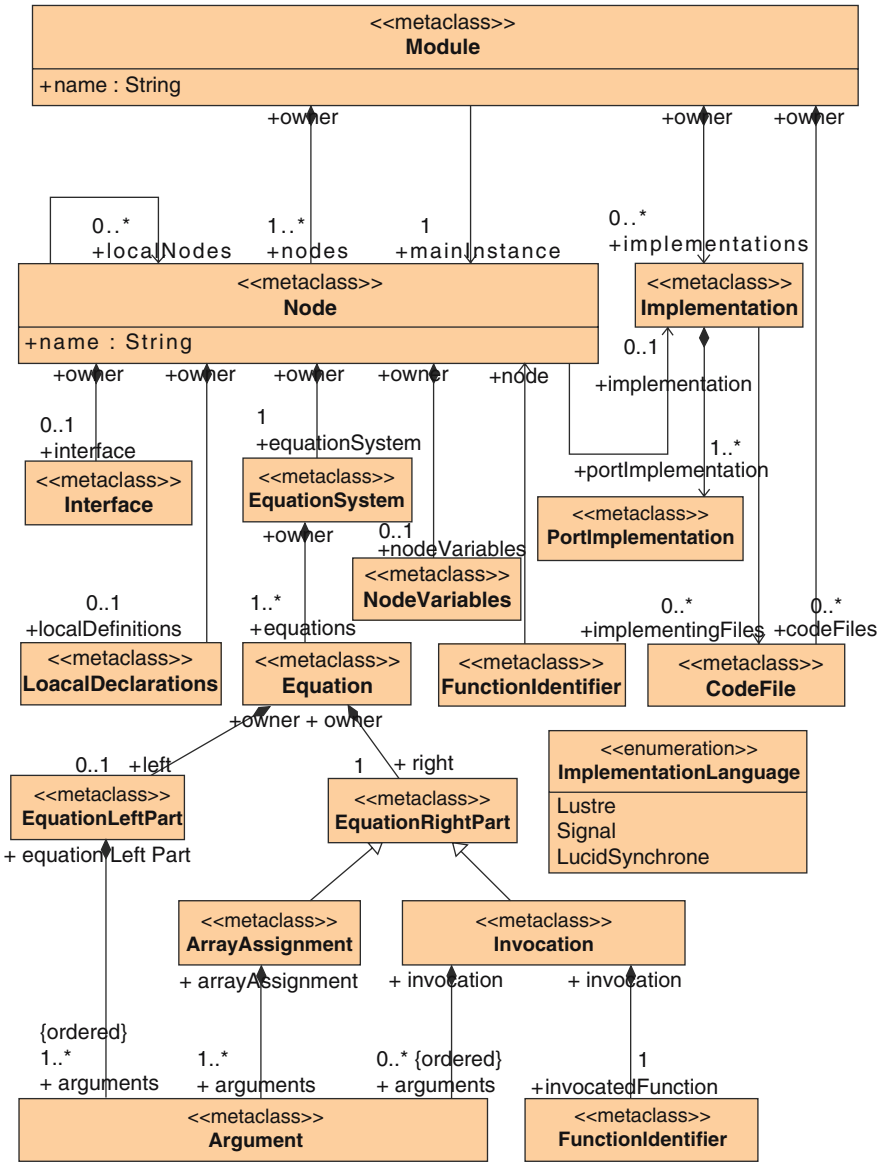


Fig. 13.4 Extract of the synchronous metamodel: `Node`, `Equation`

`LocalDeclaration`. `EquationSystem` is the node body that fulfills the functionality through a composition of at least one synchronous `Equation`. All `Nodes` are grouped in a `Module`, which represents the whole application. It contains one `Node` as the main `Node` of the application. Each `Node` is either defined in the `Module` or linked to an external function through IP deployment.

13.3.4 Remarks

`Nodes` that are not defined in the `Module` should be deployed. The equivalents of these nodes are GASPARD elementary tasks. An `Implementation` associated with a `Node` contains the information of the external function. Parameters of external function are represented by `PortImplementations`. Their orders are defined in the `Implementation` so that parameters are passed correctly to the application. An `Implementation` is associated with at least one `CodeFile`, which represents the implementation of the external function.

Synchronous models, which conform to this metamodel, act as intermediate models between data-parallel applications and data-flow languages. Their parallel compositions preserve the parallelism, and their modularity and re-usability ensure hierarchical compositions of original GASPARD models. This modeling is also generic enough so that it will not suffer from the complexity and particularity of target languages. Moreover it enables potential improvements, for instance, the integration of application control inspired by [13].

13.4 Model Transformations

Only GASPARD models with the infinite dimension at the highest hierarchy can be transformed into synchronous models. The infinite dimension is translated by a logical time in the reactive style of synchronous languages. So in synchronous models, there are no more infinite dimensions. The multidimensional arrays are translated into array-type signals. Parallelism in GASPARD can be easily modeled in synchronous models with the help of the composition operator defined in synchronous languages.

Transformations of GASPARD models into synchronous specifications (typically, LUSTRE programs) consist of two steps: firstly, a transformation of GASPARD models into synchronous models; and then, the generation of synchronous code from synchronous models obtained from the first step.

13.4.1 From GASPARD Models to Synchronous Models

Some basic transformations are first given. `Components`and `ComponentInstances` are transformed into `Nodes` and `Equations` respectively. `Ports`, `PortInstances` and `DefaultLink` connectors in a `Component` are transformed into `Signals`, whereas `Tiler` connectors are transformed into Equations as well as `Nodes`.

13.4.1.1 Transformation Rules

All the rules can be represented through a tree structure (see Fig. 13.5). The unique initial (root) rule is *GModel2SModel*. It transforms a whole deployed GASPARD application into a synchronous module. This rule then calls its sub-rules:

GTiler2SNode, *GApplication2SNode*, *GACI2SNode*, etc. *GApplication2SNode* has also three sub-rules: *GRepetitive2SEquationSystem*, *GCompound2SEquationSystem* and *GElementary2SEquationSystem*. Note that not all rules are given in the Fig. 13.5 due to lack of space (see [18] for details). In the following, only rules presented in the Fig. 13.5 are described. Among them, *GTiler2SNode* and *GRepetitive2SEquationSystem* are a little more detailed. The other rules are constructed in the same way.

- *GTiler2SNode* (see Fig. 13.6 in which each element is numbered). It is a rule for the transformation of `tiler` connectors into synchronous input or output tiler `Nodes`. An input tiler `Node` is taken as an example for the construction of a synchronous node. First of all, the `Node` (numbered 1) is created and is associated with its `Module`. The `Port` and `PortInstance` connected by this tiler are then transformed into input and output `Signals` respectively. One `Port` corresponds to one input signal, and one `PortInstance` corresponds to several output signals, whose quantity, *n*, is calculated from the repetition space defined in its connected `ComponentInstance`. The input signal is associated with *n* `SignalUsages` (4) and an output signal is associated with a `SignalUsage` (8). `Interface` (2) is then created and associated with `SignalDeclarations` (3, 9) that are associated with signals. Note that there are no `LocalDeclarations` in this node. Next, an `EquationSystem` contains *n* `Equations` (5). In each `Equation`, the `EquationLeftPart` has an `Argument` (6) which is associated with a `SignalUsage` of an input signal. `EquationRightPart` is directly an `ArrayAssignment`. Its `Argument` (7) is associated with a `SignalUsage` (8) of a corresponding output.

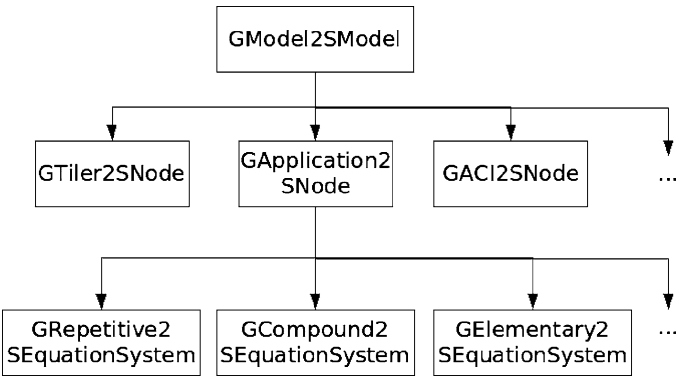


Fig. 13.5 Hierarchy of the transformation rules

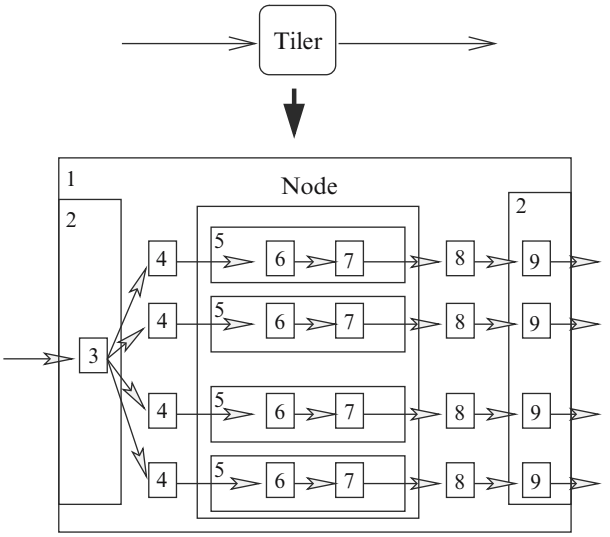


Fig. 13.6 Transformation of the tiler

- *GApplication2SNode*. It transforms application components into `Nodes`. However, all the elements in these `Nodes` are generated by its three sub-rules, which transform internal structures in the `Component` into an `EquationSystem`.
- *GACI2SNode*. It transforms the unique main `ApplicationComponentInstances` into a synchronous `Node`. It is the main instance of the application.
- *GRepetitive2SEquationSystem*. (Fig. 13.7) In this rule, an `EquationSystem` is first created. And then three types of `Equation` are created: input tiler `Equations`, repeated task `Equation` and output tiler `Equations`. `Tiler` connectors are transformed into input/output tiler `Equations`, which are invocations to `Nodes` generated by *GTiler2SNode*, and the internal `ComponentInstance` is transformed into repeated task `Equation`. A relevant repeated task `Node` is then created, in which *n* equations invoke the task node corresponding to the component that declares the internal component instance. Note that hierarchical composition in GASPARD models is preserved in synchronous models by node invocations.
- *GCompound2SEquationSystem*. Each internal `ComponentInstance` is transformed into an equation. Connectors between these `ComponentInstances` are transformed into local `Signals`.
- *GElementary2SEquationSystem*. No `Equation` is created because its owner `Node` is implemented externally and `Deployment` models are used to import its external declarations. However an `Interface` is created according to the component's ports.

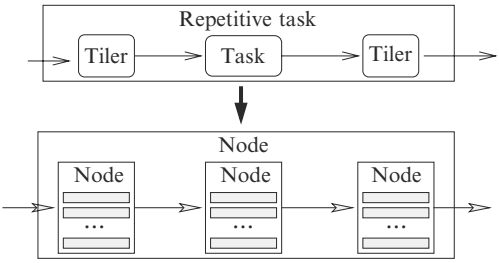


Fig. 13.7 Transformation of the repetitive task

13.4.1.2 Implementation of the Transformation Chain

GASPARD models are specified in the graphical environment MAGICDRAW, and are exported as ECLIPSE Modeling Framework (EMF) [6] models. EMF is a modeling framework and code generation facility. In the following transformation phase, these models are transformed into EMF GASPARD models. These two previous transformations will not be detailed here. Then EMF GASPARD models are transformed into EMF synchronous equational models, which are finally used to generate synchronous language code (e.g. LUSTRE code). An automated model transformation chain is then defined through the concatenation of these transformations from MAGICDRAW UML models to data-flow languages (Fig. 13.8).

These transformations were implemented with the help of specifications, standards and transformation languages. Some of them are briefly presented in this paper. MOF QVT [14] is the OMG standard on model query and transformation, which is respected in transformations presented here. Several other transformation languages and tools, such as ATL [10] and KERMETA [12] already exist. ATL is a model transformation language (a mixed style of declarative and imperative constructions) designed w.r.t. QVT. KERMETA is a metaprogramming environment based on an object-oriented Domain Specific Language. But these two languages lack of extension capability especially when some external functions are needed to be integrated into the transformation. EMFT (Eclipse Modeling Framework Technology) project was initiated to develop new technologies that extend or complement EMF. Its query component offers capabilities to specify and execute queries against EMF model elements and their contents. The MOMOTE tool (MOdel to MOdel Transformation Engine), which is based on the EMFT QUERY and is integrated into GASPARD, is a JAVA framework that allows to perform model to model transformations. It is composed of an API and an engine. It takes input models that conform to some metamodels and produces output models that conform to other metamodels. A transformation by MOMOTE is composed of rules that may call sub-rules. These rules are integrated into an ECLIPSE plugin. In general, one plugin corresponds to one transformation. During model transformations, these plugins are automatically invoked one by one.

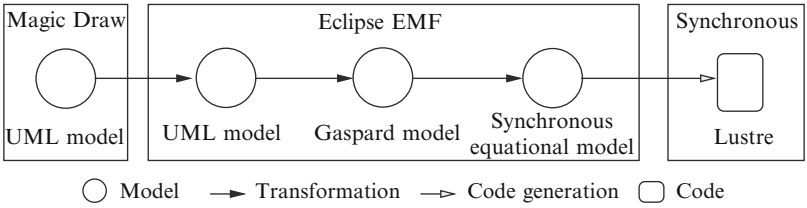


Fig. 13.8 The detailed transformation chain

13.4.2 Synchronous Code Generation from Models

The implemented code generation from synchronous models is template based. EMF JET [7] and MOCODE are used to build code generators (three generators for three data-flow languages). JET is a template based code generation tool. User defined templates in JET are used to generate JAVA implementation classes. Then, the latter can be called to generate target code. MOCODE (MOdels to CODe Engine), which works with JET, is also a tool integrated into GASPARD. It consists of an API with an engine that enables to perform model to text transformation. It takes a set of models as inputs, and then its engine recursively takes out elements from input models and executes a corresponding JAVA implementation class on them. These JAVA classes finally generate target code.

13.5 An Application Example

Examples of matrix processing, which averages the patterns from inputs, are intuitive, but they are typical to show the transformation and the application domain. One of the examples implemented is illustrated in Fig. 13.9, which takes a flow of (4, 4)-array, and produces a flow of (2, 2)-array. For each step in the flow, the average computing block has four repetitions, each of which takes a (2, 2)-sub-arrays from the input array, then carries out the computing, and produces a (1, 1)-sub-array. Finally all of the (1, 1)-arrays from the four repetitions then construct the output (2, 2)-array of the application.

The deployment of the matrix average IP (`TASK`) is illustrated in Fig. 13.10. This deployment indicates where to find the LUSTRE code that implements this IP. The physical LUSTRE code is represented by the `CodeFile`, and it is associated to the elementary task by the component `AbstractSoftwareImplementation`, which is composed of at least one `SoftwareImplementations`. This means one elementary task may have several different implementations (in different languages or through different algorithms). The `SoftwareImplementation` contains the deployment information, for example, the elementary function name,

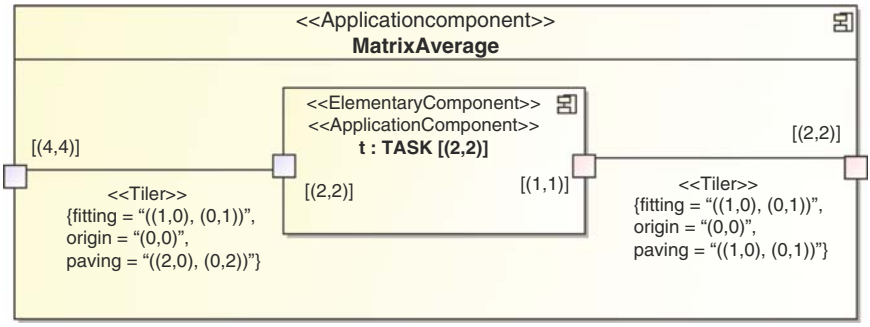


Fig. 13.9 An example of matrix average computation

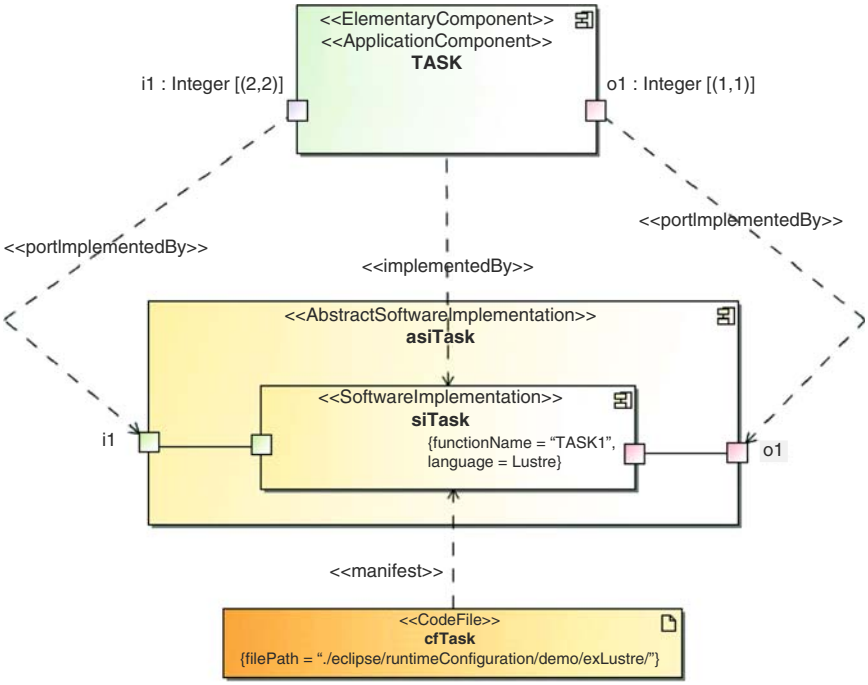


Fig. 13.10 The deployment of the matrix average IP

the language of its implementation. Other deployment information, such as ports, etc., can be found through the references (`portImplementedBy`, `implementedBy`) between `AbstractSoftwareImplementation` and `ElementaryComponent`.

The transformation chain and the generated code is illustrated in the video located in [11]. The extract of the generated code can also be found in [18].

13.6 Conclusions and Perspectives

In this paper, we proposed a synchronous metamodel and presented model transformations from data-intensive applications specified in GASPARD into synchronous languages, particularly the LUSTRE language, through a MDE approach. The code in java and rules of the implemented transformations adds up to about 5,000 lines in ECLIPSE.

Some illustrative examples of the transformation have been implemented. Due to space problem, only one is showed in this paper. Other more complicated examples can be found in [18].

Simulation and validation issues are also addressed with the generated code. Functional simulation and verification of deadlock absence on the original design have been carried out. Whereas the synchronizability analysis requires the introduction of clocks in GASPARD. One of the future work concerns the integration of control (inspired by [13]) in GASPARD models and their transformation into synchronous languages for automatic verification. More analysis details can be found in [8, 18]. Finally, the way all these analysis results can be exploited by GASPARD users is a challenging perspective from a practical point of view.

References

1. Amar, A., Boulet, P., Dumont, P.: Projection of the ARRAY-OL specification language onto the kahn process network computation model. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, NV (2005)
2. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. In: Proceedings of the IEEE 91(1), 64–83 (2003)
3. Boulet, P.: ARRAY-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, http://hal.inria.fr/inria-00128840/en/ (2007)
4. Brunette, C., Talpin, J.-P., Besnard, L., Gauthier, T.: Modeling multi-clocked data-flow programs using the generic modeling environment. In: Synchronous Languages, Applications, and Programming. Elsevier, Vienna Austria, (2006)
5. Dumont, P., Boulet, P.: Another multidimensional synchronous dataflow: Simulating ARRAY-OL in PTOLEMY II. Tech. Rep. 5516, INRIA, www.inria.fr/rrrt/rr-5516.html (2005)
6. Eclipse: Eclipse Modeling Framework. http://www.eclipse.org/emf
7. Eclipse: EMFT JET. http://www.eclipse.org/emft/projects/jet
8. Gamatié, A., Rutten, E., Yu, H., Boulet, P., Dekeyser, J.L.: Synchronous modeling of data intensive applications. Research Rep. 5876, INRIA. http://hal.inria.fr/inria-00001216/en (2006)
9. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE 79(9) (1991)
10. INRIA Atlas Project: ATL. http://www.sciences.univ-nantes.fr/lina/atl/
11. INRIA DaRT Project: Presentations and demonstrations: GASPARD2 towards LUSTRE. http://www2.lifl.fr/west/DaRTShortPresentations
12. INRIA Triskell Project: KERMETA. http://www.kermeta.org/
13. Labbani, O., Dekeyser, J.L., Boulet, P., Rutten, E.: Advances in Design and Specification Languages for SoCs, Selected contributions from FDL'06, chap. UML2 Profile for Modeling Controlled Data Parallel Applications. Springer, TU Darmstadt, Germany (2007)

14. Object Management Group (OMG): MOF Query/Views/Transformations (QVT). http://www.omg.org/cgibin/doc?ptc/2005-11-01 (2005)
15. INRIA DaRT Project: GASPARD. http://www.lifl.fr/west/gaspard/
16. Rioux, L., Saunier, T., Gerard, S., Radermacher, A., de Simone, R., Gautier, T., Sorel, Y., Forget, J., Dekeyser, J.L., Cuccuru, A., Dumoulin, C., André, C.: MARTE: A new profile rfp for the modeling and analysis of real-time embedded systems. In: UML-SoC'05, DAC 2005 Workshop UML for SoC Design. Anaheim, CA (2005)
17. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2) (2006)
18. Yu, H., Gamatié, A., Rutten, E., Dekeyser, J.L.: Model transformations from a data parallel formalism towards synchronous languages. Research Report 6291, INRIA. http://hal.inria.fr/inria-00172302/en/ (2007)

Chapter 14
UML and SystemC – A Comparison and Mapping Rules for Automatic Code Generation

Per Andersson and Martin Höst

Abstract Today embedded system development is a complex task. To aid the engineers new methodologies and languages are emerging. During the development the system is modeled using different tools and languages. Transformations between the models are traditionally done manually. We investigate the automation of this process, specifically we are looking at automatic UML to SystemC transformation. In this paper we compare UML and SystemC, focusing on communication modeling. We also present mapping rules for automatic SystemC code generation from UML. The mapping has been implemented in our UML to SystemC code generator.

