Aspects in Hardware – What Do They Look Like?

Michael Engel
Technische Universität Dortmund
Embedded System Software
michael.engel@tu-dortmund.de

Olaf Spinczyk
Technische Universität Dortmund
Embedded System Software
olaf.spinczyk@tu-dortmund.de

ABSTRACT

Aspect-oriented programming methods today have gained a significant following in the area of object-oriented high-level programming languages since their invention more than ten years ago. More recent developments have also found use cases for AOP in procedural programming languages operating at the system programming level. However, if one digs further down towards the hardware layer, only few signs of AOP usage can be found so far.

This paper motivates the use of aspect-oriented approaches in hardware development, which today is mostly done in domain-specific hardware description languages (HDLs). These languages deviate from the programming language model by providing explicit notions for concurrency and time, resulting in synthesizable circuit descriptions that can be turned into a piece of hardware. A survey of crosscutting concerns in hardware descriptions and a first definition of join-points and pointcuts for HDLs is augmented by an aspect-related analysis of a production-quality hardware component and an overview of current developments regarding AOP and hardware development.

1. INTRODUCTION

Hardware description languages (HDLs) are used to create a formal description of electronic circuits. HDLs cover a large part of the design process, ranging from the operation of the circuit, its design and organization to tests and verification methods. Figure 1 shows the suitability of various HDLs in the stages of the hardware design process.

The description of hardware components can reside of any of three levels (or a combination thereof). The lowest level is the transistor level, which specifies single transistors and their interconnects. A more abstract way of modeling hardware components is available using logic gates (like AND, OR, XOR); still more comfortable is a description on register-transfer level (RTL), which enables a high-level description of synchronous digital circuits.

Hardware descriptions of complex systems often consist of many thousand lines of HDL code, which are split over dozens of files implementing different components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACP4IS '08 March 31st 2008, Brussels, Belgium.
Copyright 2008 ACM [to be supplied] ...$5.00.

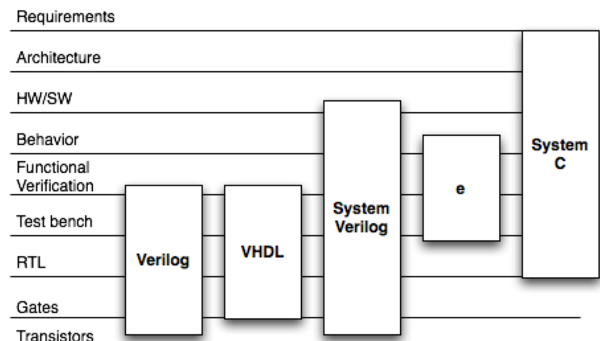

Figure 1: A Comparison of HDLs

In contrast to most (software) programming languages, HDLs provide explicit notations for expressing time and concurrency – two essential attributes of hardware designs.

One of the most commonly used HDLs is VHDL[1], which was standardized in IEEE standard 1076 [16]. VHDL allows the description of structural, physical and behavioral characteristics of digital systems. As such, the approach to describing circuits is rather different from normal sequential programming. It is used to create gate-level and register transfer logic designs. In addition, it serves as a tool for writing test benches and to perform functional evaluation. In this paper, we concentrate on the hardware design aspect of VHDL on the RTL level, where multiple components from various sources are often combined to form a complex system-on-chip. Similarities of hardware description languages and programming suggest using an aspect-oriented approach to handle these crosscutting concerns. Here, the differences in semantics between programming languages and hardware description languages have implications for aspects in HDLs, which are discussed below.

The paper is structured as follows. Section 2 gives an introduction to VHDL. In section 3, VHDL is analyzed as to possible crosscutting concerns compared to programming languages and a first model for aspects in VHDL is presented. Section 4 gives an analysis of cross-cutting concerns in a production-quality VHDL description. Related work is discussed in Section 5. Section 6 concludes the paper and outlines our goals for future research.

