A Case Study of AOP and OOP applied to digital hardware design

Tiago R. Mück[1], Michael Gernoth[2], Wolfgang Schröder-Preikschat[2], Antônio A. Fröhlich[1]

[1]Software/Hardware Integration Lab
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil

[2]Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
Erlangen, Germany

{tiago,guto}@lisha.ufsc.br,{gernoth,wosch}@cs.fau.de

Abstract. *In this paper we explore a SystemC-based hardware design method which uses aspect-oriented programming concepts. We have designed a synthesizable resource scheduler at register transfer level by using only features available in the SystemC synthesizable subset. The results show that aspect-oriented programming applied to digital hardware design provides a better separation of concerns at the cost of a negligible overhead.*

1. Introduction

The complexity of embedded system design is increasing much faster than the design and verification capability of developers. This has led to the introduction of solutions and methodologies that had been successfully deployed in the scope of large-scale software systems. For example, *object-oriented programming* (OOP), which is supported in the hardware domain by languages like SystemC.

In this paper we explore the use of *Aspect-oriented programming* (AOP) techniques for hardware design. We redesigned the hardware implementation of an operating system task scheduler [Marcondes et al. 2009] by leveraging on SystemC features in order to enable the use of OOP and AOP concepts. AOP was applied by using a domain engineering strategy that yields components in which the dependencies from the execution scenario are encapsulated as *aspects*. In order to obtain an efficient and synthesizable component, our scheduler was designed at the *register transfer level* (RTL) by using standard C++ metaprogramming features within the SystemC synthesizable subset [OSCI 2010].

The remaining of this paper is organized as follows: section 2 introduces AOP concepts and gives a contextualization of its use on the hardware domain; section 3 presents a discussion about related work; section 4 presents our design artifacts; sections 5 and 6 describe the implementation of our scheduler and show our experimental results; section 7 closes the paper with our conclusions.

2. Aspect-oriented programming

In the software domain, the use of machine code had evolved naturally to procedural languages and then to OOP. An enormous productivity improvement resulted from the increased level of abstraction. However, OOP still have some limitations in the way it allows

a complex problem to be broken up into reusable abstractions. Even though most classes in an object-oriented model will perform a single function, they often share common, secondary requirements with other classes. The implementation of these *crosscutting concerns* is scattered among the multiple abstractions, thus breaking the encapsulation principle. AOP is an elaboration over OOP to deal with the crosscutting concerns. AOP proposes the encapsulation of these concerns in special classes called *aspects*. An aspect can alter the behavior of the base code by applying *advices* (small pieces of code defining additional behavior) in specific points of a program called *pointcuts*. Some extensions to OOP languages have been proposed to support these new concepts. For example, AspectJ [Kiczales et al. 2001] and AspectC++ [Spinczyk et al. 2002] extend Java and C++ with full support for AOP features. They provide both new language constructs and an *aspect weaver*, a tool responsible for applying the advices to the base code before it is processed by the traditional compiling chain.

Recently, there has been a growing interest in high-level methodologies for hardware design as well. An example of a *hardware description language* (HDL) which supports OOP is SystemC; a C++ based modeling platform and language supporting design abstractions at the register transfer, behavioral, and system levels [Panda 2001]. However, analogous to software, in hardware some system-wide cross-cutting concerns cannot be elegantly encapsulated. For example, in complex circuits, interconnection of several entities is realized by introducing buses. A bus physically interacts with other components (e.g. CPU, DMA, ...), but it is difficult to use a module or a class to encapsulate the bus because its interface and arbitration method has to be implemented in every attached component. Other examples of crosscutting concerns in hardware designs can be also found in parts of a system related to its overall functionality or the implementation of non-functional properties (e.g. clock handling and hardware debugging through JTAG scan chains). Even with the introduction of OOP in hardware, this scattered code is hard to maintain and bugs may be easily introduced. The introduction of AOP to hardware design is expected to provide the easy encapsulation of cross-cutting concerns and an increase in the overall design quality.