Keywords code generation, UML, systemC

14.1 Introduction

Today there is a never ending demand for new functionality to be included in embedded systems such as mobile phones. This leads to increased design complexity. To overcome the increased system complexity new design methodologies, such as model driven architecture, have been introduced. In parallel with this, new languages, i.e. SystemC [2, 3], for system level modeling and simulation have also emerged. Combining new methodologies and new languages is a promising approach to manage the increasing system complexity. This is the focus of the MARTES (Model-Based Approach for Real-Time Embedded Systems development) project.[1] In the project we investigate how UML and SystemC can be used together when the ideas of Model Driven Architecture are applied. One of the tasks of the project is to investigate how transformations from UML to SystemC can be

Lund University, P.O. Box 118, SE-221 00 Lund, Sweden
Email: Per.Andersson@cs.lth.se and Martin.Host@cs.lth.se

[1] www.MARTES-ITEA.org

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

automated and supported by tools. During this research we are developing a prototype tool, which manage the UML to SystemC transformations and code generation, as an add-in to the Telelogic TAU UML2 modeling tool.[2] This part of the MARTES project is carried out in close cooperation between Lund University and Telelogic. In this research there are a number of decisions that needs to be taken related to the detailed requirements on the developed tool. It is crucial to take the right decisions concerning what functionality to include in the tool. This is achieved by developing the tool iteratively. Different versions are developed after each other, and every version is evaluated in order to decide what additional functionality to include in the next version. Evaluations are a very important part of the development of the tool. The evaluations are being carried out together with other partners in the MARTES project, in the context of case studies in industrial projects. In this paper we present the work and results from developing the first version of our UML to SystemC code generator. We start with a summary of related work in Section 14.2. We compare the constructs and semantics of UML and SystemC in Section 14.3. Based on this comparison we have developed a set of mapping rules which are detailed and motivated in Section 14.4. Our implementation of the mapping rules is presented in Section 14.5 and practical experience can be found in Section 14.6. Finally the paper is concluded in Section 14.7.

14.2 Related Work

Earlier publications on UML to SystemC mapping [4, 7] suggest that, to a large extent, there is a one to one relation between concepts in the two languages. For example a UML class is mapped to a SystemC module. This is not always desirable, sometimes UML classes are used for data encapsulation and in these cases they should remain as classes in the SystemC model. Only UML classes with ports, and/or with architecture should be mapped to SystemC modules.

Riccobene et al. [7] address this by exposing all SystemC details in the UML model through a SystemC profile. Their approach is to use UML as an implementation language for SystemC. In addition to making all standard SystemC types available at the UML level they also extend actions in state machines to handle SC_THREAD and SC_METHOD synchronization. With their approach, the engineers must tag their UML model by adding relations to the intended SystemC elements. This is similar to the last part in our design process. In our design process engineers start with an abstract UML model, which is refined in three steps. This is further explained in Section 14.5. One difference compared to their work is that we intend to automate most of this part in our process, minimizing the design effort. Another problem with bringing too many of the SystemC details into the UML model is the semantic differences between the languages, as discussed in Section

[2] www.telelogic.com

14.3. This will lead to problems during co-simulation of pure UML models with models applying the SystemC profile. It will also be problematic to generate code for different targets, i.e. hardware and software. As far as we know no one has published a semantic comparison of these two languages.

14.3 Language Comparison

In this section we compare the UML and SystemC languages. The comparison is based on UML 2 [1, 6] and SystemC 2.2 [2, 3]. The purpose of the comparison is to find and motivate mapping rules for automatic SystemC code generation from UML. The focus is on concepts which are equivalent in both languages as well as concepts and constraints which are only present in one of the two languages. When we refer to concepts which are similar in both languages we use the notation UML name/SystemC name, for example class/module. Also we refer to a class which inherits from sc_module as a SystemC module and any class inheriting from sc_interface as a SystemC interface.

14.4 Composition

UML and SystemC are similar from a structural point of view. Both languages have the concepts of package/name space which can be used to group most other language constructs. In real models packages/name spaces are mainly used to group declarations of classes/modules. A package/name space cannot be instantiated. Any instantiations done in a package/name space will result in one instance in the system, with limited visibility to the package/name space. In this paper we focus the discussion around the small system shown in Figs. 14.1 and 14.2.

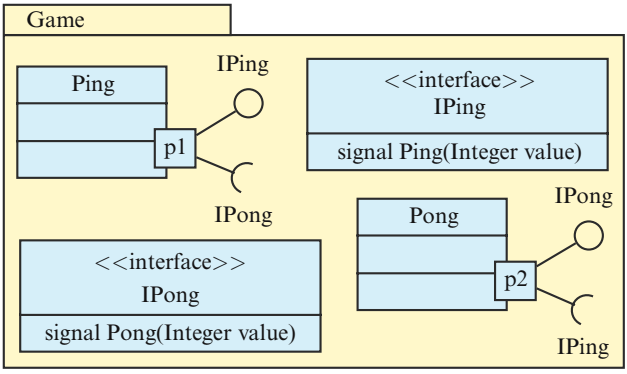


Fig. 14.1 Structure in a UML design

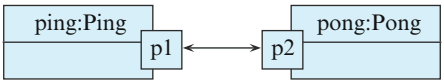


Fig. 14.2 Communication in a UML design

We will later show the SystemC code generated from it. In this system the package Game encapsulates the declarations of the classes Ping and Pong as well as the interfaces IPing and IPong. A SystemC module is very similar to a UML class. In fact a SystemC module is defined as a C++ class with some predefined methods and attributes. Thus a SystemC module can have attributes and methods and also it can inherit from one or more classes and/or modules. There are a few properties which can be assigned to a UML class which cannot be expressed in the SystemC language. One such is abstract, but this can be emulated by making one of the methods in the class abstract. We believe the minor limitations of a SystemC module are negligible in practical use, so we treat UML classes as equivalent to SystemC modules. Both UML classes and SystemC modules can contain references to other objects making it is possible to communicate between classes by method calls. This is however not the intended means to model communication in neither language. Instead communication should pass through ports connected using connectors/channels. A port is part of an class/module and defines its communication interface. In UML a port has a required and a realized interface indicating which signals it will send and receive. Both the required and realized interface can be composed of a list of UML interfaces. Some tools also allow signal lists. In SystemC a sc_port must have exactly one SystemC interface. The interface details which methods the module will call on the connected channel. A SystemC port corresponds to the required interface of a UML port. The equivalent of the UML realized interface is a SystemC sc_export. A SystemC sc_export is part of a module and has exactly one interface, which details the calls the module will implement.

14.5 Communication

The way communication is commonly modeled is quite different in UML and SystemC. In UML communication is modeled using signals, asynchronous messages that can carry data. The signals are sent through the ports of a class. The destination of a signal is determined by the connectors of the model. In Fig. 14.2, any signal sent from the object ping will be forwarded to the object pong. At the receiving object the signal is stored in a queue, from where it later will be consumed by the behavior of this class. A UML connector only provides routing information for signals, it does not model the communication mechanism, i.e. a network or a bus. If this is to be included in the model it must be done using classes. Note also that a UML connectors are primarily a relation between two objects and not between classes. A UML port can have several connectors, and a connector connects exactly two ports.

SystemC channels are a central part of communication modeling in SystemC. They implement the behavior of the communication mechanism, as follows. During

initialization each port is connected to a channel. At this time the port saves a reference to the channel. Later during simulation the ports will be transparent, forwarding any operation to the channel (this is done by overriding the – > operation). This design implies some constraints; a port may only connect to a channel which implement its interface, and also a port can only connect to one channel. By default there is no limit to how many ports that can connect to a channel, but it is possible for a channel to limit the number of ports connecting to it. Since a message sending is realized as a method call in a channel, it is not possible for two modules to communicate without an intermediate channel. Also SystemC does not allow ports to be used to make methods in a module available to other modules. For this purpose sc_export was added to the language. Using sc_export a module can encapsulate a channel and export its interface to other modules. This makes it possible for a sc_port in one module to connect to a sc_export in another module without creating any intermediate channel.

14.6 Mapping Rules

Considering the difference in communication modeling it is clear that there does not exist a trivial, one to one mapping from UML to SystemC. Some UML constructs are however so similar to SystemC that we suggest that they should be replaced with the corresponding SystemC construct during the mapping process. Table 14.1 lists some of these constructs. We base our SystemC code generator to a large extent on Telelogic's C++ code generator [8]. This is possible since SystemC is a library and a simulation engine implemented on top of C++. The implementation of our code generator is explained further in Section 14.5. In this section we focus on the mapping rules that are unique for SystemC and refer to [8] for details regarding mapping of the parts of the UML language not covered here.

The asynchronous communication of UML signals implies that there must exist a message queue somewhere in the communication. In UML this is located in the receiving class. When comparing to the predefined channels in SystemC, sc_fifo comes closest. However, there are some limitations which make it less suitable.

First, in a UML state machine it is possible to wait for one of several signals, i.e. several transitions, with different triggers, from the same state. When the state machine is in such state, the triggering signals might arrive on different ports. This leads to the need to do blocking reads on several fifo queues at the same time.

Table 14.1 Mapping rules for equivalent concepts

UML	SystemC
Package	Name space
Active class	sc_module
Class with ports	sc_module
Other classes	C++ class

This is not possible. In SystemC it is possible to wait for one of several events to occur, but when the call to wait returns it is not possible to determine which event that actually occurred. The concept of events in SystemC is similar to the wait() and notify() synchronization mechanism found in, for example, Java. The second problem with sc_fifo is its limitation to connect only one sender and one receiver. This is the same constraint as a UML connector. The problem is that several UML connectors can connect to the same port, but a SystemC port can only connect to one channel. A UML connector does not provide any message queue; instead it connects the sending port with the queue inside the receiving class. This has the same semantic as connecting a SystemC port to a channel, assuming that the channel will provide a message queue.

The observation that a UML connector has the same semantic as connecting a SystemC port to a channel is one motivation for our mapping. We map a UML connector to code which connects the sending modules port with the channel containing the message queue of the receiving module. For this to work, we need a channel which implements a message queue and allows multiple connecting senders. Also, to solve the first problem with sc_fifo, this queue should be shared among all ports of the receiving module. There is no SystemC channel which meets these needs, so we generate one for each generated SystemC module. The channel will handle all incoming signals to the module.

The structure of a generated SystemC module is shown in Fig. 14.3. The module is composed of one or more threads, one channel, and one or more ports and/or exports. When a message arrives at a sc_export, a method in the channel will be invoked and the message will be stored in the channels message queue. The threads in the module will later consume the message using the channels blocking read() method. The threads can also send messages through a sc_port. A message sent through a sc_port will arrive at a sc_export of another module.

Table 14.2 lists the UML sources for different SystemC constructs. How we generate the components of the SystemC module will be detailed below. For each realised interface of the UML class we generate one sc_export and connect it to the modules

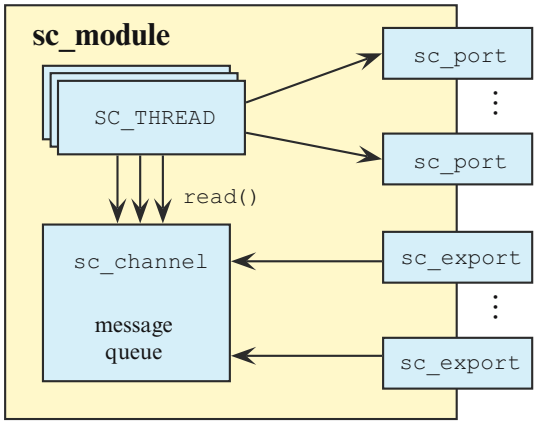


Fig. 14.3 The structure of a generated SystemC module

Table 14.2 The table lists SystemC parts and the UML constructs they are created from

SystemC part	UML source
sc_port	Port, required interface
sc_export	Port, realized interface
sc_channel	Interface, signal
sc_interface	Interface, signal
constructor of sc_module	Port, channel, state machine. Attribute initialization have more sources
SC_THREAD	State machine

```
1 Ping ping("Ping");
2 Pong pong("Pong");
3 ping.p1_port(pong.p2_export);
4 pong.p2_port(ping.p1_export);
```

Fig. 14.4 Code generated from Fig. 14.2

channel. The channel implements all realized interfaces with one method for each signal. The method stores the parameters of the signal in the channels message queue. This queue is implemented using a C++ std::deque. The channel also provides an blocking read, used by the modules threads, i.e. the methods generated from the state machine of the active UML class. To clarify, let us look at an example. The UML diagram in Fig. 14.2 will generate the SystemC code shown in Fig. 14.4.

In the first two lines the module instances are created. Lines three and four connect the ports and exports of the generated module instances. Lines three and four are generated from the UML connector. Commonly line one and two will be attributes in a module and line three and four will be part of that modules constructor.

Next we will examine the SystemC declaration of module Ping, shown in Fig. 14.5. This originates from the UML view in. The UML port p1 is mapped to a sc_port and a sc_export at line 3–4. The interfaces IPing and IPong are generated from the UML interfaces. This mapping is detailed below. Lines 6–16 contain the declaration of the SystemC channel, which contains the message queue of the Ping module. The channel is instantiated at line 17. The method Ping at line 15 originates from the UML signal Ping and is part of the IPing interface inherited at line 8. The implementation is on lines 21–26. At line 30, in the constructor of Ping, p1_export is bound to the channel instance Ping_channel. With the generated code Figs. 14.4 and 14.5, the module instance pong can send a Ping signal carrying the value three, using the syntax p2_port- > Ping(3).

14.7 Interfaces and Signals

The mapping of UML classes, ports and channels detailed above is not enough to generate code which compile. The SystemC interfaces and data structures for storing signals in the message queue are missing. These are generated from the UML

```
SC_MODULE(Ping ){
  public:
  sc_export<IPing> p1_export;
  sc_port<IPong> p1_port;
  /*--- channel ---*/
  class Channel_class:
              public sc_channel,
              public IPing{
    private:
    std::deque<UML_signal  *> queue;
    sc_event e;
    public:
    Channel_class(sc_module_name name)
    UML_signal *read()
    void Ping(int value)
  }
  Channel_class Ping_channel;
  /*--- state machine behavior  ---*/
  void Ping_thread()
}
void Ping::Channel_class::Ping(int
                                value){
  queue.push_back(   new
                     Ping_signal(value)
  e.notify()
}
Ping(sc_module_name name):
      sc_module(name)
      Ping_channel("Ping_channel")
  p1_export(Ping_channel)
  SC_HAS_PROCESS(Ping)
  SC_THREAD(Ping_thread)
}
```

Fig. 14.5 Code generated from Fig. 14.1

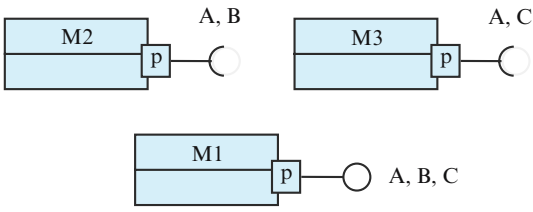


Fig. 14.6 UML classes with interface lists

interfaces and signal. For each UML interface a SystemC interface will begenerated. The generated interface will contain one method for each signal in the UML interface. The method will have the same parameters as the original signal. In addition to the method each signal will also generate a class with one attribute for each signal parameter. The code generated from the UML interface IPing in Fig. 14.1 is shown in Fig. 14.6.

```
class IPing: public sc_interface{
 public:
 virtual void Ping(int value)= 0;
 class Ping_signal: public UML_signal{
  public:
  int value;
  inline Ping_signal(int value):
                      value(value){
 }
}
```

Fig. 14.7 Code generated from Fig. 14.1

In SystemC a port and export can only have one interface and for a port to connect to an export/channel its interface must be implemented by the channel. Now look at Fig. 14.7.

UML port M1::p has three realised interfaces A, B, C. Ports M2::p and M3::p have required interfaces A, B and A, C. Port M1::p will be mapped to an SystemnC export while M2::p and M3::p will generate SystemC ports. The generated export and ports need one interface each, here named if1, if2, and if3. A trivial attempt would be for the generated interfaces to inherit directly from from A, B and C, i.e. class if1:A, B, C{}, class if2:A, B{}, and class if3:A, C{}. Now neither if2 nor if3 is a subtype of if1 and the ports can not connect to the export. A working solution is for if1 to inherit form if2 and if3. But now, due to the double inheritance, if1 will contain two instances of A. Also, with this mapping an exports interface will change as ports connect to it making it unpractical to generate code for parts of a system, or to distribute IP-cores in binary form, since they need to be recompiled when used.

To solve this we suggest that one UML port should be mapped to several SystemC ports, one for each interface it requires. The list of realized interfaces can still be mapped to one export with one interface, i.e. class if1:A, B, C{}. With this mapping M2 will have two ports, one for interface A and one for B. Both can connect to M1::p. This solves the problems mentioned above while preserving the type hierarchy among interfaces in the UML model.

14.8 Mapping Process

During initial system modeling, a pure UML model is used. Though it is possible to define a set of mapping rules from a pure UML model directly into SytemsC code, it would give the engineer little influence on the mapping and most likely a less satisfactory result. Instead we divide the mapping into three steps, as depicted in Fig. 14.8.

ll models are available and editable. This makes it possible for the engineer to have full control over the relevant details for the system under development and have the tool manage all remaining details.

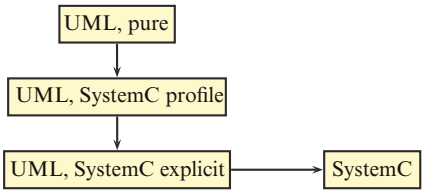


Fig. 14.8 Our three step code generation

Step 1, vertical refinement transformation: In this step an initial UML description is refined to a UML description, which follows a UML profile for SystemC. This step will, at least partly, be carried out manually. To minimize the design effort it should not be required to tag the whole model. This means that a set of default values for the SystemC specific attributes of the UML profile, must be defined. The default values will in most cases provide a satisfactory mapping in the following transformation steps.

Step 2, vertical refinement transformation: In this step the model is transformed into a new UML description that only includes UML constructs with direct representations in SystemC, i.e. classes, attributes, inheritance, etc. Other constructs such as state machines are translated to the target language. During this step we transform each state machine to a class with methods that implement the behavior of the states and transitions. In the first version of the tool, the resulting model will be a un-timed functional model. The mapping rules for UML classes, ports, channels, signals and interfaces are given in Section 14.4. A complete list of UML constructs which are removed during this transformation is beyond the scope of this paper. In addition to removing UML only concepts, we also make all relations in the model explicit. When a class is made active in UML it implies that the class will have its own thread of execution. In SystemC this is realized using SC_THREAD or SC_METHOD which implies that the class is an instance of the SystemC class sc_module. During this transformation all such implicit relations are made explicit. For example, we add a generalization relation to the SystemC class sc_module from all active UML classes. In the first version of the tool the resulting model is a un-timed functional model.

Step 3, horizontal transformation: In this step the UML model resulting from step 2 is transformed into a corresponding SystemC code. This transformation is a one to one correspondence between the UML model to the resulting SystemC code, i.e. this is a “pretty print” of the UML model. This step is implemented using the existing C++ code generator from Telelogic and thus reuse its support for scope rules, header-file inclusion and make file generation without any modifications. If the generated code is to be read by humans it is desirable to use the common SystemC macros when applicable. This requires a slight customization of the syntax of the generated code. We do this using an agent, a mechanism which makes it possible for third party executables to interact with the C++ Code generator in Telelogic Tau G2. Our agent generates SystemC like module declarations, instead of a C++ class declarations, SC_MODULE(MyModule){…} instead of class MyModule:public sc_module{…}.

14.9 Experimental Validation

The mapping rules and code generator presented in this paper have been use by VTT to extend their workload-based performance simulation [5]. The automatic mapping from UML to SystemC makes it possible to partially reuse existing UML application models, removing the need for separate work load models. VTTs experience is that our SystemC code generator is useful in practice and simplifies the engineers work in their model based design flow, see [5].

14.10 Conclusions

Combining new methodologies and new languages is a promising approach to overcome the increased complexity of today's embedded systems. This is the driving force in the MARTES project. In this paper we compare UML and SystemC. The comparison reveals that the communication is modeled quite different in the two languages. Based on our observations we present mapping rules for automatic SystemC code generation from a UML model. We also present our transformation technique, composed of two vertical and one horizontal transformations. Using our transformation technique it is possible to reuse large parts of a code generator for other target languages similar to the target languages of the code generator, i.e. the implementation of our SystemC code generator uses a large part of Telelogic's C++ code generator.

References

1. Eriksson, H., Penker, M., Lyons, B., Fado, D.: UML 2 Toolkit. OMG Press, Indianapolis, IN (2004)
2. Grötker, T., Liao, S., Marin, G., Swan, S.: System Design With SystemC. Kluwer, Norwell, MA (2002)
3. IEEE: IEEE Standard SystemC Language Reference Manual. IEEE Standard 1666–2005 (2006)
4. Nguyen, K. D., Sun, Z., Thiagarajan, P. S., Wong, W.: System driven SoC Design Via Executable UML to SystemC. Real-Time Systems Symposium (2004)
5. Kreku, J., Hoppari, M., Tiensyrjä, K., Andersson, P.: SystemC Workload Model Generation from UML for Performance Simulation. Proceedings of Forum on specification and Design Languages (FDL) (2007)
6. Piltone, D., Pitman, N.: UML 2.0 In a Nutshell. O'Reilly Media inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2004)
7. Riccobene, E., Scandurra, P., Rosti A. Bocchio, S.: A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. Design Automation and Test Europe (DATE) (2005)
8. Telelogic, PO Box 4128, Kungsgatan 6, SE-203 12 Malmö, Sweden: C++ Application Generator Reference

Chapter 15
An Enhanced SystemC UML Profile for Modeling at Transaction-Level

S. Bocchio[1], E. Riccobene[2], A. Rosti[1], and P. Scandurra[2]

Abstract This chapter describes a UML2 profile for the SystemC language, which takes into account the language improvements as specified in the IEEE 1666 SystemC Standard and effectively provided in the SystemC 2.2 simulator as foundation for Transaction-Level Modeling (TLM). The profile is a set of modeling constructs which lift both the structural and behavioral features of SystemC to UML level. It is part of a model-driven HW-SW co-design methodology based on the UML2, a SystemC UML profile for the HW side, and a multi-threaded C UML profile for the SW side, which allows modeling of the system at higher levels of abstraction (from a functional executable level to Register Transfer Level) and supports automatic code-generation/back-annotation from/to UML models.

Keywords Embedded systems, system-level design, SystemC, UML, UML profiles

15.1 Introduction

To increase the design productivity and tackle the ever growing system complexity, the Electronic Design Automation (EDA) communities are pushing a shift in design entry level for the Embedded Systems (ES) and Systems on Chip (SoC) development. New more abstract design methodologies and languages – far beyond the capabilities of existing HW description languages, like VHDL and Verilog, operating at the low Register-Transfer-level (RTL) – are needed in order to handle a design task which should allow the convergence of both HW and SW facets, as well as better reuse and integration of pre-designed components (the Intellectual Properties).