[1]VLSI Hardware Description Languages – VLSI = "Very Large Scale Integrated (Circuits)"

2. AN OVERVIEW OF VHDL

Descriptions of hardware components in VHDL are simple text files containing control structures and commands that are usually processed by a specific compiler system. Based on a VHDL description, a circuit can be created ("synthesized") by the compiler. In addition, simulators use interpretation of VHDL to enable the "execution" of a hardware description, which allows the designer to validate the design of a system prior to fabrication. VHDL includes several features to improve simulation, e.g., to specify delays of components, which are not synthesizable.

VHDL supports both structural and behavioral description of a system at multiple levels of abstraction. Structure and behavior are complementary methods to describe a system – the behavior of a system does not include information about its structure or the components that it includes whereas there are many different ways to structure a system to provide the same behavior.

The behavioral properties of a circuit are shown in fig. 2. The half-adder consists of two gates; an event on wire "a", changing from 1 to 0, changes the outputs of both gates after a given propagation delay (in this case, 5ns); both gates and the connecting wires have inertia. An additional property of this circuit is its concurrency. Both the `XOR` and the `AND` gate compute new output values concurrently whenever an input signal changes its state.

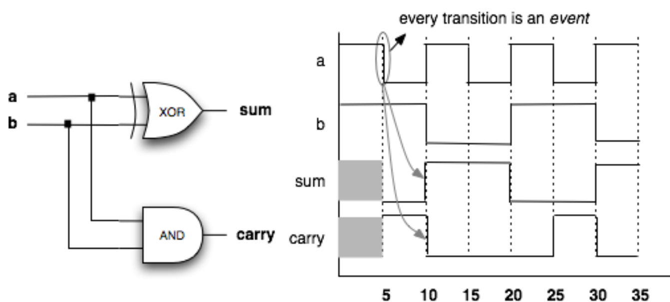

Figure 2: A Half-Adder

In a VHDL description, the following items can be specified:

- The components of a circuit
- The interconnection of components
- The behavior of components regarding their input and output signals

So, in contrast to programming languages, a VHDL system is data driven. Events on signals lead to computations that may generate events on other signals. A signal can initiate concurrent computations in different, possibly unrelated parts of a circuit.

Basic Language Concepts

Signals in VHDL describe wires. Like variables in programming languages such as C, they can be assigned values. In VHDL, values include binary bits (0 or 1) and arrays of bits, but also more complex descriptions that closer model real physical hardware are possible. For example, a tri-stated line[2] has the additional value "Z"; a dont-care bit[3] is denoted by the special value "X".

Like variables, signals also have an associated time value. A signal receives a value at a specific point in time and retains that value until it receives a new value at a future point in time. The sequence of values assigned to a signal over time is the *waveform* of that signal.

Components of a digital system are described using *design entities*. A description consists of two parts: the *interface* to the design is contained in the *entity declaration*, whereas the internal behaviour of the design is given in an *architecture* construct. The entity for an half-adder from fig. 2 looks like this:

```
entity half_adder is port(
    a, b: in bit;
    sum, carry: out bit);
end half_adder;
```

Here, `half_adder` is the name given to the design entity; the input and output signals, `a`, `b`, `sum` and `carry` are referred to as *ports*. Each port has a type (e.g., `bit` for single bits or `bit_vector` for arrays of bits) and a mode (`in` for signals entering the entity, `out` for signals generated by the entitiy or `inout` for bi-directional signals).

So far, this is no different to the declaration of a function. The implementation of the entity is contained in the architecture construct. Unlike functions, inside of an architecture description, all actions take place concurrently. Signal assignment statements specify the new value and (for simulation purposes) the time at which the signal is to acquire this value. The textual order of the concurrent signal assignments does not effect the results. The architecture for the half-adder entity looks like this:

```
architecture half_adder_arch of half_adder is
  begin
    sum <= (a xor b) after 5 ns;
    carry <= (a and b) after 5 ns;
  end half_adder_arch;
```

In addition to input and output signals, an architecture may also contain signals local to that architecture. These are declared before the `begin` statement of the architecture.

Control Structures

To enable a more high-level description of a system, control structures allow for conditional setting of signals.