3. Related work

Several works have already proposed the use of AOP concepts for hardware design. In [Engel and Spinczyk 2008] the authors discussed the nature of crosscutting concerns in VHDL-based hardware designs. They have proposed a hypothetical AOP extension for VHDL in which the execution of a process and the setting of a signal are used as pointcuts. However, the work lacks a concrete implementation and an evaluation of the impact of AOP in the design. In [Bainbridge-Smith and Park 2005] the authors discussed how the separation of concerns may relate to different levels of algorithmic abstraction. They have mentioned the development of ADH, a new HDL based on AOP, but further details about ADH are not mentioned. In [Burapathana et al. 2005] the use of AOP concepts to sequential logic design was proposed. Nevertheless, they focused on very simple and low level examples like flip-flops and logic gates on which only the clock can be feasibly handled as a crosscutting concern.

There are also several works which proposed the use of AOP concepts mostly for hardware verification. In [Endoh et al. 2008], AOP was used to enable assertion-based verification in high-level hardware design, in which assertions are based on pointcuts in-

stead of specifiers to signal changes. They have designed and implemented two assertion languages with pointcut-based assertions, ASystemC and ASpecC, which work alongside SystemC and SpecC, respectively. ASystemC uses pointcut of AspectC++, and its implementation translates assertions into aspects of AspectC++. [Kallel et al. 2010] also proposed the use of SystemC and AspectC++ to implement assertion checkers. The authors focused on the verification of transaction-level models (TLM) in which transaction state updates are used as pointcuts. They provide a framework in which the user verification classes extend the base aspect classes that implement the pointcuts and verification.

Other proposal that focus only on hardware verification can be seen in [Vachharajani et al. 2004], in which the authors developed the *Liberty Structural Specification Language* (LSS). In LSS each module can declare that its instances emit certain events at runtime. These events behave like pointcuts of AOP. Each time a certain state is reached or a value is computed, the instance will emit the corresponding event and user-defined aspects will perform statistics calculation and reporting. [Liu et al. 2010] also proposed AOP-based instrumentation, but focusing high-level power estimation. They have developed a methodology based on SystemC in which AspectC++ is used to define special power-aware aspects. These aspects are used as configuration files to link power aware libraries with SystemC models.

Other works provide AOP features not only for verification, but also for the actual design of hardware. [Déharbe and Medeiros 2006] present and assess possible applications of AOP in the context of integrated system design by using SystemC with AspectC++. Differently from the works discussed previously, they showed how AOP can be used to encapsulate some functional characteristics of hardware components. They modeled as aspects the replacement policy of a cache, the data type of an FFT, and the communication protocol between modules. However, only simulation results are shown and they do not compare the implementation of aspect-based components against components with all the functionalities hard-coded. In a similar work, [Liu et al. 2009] implemented a SystemC model for a 128-bit floating-point adder and described the implementation of the same model using AOP techniques. But, synthesis results are not provided and the two models are compared only in terms of functionality to show that the AOP design works like the original SystemC-only design.

Other works in this area follow different approaches. The *e* programming language [Vax 2007] was designed for modeling and verification of electronic systems and some of its mechanisms can be used to support AOP features. Apart from its OOP features, *e* has some constructs to define the execution order of overloaded methods in inherited classes, which can be used to define pointcuts and implement aspects. Indeed, this can be used to implement the behavior of hardware components, but *e* is more focused in high-level specification and there is not any tool support for synthesis.

4. Designing a hardware using AOP

Similarly to previous works, we also based our approach on methodologies which have been used in the software domain. The *Application-driven Embedded System Design* (ADESD) [Fröhlich 2001] methodology elaborates on commonality and variability analysis—the well-known domain decomposition strategy behind OOP—to add the concept of aspect identification and separation at early stages of design. It defines a domain

engineering strategy focused on the production of families of scenario-independent components. Dependencies observed during domain engineering are captured as separate *aspect*, thus enabling components to be reused on a variety of execution scenarios with the application of proper *aspects*. This aspect weaving is performed by constructs called *Scenario adapters*[Fröhlich and Schröder-Preikschat 2000].