[1] STMicroelectronics R&I, Agrate Brianza, Italy; Email: {sara.bocchio,alberto.rosti}@st.com

[2] University of Milan – DTI, Crema, Italy; Email: {riccobene, scandurra}@dti.unimi.it

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

Recently, the Unified Modeling Language (UML) [15] and its extension mechanism is receiving significant interest in the hardware community, since it allows UML customization towards the definition of a family of languages targeted to specific application domains (telecommunications, aerospace, real time computing, automotive, System-on-Chip, etc.) and levels of abstraction. This is confirmed by several standardization activities controlled by the OMG, such as: the Schedulability, Performance, and Timing Analysis (SPT) profile [17]; the recent UML for SoC Forum (USoC) [18] in Japan founded by Fujitsu, IBM/Rational, and CATS to define a set of UML extensions to be used for SoC design; the SysML proposal [24] which extends UML towards the Systems Engineering domain, and the MARTE (Modeling and Analysis of Real-Time Embedded systems) initiative [16].

Along the same research line, we can mention the recent model-driven HW-SW co-design methodology in [2, 5]. According to the emerging Model Driven Engineering (MDE) approach, a new design flow is proposed for ES development. It is based on the UML 2.0 to be used in a platform-independent manner to provide a first high-level functional specification of the whole system, a SystemC UML profile to be used for the HW description at several abstraction levels on top of the RTL level, and a multi-threaded C UML profile to specify the SW application. Moreover, to foster this methodology in a systematic way and combine all the involved notations together in a seamless manner, in [6] a design process, called UPES (Unified Process for Embedded Systems), is defined by extending the conventional Unified Process (UP) of UML, together with the UPES sub-process, called UpSoC, for refining the HW platform model. Furthermore, a HW/SW co-design environment [4] was developed on top of the UML visual modeling Enterprise Architect (EA) tool [9], to assist the designer across the refinement steps in the UML modeling activity regarding the HW part, from a high-level functional model of the system down to the RTL level, and supports forward and reverse engineering of C/C++/SystemC code.

The SystemC UML profile [6, 25] is the key point of this model-driven co-design methodology for ES. It is a consistent set of modeling constructs which lift both the structural and behavioral features (including processes, events and time features) of SystemC to UML level, while providing unification in the overall UML modeling activity. This last starts from the definition of an abstract UML model (or PIM – platform independent model) describing the general functionality of the system, and continues with subsequent refinements of the PIM (or of portions of it) into platform specific models (or PSMs) through a sequence of model transformations. For the HW components, this sequence of PSMs goes from a high level functional un-timed/timed model of the system down to a transaction-level model, to a behavioral model, to a bus-cycle accurate (BCA) model, to a final RTL model for the synthesis of an end-product integrated into a chip. The UML profile for SystemC allows using UML at PSM level, provides unification between PIM and PSMs, and allows automatic encoding of PSMs into final SystemC code.

The choice of SystemC as implementation language is intentional, and mainly due to the fact that SystemC is becoming one of the most important system-level languages for SoC design. In 2006, SystemC received a major revision (2.2) and

became IEEE Standard [5]. This last revision includes new structural (`sc_export` and `sc_event_queue`) and behavioral (dynamic processes, fork/join synchronization, etc.) features required for modeling at transaction-level according to the OSCI [19] standard TLM 1.0 API.

To align the SystemC profile with the standard IEEE [25] and support refinement towards implementation in SystemC 2.2 according to the OSCI TLM standard, the SystemC UML profile described in [1] has to be reviewed. In this chapter, we present a UML2 profile for the SystemC 2.2 release. It extends the profile in [6, 25] with the new improvements specified in the IEEE 1666 SystemC Standard. The structural features of the SystemC UML profile in [1] have been extended including the new features of ports connection and event queue handling, while for the behavioral part, we extend the SystemC Process State Machines (an extension of the UML state machine formalism introduced as part of the SystemC profile to model the reactive and concurrency behavior of SystemC processes) with the new enhancements in SystemC 2.2 for dynamic processes, i.e. processes created at runtime as children processes of running processes. This last extension required to fix some UML state machines semantic variation points to capture the operational semantics of the dynamic SystemC processes. This new profile allows for modeling at Transaction-Level (TLM) of abstraction with the OSCI TLM 1.0 library. Moreover, according to the reviewed version of the profile, the code generator of the HW-SW co-design environment in [4] has been updated to guarantee straightforward generation of efficient SystemC 2.2 code from diagrammatical UML models developed by using the SystemC profile.

The remainder of this chapter is organized as follows. Section 15.2 sketches some fundamentals of the SystemC 2.2 language assuming the reader familiar with the SystemC language. Section 15.3 introduces basic concepts underlying the SystemC UML profile along with the enhanced structural and behavioral features of the profile. Section 15.4 describes the code generation facility for diagrammatical models developed using the SystemC UML profile, while Section 15.5 presents some case studies. Related work and conclusions are given in Sections 15.6 and 15.7, respectively.

15.2 SystemC Background

The SystemC language is an open standard that is imposing as the reference language in ESL (Electronic System Level) design; it is controlled by the OSCI group [19] made of different companies in the EDA area.

SystemC is defined in terms of a C++ class library for modeling in terms of C++ programs, and provides an event-based and discrete-timed simulation kernel.

SystemC provides constructs for modeling the system structure (`sc_module` and `sc_channel`), the communication (`sc_port`, `sc_interface`, `sc_event`), the concurrent behavior through processes (`sc_method` and `sc_thread` processes) and a set of data types for hardware data.

In 2006, SystemC received a major revision (2.2) and became IEEE Standard [25]. This last revision includes new structural features (`sc_export` and `sc_event_queue`) and behavioral features (dynamic processes, fork/join synchronization, etc.) required for modeling at transaction-level towards hardware-software implementation according to the OSCI TLM standard.

SystemC has been involved into several SoC design flows at industrial level, exceeding, for its expressivity, the capabilities of traditional Hardware Description Languages (HDLs). It permits to design at system level supporting different abstraction levels (un-timed/timed functional, TLM, behavioral, BCA, and RTL), thus allowing design refinement in a unique modeling environment.

15.3 The SystemC UML Profile

A UML profile is to be intended as a dialect of the UML for a particular platform or application domain. The UML profiles mechanism is a standard way of customizing the UML by adding a set of stereotypes, tags and constraints. Stereotypes define how the syntax and the semantics of an existing metaclass of the UML metamodel are extended for a specific domain terminology or purpose. Tag values are user-defined properties of a stereotype to add further attributes to the extended metaclass. Constraints are expressed as formulas in the Object Constraint Language (OCL) and serve to add static semantic restrictions to the extended UML modeling element.

For defining profiles, UML2 is endowed with a standard graphical notation which is easily supported by UML visual modeling tools. A profile is denoted as a package with the keyword «`profile`». Within the profile package, a class of the UML metamodel that is extended by a stereotype is labeled as a conventional class with the keyword «`metaclass`». A stereotype is denoted as a class with the keyword «`stereotype`». The extension relationship between a stereotype and a metaclass is depicted by an arrow with a solid black triangle pointing toward the metaclass box. When applied to an element in a UML model, a stereotype is shown as a keyword consisting in the name of the stereotype within a pair of guillemets, near the symbol of the UML element or with a special icon defined for it (if any) in place of the conventional symbol for the element.

A UML2 profile for SystemC 2.0 already exists [1]. In the next two sections, an extension of this profile is presented (we assume the reader familiar with SystemC 2.0) in a lightweight manner by describing some new structural features (`sc_export` and `sc_event_queue`) and behavioral features (dynamic processes and fork/join synchronization) capturing the semantics as specified in the IEEE 1666 SystemC standard and implemented in the SystemC 2.2 execution engine [5]. This extension is dictated by the necessity to align the profile definition with the standard IEEE [5] in order to include the new SystemC constructs for modeling systems, communication, hardware and software at the transaction-level, and supporting refinement towards hardware-software implementation according to the OSCI TLM standard.

15.3.1 Enhanced Structural Features

In SystemC 2.2, a port (`sc_port`) may be bound to a channel either directly, or indirectly by being bound to another port (according to a parent-to-child module relationship) or to a `sc_export` port (export, in brief). Figure 15.1 shows the `sc_port` and `sc_export` stereotypes. Note that, a further tag policy (an element of the enumeration type `sc_port_policy`) has been added to the sc_port stereotype; it is used to determine the rules for binding multi-ports and the rules for unbound ports, as specified in the new SystemC version.

An export defines a set of services (as identified by the interface type of the export) that are provided by the module exposing the export. Providing an interface through an export is an alternative to a module which simply implements the interface. The use of explicit exports exposed by a module instance allows a single module to provide multiple interfaces in a structured manner: the underlying interfaces are implemented somewhere within the module, e.g. by a child channel instance.

The `sc_export` stereotype maps the notion of SystemC export port directly to the notion of UML port, plus some constraints. An export port can have exactly one provided interface – the type of the export – and no required interfaces. An export can only be bound to a channel derived from the type of the export or to another export (provided that this export itself is directly or indirectly bound to a channel) with a type derived from the type of the export. Similarly to the `sc_port` notation, an export port is shown as a small square symbol with the port name and the keyword «`sc_export`» nearby. Alternatively, an export can also be shown as a small triangle icon with the port name. In both cases, the provided interface is shown by a circle or ball, labeled with the name of the interface, attached by a solid line to the export port.

In the new profile definition, the semantics of connectors `sc_connector` and `sc_relay_connector` has been extended in order to represent three new possible bindings: *port-to-export*, *export-to-channel*, and *export-to-export*. To be precise, the `sc_connector` stereotype, originally provided as extension of the UML connector to explicitly bind a port to a channel (*port-to-channel*), now can be used to directly bind a port to an export provided that the export exposes the interface

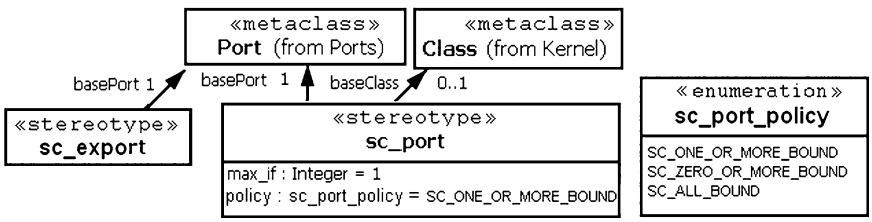


Fig. 15.1 `sc_export` and `sc_port` stereotypes

required by the port. Similarly, a `sc_relay_connector`, originally defined to represent the *parent-to-child* port binding, now is also used to bind an export to a channel, and also to bind an export to another export. Both connectors are binary, i.e. a connector specifies a link that enables communication between two instances only. All connectivity rules are provided in terms of OCL constraints defined over the involved classes of the UML metamodel. Figure 15.2 shows an example of application of these stereotypes in a UML class diagram to model the hierarchical structure of a `Top` module made of two sub-modules, `Caller` and `Middle`, connected via a port-to-export binding.

Figure 15.3 shows the `sc_event_queue` stereotype definition together with the one for the `sc_event` stereotype. They represent SystemC events in terms of special UML signals whose notification generates signal events (instances of the

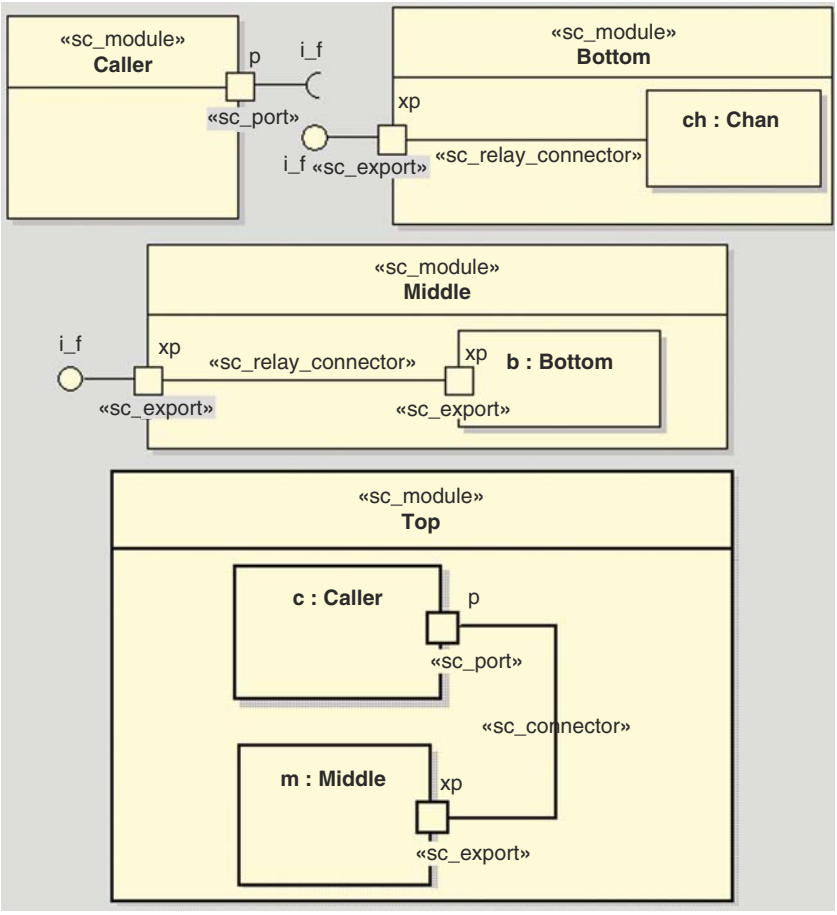


Fig. 15.2 Example of structural modeling with `sc_esport`

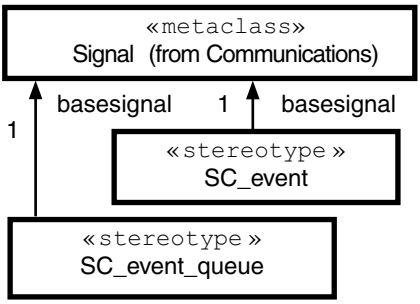


Fig. 15.3 `sc_event_queue` stereotype

class `SignalEvent` in the UML metamodel) to be put in the input pool of the processes to be activated/resumed. In particular, the `sc_event_queue` stereotype denotes a structured signal, namely, an event queue which can have multiple notifications pending. For an event queue only delta-cycle delayed and timed notifications are allowed. A `sc_event_queue` cannot be used in most contexts requiring a `sc_event` but can be used to define the static sensitivity of processes. The mechanism used to queue event notifications shall be implementation-defined, with the proviso that an event queue must provide a single default event that is notified once for every notify action for the event queue. Effective user-named signal instances are declared with the stereotype keyword «`sc_event`» within the attribute compartment of a module's class or a channel's class. The label for a trigger on a state machine transition denoting a `sc_event` signal may explicitly indicates the name of the specific `sc_event` instance whose notification causes the triggering of the transition. The same notation is used for a `sc_event_queue` structured signal.

15.3.2 Enhanced Behavioral Features

Processes are the basic mechanism in SystemC for representing concurrent behavior. Two kinds of processes are available: methods and threads. Clocked threads are a specialization of threads. Each kind of process has a slight different behavior, but basically all processes: run concurrently, are sequential, and are activated (if terminated or simply suspended) on the base of their own sensitivity, which consists of an initial list of zero, one or more events – the *static sensitivity* of a process – and can dynamically change at run time realizing the so called *dynamic sensitivity* mechanism.

The SystemC UML profile defines two processes stereotype «`sc_method`» and «`sc_thread`» (see [1] for details); both extend the `Operation` and the `StateMachine` UML metaclasses. This double extension allows us to associate an operation to its behavior specified in terms of a (method) state machine. Special state and action stereotypes are added to support the behavioral features mentioned

above. These stereotypes and their associated OCL constraints lead to a variation of the UML state machine formalism: the *SystemC Process State Machines*. This formalism allows modeling the control flow and the reactive behavior of processes (methods and threads) within modules, dealing with concurrency, synchronization and timing aspects.

A process state machine can contain the definition of local variables. Two particular states (initial and final) are used to model start and termination of the process behavior. The behavior is modeled by states, transitions, and actions. States can contain simple actions or activity which must obey the syntactic rules and take the semantics of the C++/SystemC language (the action or surface language). The semantics of basic C/C++ control structures, like `if` conditions, `while` loops, etc., is captured in terms of stereotyped choice pseudostates (see for example the `while` loop in Fig. 15.4).

The example in Fig. 15.4 also shows a `static_wait`-stereotyped state. It captures the SystemC semantic of a `wait()` statement with no arguments.

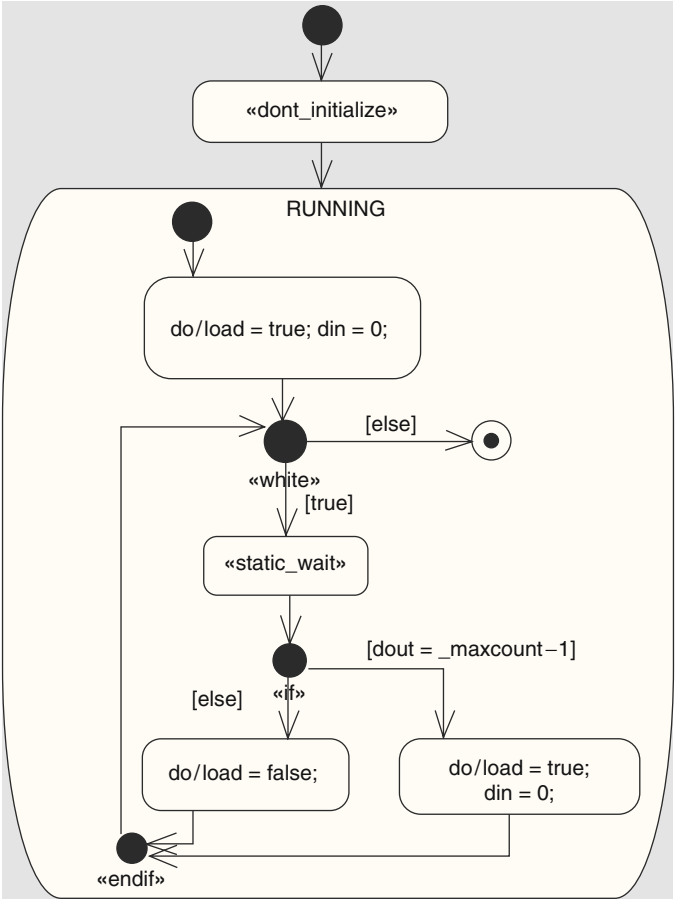


Fig. 15.4 Example of thread process state machine

In general, to model the dynamic sensitivity mechanism of a thread process two possible wait-stereotyped states are available (see Fig. 15.5): the first one is a wait on the static sensitivity list, the second is a wait on a dynamic sensitivity list characterized by an event condition `e*`. Figure 15.6 shows how a `wait(e*)` call is modeled in UML for all possible forms of the event `e*`: a single timed event, a single signal event, a single event with timeout, an AND-list of signal events, an

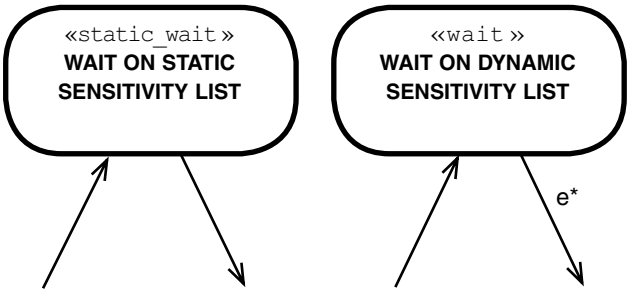


Fig. 15.5 Static and dynamic wait

SystemC	UML Notation
`wait(e)` wait for an event `e`	e «wait» waiting for e
`wait(t)` wait for `t` time units	t «wait» waiting for t
`wait(e,t)` wait for an event with timeout	e, t «wait» waiting for e with timeout t
`wait(e1&..&eN)` wait for an AND-list of events	$e_1, \dots, e_N$ «and» «wait» waiting for e_1&...&e_N
`wait(e1|..|eN)` wait for an OR-list of events	$e_1, \dots, e_N$ «wait» waiting for e_1|...|e_N
`wait(t,e1&..&eN)` wait for an AND-list of events with timeout	$e_1, \dots, e_N$ «and» t «wait» waiting for e_1&...&e_N
`wait(t,e1|..|eN)` wait for an OR-list of events with timeout	$e_1, \dots, e_N$ t «wait» waiting for e_1|...|e_N

Fig. 15.6 Dynamic sensitivity of a thread process

OR-list of signal events, AND-list of signal events with timeout, and OR-list of signal events with timeout. Similar constructs have been defined to model the dynamic sensitivity mechanism of a method process.

We choose to use the state machines with respect to other UML behavioral diagrams (like the activity diagrams) because this kind of diagram provides a behavioral pattern appropriate for modeling the *reactive* and *hierarchical* behavior of SystemC processes, which can be activated by triggering external synchronization events. Moreover, according to the OMG specification [15], state machines are sequential as far as their internal behavior is concerned, but any state machine is concurrent with respect to the other state machines of the system. Indeed, UML state machines can be used for modeling simple functions that execute under the control of processes, and it is also possible to represent the SystemC synchronization mechanism for suspending/resuming a process in terms of stereotyped states and events.

We extend here the SystemC process state machines by adding specialized submachine states and orthogonal regions within a state machine to model the notion of *process hierarchy* expressed in SystemC in terms of *dynamic processes*. A dynamic process is a process created at run-time during execution, as child process of a method process or a thread process, or a clocked thread process. A dynamic process can in turn create other processes dynamically. The SystemC 2.2 release supports the notion of dynamic process by introducing the concept of spawned process, i.e. a process (a child process) created by another process (the parent process) by invoking the predefined function `sc_spawn`. In the Systemc UML profile, the dynamic creation of such a process – a dynamic spawned process- is denoted in the state machine diagram associated to the parent process by means of a submachine state[1] labeled with the stereotype «`sc_spawn`» (see Fig. 15.7 for the stereotype definition). The state machine referenced by the submachine state specifies the functionality of that dynamic process.

After the creation of a spawned process, the parent process and the new child process proceed in parallel, unless a specific synchronization schema is explicitly provided by the designer by means of notification of events. This natural asynchronism is reflected at UML level in the state machine diagram of the parent process by the use of orthogonal regions. To be precise, a process state machine which dynamically creates processes is represented by a state machine with two or more regions (see Fig. 15.8). One region contains the behavior specification of the parent process, while the others contain exactly one «`sc_spawn`» submachine state each. The overall process creation (i.e. the invocation of the SystemC `sc_spawn` function) is denoted by a fork vertex in the parent region with two outgoing transitions: one entering in the «`sc_spawn`» submachine state of the child process, and one entering in some state of the parent region to continue the specification of the parent process behavior after the process creation. Therefore, the submachine state

[1] In UML, a submachine state specifies the insertion of the specification of a submachine state machine. The state machine that contains the submachine state is the container.

