An Integrated SystemC Debugging Environment

Frank Rogin* Christian Genz† Rolf Drechsler† Steffen Rülke*

* Fraunhofer Institute for Integrated Circuits, Division Design Automation, 01069 Dresden, Germany
{frank.rogin, steffen.ruelke}@eas.iis.fraunhofer.de

† University of Bremen, Institute of Computer Science, 28359 Bremen, Germany
{genz, drechsle}@informatik.uni-bremen.de

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 [1]. *Our work mainly focuses on developing an easy to handle interface which supports debugging and system exploration of SystemC designs.*

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 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 a HDL simulation can do. All these features make SystemC an excellent approach for modelling SoCs and allow to implement 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 provide 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, dataflow 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 crossprobing 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 paper is organized as follows. Section 2 discusses related approaches and tools which allow to debug SystemC designs. In Section 3 the general architecture of our IDE is described in more detail while Section 4 considers the provided debugging interface and the graphical frontend and its debugging support. In Section 5 we illustrate some IDE features exemplarily and demonstrate their feasibility using a short example. Finally, Sec-

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

tion 6 concludes the paper and gives a perspective on future work.

2 Related Work

Debugging SystemC models requires hybrid techniques that grant access to design components quickly but also allow to evaluate 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 crossprobing 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 that accomplishes 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, crossprobing 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 crossprobing 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, and
- offer 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, and
- a 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.

3 Debugging Environment

Our IDE consists of three components. Each of these components realizes a particular task. As sketched in Figure 1 our debugging flow starts at the original system description which is being compiled to an executable.

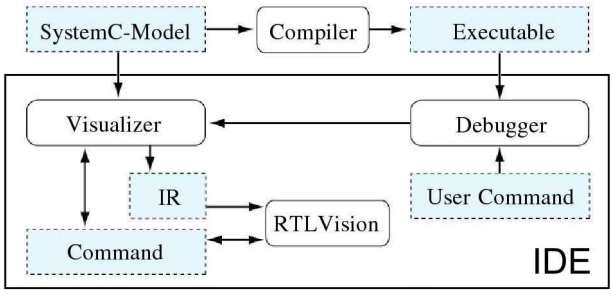

Figure 1. Architecture of the IDE

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.

4 Debugging Features

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 1 assembles a list of visualized high-level 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.

Table 1. Visualized 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 Figure 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.

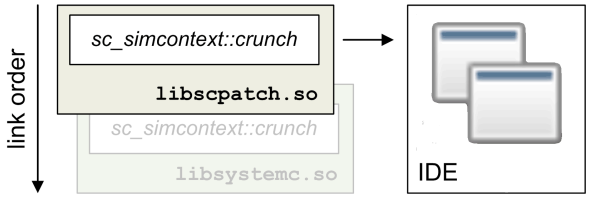

Figure 2. Preloading kernel methods

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 Figure 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,
- crossprobing,
- path fragment navigation, and
- 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, and
- annotating values to signals and ports.

5 Practical Application

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]. Figure 3 shows an example debug session simulating this design. The different views allow to explore 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.

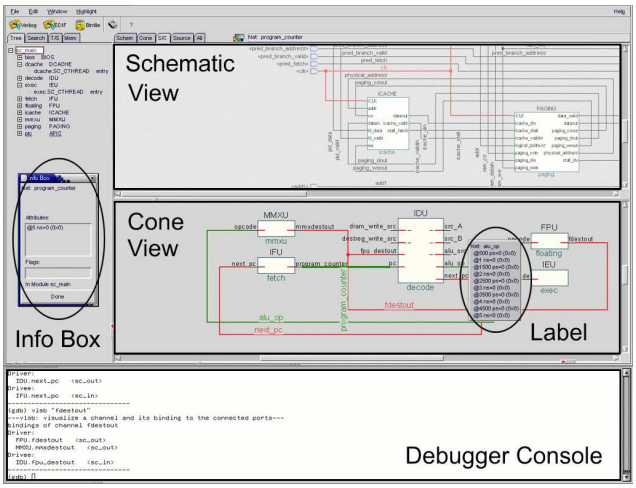

Figure 3. Example debug session

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

The **vlsb** command (Table 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. Figure 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"
```

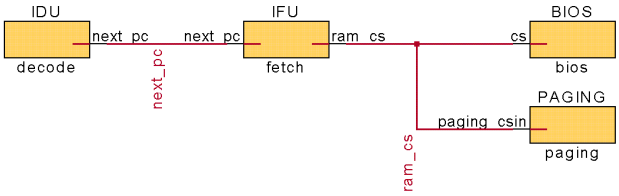

Figure 4. Debug command vlsb

The **vtrace_at** command (Table 1) is a typical representative of the monitoring command type. It traces the given signal or port and records the actual value at the specified simulation time stamp. The logged value is attached as label text 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. Figure 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"
```

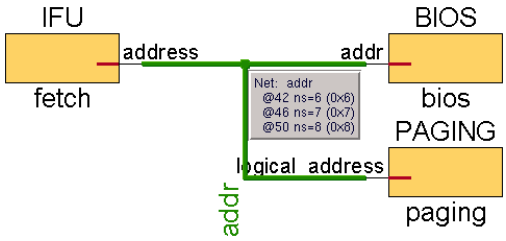

Figure 5. Debug command vtrace_at

Table 2 underlines the efficiency of our non-intrusive, patch-free approach using library interposition (Section 4.1) while illustrating the performance of the **vtrace** command (Table 1). So, the observation of 750000 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%.