The design artifacts proposed in ADESD were implemented and validated on the *Embedded Parallel Operating System* (EPOS) [Fröhlich 2001]. EPOS aims to automate the development of dedicated computing systems, and features a set of tools to select, adapt, and plug components into an application-specific framework, thus enabling the automatic generation of an application-oriented system instance. EPOS is implemented in C++ and leverages on *generic programming* [Czarnecki and Eisenecker 2000] techniques such as *static metaprogramming* in order to achieve high reusability with low overhead.

The next sections describe some of these design artifacts and how we have applied them in the implementation of our hardware scheduler.

4.1. Scenario adapters

Scenario adapters were developed around the idea of components getting in and out of an execution scenario, allowing actions to be executed at these points, therefore, a scenario must define at least two different operations: *enter* and *leave*. These actions must take place respectively before and after each of the component's operation in order to setup the conditions required by the scenario. For example, in a compressed scenario, enter would be responsible to decompress the component's input data, while leave would compress its outputs.

In the software domain, components are objects which communicate using method invocation (considering an OOP-based approach) and the execution of all operations are naturally sequential, so the scenario adapters were originally developed to provide means to just efficiently wrap the method calls to an object with enter and leave operations. However, in the hardware domain, components have input and output signals instead of a method or function interface, and all operations are intrinsically parallel. These different characteristics required some modifications on the original scenario adapter. The new scenario adapter is shown in figure 1.

SystemC defines hardware components by the specialization of the *sc_module* class. Components communicate using special objects called *channels*. SystemC channels can be used to encapsulate complex communication protocols at register transfer or higher levels of abstraction. However, these complex channels lie outside the SystemC synthesizable subset, so we use only *sc_in* and *sc_out*, which define simple input and output ports for components. Methods which implement the component's behavior must be defined as SystemC processes. In our examples we use SystemC clocked threads (*SC_CTHREAD*), in which all operations are synchronous to a clock signal. The implementation of the *Component::controller* method in figure 1 shows the common behavior of a *SC_CTHREAD*. SystemC *wait()* statements must be used to synchronize the operations with the clock, in other words, all operations defined between two *wait()* statements occur in the same clock cycle.

Using these constructs, we define each aspect as a single and independent hardware component (*Aspect* class). *Enter* and *leave* operations are defined using a simple