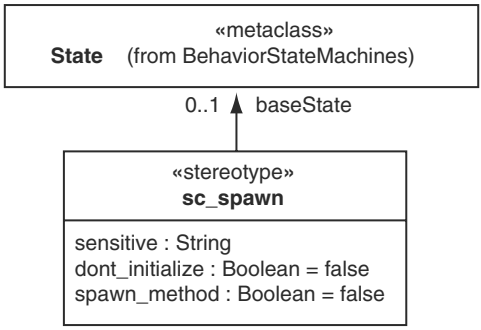


Fig. 15.7 `sc_spawn` stereotype

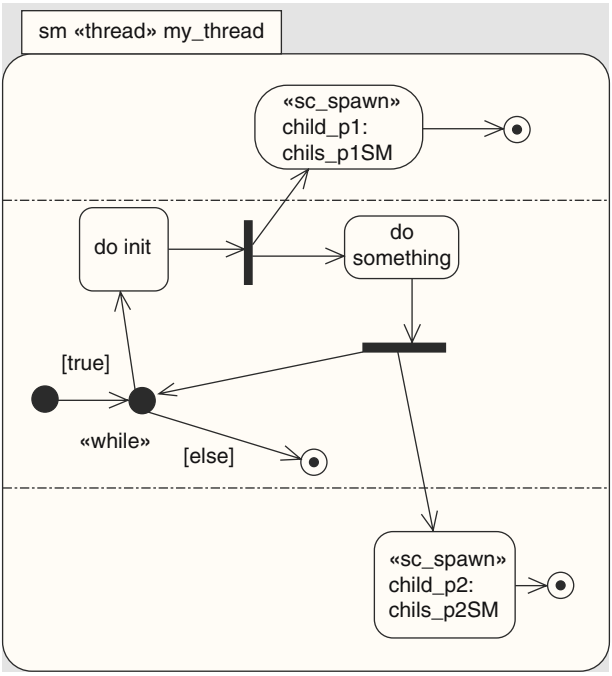


Fig. 15.8 A thread process spawning dynamically two processes within an infinite loop

(and therefore its reference process state machine) is exclusively entered via a fork vertex departing from the parent region, and can be exited either as a result of reaching its final state (normal case) or via a join vertex in the parent region (in the case of a fork/join schema, see the last paragraph below). No entry/exit points can be defined for a «`sc_spawn`» submachine state.

A special case of synchronism for thread processes is when the parent process wants to wait for the termination of a child process, for example, to get any return

values from the child process execution and then resumes and continues its own execution. In this case, the parent process has to wait for the `terminated_event` of the underlying child process instance that is automatically notified when the child process terminates.

The `sc_spawn's` tagged values are used to specify some spawn options which determine certain properties of the spawned process instance. In particular, as for thread and method processes, the sensitive and `dont_initialize` (false, by default) tags are used to declare the static sensitivity list (if any) and the initialization status of the spawn process, respectively. The boolean tag `spawn_method` being set to true indicates that the spawned process is a method process, and therefore the associated process state machine shall be a method state machine. By default, this tag is set to false, i.e. by default a spawned process is a thread process. It is not possible to spawn a dynamic clocked thread process.

A spawned process, in contrast to ordinary processes, allows the passing of arguments and a return value to and from it. UML2 supports the concept of parameterized behavior for all the kinds of behavior in UML (activities, state machines, etc.); this means that when a process state machine is invoked as behavior of a spawned process, its parameters (if any) are created and appropriately initialized (by the caller process) according to their direction `in` and `inout`. When the state machine of the spawned completes its execution, a value or set of values is returned corresponding to each parameter with direction `out`, `inout`, or `return`.

SystemC 2.2 introduces also the macros `SC_FORK` and `SC_JOIN` to be used in pairs within a thread process to enclose a set of calls to the function `sc_spawn`. The parent thread control leaves the fork-join construct when all the spawned processes are terminated; this means that during the execution of the spawned processes the parent process is not running. We use the UML fork/join pseudo-states to model these macros, as shown in Fig. 15.9: a pair of fork/join for two spawned processes;

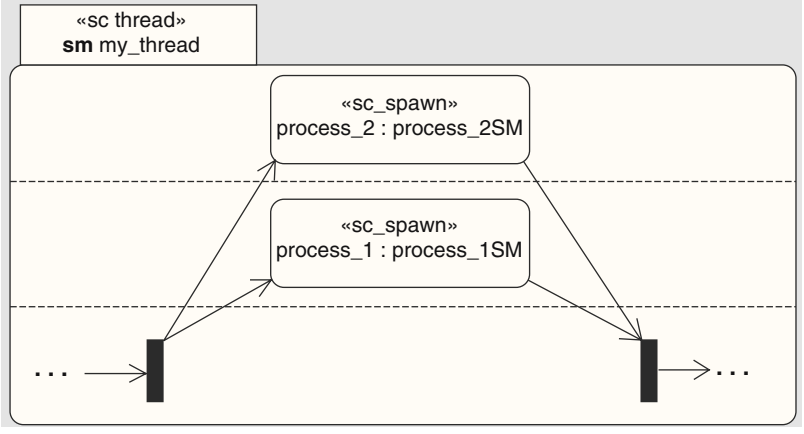


Fig. 15.9 `sc_fork` and `sc_join`

both the fork/join bars are within the region of the parent (thread) process. After termination of the two spawned processes, the parent thread continues to execute.

15.4 Generating SystemC Code from Model Patterns

We developed a prototype tool based on the EA [9] UML visual modeling tool as front-end for consolidated lower level co-design tools (see [4]). This tool consists of two major parts: a development kit (DK) with design and development components, and a runtime environment (RE) represented by the SystemC execution engine. The DK consists of a UML2 modeler supporting the UML profile for SystemC and a UML profile for multi-thread C, and translators for forward/reverse engineering to/from C/C++/SystemC.

We further extended the SystemC code generator by including new code generation rules for the enhanced structural and behavioral features of the profile. The task of the generator is to inspect the elements in the UML model via their connections and create the corresponding modules structures and processes behavior in SystemC. In particular, from the process state machines, the generator follows and combines specific model patterns. The result is a complete working code, without the need for post-generation code modifications or additions.

15.5 Case Studies

We have developed several different case studies, some taken from the SystemC distribution like the Simple Bus design, and some of industrial interest. The Simple Bus case study is a well-known transaction-level example, designed to perform also cycle-accurate simulation. It is made of about 1,200 lines of code that implement a high performance, abstract bus model. We modeled the Simple Bus system in a forward engineering flow in order to test the code generator. The UML description using our SystemC profile consists of about 15 diagrams among class diagrams and process state machines.

To test the expressive power of the profile in representing a variety of architectural and behavioral aspects, we modeled the On Chip Communication Network (OCCN) API [13], a parameterized and configurable SystemC library of about 14,000 lines of code. The OCCN design has been imported automatically from the C++/SystemC code into the EA-based modeler by the reverse engineering facility, then refined using the modeling constructs of the SystemC UML profile. We have used this example to test the reverse engineering flow.

In [3], we present an example related to a system composed of a VLIW processor developed in ST, called LX, with some dedicated hardware for an 802.11b physical layer transmitter and receiver described at instruction level. The UML model of this application is a function library encapsulated in a UML class which

provides, through ports, the I/O interface of the SW layer to the HW system. This class is then translated to C/C++ code and the resulting code is executed by the LX ISS wrapped in SystemC for HW/SW co-simulation at cycle accurate level. The UML wrapper of the LX ISS is modeled with the SystemC UML profile, in order to generate a SystemC wrapper for the ISS and to allow a HW/SW co-simulation at transaction or cycle-accurate level.

15.6 Related Work

The possibility to use UML 1.x for system design started in 1999 [11, 12]. The general opinion, at that time, was that UML was not mature enough as a system design language. Nevertheless significant industrial experiences and research developments on how to use UML within a system design process started, trying to solve the limitations of the language. As part the OMG profile initiatives mentioned in the introduction, we here reference some relevant works for UML modeling and code generation in the area of embedded systems and SoC design.

YAML [23] is one the first tool based on UML which provides a skeleton-based generation of SystemC code. In [10] an extension of UML 1.x is presented to design embedded real-time applications. UML is conceived as a specification language that allows describing different facets of the system. The proposed approach relies on the concept of *platform based design*. The fundamental idea is to adapt UML for the design of embedded software, providing a proper notation and an associated semantics to use UML diagrams for modeling different facets of the system. The methodology specifies a set of UML diagrams to capture the functionality (use cases, class, state machines, activity and sequence diagrams) and to refine it by adding proper MOCs. However, no code generation facility is provided. Another approach to the unification of UML and SoC design is the HASoC (Hardware and Software Objects on Chip) [8] methodology based on the UML-RT profile [17, 21]. The design process starts with an *uncommitted model* and after a *committed model* is derived by partitioning the system into software and hardware, and then mapped onto a system platform. From these models a SystemC skeleton code can be also generated, but to provide a finer degree of behavioral validation, detailed C++ code must be added by hand to the skeleton code. All the works mentioned above could greatly benefit from the use of new constructs available in the UML2.

A Model Driven Architecture (MDA) [14] approach for SoC design is presented in [7] in the specific context of *Intensive Signal Processing*. The application and the architecture are specified in UML as separate platform independent models; according to the Y chart diagram concept, it is then possible to apply model transformations and deploy platform specific models, among which SystemC.

SysML [24] is a conservative extension of UML2 for a domain- neutral representation (i.e. a PIM model as in MDA [14]) of system engineering applications. It can be involved at the beginning of the design process, in place of the UML, for

the requirements, analysis, and functional design workflows. So it is in agreement with our UML profile for SystemC, which can be thought (and effectively made) a customization of SysML rather than UML. Similar considerations also apply to the MARTE proposal [16]. The standardization proposal [18] by Fujitsu, in collaboration with IBM and NEC, has evident similarities with our SystemC UML profile, like the choice of SystemC as a target implementation language. However, their profile support neither constructs for modeling behavior nor a time model.

Recently, a set of papers deal again with the issue of SystemC code generation from UML diagrams. In [22], for example, the authors propose the use of UML activity diagrams to model data flows. This approach is similar to our one with the difference of using activities diagrams instead of state machines for modeling the system behavior. Code generation is supported for the Handel-C language. In [20], a mapping from SysML to SystemC is proposed. Their aim is to obtain a SystemC code that resembles the behaviour of the original UML model, whereas we extend the UML accordingly to the SystemC execution semantics.

15.7 Conclusions

We extend the UML2 profile for SystemC [1] in order to capture the advanced features of the SystemC IEEE Std [25] concerning ports connection, event queue handling and concurrent aspects of dynamic and hierarchical processes. The main target of this UML profile is to provide a means for SW and HW engineers to improve the current industrial SoC design flow joining the capabilities of UML and SystemC to operate at system-level. This enhanced SystemC UML profile allows modeling at TLM level and, specifically, at a certain number of TLM sub-levels through the OSCI TLM 1.0 API, as well as the new TLM 2.0 proposal. As future work, we are exploring the possibility to define a formal refinement methodology with precise abstraction/refinement patterns for modeling at transaction-level, thus enabling users to efficiently develop SoC virtual prototypes at UML level before physical implementation and making the UML-based environment the ideal framework for high-level system modeling and validation.

References

1. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2005) A UML 2.0 Profile for SystemC. STMicroelectronics TR, AST-AGR-2005-3.
2. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2005) A SoC Design Methodology Based on a UML 2.0 Profile for SystemC. In: Proceedings of Design, Automation and Test in Europe (DATE'05).
3. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2005) A SoC Design Flow Based on UML 2.0 and SystemC. In: Workshop UML-SoC'05 at DAC'05.
4. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2006) A Model-driven Design Environment for Embedded Systems. In: Proceedings of Design Automation Conference (DAC'06).

5. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2007). A Model-driven Co-design Flow for Embedded Systems. In: Advances in Design and Specification Languages for Embedded Systems (Best of FDL'06), Springer. Netherlands
6. Bocchio S., Riccobene E., Rosti A., Scandurra P. (2007) Designing a Unified Process for Embedded Systems. In: Proceedings of International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07).
7. Dumoulin C. P., Boulet M. P., Dekeiser J. L. (2003) MDA for SoC Embedded System Design, Intensive Signal Processing Experiment. In: Proceedings of SIVOES-MDA'03.
8. Edwards M. D., Green P. (2003) UML for Hardware and Software Object Modeling. In: UML for real design of embedded real-time systems, pages 127–147.
9. The Enterprise Architect Tool. www.sparxsystems.com.au.
10. Rong Chen. et al. (2003) UML and platform-based Design. In: UML for Real design of Embedded Real-Time Systems, Kluwer, Norwell, MA, USA.
11. Martin G. (1999). UML and VCC. Cadence Design Systems, Inc., White Paper.
12. Martin G., Lavagno L., Guerin J. L. (2001) Embedded UML: A Merger of Real-time UML and Co-design. In: Proceedings of CODES'01.
13. The OCCN Project: http://occn.sourceforge.net/.
14. OMG, Model Driven Architecture (MDA). http://www.omg.org/mda/.
15. OMG. UML 2.1.1 Superstructure Specification. www.uml.org.
16. OMG. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), ptc/07-08-04 (Beta 1).
17. OMG. UML profile for Schedulability, Performance, and Time, formal/03-09-01.
18. OMG. UML Profile for System on a Chip (SoC), formal/06-08-01, v1.0.1
19. The Open SystemC Initiative. www.systemc.org.
20. Raslam W., Sameh A. (2007) Mapping SysML to SystemC. In: Proceedings of the Forum on Specification and Design Languages (FDL'07).
21. Selic B., Rumbaugh J. (1998) Using UML for Modelling Complex Real-Time Systems. ObjecTime Limited/Rational Software White Paper.
22. Schattkowsky T., Hausmann J. H., Engels G. (2006) Using UML Activities for System-on-Chip Design and Synthesis. In: Proc. of the ACM/IEEE International Conference on Model-driven Engineering Languages and Systems (MoDELS'06). Genova, Italy.
23. Sinha V. et al. (2000) YAML: A Tool for Hardware Design Visualization and Capture. In: Proc. of the 13th International Symposium on System Synthesis, IEEE Press. Madrid, Spain.
24. SysML. http://www.sysml.org/.
25. SystemC Language Reference Manual. IEEE Std 1666–2005, 31 March 2006.

Chapter 16
SC² StateCharts to SystemC: Automatic Executable Models Generation

Marcello Mura and Marco Paolieri

Abstract The recent development of embedded systems calls for the necessity of a complete framework for design and simulation of applications that span through all levels of system design. Desirable characteristics of such a framework are rapidity of use, simplicity and reusability. For this purpose we already introduced a generator that converts specifications written with a subset of StateCharts to behavioral SystemC [16, 17]. We present here the new version of our tool: most of the limitations of the previous versions have been overcome, the considered subset of the StateCharts formalism has been extended and the target has been changed from behavioral to Register Transfer Level (RTL) SystemC. A major enhancement of this new version is the possibility of obtaining various module instances starting from a single specification, which is vital in some contexts (e.g. Wireless Sensors Networks simulation). The semantics chosen for our StateCharts diagrams is clearly described. The generation of executable models, as well as the kernel template of the generated code, are discussed in detail.

16.1 Introduction

The possibility of generating customized simulators to model a relevant subset of systems in a very effective way could open interesting scenarios in early design phases (even before Hardware/Software partitioning [12]), especially when intrinsic complexity related to the projects is such that people with different expertise need to cooperate. In fact within this kind of framework it is possible to design, in a very short time, virtual prototypes that can be used for ***requirements formalization*** and ***validation***. Moreover systems under development could be extensively tested from the very beginning up to advanced design stages with the same tool, incrementally integrating the model level of definition. Functional and non-functional

ALaRI, Faculty of Informatics, University of Lugano, Switzerland;
Email: muram@alari.ch

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

properties can be analyzed, e.g. using such kind of instruments we analyzed power consumption of a networking protocol in [14, 15] and of the cache memory of a microprocessor in [16].

In our work the emphasis is on the model: our main contribution is in fact a model-based generator of simulators that – starting from dynamic information about a system expressed with a convenient subset of the StateCharts formalism – generates well structured RTL SystemC code for simulation. The framework is organized in a way that it is possible to iteratively refine models up to a point that the generated code is very near to the synthesizable level. During this process results can be compared, allowing for an easier development process.

In Section 16.2 related work is described. The semantics of the StateCharts dialect we use are presented in Section 16.3 and compared with the most important variants. The methodology for extracting information from UML diagrams and using it to create SystemC models is briefly outlined in Section 16.4. Section 16.5 presents a major innovation of our work: the possibility of performing multi-instance simulation. A small example showing the most noticeable features of our framework as well as the introduction of a shell console is illustrated in Section 16.6. Conclusions and further work are outlined in Section 16.7.

16.2 Related Work

In the past ten years there has been a consistent research effort on this subject, leading also to commercial software products. I-Logix StateMate [1] generates executable models starting from UML diagrams, MATLAB Stateflow [2], does the same starting from a concurrent FSM formalism similar to that of StateCharts. In [4] the translation of StateCharts into Hierarchical Finite State Machines (HFSMs) is explored in order to build test cases for the corresponding VHDL realization. StateCharts formalism is also very appropriate for the formal validation of models. In particular, automatic translation into ***Promela/SPIN***, a language used for automatic model checking, was presented in [5, 10, 13]; recently an interesting approach to this problem was reported in [9]. The present research effort aims at building a framework for the generation of simulators. It differs from the commercial products ([1, 2]) first of all for the choice of SystemC as a target language for the generated models [11] so that they can be inserted in already existing SystemC simulation frameworks. It has simpler semantics allowing for an easier customization. As a result the simulator code is clearly structured and easy to understand and manage. Moreover it is possible to use the generated model as an entry point for successive refinement phases leading possibly to HW synthesis. The output can be reduced to a minimum, therefore simulations are quicker and this greatly extends the range of applicability (i.e. contexts in which simulations for long periods of time are necessary). The use of SystemC is particularly indicated for modeling purposes, e.g. in [18] and in [20] SystemC code is generated starting from UML representations, with the final purpose of creating a hardware synthesis. Differences between these

works and our generator are evident. In [18] Transaction Level Modeling (TLM) SystemC code is generated; in [20] a simple one-to-one binding between specifications and generated code is studied. As a result the expressiveness of the modeling language is reduced and models need to be conceived in a way that is very near to SystemC generated code. We started from the concepts explained in our previous work [16, 17] but the tool has been completely modified. The template of the generated code is very different (from behavioral to RTL SystemC) and this has a clear impact on the performance of generated code. The subset of StateCharts semantics represented is extended with insertion of hierarchical states, history and interlevel transitions. Moreover the possibility of generating multiple interacting instances from a single model is provided. This innovative contribution represents a major enhancement as it allows easy generation of executable models for a wide range of multi-instances domain (e.g. networking where a lot of indistinguishable devices may operate).

16.3 Statecharts Semantics Overview

StateCharts are a formalism introduced more than twenty years ago [7] and represent a very powerful instrument for the design of systems. They are derived from FSMs with some extensions such as the concept of hierarchy, the possibility of modeling concurrency, of broadcasting communication to all the concurrently running machines and of adding code to complete behavioral description of states. In particular code may be inserted such that it is executed when a transition triggers (***action***), when entering a state (***entry-activity***), while in a state (***do-activity***) or when exiting a state (***exit-activity***). Since their introduction there have been many attempts to give well defined semantics to StateCharts. The main issue was to define declarative semantics (e.g. [19]) – possibly a denotational one with a compositional approach – corresponding to the operational one that was firstly proposed. Given that StateCharts are not an official language, in a short time a large number of variants were developed. In [3] the possible different aspects were analyzed and a summary of more than 20 different semantics developed for the formalism was given. Even later some more approaches were taken, the most remarkable one being the semantics behind StateMate [1] that was clearly exposed in [6].

When dealing with StateCharts it is therefore necessary to explicitly specify the semantical choices that have been adopted. We decide to focus on clarity and on usability of the formalism. For this reason only a subset of the semantics defined in [6] has been chosen. The basic idea that has been followed is that of using StateCharts in order to facilitate the notation of Concurrent FSMs and to make it more readable. Therefore every operation defined in StateCharts has its immediate counterpart in terms of concurrent state machines. While this on one side reduces the expressivity of the formalism, on the other keeps a strict contact with possible implementations and allows to use diagrams created for the simulator in later phase of development as a reference point or even – once the framework will be completed

– for HW synthesis. With reference to the possible options summarized in [3] we have decided not to use ***Perfect Synchrony Hypothesis***; therefore events happening in a time instant are accounted for in the following one. This in order to maintain ***Causality***, to avoid ***Self-Triggering***, ***Instantaneous States*** and consequently multiple entrance or exits to/from states and infinite sequence of transitions in a single time instant. It is easy to notice that because of this decision transitions happening simultaneously are constrained to be in different parallel components of the StateCharts. The effect of a transition can also be contradictory to its cause without any problem as the two refer to two different time instants. This approach (in accordance with [6] and in opposition to [19]) is particularly suited to the HW context as confirmed by the fact that a similar approach is found in HDLs (e.g. VHDL). Another central point is that of ***Interlevel Transitions*** and the use of ***History***. On one side semantics comprising these aspects tend to have problems in terms of ***compositionality***, because information regarding internal states needs to be exported. On the other side they allow to model in an intuitive way complex systems reducing the number of states. Therefore we have decided to support these characteristics in our model, the designer is free to use them or not depending on the kind of model and the level of the design procedure.

Negated Triggers and in general ***Logical Composition of Triggers*** are supported by our semantics, reverse polish notation is used. We do not use any implicit ***State Reference***: if the entrance, presence in a state and exit needs to be usable by other parts of the StateCharts, explicit events should be put respectively in entry, do and exit activities. This choice allows quicker simulations as a lot of redundancies are removed. ***Discrete Events*** are used, i.e., events are valid only in the instant they appear; given that ***Instantaneous States*** are not allowed in our semantics, duration of events is not an issue. The only priority scheme we have is that ancestor states transitions have priority over descendant states ones; non preemptive interrupt is used to this end. Transitions happen in null time, time can pass only within states. A timer has been used with the keyword ***timer(t)*** with the obvious meaning that after time ***t*** elapsed while in a given state the transition having the timer as a trigger executes.

Determinism of the model is left to the designer. In our formalism it is possible to specify non-deterministic behavior (e.g. two different events triggering two different transitions happen in the same instant). Whereas non-determinism is considered by some one of the main drawbacks of the formalism, it allows representing several aspects of complex systems. The increase in complexity (possible exponential explosion of states) is well compensated by the extended expressiveness. A central aspect is the injection of external code by means of ***Actions*** and ***Activities***. This may involve adding complex behavior and complex processing of variables. Given that a full concurrent environment is provided, racing condition on variables may appear. Blocking racing is not hard, but may not be the best solution, in particular when the order of execution of the accesses to variables does not influence its final value. In our semantics different accesses to the same variable in the same instant are put in sequence (in random order). There is the possibility – for debugging purposes – of detecting racing conditions.