A multiplexer that selects its output signal from one of four input signals according to selection bits can be described as follows using conditionals:

```
entity mux4 is port(
  in0, in1, in2, in3: in bit;
  sel0, sel1: in bit;
  Z: out bit_vector(7 downto 0);
end mux4;
architecture behavior of mux4 is
  begin
    Z <= in0 after 5 ns when S0='0' and S1='0' else
         in1 after 5 ns when S0='0' and S1='1' else
         in2 after 5 ns when S0='1' and S1='0' else
         in3 after 5 ns when S0='1' and S1='1'
  end behavior;
```

[2]A signal that can attain a high-impedance state in addition to voltage levels for 0 and 1

[3]Indication that specific bit is not relevant, e.g., in comparisons

Here, the first conditional found to be true determines the value set on the output. A similar statement reminiscent of `switch` in C-like languages is available with the *selected signal assignment*:

```
with address select
  reg_out <= reg0 after 5 ns when "000", ...
```

Processes

In cases when component behavior cannot simply be modeled as delay elements, processes provide a sequential execution of assignments similar to functions in regular programming languages. Processes may contain variables that can be assigned values; unlike signals, variable assignments take effect immediately. In addition, control-flow is possible in the form of `if-then-else`, `case` and `loop` statements. Signal assignments to external signals is also possible.

To determine when a process is to be executed, a *sensitivity list* specifies the signals which determine when the process executes. The half-adder can also be described using a process:

```
entity half_adder is port(
  a, b: in bit;
  sum, carry: out bit;
end half_adder;

architecture behavior of half_adder is
  begin
    sum_proc: process(a, b) begin
      if (a = b) then
        sum <= '0' after 5 ns;
      else
        sum <= (a or b) after 5 ns;
      end if
    end process;

    carry_proc: process(a, b) begin
      case a is
        when '0' => carry <= a after 5 ns;
        when '1' => carry <= b after 5 ns;
        when others => carry <= 'X' after 5 ns;
      end case;
    end process;
  end behavior;
```

Here, two processes `sum_proc` and `carry_proc` are defined that are started when either of the input signals `a` or `b` changes its state. Both processes execute in parallel, the instructions inside each process are executed sequentially.

Loops in VHDL are restricted to run over pre-determined ranges of values. `for` and `while` loops are available:

```
for index in 1 to 32 loop
  ...
end loop

while (j < 32) loop
  ...
  j := j + 1;
end loop;
```

Since we can only give a short overview of VHDL in the context of this paper, we would like to refer the interested reader to the VHDL cookbook[2] for detailed information.

3. ASPECTS IN VHDL

Since describing circuits VHDL takes a different approach to regular programming languages, the nature of crosscutting concerns is different. However, existing similarities between HDLs and programming languages give some hints as where to search for crosscutting concerns[10].

Crosscutting Concerns

Entities are the unit of encapsulation in VHDL. They can roughly be compared to classes in object-oriented programming. The components described in entities are often modeled after traditional hardware design methods – an entity represents the description of what would former have been a separate integrated circuit. Thus, these entities form the primary concerns of most designs.

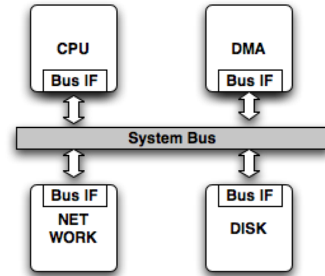

Figure 3: A Bus System

In complex circuits, interconnection of several entities is realized by introducing *buses* consisting of several signals common to all connected components. Fig. 3 shows a bus system with four connected components. A bus is a concern of its own, consisting of a set of wires and agreed-upon protocols that define the arbitration methods of the bus, i.e., which component connected to the bus may transfer data on the bus at a given time. In traditional designs, the bus interface and arbitration method has to be implemented in every attached component separately; thus, changes to the bus structure (like a different bus width) or behavior (changing the arbitration method) will require changes in all components, as shown in fig. 3.

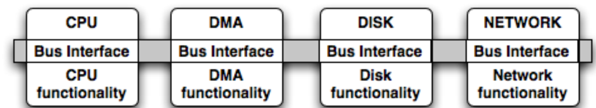

Figure 4: Crosscutting Concerns in a Bus System

Further examples of possible crosscutting concerns in many hardware designs can be found in parts of a system that control or observe its overall functionality or implement non-functional properties.