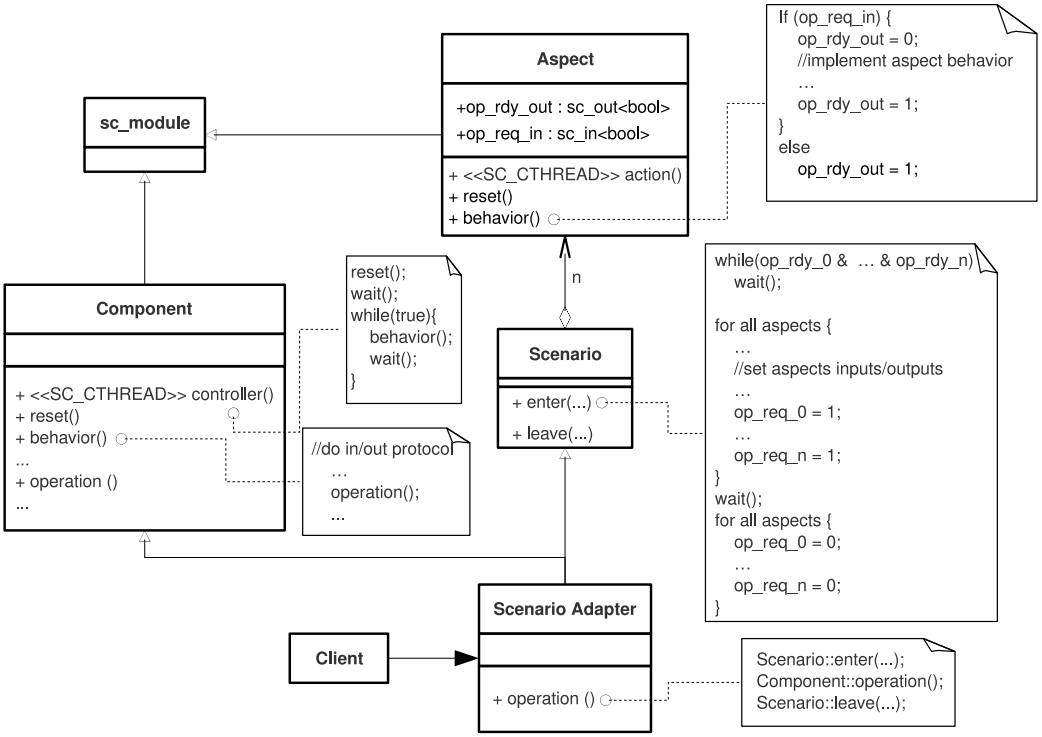

Figure 1. UML class diagram showing the general structure and behavior of a scenario adapter.

handshaking protocol (*op_rdy_out* and *op_req_in* signals) to trigger its execution. The remaining input/output ports define which operation are being triggered (this is specific of each aspect). With this kind of handshaking communication protocol we can produce more reusable components, since the number of clock cycles it requires for each operation is hidden by the protocol, thus making it easier to synchronize component execution with the rest of the design.

The *Scenario* class incorporates, via aggregation, all of the aspects which define its characteristics. It defines *enter* and *leave* methods to encapsulate the implementation of the handshaking protocol which trigger the aspects. Figure 1 shows how the scenario's *enter* operation is implemented. All aspects are triggered at the same time and executes in parallel, however, if required by the scenario, this can be modified in order to execute each aspect sequentially at the cost of additional clock cycles.

The adaptation of the component to the scenario is performed by the *Scenario Adapter* class via inheritance. This adaptation is possible through the separation of the component's input/output protocol from the implementation of its behavior. A SystemC process (*controller* method) handles the input/output protocol (*behavior* method) and calls the requested operations, which are each implemented in its own methods. These methods are overridden in the *Scenario Adapter* class. Notice that, although scattered through a class hierarchy and different methods, all operations (from the handling of the component's input/output protocol, to the triggering of the aspects) executes inside the *controller SC_CTHREAD* process. For the proposed scheme to work, *wait()* statements are also used to schedule the operations among the clock cycles, instead of defining explicit state machines. If the latter is used, it would not be possible to elegantly implement the structure

described in figure 1, since a state machine would require manual intervention to add the operation defined by the scenario.

4.2. ADESD and classic AOP

Several previous works have already discussed aspect-oriented hardware design using SystemC and proposed solutions based on classic AOP concepts using the well known AspectC++ language. Indeed, AspectC++ provides more powerful mechanisms for aspect implementation then ADESD, especially when it comes to the definition of the pointcut, however, this additional mechanisms are usually either unnecessary or can be efficiently replaced. For example, the aspects implemented in *Déharbe and Medeiros* [Déharbe and Medeiros 2006] (section 3) could be more elegantly implemented using other standard C++ features like inheritance and templates parameters. In the scope of ADESD, we can say that scenario adapters can be used to implement *homogeneous crosscutting* [Jun et al. 2009] (the process of adding the same behavior for all classes). *Heterogeneous crosscutting* [Jun et al. 2009](when concern is specific to a certain component or family of components) can be easily implemented with standard OOP (e.g. inheritance). Additionally the implementation of ADESD's mechanisms can be realized using only standard SystemC features. Previous works focus on tools and languages which were deployed only for software development (e.g. AspectC++), which limits its use for the generation of synthesizable hardware.

5. Scheduler implementation

Our case study is based on a previous implementation of the EPOS scheduler described by [Marcondes et al. 2009], which described a task scheduling suitable for hardware and software implementation. However, the original VHDL implementation was not susceptible to the same mechanisms that render its software counterpart flexible and reusable. The new System-based hardware scheduler is described below.