16.4 Generation of Simulators

The overall framework has been developed exploiting a compiler like structure: it can be seen as composed of a front-end in charge of extracting all the useful information from the XMI – exported from the Poseidon[1] UML suite – and turning it into an Intermediate Representation (IR), and a back-end part where the IR is transformed in the SystemC code of the simulator. This kind of approach guarantees the possibility to easily adapt SC[2] to different XMI dialects just by modifying the front-end or obtaining the simulator's source code in another programming language just changing the back-end. Whereas the compiler-like structure has been inherited from [17] both the IR and the final template are deeply different and represent innovative contributions as will be detailed in the following sections.

16.4.1 Front End and Intermediate Representation

We decided to define IR through an XML-grammar, for the following reasons:

- It is easier to make transformation between different XML representations.
- Tools for parsing, syntax checking and translation of XML are available and free (e.g. we used Saxon[2]).
- XML Schema Definition (XSD) makes it possible to define a well-structured XML grammar and to validate it against any input XML file.

Whereas in our previous tools the IR was represented in Graph eXchange Language (GXL), we now decided to radically change approach. Taking into strict consideration the fact that no standard XML format for Statecharts exists we defined our own grammar – able to fully describe the Statecharts formalism – fitting the complete semantics. Moreover we provide – using XML Schema – debugging features, giving the possibility to validate a StateCharts model before the code generation process starts. As a first step we defined the grammar in the Backus-Naur Form (BNF) as shown in Fig. 16.1. Grammar definition was performed taking into consideration the compilation process. The StateCharts diagrams are seen as composed by a list of FSMs and additional information. Additional information consists of the variables used inside the FSMs and the events triggered – that must be declared beforehand. FSMs are considered as a list of states and a list of transitions. A state can be either a simple state, a FSM – in case of hierarchical states – or a parallel execution of multiple states. It is apparent that the symbol < andstate > in Fig. 16.1 is inessential, but it has been inserted in order to export some redundant information and make the automatic generation easier. Simple states and transitions are further

[1] http://www.gentleware.com

[2] http://saxon.sourceforge.net/

```
<statecharts> ::= <fsmList><additionalInfo>
<additionalInfo> ::= <varList><eventList>

<fsmList> ::= <fsm>|<fsmList><fsm>
<varList> ::= var|<varList>var
<eventList> ::= event|<eventList>event

<fsm> ::= <stateList><transitionList>
<stateList> ::= <state>|<stateList><state>
<transList> ::= <trans>|<transList><trans>

<state> ::= <fsm>|<andstate>|<simplestate>
<andstate> ::= <fsmList>

<simplestate> ::= entry_act do_act exit_act
<trans> ::= source destination trigger guards action
```

Fig. 16.1 StateCharts Backus Naur Form grammar

decomposed into atomic components as clearly shown. It is possible to use logical combination of events – expressed in reverse polish notation – as triggers.

The structure described above has been directly defined in terms of XSD rules. Having an XML Schema of the IR it is possible to validate syntactical and grammatical correctness of any instance through an already existing XML Schema validator (e.g. we used Saxon). This is a key feature – and represents an enhancement with respect to our previous releases and available solutions – as it allows early identification of a wide class of mistakes before compilation of SystemC code speeding up the debugging process. Whereas in previous versions information was only gathered from StateCharts diagram and variables were automatically recognized, in order to enhance capabilities of our models we use also Class diagrams. They serve for separating different parts of the model and give the possibility to have more details in declaration of variables and events. Information from Class Diagrams is collected in the < additionalInfo > element of the BNF grammar.

16.4.2 Back End and Generated Code

The previously described IR contains information for generation of all SystemC code. The process happens through a series of Extensible Stylesheet Language Transformation(XSLT) [8] passes that represents the most natural way to translate an XML format. The simulator is coded at RTL level. In Fig. 16.2 a piece of pseudocode illustrating the template of each < fsm > is shown. There is a one-to-one mapping between the number of < fsm > instances in the grammar in Fig. 16.1 and the number of such SC METHODs in the executable model. In case the StateCharts have a flat structure (i.e. without hierarchy) the number of < fsm > s is equal to the number of concurrent machines. In case hierarchy is used there is one more < fsm > for each substate – i.e. simple state or machine – in the model. This greatly reduces the number of SC METHODs running concurrently as on our first release there

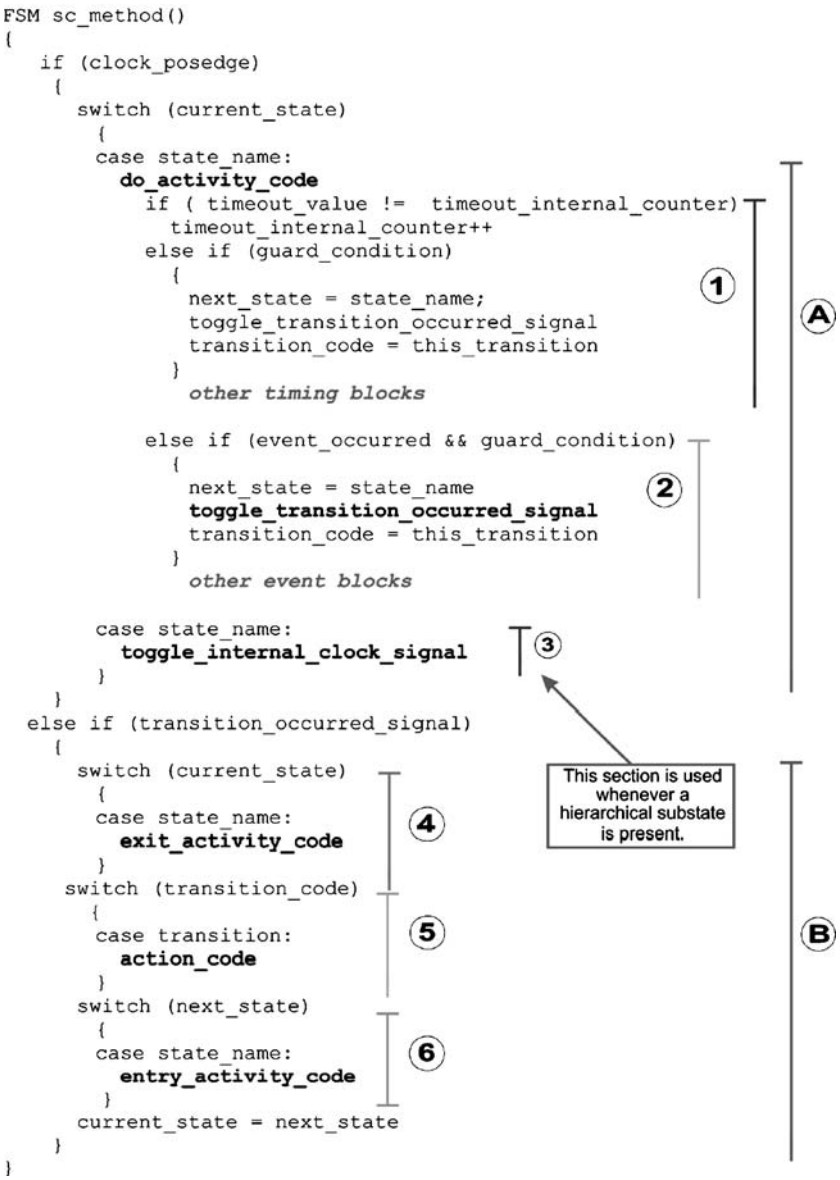


Fig. 16.2 Pseudo-Code template for each < fsm > in the StateCharts. timer blocks (1), events blocks (2), internal signals to propagate across substates (3) are the main sections of code devoted to management of states. The switches for selecting the appropriate exit activity (4), action (5) and entry activity (6) corresponding to a transition are highlighted in the bottom part

were as many SC THREADs as states. Reduction in complexity is even more robust as in [16] we found that it is possible to create a better performing system using two SC METHODs instead of each SC THREAD.

The SystemC model code is organized in SC MODULEs. Each SC MODULE corresponds to an Independent Concurrent State Machine (i.e. not forming an

ANDSTATE). Inputs and Outputs – i.e. variables and events – of these SC MODULEs are accurately defined, reducing therefore the number of ports (i.e. high simulation efficiency). The code is split into two parts (A and B in Fig. 16.2), the first one describes behavior while inside a state, the other during transitions. The operations inside the states are executed at the rising edge of the clock. At every clock cycle all the do activities are executed. Timer blocks – one per each outgoing transition triggered by a timer (t) – and events blocks – one per each outgoing transition triggered by an event – are checked. If conditions for a transition hold, the appropriate signal is toggled and it causes in the following cycle the execution of the code managing the transition. This code is very clear: a group of switches evaluates current state, transition code and next state in order to execute the right exit activity – dependent on current state –, action – dependent on transition code – and entry activity – dependent on next state. This ensures that transitions happen in null time and all the related code is executed in the same instant according to StateCharts semantics.

Hierarchy is treated with the use of multiple FSM SC METHODs in the same SC MODULE. There is no theoretical limitation on the depth of hierarchy. Anyway abuse of this possibility will slow down performance (as the number of concurrent SC METHODs increases). The clock is used by the first level of hierarchy, then each other level is sensitive to an "internal clock" triggered by the immediately lower hierarchical level. With this solution if a state has multiple hierarchy levels, all the levels are executed in consecutive cycles in the same time instant. This is not possible using only one clock. Restoring the initial state on exiting hierarchical machines is performed only if no history is present otherwise the last valid state is kept for the next entrance.

Management of events and variables is complicated by the presence of hierarchy. Even though we found a mechanism that ensures no time instants are lost in taking execution across the hierarchical levels, elapsing of cycles can cause erroneous processing of events. For dealing with such issue we designed a generic module that needs to be instantiated for each event. Events are represented by a logic one on the corresponding signal. Modules sample on the clock negative edge – so that elapsing of cycles does not have any impact – the OR of signals from all the FSMs that can fire the corresponding event.

This solution has a price in terms of higher complexity as every instantiated module for management of events costs one more SC METHOD. As long as variables are concerned there is the problem of multiple writing accesses to the same variable. Therefore a generic module (working as a bus arbiter) is necessary to take care of updating the variable value whenever a modification is required. If more modifications are required at the same instant only one can be performed (non deterministically chosen). Given that the application for our tools started from power estimation of complex systems – i.e. multiple states can give a contribute to power consumption in the same instant – there is the possibility of using a different kind of variable that can accept multiple inputs in the same clock cycle, resulting in the sum of the inputs. A module (i.e. a multi input adder) manages these of variables.

16.5 Multi-instantiation

We found that an important requirement that was not met by our previous versions [16, 17] and similar works [18, 20] is the possibility of creating multiple instances starting from a single model. This same exigence holds for many environments (e.g. systems following the client-server paradigm). Variables and events should be divided into two groups, those that are global and serve all the simulator and those that are only used by the instance that includes them. This way it is possible to make the instances work autonomously one to the other and interact only when they share access to global fields. As an example – in the context of wireless communication – all the devices wake up and listen when a beacon event is triggered, but in case of a single device communication the event that triggers its change of state should not trigger any other device. Our modular template allows for easy generation of multiple instances. In fact the issue is reduced to a routing problem, as the proper signals must be routed to the proper modules.

Global signals (variables) are routed everywhere therefore they are easily managed. Such a division requires the user to separate the variables/events inside the classes. The variables and the events that are declared public have global scope in the simulator, whereas those private are only visible inside the particular instance. On the other hand variables and events that are declared public can be accessed by all the machines. In the case of wireless communication systems – as an example – the channel and the synchronization signals are public, events causing a radio to transmit a message (or variables that indicate the length of the message) are obviously local to the instance that generates them. Classes are linked to StateCharts representing their behavior, in this way it is possible to group multiple instances functionalities just representing their behavior once. This is a major enhancement as in our previous works it was necessary to replicate StateCharts to have more instances, and moreover the management of events and variables of these replicas was cumbersome.

When generating the simulator it is possible to give one or more UML files as input. Each file contains a class diagram with the declaration of events/variables and the corresponding StateCharts diagrams. The framework creates an instantiatable object per file and it is possible to create as many instances of each one as needed. The variables/events in the class diagrams are checked per name, public variables/events of different files that have the same name are all grouped as a single variable. The main limitation of this scheme is that for the moment it is not possible to define relationships between single instances when the sharing of variables is involved. It is possible, e.g., to define a public channel variable that can be used by all the instances, but it is not possible to decide that two particular instances share a variable, whereas four others share another one. In order to obtain such behavior it is necessary to create a complex StateCharts that through some guard condition can decide in which way to operate. This is of course not optimal as it causes a noticeable increase in the number of events and variables, and complicates the design phase. We are planning to improve this aspect in the future.

16.6 A Simple Example

It is also useful to illustrate another new feature of our tool: the console shell. Whereas in previous versions it was necessary to create State Machines on top to pass events and act on variables, now it is possible to do these operations from the console line. This enhances usability of the tools specially when models are created as inserting various debugging patterns is much quicker (does not require modifying diagrams, creating the model and compiling it). The console is an interface between the user and the SystemC simulation engine. In order to illustrate all the concepts explained above we show a simple example (see Figs. 16.3 and 16.4). The scenario is that of pressure and temperature monitors. The environment is represented as a global machine. We just introduced an exemplary simple machine that changes temperature and pressure parameters following a simple pattern.

Monitors can access the pressure and temperature variables and have visibility over the events fired by the global machine, but they work independently one to the other. It is possible to instantiate as many monitors as desired. The number of < fsm > for this example is three for the environment (global) and five for each instance of the monitors. Therefore $3 + 5 \times$ #instances SC METHODs are necessary. It is necessary to add one SC METHOD per distinct variable and event.

```
Enter Command:
fire on_signal[1] 10
added firing event on_signal[1] time 10
LIST COMMAND INSERT EVT: on_signal[1]
Enter Command:
fire sample_evt[2] 10
added firing event sample_evt[2] time 10
LIST COMMAND INSERT EVT: sample_evt[2]
Enter Command:
fire read_evt[1] 11
added firing event read_evt[1] time 11
LIST COMMAND INSERT EVT: read_evt[1]
Enter Command:
go 15
... ... ... ... ... ... ... ... ... ... ... ...
... ... ... ... ... ... ... ... ... ... ... ...
fsm: device[1] ENTRY state: IDLE time: 10
fsm: device[2] ENTRY state: READING time: 10
... ... ... ... ... ... ... ... ... ... ... ...
... ... ... ... ... ... ... ... ... ... ... ...
time :10 THE PRESSURE IS 10
time :10 THE TEMPERATURE IS 0
... ... ... ... ... ... ... ... ... ... ... ...
... ... ... ... ... ... ... ... ... ... ... ...
fsm: device[1] ENTRY state: READING time: 11
... ... ... ... ... ... ... ... ... ... ... ...
   ... ... ... ... ... ... ... ... ... ... ... ...
```

Fig. 16.3 A brief example showing the use of the console shell is shown. Means of firing events (fire command) for the various instances and running the simulator (go command) are illustrated. Important lines have been extracted from output

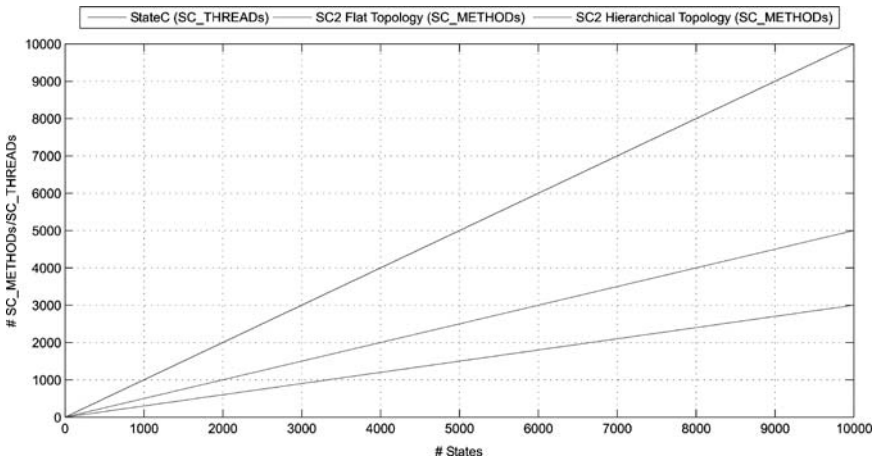


Fig. 16.4 StateCharts of an example. Monitoring devices can be instantiated, whereas the environment has global scope

It is very difficult to give an accurate performance comparison between this tool and older versions, as they operate in distinct way, and they can deal with a different subset of the formalism. Using flat hierarchy machines (the only ones manageable by our previous version) simplified events management is possible, but clarity of the models is underpinned. Moreover performance of the resulting simulator depends also on the kind of model. Therefore we give some general indication suggesting that the new approach is very beneficial in terms of performance. In Fig. 16.5 reduction in number of concurrent processes running is clearly shown. As far as execution time is involved the weight of a SC METHOD is about one third that of a SC THREAD.

16.7 Conclusions and Future Work

In this paper we discussed the new version of our tool. A lot of innovations have been introduced in the whole process and as a result the representable subset of StateChart is greatly extended (e.g. hierarchical states, history, interlevel transitions). Moreover an XML grammar for StateCharts has been designed and used as Intermediate Representation in the process of model generation. Resulting models are more powerful and new interesting features have been inserted as the possibility of instantiating multiple objects from a single model and the creation of a shell internal to the model for improving its easy of use.

Future work will involve refinement of template in order to map it to a VHDL synthesizable code, optimization to improve performance and improvement of the

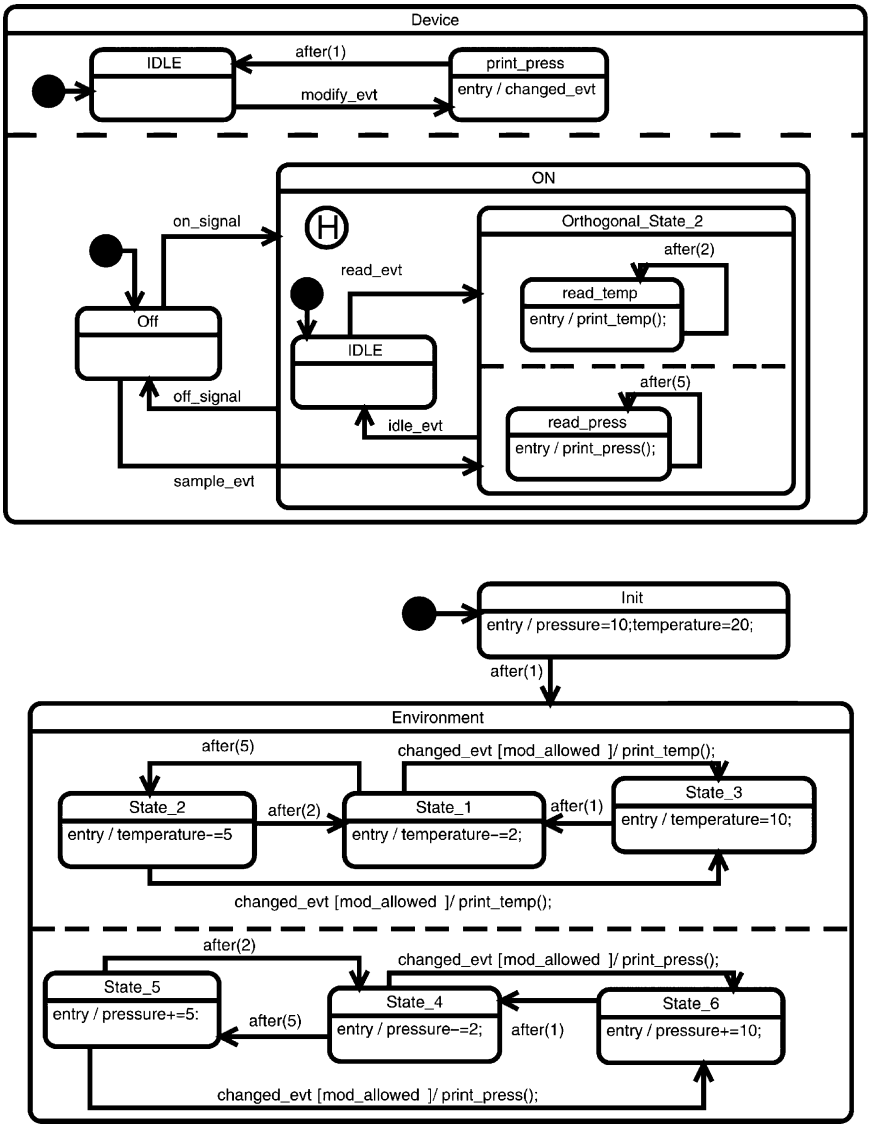


Fig. 16.5 Comparison in the number of generated concurrent processes per state in the model. The line has been drawn considering multi-instantiation in an average case

mechanism for multiinstantiation. In particular we will work on finding a mechanism for minimizing the concurrent SC METHODs running in case of StateCharts with hierarchy and on overcoming the limitation in model instantiation.

Acknowledgement The authors would like to thank Professor Marc Engels for his precious advices, his kind support and his valuable feedback.

References

1. http://www.ilogix.com/sublevel.aspx?id = 74.
2. http://www.mathworks.com/products/stateflow/.
3. M. Von Der Beek. A comparison of StateChart variants. In *Formal Techniques in Real-Time and Fault tolerant Systems*, 1994.
4. F. Fummi, M. G. Sami, and F. Tartarini. Use of Statecharts-Related description to achieve testable design of control subsystems. In *Proc. GLSVLSI*, 1997.
5. S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Formal Aspects of Computing*, 51, 2002.
6. D. Harel and A. Naamad. The STATEMATE semantics of StateCharts. *ACM Transactions on Software Engineering and Methodologies*, 1995.
7. D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 1987.
8. M. Kay. *XSLT 2.0 Programmer's Reference* (*Programmer to Programmer*). WROX, 3 edition, Aug. 2004.
9. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker. *Journal of Logic and Algebraic Programming*, 11, 1999.
10. J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. *ase*.
11. Grant Martin. SystemC and the future of design languages: Opportunities for users and research. In *Proc. SBCCI*, 2003.
12. G. De Micheli and R. K. Gupta. Hardware/Software co-design. In *IEEE Proceedings*, Mar. 1997.
13. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in promela/spin. In *Proc. WIFT*, 1998.
14. M. Mura. Ultra-low power optimizations for the ieee 802.15.4 networking protocol. In *Proc. MASS*, 2007.
15. M. Mura, M. Paolieri, F. Fabbri, L. Negri, and M. G. Sami. Power modeling and power analysis for IEEE 802.15.4: a concurrent state machine approach. In *Proc. CCNC*, 2007.
16. M. Mura, M. Paolieri, L. Negri, and M. G. Sami. Statecharts to SystemC: a high level hardware simulation approach. In *Proc. GLVLSI*, 2007.
17. L. Negri and A. Chiarini. StateC: a power modeling and simulation flow for communication protocols. In *Proc. FDL*, Sept. 2005.
18. K. D. Nguyen, Z. Sun, P. Thiagarajan, and W. Wong. Model-driven SoC design via executable UML to SystemC. In *Proc. RTSS*.
19. A. Pnueli and M. Shalev. What is in a step: on the semantics of StateCharts. In *Proc. TACS*, 1991.
20. Chen Xi, Lu JianHua, Zhou ZuCheng, and Shang YaoHui. Modeling SystemC design in UML and automatic code generation. In *Proc. ASP-DAC*, 2005.