# of traced signals	Slow down over simulation time (# of observed data sets)		
	1000 ns	2000 ns	3000 ns
0	1.0	1.0	1.0
5	1.3 (10000)	1.3 (20000)	1.4 (30000)
50	1.8 (100000)	1.8 (200000)	1.8 (300000)
75	2.3 (150000)	2.6 (300000)	3.0 (450000)
100	3.0 (200000)	3.2 (400000)	3.6 (600000)
125	3.2 (250000)	3.9 (500000)	4.0 (750000)

Table 2. Exemplary performance slow down

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. Therefor, we use several exploration and visualization features (Section 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 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 shows subsequently that the only port reported by **vlsio_rx** is connected to the signal **alu_op** (Figure 6).

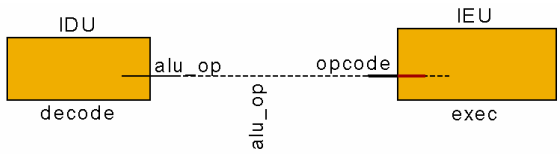

Figure 6. Tracking down the op-code signal

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 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 1) is applied (Figure 7):

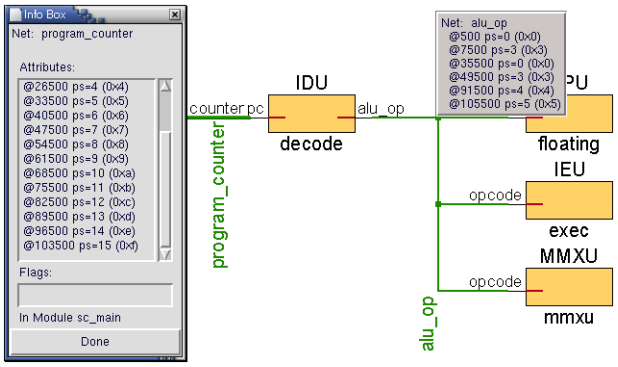

Figure 7. Exploring traced signals

```
(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 til 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.

A conventional debug procedure would set several breakpoints on the right positions into the instruction decoder and the ALU. On any stop of this 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.

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).

Acknowledgement

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

References

[1] ARM Ltd. MaxSim Developer home. www.arm.com.

[2] D. Berner, H. Patel, D. Mathaikutty, J.-P. Talpin, and 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, and G. Bois. A Methodology for Interfacing Open Source SystemC with a Third Party Software. In *Design, Automation and Test in Europe*, pages 16–20, 2001.

[4] Concept Engineering. home page. www.concept.de.

[5] CoWare. Platform Architect. www.coware.com.

[6] Eclipse Foundation. project home. www.eclipse.org.

[7] C. Eibl, C. Albrecht, and R. Hagenau. gSysC: A graphical front end for SystemC. In *European Conference on Modelling and Simulation*, pages 257–262, 2005. Source available at www.iti.uni-luebeck.de/ albrecht/gSysC/.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design pattern - elements of reusable object-oriented software. In *Addison Wesley Professional Computing Series*, 1999.

[9] C. Genz, R. Drechsler, G. Angst, and L. Linhard. Visualization of SystemC Designs. In *IEEE International Symposium on Circuits and Systems*, pages 413–416, 2007.

[10] GNU debugger. home. www.gnu.org/software/gdb.

[11] D. Große, R. Drechsler, L. Linhard, and G. Angst. Efficient automatic visualization of SystemC designs. In *Forum on Specification and Design Languages*, pages 646–657, 2003.

[12] M. Moy, F. Maraninchi, and 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*, pages 26–35, 2005.

[13] M. Moy, F. Maraninchi, and L. Maillet-Contoz. PINAPA: An extraction tool for SystemC descriptions of systems-on-a-chip. In *ACM international conference on Embedded software (EMSOFT '05)*, pages 317–324, 2005.

[14] OSCI. SystemC home page. www.systemc.org.

[15] F. Rogin, E. Fehlauer, S. Rülke, S. Ohnewald, and T. Berndt. Non-Intrusive High-level SystemC Debugging. In *Advances in Design and Specification Languages for Embedded Systems*. Springer, July 2007.

[16] A. Wieferink, M. Doerper, T. Kogel, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and 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, pages 3–11, Jan. 2005.