Figure 2 shows a simplified view of the task scheduling model. In this design, the task is represented by the class *Thread* and defines the execution flow of the task, implementing the traditional functionality (e.g. suspend and resume operations). The classes *Scheduler* and *SchedulingCriteria* define the structure that realizes the task scheduling. Traditional design and implementations of scheduling algorithms are usually done by a hierarchy of specialized classes of an abstract scheduler class, which can be further specialized to bring new scheduling policies to the system. In order to reduce the complexity of maintenance of the code (generally present in such hierarchy of specialized classes), as well as to promote its reuse, the design detaches the scheduling policy (criteria) from its mechanisms (lists implementations) and also detaches the scheduling criteria from the thread it represents. This is achieved by the isolation of the element's comparison algorithm of the scheduler in the criteria.

5.1. Hardware implementation

The separation of the mechanism from the scheduling policy was fundamental for the construction of the scheduler in hardware. The hardware scheduler component implements only the mechanisms that realize the ordering of the tasks, based on the selected policy. In this sense, the same hardware component can realize distinct policies.

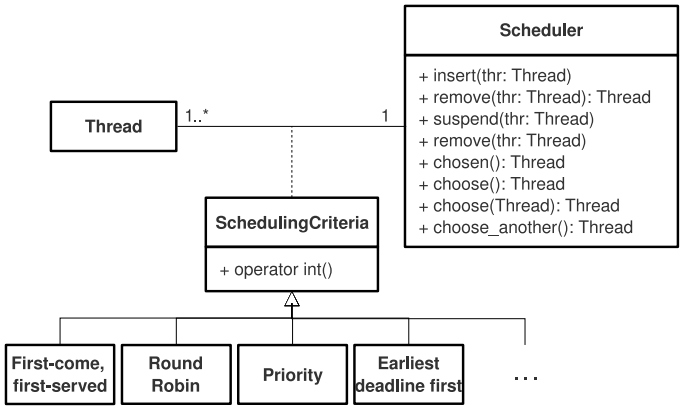

Figure 2. Simplified UML view of the task scheduling model

The implementation of the scheduler in hardware follows a well-defined structure. It has an internal memory that implements an ordered list. One process (*Controller*) is responsible for interpreting all the data received by the interface of the component in hardware and then to activate the process responsible for implementing the functionality requested by the user (through the command interface register). This implementation, as the software counterpart, realizes the insertion of its elements already in order, that is, the queue is always maintained ordered, following the information that the *SchedulingCriteria* provides.

5.2. Aspects implementation

We have implemented aspects for debugging. Unlike previous works [Vachharajani et al. 2004, Liu et al. 2010], which focused on simulation-time tracing and logging, we have implemented aspects for on-chip debugging. Figure 3 shows the debugged family of hardware aspects. The class *DebuggedCommon* defines common ports for all aspects. Besides the ports used for clock and reset, it defines outputs for a JTAG debug protocol (*trigger_out* and *data_out*) and for the enter/leave protocol (*op_rdy_out* and *op_req_in*). The input values for the ports defined by the subclasses determine which operation will be triggered.

The aspects implemented define the following debugging functionalities: *Watched* causes the state of a component to be dumped every time it is modified; *Traced* causes every operation execution to be signalized; and *Profiled* counts the number of clock cycles used by the component for each operation.

5.3. Scenario adapter implementation

Figure 4 shows how we applied the aspects to the scheduler using a scenario adapter (for simplicity, some details, such as methods, ports, and hierarchies, are omitted). The implementation follows the guidelines depicted in figure 1. The class *Scheduler* defines the scheduler component. The *controller* SystemC process is responsible for reading the component inputs and calling the method which implements the corresponding operation. The class *AdaptedScheduler* implements the scenario adapter. It inherits from the *Scheduler* and *Debugged* classes, and redefines the operation methods by adding calls to the *enter* and *leave* methods of *Debugged*. The *Debugged* class defines the scenario and its methods implement the handshaking protocol that triggers the aspects components.