Chapter 17
Asynchronous On-Line Monitoring of Logical and Temporal Assertions

K. Morin-Allory[1], L. Fesquet[1], B. Roustan[2], and D. Borrione[1]

Abstract PSL is a standard formal language to specify logical and temporal properties under the form of assertions. This paper presents the synthesis of PSL assertions into asynchronous hardware monitors that can be linked to the circuit under monitoring. The checker synthesis is based on a systematic interconnection of asynchronous primitive monitors corresponding to PSL operators. The asynchronous monitors are implemented with quasi delay insensitive logic which gives reliable and robust monitors in the case of truly asynchronous events, temperature or voltage variations. These monitors are applicable to a wider range of verification tasks such as the communications among globally asynchronous modules or in safe or secure applications.

Keywords PSL, SVA, hardware monitors, asynchronous circuits

17.1 Application Context

New design paradigms are required for large systems on a chip, among which the systematic use of software and hardware "platforms", and rigorous specification, verification and test methods. In this context, the use of declarative assertions, to specify the expected functional and temporal properties of modules and/or their environment, is recognized as a valuable, time saving technique [12] that can be carried across description levels and serve a wide range of usages. Assertions are useful for specifying constraints for correct IP utilization, the results delivered by IPs, the correct expected design behaviors, etc. As a Boolean property expected to

[1]TIMA Laboratory, 46 avenue F. Viallet, 38031 Grenoble, France;
Email: {name.surname}@imag.fr

[2]ENSERG/INPG, 3 parvis L. Néel, 38016 Grenoble, France; Email: roustanb@enserg.fr

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

be true, an assertion can be evaluated by simulation, emulation or formal verification. An assertion can also be seen as a high level functional specification for a circuit primarily intended for snooping on events over time.

Several formalisms have been developed to ease writing temporal and logical properties, among which SystemVerilog Assertions and PSL are IEEE standards [13, 14]. Synthesizing an asserted property as a *monitor*, and interconnecting the design and the monitor, is a common technique to design validation and online circuit testing that promises to become increasingly useful for large embedded systems.

The on-going work reported in this paper aims at automatically generating truly asynchronous and synthesizable monitors from PSL assertions, for online checking of circuits in normal operation. Moreover, the monitors can easily be simulated and emulated on a hardware platform. The design debugging on a FPGA board is also an obvious application of our method, with the advantage of permitting full operation speed.

In this context, many applications are foreseen. Some examples are given below:

- Monitoring large systems built from synchronous IP's: one difficulty in debugging "globally asynchronous locally synchronous" systems is the correctness of communications. Asynchronous monitors are needed to pinpoint erroneous transactions between modules that belong to different clock domains.
- Monitoring inherently asynchronous events, guaranteeing that an appropriate response is given, irrespective of the events delay.
- Safely monitoring circuits in harsh environments thanks to the intrinsic robustness of asynchronous logic.
- Monitoring secure chips, such as cryptoprocessors, in order to detect side-channel attacks using fault injections.

17.2 State of Art

FOCS from IBM [1, 6] was, to our knowledge, the first tool to automate the generation of register-transfer level (RTL) monitors from PSL, producing VHDL or Verilog code that can be linked to the design at hand for checking on a clock cycle basis. Although primarily intended for on-line simulation, including mixed signal simulation by other parties [2], monitors produced by FOCS are synthesizable, and can be fed to a model checker. The principles for building syntax directed monitors for clock synchronized "foundation language" PSL expression [7] and SERE's [9, 15] have been disclosed with a particular emphasis on debugging feature [8, 15]. A more formal automata theoretic construction of monitors, the so-called "temporal testers", are also built in a compositional way [17]. Cimatti et al. [10] propose another modular encoding to turn PSL properties into nondeterministic Büchi automata. Other tools are now provided by the main CAD companies, that interface

several verification engines (simulation, emulation, formal verification); various libraries of predefined checkers (CheckerWare [3], OVL [4]) are supported in addition to the standard assertion languages.

To the best of our knowledge, all synthesized checkers are clock synchronized. Checkers that pretend crossing multiple clock domains, e.g. [5], appear to be hardwired special purpose modules rather than generated from general assertions. Yet, at system level, one needs to write properties that are triggered by asynchronous events, such as interrupts, or that check communication protocols among globally asynchronous modules. The early published solutions generate software checkers linked to design models in C++ or in SystemC, that are verified by simulation [11].

17.2.1 A Modular Construction

The method we propose is modular. We started with the method initially developed in [7] for synchronous designs. We thus created a completely new library of primitive digital asynchronous components for the basic PSL temporal operators, and an interconnection technique based on hand-shaking protocols. The novelty of our approach lies in the fact that the advancement of time is seen as a sequence of events on arbitrary signals instead of occurrences of a single master clock ticks. Signal changes, rather than clock ticks, are thus considered the points in time when PSL formulas are to be evaluated. This paper is an extension of an early article [16] with some experimental results.

17.2.2 Asynchronous Logic Benefits for Monitoring

While in synchronous circuits a clock globally controls activity, asynchronous circuits activity is locally controlled using communicating channels able to detect the presence of data at their inputs and outputs. This is consistent with the so-called handshaking or request/acknowledge protocol. One transition on a request signal activates another module connected to it. Therefore, signals must be valid at all times. Asynchronous circuit synthesis must be more strict, i.e. hazard-free. In order to have very reliable monitors, we choose to implement Quasi-Delay Insensitive (QDI) circuits [18]. Indeed, these circuits are very robust to Process, Temperature and (strong) Voltage variations. Moreover, they offer nice properties such as modularity or low-power consumption.

In contrast to synchronous circuits, the QDI circuit synchronization is made locally with two asynchronous signals: a request signal and an acknowledge signal. This is done with a Muller gate which implements a “rendezvous” between these two signals. When all inputs of a Muller gate (Fig. 17.1) are equal, the output takes the input value. When inputs are different, the output holds its previous value (see Table below).

A	0	0	1	1
B	0	1	0	1
Z	0	Z^{-1}	Z^{-1}	1

A	0	0	1	1
B	0	1	0	1
Z	0	Z^{-1}	Z^{-1}	1

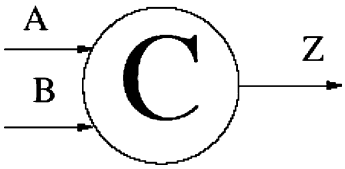


Fig. 17.1 A Muller gate

17.3 Property Specification Language

We briefly recall how properties are written, to underline the subtle differences between the synchronous and asynchronous interpretation of PSL formulas. A property is built on three types of building blocks: the Boolean expressions, the sequential expressions (SERE) that define finite-length regular patterns (called sequences) of Boolean expressions and subordinate properties that express relationship among Boolean or SERE expressions. Various operators called Foundation Language (FL) operators express temporal relationships: `until`, `always`, `before`, ... In this paper we focus on the FL operators. Our work is based on the formal semantics of the operators, defined on traces, and given in [13]. To make this paper self contained, and understandable, we briefly give an intuitive definition on a small example.

Consider the following property `P1`.

```
PSL property P1 is
  Always A → next(B until C);
```

Property `P1` means that for each evaluation cycle such that '`A`' holds, at the following evaluation cycle `B` must remain '1' until `C` holds.

The PSL semantics are defined on a trace, and some evaluation cycle. In a synchronous design, the evaluation cycle can be clock driven (`@ clock'event and clk='1'`), but in an asynchronous design, the evaluation cycle may be event driven. Thus, for a same trace a property can hold or fail. The two waveforms on Fig. 17.2 illustrate two evaluations on a same trace for property `P1`.

Top waveform: At clock edge #2 and #7, *A* holds. Starting from the next evaluation cycle (#3 and #8) *B* must hold until *C* is'1'. The property is not verified since *B* does not hold at #8: the second evaluation fails and the whole assertion is not verified.

Bottom waveform: For this waveform, the evaluation cycle is event driven: each time there is an event on one of the signals involved in the property, the property is

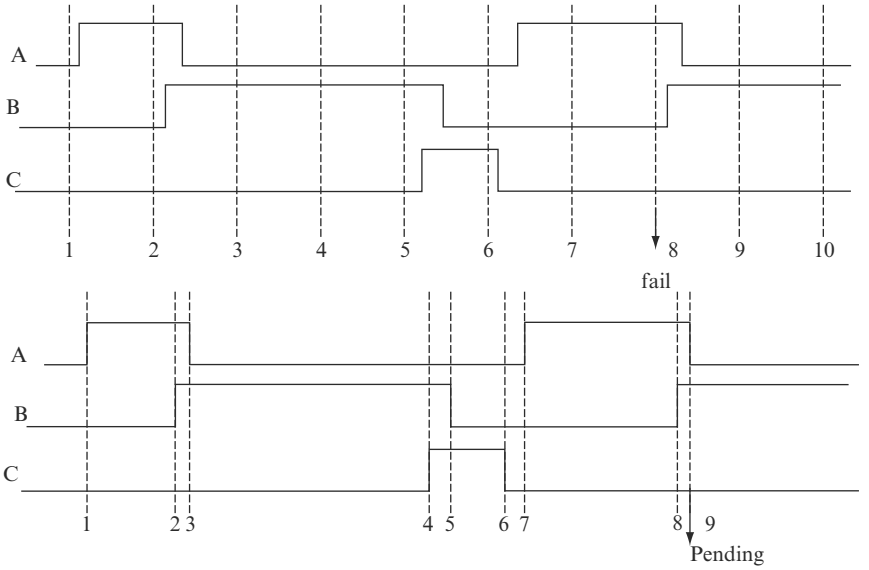


Fig. 17.2 A synchronous and an asynchronous evaluation of `P1`

evaluated. At event #7, *A* is asserted, and on the next event *B* is '1' and remains '1' until the end of the trace. Since *C* is '0', the property is pending: an extension of the trace may lead to an error or not.

The asynchronous solution we propose supports both evaluation cycles.

17.4 Monitor Generation

The monitors we build reflect the four satisfaction levels for a property : hold strongly, hold, pending and fail [13]. When implemented in hardware, the monitor outputs display the property satisfaction level, and the indication that the answer is no longer pending may be used as an interrupt to trigger further actions.

A monitor for a property P is built as a module that takes as inputs the reset, the synchronization signals (clock, hand-shake, etc.), a signal *Start* that triggers the evaluation, and the signals of the design under verification (DUV) that are operands of the temporal operators in P (see Fig. 17.3). The three monitor outputs have the following significance:

- *Checking*: a 1 indicates that output *Valid* is effective at the next synchronization time;
- *Valid*: provides the evaluation result (1 means absence of error, 0 means error);
- *Pending*: a 1 indicates that the monitor has been started and that the satisfaction result is pending; this is significant for strong operators.

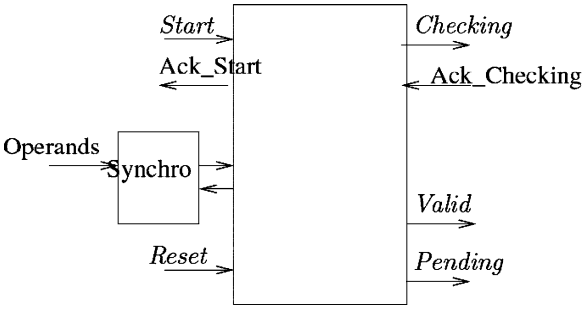


Fig. 17.3 Interface of a monitor

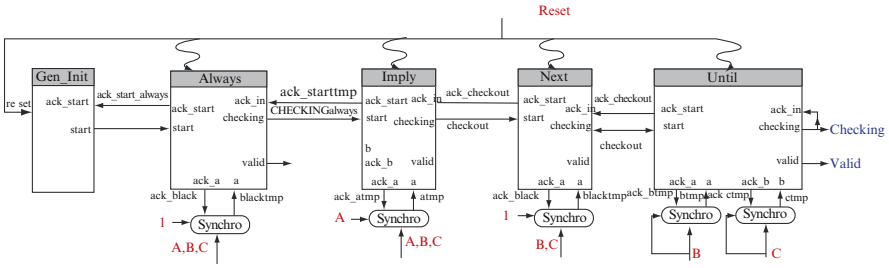


Fig. 17.4 Property monitor for P1

The synthesis method relies on:

- A library of primitive monitors, one for each PSL operator of the "foundation language".
- A systematic connection procedure to build complex monitors from primitive ones, based on the PSL expression syntax tree.

Operators that take one or two integer parameters, such as *next* or *next_a*, have corresponding generic monitors with the same parameters. In addition some operators have several variants: weak or strong, overlapping or non overlapping (e.g. *before*), in effect corresponding to several primitive monitors. All primitive monitors have, maximally, the interface shown on Fig. 17.3: there may be 1 or 2 operand inputs, there may be a pending output or not.

Figure 17.4 illustrates the construction of the monitor for property. In this monitor, we have chosen to add an event driven synchronization block. For each primitive monitor, this block takes as input the operand of the primitive monitor and all the signals involved in the sub-formula as synchronization signals: e.g. the operator "`next`" takes no operand as input (connected to '1'), and *B*, *C* as synchronization signals since they are involved in the subformula of *P*1: `next(B until C)`. This synchronization block can be substituted by any synchronization block even by a clock driven synchronization block.

As an example of library primitives, the *imply* operator is presented. The property semantics are first expressed as a Petri Net (see Fig. 17.5). The primitive monitor is then synthesized from this Petri Net description into a gate netlist including standard logical gates and Muller gates. This approach fits naturally with asynchronous logic, where an arbitrary number of modules can be assembled by means of the handshake protocol, preserving delay insensitivity.

Figure 17.6 shows the sub-circuits (identified with dashed lines) corresponding to the places in the Petri net. The three boxes contain a very simple structure which

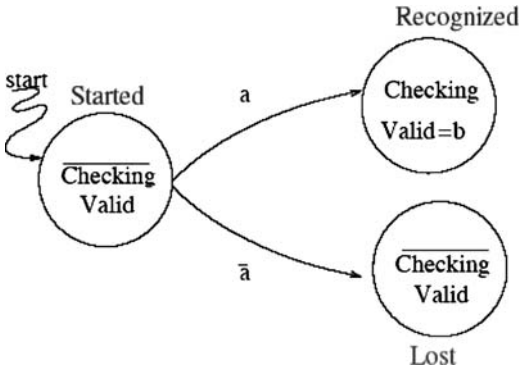


Fig. 17.5 Petri net of the imply operator

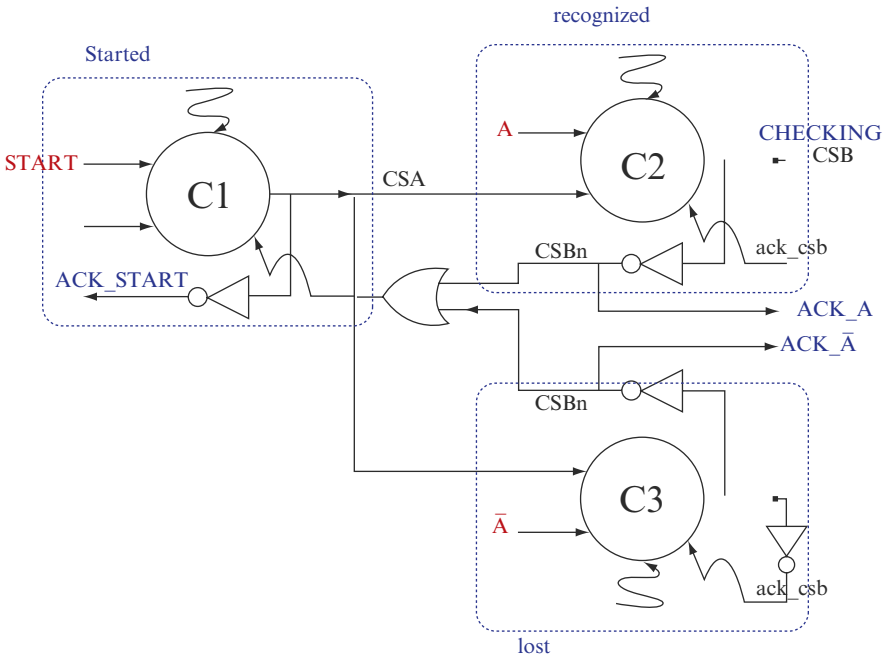


Fig. 17.6 Monitor of the imply operator

implements a rendezvous between a state signal (similar to a token in a Petri Net) and a condition signal allowing a transition between two states. This is realized with a 3-input Muller gate and an inverter. For instance, the 3-input Muller gate, located in the *Started* box, is set to 1 when the current state is *Started*. The transition to states *recognized* and *lost* is conditioned by the value of signal A.

Assume that all the Muller gate outputs are set to 0, except the output of *C1* (the monitor is in state *Started*). Assume that A is '1'. All the inputs of *C2* are 1 (the acknowledgment signal of the following state is also 1) and the output of gate *C2* goes up. The acknowledgment signal, connected to the inverted value of the Muller gate output, resets the preceding *Started* state. This is interpreted as a state change from *Started* to *Recognized*.

Last, the monitor output *Checking* is directly computed with the gate *C2*.

17.5 FPGA Implementation

17.5.1 Implementation of Assertion Monitors

To implement PSL assertions in a digital system, the designer follows the standard design flow (HDL description, synthesis, place and route) as illustrated on Fig. 17.7. The PSL assertions are extracted from the system specification. Once the PSL assertions have been extracted, the monitors are automatically generated by our dedicated platform HORUS, resulting in a netlist of property monitors. This checker netlist is then merged with the IP to be monitored using HORUS. The next steps follow the standard design flow and target FPGAs as well as ASICs.

Monitors implemented in ASIC are primarily devoted to on-line testing of the circuit in operation. In FPGA, the monitors can be used to detect design errors at the hardware or software level, the primary interest being several orders of magnitude in the verification speed compared to a simulation execution.

17.5.2 A Bus Snoop-System for Software Verification

To demonstrate the hardware asynchronous monitor principles on a real system, an experimental platform, based on an Altera FPGA (a Stratix 1s40), has been designed. The implemented architecture is described in Fig. 17.7. The Nios-Avalon architecture is based on a standard Avalon bus and has an UART serial interface, a Nios processor with a RAM and a boot ROM. The hardware monitors are connected to the bus through a small interface in order to snoop the data transactions about which the PSL properties are written. The interface also allows the Nios processor to scan the state (Pending, Hold, Fail) of the monitors. Figure 17.7 displays an experiment with one asynchronous monitor.

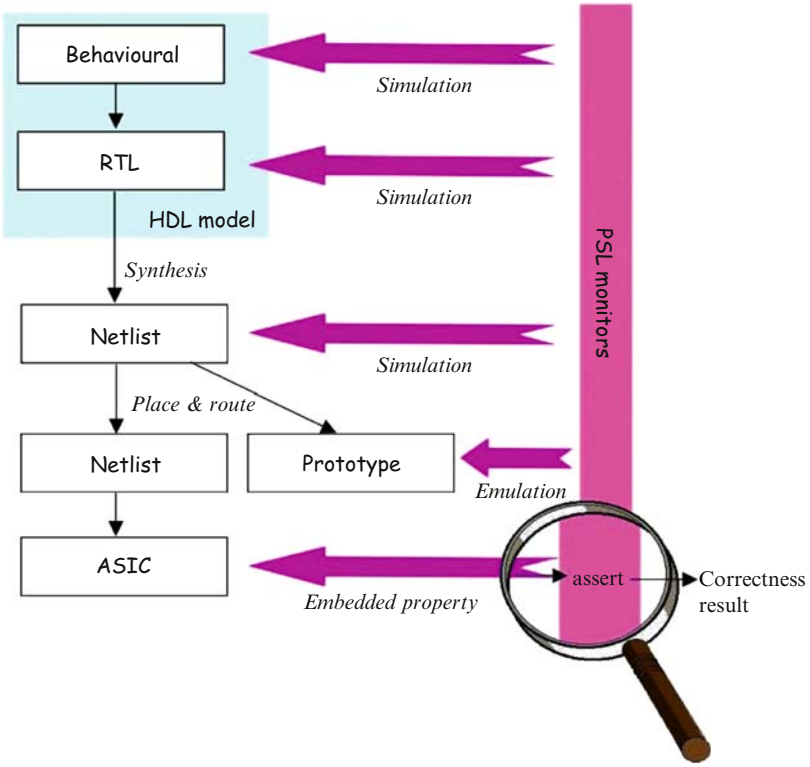


Fig. 17.7 Design flow of the Horus platform

A host computer is used to load the hardware on the FPGA (with a JTAG link not represented on Fig. 17.8). Then, the software is downloaded through the UART link and executed on the Nios processor.

Each monitor snoops its own set of signals on the Avalon Bus, and evaluates a particular property. After monitors are started, as long as all hardware monitors are in pending state, the Nios executes its program normally. When a monitor detects a Hold or Fail condition, an interrupt is generated and the Nios processor executes an exception handler. The interrupt routine performs appropriate actions for debug, e.g. read the state of the implied monitor and display it on the host computer.

17.6 Conclusion

This article aims to synthesize asynchronous checkers, described in an assertion language such as PSL or SVA, not only for debugging during simulation or emulation but also for ASIC online monitoring. The main advantages of asynchronous checkers are their intrinsic robustness to process, temperature and voltage variations

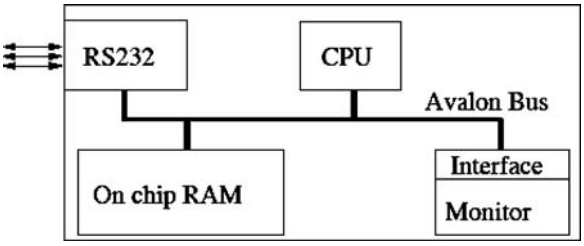


Fig. 17.8 The Nios architecture connected to one asynchronous monitor

thanks to the QDI logic. In these conditions, an abnormal behavior of the monitored circuit can be detected even if its power supply voltage is not high enough to ensure a correct functioning. Indeed, the asynchronous monitor functional correctness is warranted in a large voltage range (typically from 1.2 to 0.4 V for a 130 nm CMOS process). This can be used for monitoring critical IPs in safe or secure applications. Moreover, the delay insensitivity allows a reliable verification of transactions between modules that belong to different clock domains. The monitor generation is based on a systematic interconnection of asynchronous primitive monitors corresponding to PSL operators of the "foundation language". This approach has been successfully prototyped on standard FPGA platforms. Further works will address the monitor generation of SEREs.

References

1. www.haifa.il.ibm.com/projects/verification/Formal_Methods-Home/index.html.
2. www.dolphin.fr/medal/smash/flash/smash_flash.html.
3. www.mentor.com/products/fv/abv/0-in/index.cfm.
4. www.accellera.org/activities/ovl/.
5. www.mentor.com/products/fv/abv/0-in-cdc/index.cfm.
6. Y. Abarbanel et al. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Computer Aided Verification*, volume 1855 of *LNCS*, ISBN:3-540-67770-4, pages 538–542, Springer, London, 2000.
7. D. Borrione, M. Liu, P. Ostier, and L. Fesquet. PSL-based online monitoring of digital systems. In *Advances in Design and Specification Languages for SoCs – Selected Contributions from FDL'05*. Springer, London, 2006.
8. M. Boulé, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *24th IEEE International Conference on Computer Design (ICCD'06)*, 2006.
9. M. Boulé and Z. Zilic. Efficient automata-based assertion-checker synthesis of psl properties. In *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, Nov. 2006.
10. A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: a Modular Symbolic Encoding. In *Formal Methods in Computer Aided Design, FMCAD'06*, ISBN 0-7695-2707-8, pages 125–133, IEEE Computer Society, San Jose, CA, Nov. 2006.
11. A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. Combining system level modeling with assertion based verification. In *ISQED*. IEEE Computer Society, 2005.

12. H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer, Dordrecht, The Netherlands, June 2003.
13. IEEE Computer Society. *IEEE Standard for Property Specification Language Reference Manual, (PSL)*, Oct. 2005.
14. IEEE Computer Society. *SystemVerilog IEEE Std 1800–2005*, 2005.
15. K. Morin-Allory and D. Borrione. On-line monitoring of properties built on regular expressions sequences. In *Forum on specification & Design Languages (FDL'06)*, Sept. 2006.
16. K. Morin-Allory, L. Fesquet, and D. Borrione. Asynchronous assertion monitors for multi-clock domain system verification. In *IEEE International Workshop on Rapid System Prototyping*, pages 98–102. IEEE Computer Society, Chania, Crete, 2006.
17. A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In J. Misra, T. Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, *Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, Hamilton, Canada, August 21–27, 2006.
18. J. Sparsø and S. Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer, Dordrecht, The Netherlands, 2001.

Chapter 18
Transactor-Based Formal Verification of Real-Time Embedded Systems

D. Karlsson, P. Eles, and Z. Peng

Abstract With the increasing complexity of today's embedded systems, there is a need to formally verify such designs at mixed abstraction levels. This is needed if some components are described at high levels of abstraction, whereas others are described at low levels. Components in single abstraction level designs communicate through channels, which capture essential features of the communication. If the connected components communicate at different abstraction levels, then these channels are replaced with transactors that translate requests back and forth between the abstraction levels. It is important that the transactor still preserves the external characteristics, e.g. timing, of the original channel. This chapter proposes a technique to generate such transactors. According to this technique, transactors are specified in a single formal language, which is capable of capturing timing aspects. The approach is especially targeted to formal verification.

Keywords Transactor, formal verification, petri-net, regular expressions, embedded systems

18.1 Introduction

Developers of embedded systems face an ever-increasing complexity of their designs. At the same time, they also face an ever-decreasing time-to-market. A common way to deal with this challenge is to divide the design into several components, each component with its own responsibilities and functionality.

This divide-and-conquer technique is usually combined with an iterative top-down approach, where the system is initially defined at a high level of abstraction, leaving out most low-level details. The design is then gradually refined and more and more details are put into place. During this process, some parts of the system will be described at high level and other parts at low level.

Department of Computer and Information Science, Linköpings universitet, 58183 Linköping, Sweden; Email: {danka, petel, zebpe}@ida.liu.se

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

This situation, together with the fact that verification and test consume a significant part of the total development cost, stresses the need for efficient verification methods that target systems described at mixed abstraction levels.

The above-mentioned problem is traditionally solved in an unsystematic manner, where developers rewrite properties and modify the system in an ad hoc manner in order to match the mixed level model. Lately, a more systematic approach, involving transactors, has been proposed [4, 5].

The key issue of the problem lies in the fact that two (or more) components described at different abstraction levels cannot communicate with each other, since they, in principle, use different protocols. One component uses a more high-level protocol than the other. A transactor is a mechanism that bridges this gap by translating the high-level requests into their low-level ditto and vice versa. Moreover, evaluations have shown that using a transactor-based verification approach is more effective than a traditional RTL verification flow with respect to both fault and assertion coverage [1]. Using transactors moreover helps in reusing testbenches as well as assertions in the refinement process.

A few works have been performed in the area of automatically generating this type of transactors, based on protocol conversion techniques [2, 3]. Bombieri et al. [4] start from a master-bus-slave communication framework that contains information on how communication is carried out at different abstraction levels on the specified infrastructure (bus). From this framework, the authors extract a master, bus or a slave transactor from a high to low level or vice versa. Their extraction algorithm is based on Extended Finite State Machines. It does, however, not handle timing aspects explicitly and is only applicable on bus-based protocols.

Balarin et al. [5] use Sequential Extended Regular Expressions (SERE) to specify the relation between the two interfaces of the transactor and to automatically generate the corresponding transactor. The transactors are generated in a programming language such as C++, Verilog or SCE-MI, in order to facilitate integration with existing simulation tools. The approach supports to a lesser extent formal methods, and it completely lacks the support for time.

Protocols are often described using various kinds of regular expression-like languages. Although SEREs [5] in principle are sufficiently expressive, they do not support the notion of time. Timed Regular Expressions [6], on the other hand, lack several useful features, such as variables and conditions.

The approach proposed in this chapter combines SEREs with timed regular expressions by adding a timing feature on top of SEREs. We call the resulting language Timed SERE (TSERE). By doing this, we are able to create transactors suitable for formal verification in a component-based real-time setting with mixed abstraction levels. The approach moreover widens the scope of responsibility of transactors from a pure protocol converter to a semirefined communication channel.

The chapter is organised into seven sections. Section 18.1 introduces and motivates transactor-based verification. Next, Section 18.2 provides an overview of the proposed approach. Section 18.3 presents the Petri-net based design representation that is used throughout the chapter, and Section 18.4 defines the Timed Sequential

Extended Regular Expression language that is used for specifying transactors. Section 18.5 describes the mechanism to generate timed Petrinets from the formal description and Section 18.6 presents a few case studies. Section 18.7 concludes the chapter.

18.2 Overview

In the proposed approach, a system consists of several communicating components, as indicated in Fig. 18.1. Each component implements a well-defined functionality, and they interact with other components and the rest of the system through ports, depicted in the figure with circles at the edges of the component.

Channels are inserted between communicating components. The channels model the protocol, delays, noise and other peculiarities that can occur in the communication. They are hence only an artefact for high-level models, that will not occur or be synthesised in the final implementation. Channels can, from a modelling point of view, be regarded as a special type of components, and are depicted with dotted lines.

During the development phase, it is often desirable to check if certain temporal logic properties are satisfied in the system. Such analysis can be obtained by feeding a model of the system into a model checking tool together with properties to be verified. This procedure gives a formal proof whether the properties are satisfied in the system or not [7].

At the same time, the components are iteratively refined and more and more details are added to the system. This naturally leads to a situation where some parts of the system are more refined than others. However, it is still desirable to occasionally verify the system to ensure that the recently performed refinement steps did not violate any, possibly critical, properties.

When refining the components, the interfaces of those components are simultaneously refined. However, the interfaces are shared or connected with other components, that are not yet refined. This creates an incompatibility of interfaces between the involved components and channels. In order to overcome this problem, the channel is replaced by a transactor between the incompatible interfaces, as

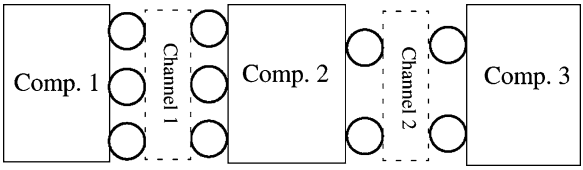


Fig. 18.1 Targeted system topology

demonstrated in Fig. 18.2. A transactor can thus be seen as a channel connecting components at different levels of abstraction, or a semi-refined channel. The transactor shall encapsulate the same external behaviour as the channel it replaces with respect to delays, noise, etc.

The transactor takes high-level requests and translates them into low-level ones, and vice versa. It is described in Timed Sequential Extended Regular Expressions (TSERE), which is both intuitive and sufficiently expressive for this purpose. The TSEREs (and thereby also the transactors) are given either by the designer himself, or, in a standardised context, by a third-party provider.

The example in Fig. 18.3 will be used to explain the approach in more detail. A sender repeatedly sends messages to a receiver over a channel. At a high level of abstraction (Fig. 18.3(a)), it takes 2 time units for the message to be transported

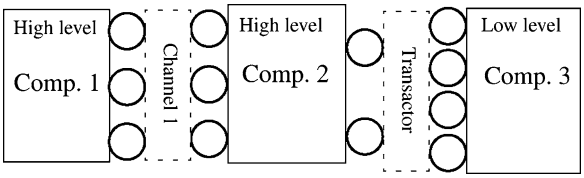


Fig. 18.2 System at mixed abstraction level with transactor

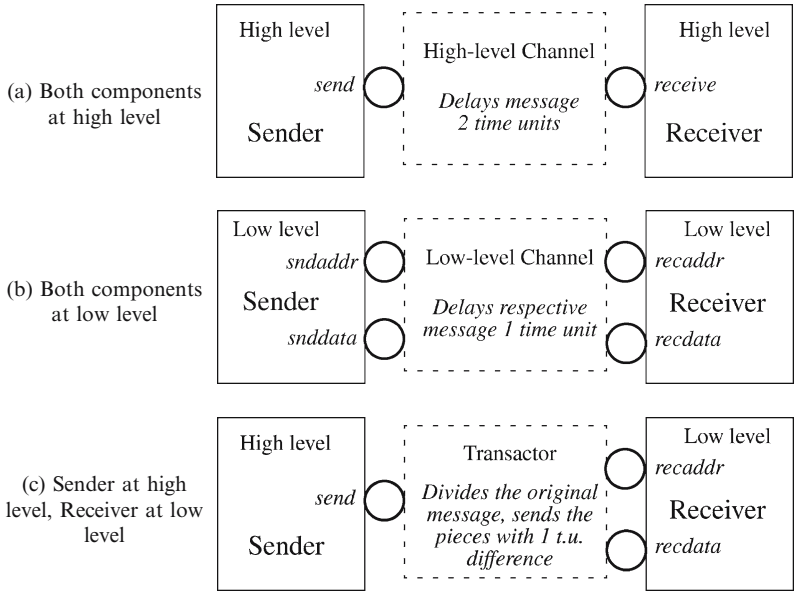


Fig. 18.3 Explanatory example

between the two components. This delay is implemented in the channel interconnecting the components.

At a low level of abstraction (Fig. 18.3(b)), the message is refined into two: address and data. The protocol that the sender and receiver have agreed upon states that these messages should be sent sequentially with 1 time unit in between. It moreover takes 1 time unit for each message to reach the receiver. The sender thus sends the data at the same time as the address reaches the receiver. It should be noted that the total timeframe for sending a message in the two abstraction levels is the same. In both cases, this takes 2 time units. Thus, the channel preserves its external behaviour between abstraction levels.

At one moment, during the refinement phase, only one of the components is refined. Assume that this component is the receiver (Fig. 18.3 (c)). At this stage, the sender and receiver adhere to different protocols and cannot communicate with either of the high-level or low-level channels. Instead, the channel is replaced with a transactor that translates the high-level message into the stipulated sequence of low-level ones. The transactor consequently has to analyse the message from the sender and divide it into two. The first message should contain the destination address, whereas the second one should contain the data. The transactor then forwards the two pieces to the receiver with 1 time unit difference.

The transactor can be said to be a mix of the two versions of the channel. It, however, also contains additional protocol information not explicit in the channels, e.g. how to split the high-level message and the time separation between the address and data transmission. Therefore, the information captured in the channels is not sufficient for formulating the TSEREs. In addition, the transactor respects the external timing behaviour of the channels.

18.3 Verification Flow and Design Representation

This section introduces the verification flow and the Petri-net based design representation used in this chapter.

18.3.1 Verification Flow

Figure 18.4 presents the overall verification flow where the work described in this chapter is put into context. The flow centers around a component-based verification methodology [7], which accepts three entities as input: a mixed-level model, transactor and Timed Computation Tree Logic (TCTL) properties [8].

The mixed-level model is obtained from traditional refinement steps of a high-level model. The designer then writes TSEREs describing the communication discrepancies arisen from the mixed abstraction levels in the semirefined design and generates a transactor out of them (the focus of this chapter). The TCTL formulas express the real-time properties to be verified.

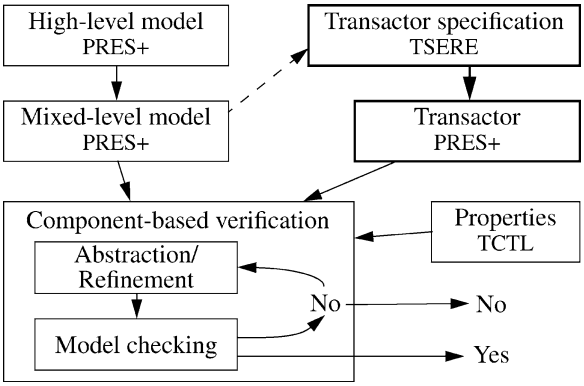


Fig. 18.4 Verification flow

In the verification methodology, an abstraction of the model is first obtained with respect to the components and channels referred to by the properties. The abstracted model is then input to the UPPAAL model checker [9], by first translating the Petri-net model [10] into Timed Automata [11], the input language of UPPAAL. If the result of the model checking was false, the model might need to be refined (relative to the abstraction done in the verification methodology, not the design itself) based on diagnostic information obtained from the model checker. In case the refinement of the abstraction fails, the properties are concluded not to be satisfied. If, on the other hand, the model checking result was true, it can be concluded that the properties hold in the model.

18.3.2 The Design Representation: PRES+

The components as well as the system as a whole are assumed to be modelled in a design representation called Petri-net based Representation for Embedded Systems (PRES+) [10]. It is a Petri-net based representation with the extensions listed below. Figure 18.5 shows an example of a PRES+ model.

1. Each token has a value and a timestamp associated to it.
2. Each transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. If the time delay interval is not explicitly stated, it is assumed to be [0..0]. In Fig. 18.5, the functions are marked on the outgoing edges from the transitions.
3. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.
4. The transitions may have guards. A transition can only be enabled if the value of its guard is true (transitions t_4 and t_5).

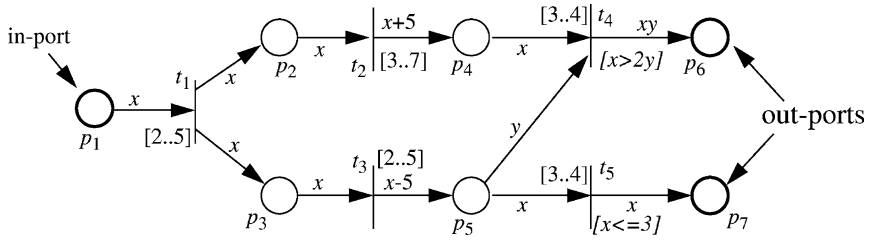


Fig. 18.5 A simple PRES+ net

Places without incoming arcs are called in-ports, and places without outgoing arcs are called out-ports. A common name for in-ports and out-ports respectively, is ports. Components are subnets of the whole model, delimited by ports.

18.4 Timed Sequential Extended Regular Expressions

The proposed approach introduces Timed Sequential Extended Regular Expressions (TSEREs) for the specification of transactors. TSEREs consist of three types of entities: basic entities, terms and operators.

18.4.1 Basic Entities

Basic entities cannot be standalone TSEREs, but constitute a part of terms. They are used as building blocks for storage, communication and computation. The three categories of basic entities are shown below:

1. Variables: a, b, c

   Variables are used to store and retrieve values. Variables are associated to a datatype. Unless explicitly stated otherwise, the datatype used in all examples is integer. The scope of a variable stretches from its first occurrence to the end of the sequence (see the sequence operator below) of that first occurrence.

2. Port labels: *!send*, *?rec*

   Port labels are used to define the interaction with other components. *!* denotes the sending of a (possibly empty) message on the subsequent out-port, and *?* denotes receiving of a message from the specified in-port.

3. Arithmetic expressions: $(a + b) \cdot 3$

   Arithmetic expressions perform a computation on other basic entities, following standard syntax. This entity allows expressing data processing.

18.4.2 Terms

Terms describe an action by combining basic entities. There are three different types of terms, listed below:

1. Assignments: $a \leftarrow 3$, $!send \leftarrow 0$, $b \leftarrow ?rec$

   The variable or out-port on the left-hand side of the arrow is updated to the value of the variable, in-port or arithmetic expression on the right-hand side.

2. Guards: $a = 4$, $?rec > 10$

   Guards compare the value of a variable or in-port with the evaluation of an arithmetic expression. If the guard evaluates to true, nothing happens. Otherwise, the TSERE fails (or, loosely speaking, reaches a dead end).

3. Delays: [0..0], [3..5]

   Delays denote the passing of time. They are expressed as intervals, with the connotation that an arbitrary amount of time from the interval may elapse. This feature is crucial in the context of real-time systems.

18.4.3 Operators

In addition to terms, TSEREs can be recursively combined to express more complex behaviour with the following operators. Assume α and β being arbitrary TSEREs.

1. Sequence: $\alpha; \beta$

   α occurs immediately before β.

2. Choice: $\alpha + \beta$

   Either α or β occurs.

3. Concurrency: $\alpha \mid \beta$, $\alpha^{|n}$

   α and β occur concurrently. The concurrency operator is not considered to have occurred until both α and β have fully occurred. $\alpha^{|n}$ denotes n concurrent copies of α.

4. Iteration: α^n, α^∞, α^*, α^+

   The iteration operators denote a sequence of recurring α. The length of that sequence depends on the type of iteration. α^n denotes a sequence of length n and $n = \infty$ signifies an infinitely long sequence. Such a sequence can only be escaped if placed inside the choice operator. α^* denotes a sequence where n is arbitrarily chosen between $0 \leq n \leq \infty$, and in the case of α^+, n is arbitrarily chosen from $1 \leq n \leq \infty$.

18.4.4 Example

Returning to the example introduced in Fig. 18.3, the high-level and low-level channels and the transactor can be expressed with the following TSEREs:

1. High-level channel: $(m \leftarrow ?send; [2..2]; !rec \leftarrow m)^{\infty}$
2. Low-level channel: $(a \leftarrow ?sndaddr; [1..1]; !recaddr \leftarrow a; d \leftarrow ?snddata; [1..1]; !recdata \leftarrow d)^{\infty}$
3. Transactor: $(m \leftarrow ?send; [1..1]; !recaddr \leftarrow m.addr; [1..1]; !recdata \leftarrow m.data)^{\infty}$

The infinite iteration on the whole expression is necessary to enable the transactor to process several requests. Without the iteration, the transactor and channels would stop working after the first request.

As another example, consider a variant of the low-level channel where either the address and data are sent simultaneously, or we receive a reset request. Equation 18.1 shows the corresponding TSERE.

$$(((a \leftarrow ?sndaddr; [1..1]; !recaddr \leftarrow a) \mid (d \leftarrow ?snddata; [1..1]; !recdata \leftarrow d)) + ?reset)^{\infty} \tag{18.1}$$

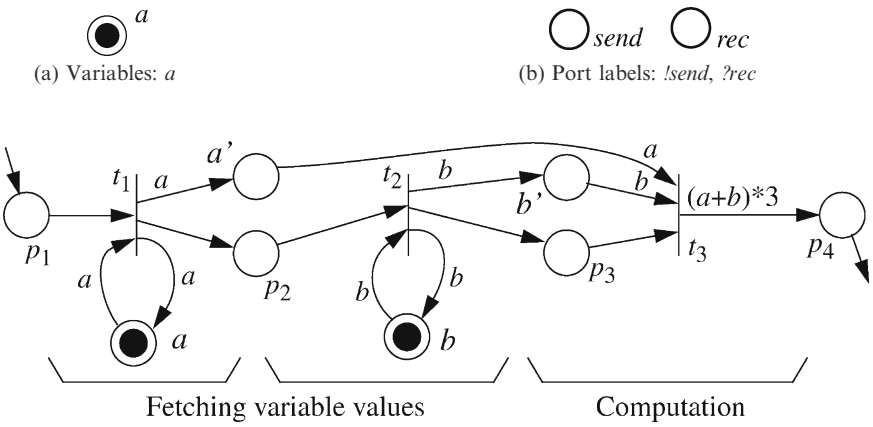
If statements can be expressed using guards together with the choice operator. In combination with iteration, this structure allows formulating bounded loops, as demonstrated in Eq. 18.2.

$$\alpha^n \Leftrightarrow i \leftarrow 0; ((i < n; \alpha; i \leftarrow i + 1)^{\infty} + (i = n)) \tag{18.2}$$

18.5 Transactor Generation

To generate a transactor is a two-step process. First, the behaviour of the transactor must be described with TSEREs. This must be done in such a way that each highlevel request is mapped onto low-level ones, while preserving the external behaviour, e.g. timing. Once a TSERE for the transactor is developed, that TSERE is automatically translated into an equivalent PRES+ model. This section provides details on how this is done.

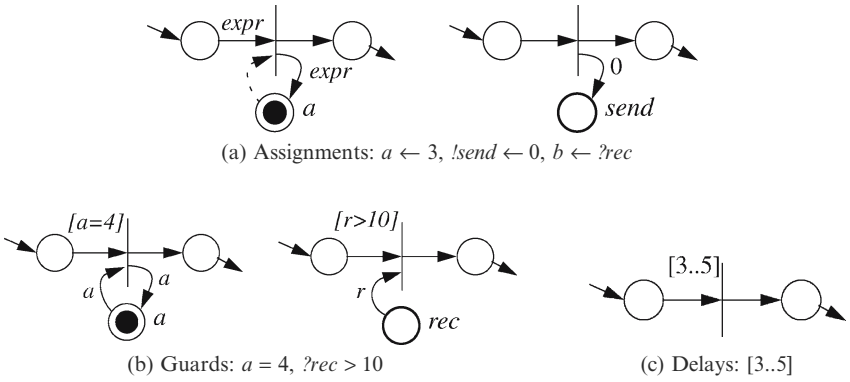
Regular expression based languages have a very strong relation with finite automata (and therefore also with PRES+), which makes such conversion relatively straight-forward [12]. Each basic entity, term and operator is mapped onto a PRES+ pattern, which directly reflects the semantics of that entity. The patterns have one entry place and one exit place, indicated in figures by a loose incoming and outgoing arc respectively. A token arriving in the entry place of a pattern enables the execution of that pattern, i.e. the occurrence of its corresponding TSERE. After executing the pattern/expression, a token should, by convention, be put in the exit place to indicate its completion. Figure 18.6 presents the patterns corresponding to basic entities, Fig. 18.7 the patterns corresponding to the terms and Fig. 18.8 the patterns corresponding to the operators.