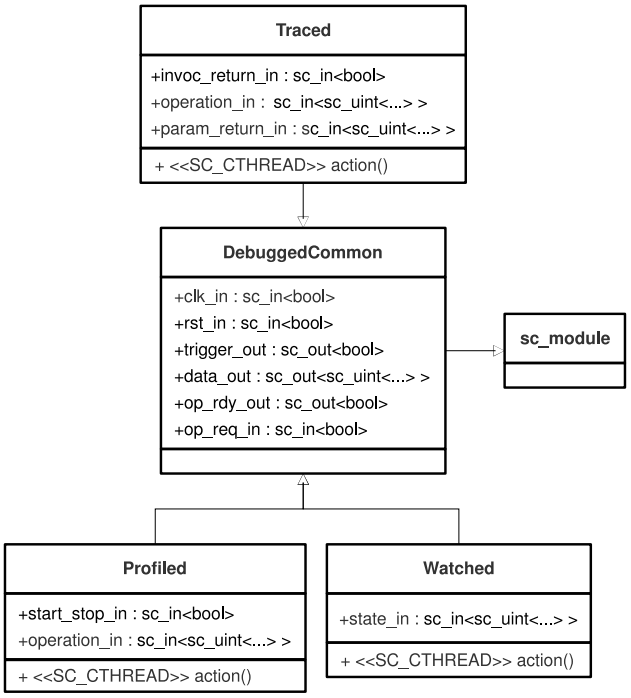

Figure 3. The debugged family of hardware aspects

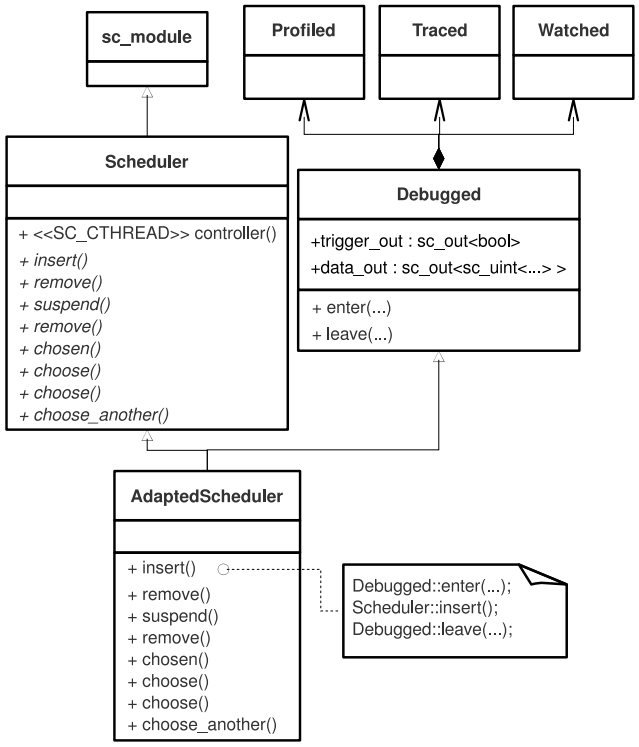

Figure 4. Scheduler modified by the scenario adapter

6. Results

We synthesized the SystemC designs described previously using Celoxica's Agility 1.3. They were synthesized to VHDL and EDIF formats targeting a Xilinx Spartan3 XC3S2000 FPGA. The final place-and-route was performed using Xilinx ISE 12.3. All the synthesis processes were executed with all the optimization enabled. Tables 1 and 2 show the results. *Normal scheduler* is the scheduler component without any modification, *Debugged scheduler—scenario adapter* is the scheduler modified with the scenario adapter while *Debugged scheduler—hand coded* is the scheduler with the aspects functionalities hand coded. Table 2 also shows the debugged family synthesized in isolation.

Table 1. Hardware resources estimated by Agility.

Parameter	Normal scheduler	Debugged scheduler hand coded	Debugged scheduler scenario adapter
4-input LUTs	2887	2978	3037
Flip Flops	663	754	842
Longest path delay (ns)	32.76	32.76	32.76

Table 2. Hardware resources used after placing and routing Agility's netlists.