(a) Variables: *a* (b) Port labels: *!send*, *?rec*

(c) Arithmetic expressions: $(a + b) \cdot 3$

Fig. 18.6 PRES+ patterns for TSERE basic entities



(a) Assignments: $a \leftarrow 3$, *!send* $\leftarrow 0$, $b \leftarrow$ *?rec*

(b) Guards: $a = 4$, *?rec* > 10 (c) Delays: [3..5]

Fig. 18.7 PRES+ patterns for TSERE terms

18.5.1 Patterns for Basic Entities

Variables are represented by a place (Fig. 18.6(a)), initially without a token. When the variable is assigned a value for the first time, and the variable enters its scope, a token containing the initial value is put in the place. From that point on, a token shall always reside in that place during the whole lifetime of the variable. The last term in the sequence, where the scope of possibly several variables ends, should consume the tokens in the places corresponding to those variables. Not storing values when not needed reduces statespace, and therefore mitigates the effects of statespace explosion. This is important for efficient model checking.

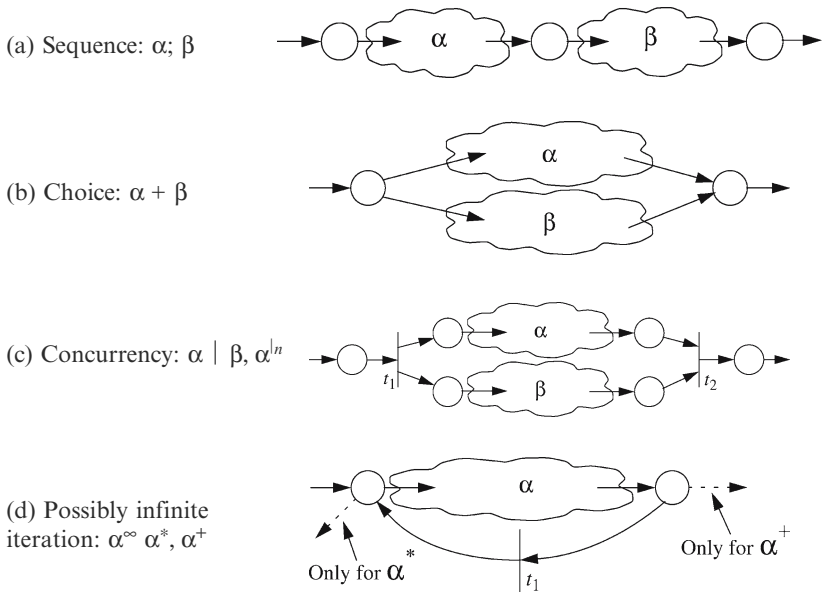


Fig. 18.8 PRES+ patterns for TSERE operators

Port labels are also modelled with a single place (Fig. 18.6(b)). These places will serve as ports of the transactor. *?* labels serve as in-ports and *!* labels as out-ports. Therefore, the transactor can only consume tokens from *?* label ports, and analogously only put tokens in *!* label ports.

Arithmetic expressions are modelled in two stages: fetching variable values and computation (Fig. 18.6(c)). The value of each variable involved in the expression must be explicitly fetched and stored in a temporary place. This arrangement is due to the fact that PRES+ transitions only are associated to one function. Without the fetching steps, the involved variables would change values to the value of the expression, which is not the desired behaviour.

The fetching of variable values is realised by transitions t_1 and t_2 in Fig. 18.6(c), for variables *a* and *b* respectively. The transitions consume the token from the variable place and immediately put it back with the same value. In the case of *?* port labels, the token is never put back. A copy of the value is moreover stored in a temporary place, *a*' and *b*' respectively. These tokens are then used in the final computation stage, transition t_3, instead of directly accessing the variable places. The fetching stages and the final computation stage are connected in a sequence with the help of intermediate places, p_1 to p_4. The result of the expression is located in the exit place of the arithmetic expression.

18.5.2 Patterns for Terms

Assignments are realised in a similar way as variable fetching, with the difference that the value of the token is updated (Fig. 18.7(a)). The new value is located in the entry place in the case of arithmetic expression, or, in the case of a constant, the transition function is set to that constant. Attention must be paid to if the assignment denotes the initial assignment to the variable in question or not. If it is, there is no token in the variable place to be consumed and consequently there shall not be an arc from the place to the transition. If the assignment is an update of an already initialised variable, the token must, on the contrary, be consumed before the update is actuated. In the case of *!* port labels, tokens are never consumed from within the transactor. As an optimization when the new value is an arithmetic expression, the assignment can be merged with the computation stage of the arithmetic expression.

Guards are implemented as variable fetching without creating a temporary copy, with the addition that the transition guard is set to the TSERE guard expression (Fig. 18.7(b)).

Delays are modelled with a transition with the time delay interval stipulated by the TSERE delay expression (Fig. 18.7(c)). The modelling of delays is preferably optimised by moving the time delay interval to the first transition of the subsequent TSERE, if such exists.

18.5.3 Patterns for Operators

The operator patterns combine several subpatterns to form a more complex behaviour. In Fig. 18.8, the subpatterns are drawn as clouds with arrows from/to its entry and exit places. The resulting complex pattern is also assigned entry and exit places, indicated in the figures in the same way as with the terms.

Sequences are realised by merging the exit place of the first subpattern with the entry place of the second (Fig. 18.8(a)). The entry place of the first subpattern becomes the entry place of the whole sequence, and the exit place of the second subpattern becomes the exit place of the whole sequence. In this way, when the first subpattern has finished executing, a token is put in the shared middle place, which enables the execution of the second subpattern.

In the pattern for the choice operator (Fig. 18.8(b)), the entry and exit places of the subpatterns are merged, so that all subpatterns share the same entry place and the same exit place. When a token appears in the entry place, this leads to the enabling of all subpatterns, out of which one is chosen randomly. If the first term of a subpattern is a guard that evaluates to false, that subpattern can naturally not be chosen.

When a token arrives in the entry place of the concurrency pattern (Fig. 18.8(c)), the entry places of each subpattern must also be marked to enable the execution of each corresponding subpattern. This is achieved by introducing an additional transition (t_1) with the entry places of all subpatterns as output and the entry place of the whole pattern as input. A similar, but contrary, construct is also inserted at the exit places (t_2),

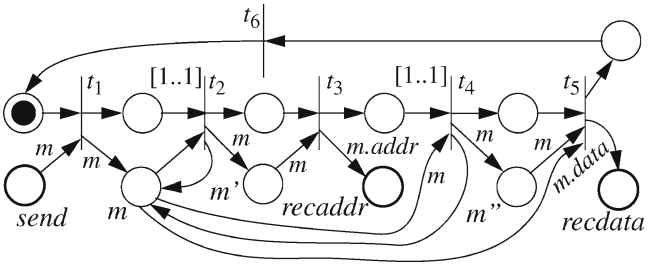
implementing the synchronisation of the subpatterns upon their completion. The concurrency operator is not considered completed until all subexpressions are completed.

Iteration is accomplished by connecting the exit place of the subpattern to its entry place via a transition (t_1 in Fig. 18.8(d)). This procedure can, in the case of α^{∞} and α^{*}, be optimised by instead merging the entry and exit places of the subpattern. The entry place of the subpattern is also the entry place of the iteration. For α^{*} iterations, the exit place is the same as the entry place, whereas for α^{+} the exit place of the iteration is the exit place of the subpattern. α^{∞} iterations do not have an exit place due to their infinite nature. Finite loops are implemented based on Eq. 18.2.

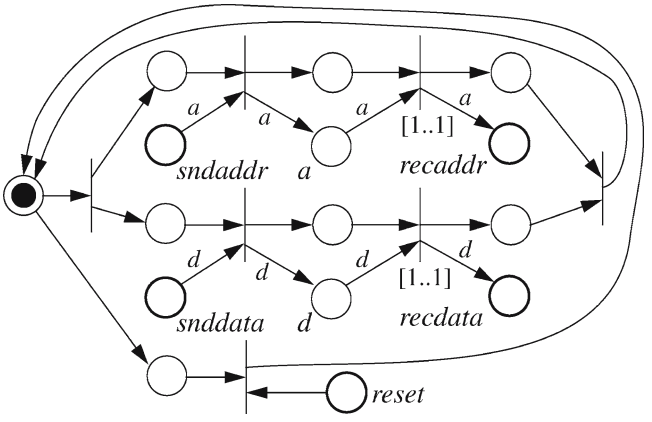
When a PRES+ model has been generated for the whole TSERE, an initial token is put in the entry place of the final model, to indicate the first term.

18.5.4 Examples

Let us continue the sender and receiver example introduced in Fig. 18.3, and where the TSEREs for the channels were listed in Section 18.4.4. Figure 18.9 provides the



(a) The generated transactor from Fig. 18.3 (c)



(b) The PRESS+ model corresponding to Eq. 18.1

Fig. 18.9 Examples of PRES+ models generated from TSEREs

PRES+ models resulting from the presented approach, including certain optimizations.

The core of the transactor is a sequence of reading and writing on ports combined with simple arithmetic expressions (Fig. 18.9(a)). Transitions t_2 and t_4 model the variable fetching stages of the arithmetic expressions, while transitions t_3 and t_5 combine the computation stages with the assignment on ports *recaddr* and *recdata* respectively (optimization). The delays are moreover added to the first transitions in the subsequent terms, in this case t_2 and t_4. It should moreover be noted how the scope of variable m is modelled. Transition t_1 realises the first assignment to m, therefore it only puts a token with the initial value in place m. As transition t_5 is the last transition in its scope, it consumes the token, no matter it needs the value or not. Transition t_6 models the infinite loop.

Figure 18.9(b) presents the PRES+ model corresponding to Eq. 18.1. Inside the iteration, there is a choice between either two concurrent statements or a single reading of reset. If the reset is not immediately present, the two concurrent sequences are launched. If the reset is present, there is a non-deterministic choice between the two options. The loop is in this figure optimised in the sense that the exit place of the choice operator is merged with its entry place.

18.6 Case Studies

The proposed approach has been applied on two examples: the example from Fig. 18.3 and an AMBA-based protocol. The models were formally verified on high, low and mixed levels of abstraction using a Linux machine with an Intel Pentium 4, 2.8 GHz processor and 2 GB of memory. The AMBA example was moreover verified with different configurations on the number of masters (M) and slaves (S). Both examples were checked for the same two properties: no deadlock and that sent messages will arrive at their destinations.

Tables 18.1 and 18.2 present the verification times in seconds for the respective example. The tables moreover indicate the sizes of the TSEREs, which define the channels/transactors, as the number of terms and operators in the expression. The size of the entire verified PRES+ model is indicated by the number of transitions. These numbers only give a hint to the size of the examples and are not directly related to verification time. These results indicate the reasonableness of the proposed approach.

Table 18.1 Results from the example given in Fig. 18.3

Abstraction level	No deadlock	Sent will arrive
High	0.12 s	0.13 s
Low	0.06 s	0.09 s
Sender high – receiver low	0.11 s	0.06 s

Table 18.2 Results from the AMBA example

M–S	Abstraction level	No deadlock	Sent will arrive
1–1	High	0.33 s	0.12 s
	Low	0.19 s	0.22 s
	M high – S low	0.19 s	0.17 s
	M low – S high	0.30 s	0.40 s
1–2	High	0.50 s	0.46 s
	Low	0.80 s	1.68 s
	M high – S low	0.24 s	0.35 s
	M low – S high	1.44 s	3.57 s
2–1	High	0.19 s	0.43 s
	Low	0.48 s	1.53 s
	M high – S low	0.38 s	0.84 s
	M low – S high	1.43 s	6.59 s
2–2	High	5.01 s	18.99 s
	Low	5.43 s	22.57 s
	M high – S low	5.39 s	17.77 s
	M low – S high	42.06 s	200.5 s

18.7 Conclusions

This chapter has presented an approach to generate transactors for real-time embedded systems, suitable for formal verification. The approach assumes a design where components communicate over channels, and that those channels capture all the characteristics of the communication. During the development, more and more components are refined leading to a model with mixed abstraction levels. In such models, the components cannot directly communicate due to protocol discrepancies. In order to overcome these discrepancies, the channels interfacing components of different abstraction levels are replaced with transactors. The behaviour of the transactors, i.e. the mapping of requests between abstraction levels, is described using TSEREs, which are automatically converted into the design representation used, PRES+. The resulting PRES+ model can then be analysed by a formal verification tool.

References

1. Bombieri N, Fummi F, Pravadelli G (2006) On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL. Proc. ACM/IEEE Design and Test in Europe, Munich, Germany, 6–10 March
2. Akella J, McMillan K (1991) Synthesizing Converters between Finite State Protocols. Proc. International Conference on Computer Design, Cambridge, MA, Oct. 15–15, pp. 410–413
3. Passerone R, Rowson JA, Sangiovanni-Vincentelli, A (1998) Automatic Synthesis of Interfaces between Incompatible Protocols. Proc. Design Automation Conference, San Francisco, CA, June, pp. 8–13
4. Bombieri N, Fummi F, Pravadelli G (2006) A TLM Design for Verification Methodology. IEEE Ph.D. Research in Microelectronics and Electronics, Otranto (LE), Italy, 11–15 June, 337–340

5. Balarin F, Passerone R (2006) Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation. Proc. Design and Test in Europe, Munich, Germany, pp. 1013–1018
6. Asarin E, Caspi P, Maler O (1997) A Kleene Theorem for Timed Automata. Proc. Logic in Computer Science, Warsaw, Poland, June, pp. 160–171
7. Karlsson D, Eles P, Peng Z (2007) Formal Verification of Component-based Designs. Journal of Design Automation for Embedded Systems 11(1):49–90
8. Alur R, Courcoubetis C, Dill DL (1990) Model Checking for Real-time Systems. Theoretical Computer Science 414–425
9. UPPAAL homepage: http://www.uppaal.com/
10. Cortés LA, Eles P, Peng Z (2000) Verification of Embedded Systems Using a Petri Net Based Representation. Proc. International Symposium on System Synthesis, Madrid, Spain, pp. 149–155
11. Alur R, Dill DL (1994) A Theory of Timed Automata. Theoretical Computer Science 126:183–235
12. Kozen DC (1997) Automata and Computability. Springer, New York.

Chapter 19
A Case-Study in Property-Based Synthesis: Generating a Cache Controller from a Property-Set

Martin Schickel, Martin Oberkönig, Martin Schweikert, and Hans Eveking

Abstract Property-based synthesis has become a more prominent topic during the last years, being used in multiple areas like, e.g. formal verification and design automation. We will show how a property-based formal specification of a cache controller for a MIPS core can be used to automatically generate a functional implementation of that controller and how additional performance information about the complete system can be gained from doing so.

Keywords Property Based Design, Synthesis, Formal Verification, Cando-Objects

19.1 Introduction

The integration of design and verification effort has strongly improved during the last decade. Many EDA companies require their designers to include assertions into the hardware descriptions – a technique known as assertion-based design (ABD). Also, formal specifications, consisting of properties and assertions, are no longer only developed during the verification of a design, but also before and during its creation. Looking at this development, the obvious question is whether those formal specifications used to verify designs can also be used to automatically generate hardware implementing the properties, thereby assuring a golden model which is correct by construction.

In the last years, some significant progress has been made in this area, enabling the automatic generation of prototype models from ever larger and more complex sets of properties. In using this approach, we can assure that a design verified using

Computer Systems Lab, Darmstadt University of Technology Darmstadt, Germany;
Email: {schickel,oberkoenig,schweikert,eveking}@rs.tu-darmstadt.de

E. Villar (ed.) *Embedded Systems Specification and Design Languages*,

a complete set of properties will be working exactly as the golden model generated from them, thereby formally relating the until now unrelated specification languages for models and verification.

In the following sections we will discuss the results of our experiments with a set of properties describing the functionality of a cache controller for a MIPS. Using these properties we wanted to reach two different goals:

Firstly, we wanted to know whether it was possible to generate a functioning simulation model of the cache controller and simulate it together with a MIPS core. Secondly, we wanted to see whether we would be able to derive information about the behavior of a system consisting of a MIPS core and a cache controller adhering to the property-set we had.

We used the CandoGen-tool [1] from Darmstadt University described, e.g. in [2] by Schickel et al. This tool is capable of generating VHDL-descriptions of so-called *Cando-Objects* from sets of finite properties written in PSL [3, 4] or ITL [5]. These Cando-Objects are in essence black-boxed designs whose behavior is restricted by the properties they were generated from (hence their name: "Can do anything not disallowed").

However, there have been other efforts to automatically synthesize executable hardware from properties: the ProSyd project and BlueSpec.

The ProSyd project was founded to research possible improvements in property-based system design. One of the deliverables was a tool capable of synthesizing functioning hardware from arbitrary PSL properties. The tool first constructs a finite state machine from the properties, and then translates the machine into a hardware description language. While the results are very good when the properties only describe a system's control path, the used methods' complexity is unsuitable for the generation of data paths [6]. Since our properties include the data path, this tool is unsuitable for us.

BlueSpec is a company founded by Arvind Mithal from MIT. It utilizes the patented term-rewriting-system [7] to translate properties written in BlueSpec-SystemVerilog into functioning hardware. This method is known to be highly efficient and often produces results better than human designers, but it requires the user to write properties in a different style than that used when writing verification properties. Therefore verification properties cannot be used for synthesis using this method. Since our properties were verification properties written in another language (i.e. PSL and ITL), this tool was also unsuitable.

19.2 The Cache Controller Properties

For our experiment we had obtained a MIPS core from opencores.org [8] and a set of properties describing the functionality of a simple cache controller, which had to be transparent in order to use the non-modified MIPS design. The set of cache properties describes a fully associative cache model (i.e. the definition of *cache-hit* was basically 'any cache-cell has valid data for a given address'). A least recently

used (LRU) policy was specified as well as a write-through technique. The size of the cache was determined to be eight cache lines of eight 32 bit-instructions (8 × 256 bit), but could not only be used to cache instructions, but also to cache data needed during the pipeline's execution step.

The properties for the cache can be categorized in five functional groups:

- Manager &- Cacheline validity correct?
- WriteData &- Write Instructions handled correctly?
- Replacement &- LRU algorithm working correctly?
- Instruction &- Read Instruction handled correctly?
- Memory &- Read Data handled correctly?

One example property is illustrated in Fig. 19.1. It describes the reset behavior of the memory group. It is written in VHDL-flavored ITL.

19.3 Experimental Results

All the properties could be transformed into VHDL descriptions of a working circuit model incorporating all the described functionality. The transformation runtimes are listed in Table 19.1.

The time spent on the properties in the *manager* group was fairly long. This can be explained by CandoGen's current internal use of BDDs which may become rather complex when the number of variables grows larger than 300 as is the case when checking whether a cache-hit has occurred. This is due to the BDD-explosion which occurs prominently when shift- and multiplication operations are concerned. The effects might be countered by using AIGs [9] to replace or complement the BDD-representation of the circuits. A hybrid AIG/BDD-system might combine the strengths of both representation methods.

```
property reset is
assume:  at t:   reset='1';
prove:   at t+1: wait_for_mem='0';
         at t+1: update_least_recent_mem='0';
         at t+1: update_cache_info_mem='0';
         at t+1: mem_req_read='0';
end property;
```

Fig. 19.1 Sample property

Table 19.1 Model generation data

Module	#Props	Lines of code	Runtime (min)
Manager	4	93	321
WriteData	4	89	5
Replacement	5	135	2
Instruction	5	239	46
Memory	6	134	13

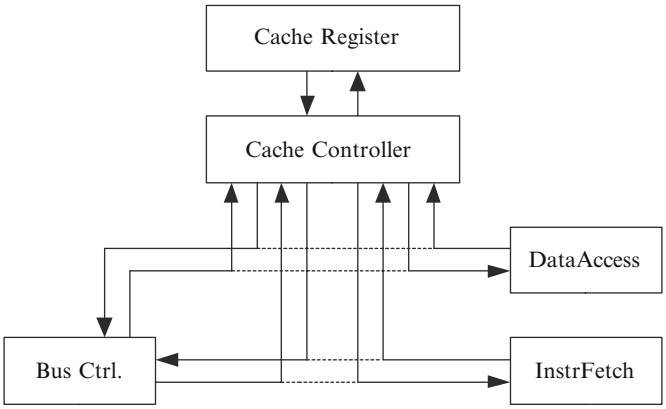


Fig. 19.2 Connection of controller to MIPS core

The generated VHDL models could then be connected to form the complete cache controller and be simulated together with the MIPS core. To do so, the cache controller was connected to the core's memory interface as shown in Fig. 19.2. The dotted lines mark the original connections.

The simulation of small precompiled and preloaded programs during the course of directed testing worked well and showed a full functionality of the cache, reducing the average memory access latency.

The last step was the verification or formal deduction of system level properties. Since one of the most prominent properties of a cache is the acceleration of memory accesses, we decided to write properties to examine the memory access speedup. On the original design, it can be proven, that any memory access has the same latency as was specified within the memory description.

When the cache controller is attached to the design, this property does not hold anymore. A counter-example shows that when consecutive areas of memory are addressed the memory access may be completed more quickly. By relaxing the property to allow for completion within a certain timeframe we can quickly determine the effect of the cache to be between −3 to +1 cycles latency. The latter results from the cache's property to read complete cache lines, which may prove problematic when memory accesses are sufficiently random. The proof of these properties was completed within negligible time (less than 1 min per property).

19.4 Conclusion

We have shown that it is possible to automatically generate hardware from properties and used the generated model during simulation and to prove system properties. Future research will include synthesizability of complete processor cores from verification properties.

Acknowledgments The research leading to this publication was conducted within the scope of the FEST project jointly funded by the German ministry of research and education and industry partners.

References

1. M. Schickel, V. Nimbler, M. Braun and H. Eveking: *CandoGen – A Property-Based Model Generator*, University Booth, Nice, France, Date'07.
2. M. Schickel, V. Nimbler, M. Braun and H. Eveking: *On Consistency and Completeness: Exploiting the Property-Based Design Process*, Proc. of FDL'06.
3. *Property Specification Language, Reference Manual, Version 1.1*, Accellera, 2004, http://www.eda.org/vfv/docs/PSL-v1.1.pdf.
4. C. Eisner and D. Fisman: *A Practical Introduction to PSL*, Springer, New York, 2006.
5. *User Documentation: OneSpin MV 360 – Version 4.1*, OneSpin Solutions GmbH, 2006.
6. ProSyd Project Deliverable 2.3/1: *Evaluation of tools and methodologyfor property-based logic synthesis*, www.prosyd.org.
7. A. Mithal, J. Hoe. *Digital Circuit Synthesis System*, U.S. Patent U.S. 6,597,664 B1, 7/2003.
8. http://www.opencores.org/projects.cgi/web/minimips/overview
9. V. Paruthi and A. Kuehlmann: Equivalence checking combining a structural SAT-solver, BDDs, and simulation, in ICCD'2000.