Parameter	Normal scheduler	Debugged scheduler hand coded	Debugged scheduler scenario adapter	Profiled	Traced	Watched
4-input LUTs	2942	3042	3097	30	40	11
Flip Flops	663	754	842	28	41	12
Occupied Slices	1563	1668	1685	21	22	8
Longest path delay (ns)	23.28	22.58	23.28	4.79	6.00	4.70

The results show that the use of scenario adapters yields a very low overhead in terms of both resource consumption and performance. For the scenario-adapted scheduler, the number of occupied slices is about 1% higher than the hand-coded scheduler. This overhead comes basically from the additional signal and registers required by the handshaking protocol that is used to trigger the aspects, which is not required when everything is coded within a single SystemC process. The difference in performance (given by the longest path delay) is about 3%. Curiously, in the final place-and-routed designs, the hand-coded scheduler has the smaller longest path delay. This may be the result of some optimization algorithm applied in the place-and-route backend.

7. Conclusion

In this paper we have shown how a domain engineering strategy and AOP techniques can be applied to design and implement a flexible task scheduler in hardware. The scheduler's dependencies from a debugging execution scenario were encapsulated in aspects and further applied to the core component through the use of a scenario adapter, thus providing a better separation of concerns. The results showed that our design artifacts can be synthesized and introduce a negligible overhead in the generated components.

Acknowledgments

This work was partially supported by the *Coordination for Improvement of Higher Level Personnel* (CAPES) grant, projects RH-TVD 006/2008 and 240/2008, and by the *German Research Council* (DFG) under grant no. SCHR 603/7-1.

References

Bainbridge-Smith, A. and Park, S.-H. (2005). ADH: an aspect described hardware programming language. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 283 – 284.

Burapathana, P., Pitsatorn, P., and Sowanwanichkul, B. (2005). An Applying Aspect-Oriented Concept to Sequential Logic Design. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 819–820, Washington, DC, USA. IEEE Computer Society.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Déharbe, D. and Medeiros, S. (2006). Aspect-oriented design in systemC: implementation and applications. In *Proceedings of the 19th annual symposium on Integrated circuits and systems design*, SBCCI '06, pages 119–124, New York, NY, USA. ACM.

Endoh, Y., Imai, T., Iwamasa, M., and Kataoka, Y. (2008). A pointcut-based assertion for high-level hardware design. In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '08, pages 4:1–4:6, New York, NY, USA. ACM.

Engel, M. and Spinczyk, O. (2008). Aspects in hardware: what do they look like? In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '08, pages 5:1–5:6, New York, NY, USA. ACM.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Fröhlich, A. A. and Schröder-Preikschat, W. (2000). Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA.

Jun, Y., Tun, L., and Qingping, T. (2009). The application of Aspectual Feature Module in the development and verification of SystemC models. In *Specification Design Languages, 2009. FDL 2009. Forum on*, pages 1 –6.

Kallel, M., Lahbib, Y., Tourki, R., and Baganne, A. (2010). Verification of systemc transaction level models using an aspect-oriented and generic approach. In *Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010 5th International Conference on*, pages 1 –6.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK. Springer-Verlag.

Liu, F., Mohamed, O. A., Song, X., and Tan, Q. (2009). A case study on system-level modeling by aspect-oriented programming. In *Proceedings of the 2009 10th International Symposium on Quality of Electronic Design*, pages 345–349, Washington, DC, USA. IEEE Computer Society.

Liu, F., Tan, Q., Song, X., and Abbasi, N. (2010). AOP-based high-level power estimation in SystemC. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 353–356, New York, NY, USA. ACM.

Marcondes, H., Cancian, R., Stemmer, M., and Fröhlich, A. A. (2009). On the Design of Flexible Real-Time Schedulers for Embedded Systems. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 02*, CSE '09, pages 382–387, Washington, DC, USA. IEEE Computer Society.

OSCI (2010). Systemc synthesizable subset draft 1.3.

Panda, P. R. (2001). SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 75–80, New York, NY, USA. ACM.

Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002). AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Vachharajani, M., Vachharajani, N., and August, D. I. (2004). The liberty structural specification language: a high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 195–206, New York, NY, USA. ACM.

Vax, M. (2007). Conservative aspect-orientated programming with the e language. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 149–160, New York, NY, USA. ACM.