This includes scan chains like JTAG which provide for an external ability to introspect the digital system, clocks that are derived from a central master clock and used in several entities alike, and other central control signals like a global RESET line.

Of course, the precise nature of the crosscutting concerns in a given design depends on the chosen modularization of the overall system. Described above are typical crosscutting concerns that are derived from traditional, circuit-driven approaches to hardware development with VHDL.

Possible Join-Point Types

In aspect-oriented programming languages like AspectC++ [15] or AspectJ [9], join-points are mostly related to events

taking place in the sequential control flow of a system. This model is not appropriate for describing join-points in VHDL, since it does neither provide for concurrently executing processes. In addition, many join-point descriptions are based on the notion of method calls. In VHDL, processes – which can be considered as the most similar element to methods in programming languages – are not called *explicitly*, but rather *implicitly* by setting a signal that is included in the processes sensitivity list in some other component of the hardware design.

As a consequence, a first, simple, join-point model for VHDL includes join-points at

- the *execution* of a process and
- the *setting* of a signal.

One can imagine more advanced types of join-points, e.g. including the execution of commonly used control structures like loops or case-statements. However, even these simple join-points can be quite useful when designing hardware. Section 4 describes a real-world example that makes use of these join-point types.

Pointcut Descriptions

Analogous to the aforementioned AOP languages, pointcuts for VHDL consist of a set of join-points combined using first-order logic formulae and wildcards to create a set of join-points.

In the case of VHDL, its inherent concurrency has interesting consequences. Consider, for example, combining join-points at the setting of a signal using the "and" operator. In programming languages, a combination of two `call(...)` join-points using an `&&` operator normally would not match anything. In VHDL, however, signals can be set concurrently, so a pointcut combining two such join-points would actually make sense.

Additionally, it would be useful if one was able not only to describe single events, but rather to capture a sequence of events. Here, temporal logic could be used to create more complex pointcut descriptions[1]. This is also an interesting area for future research.

Special care must also be taken when combining join-points that reside in concurrent vs. sequential parts of the VHDL code. This is discussed in detail below.

Types of Advice

As a first approach, the types of advice useful in an aspect-oriented extension of VHDL can be modeled after well-known advice types. However, due to the concurrent nature of VHDL, *before* and *after* advice are only well-defined in the context of processes.

An *around* advice, however, can replace a signal assignment in a concurrent environment. Depending on the context, the *proceed* statement contained in the advice code will have to behave differently. If the related join-point is inside of a process, it will cause the original code (e.g., a signal assignment) to take place at the point in time *proceed* is executed. Outside of a process, however, timing is not relevant, thus proceed will create an additional, *concurrent* signal assignment.

One important question is whether the code inside an advice is concurrent or sequential. If a pointcut for an advice includes join-points in concurrent structures, including concurrent code in the advice will result in additional, possibly conflicting, signal settings. Using a process inside that advice would be analogous to including the signal set at the related join-point in the sensitivity list of that process.

If, however, a pointcut for an advice contains join-points in sequential VHDL code, then the execution of the advice code as a sequential process in the context of the original process containing the join-point is the only useful alternative.

Overall, the most important differences between AOP in hardware description languages to classical aspect-oriented programming languages are caused by the implicit conocurrency in HDLs (which may also be a problem for aspects in parallel programming languages). While a process is similar to a kind of advice, crosscutting concerns can still be found inside the code of a process. These properties are demonstrated using VHDL code in the following section.

4. CODE ANALYSIS: A UART DESCRIPTION

In analogy to the open source software movement, the approach of describing hardware using a HDL has led to an open hardware movement. This gives the chance to take a look at HDL implementations of complex circuits. Such an analysis was formerly impossible since hardware developers were guarding their intellectual property.[4]

In this section, we verify our assumptions about crosscutting concerns by examining an implementation of a UART[5] as an example of a circuit used in real-world applications.

A UART – Universal Asynchronous Receiver and Transmitter – is a circuit used for bit-serial communication between computer systems and peripheral devices like modems or printers. One common UART is the 16550, which is available as an open source VHDL implementation from QuickLogic, Inc. [11].

The 16550 UART consists of five interconnected entities (see fig. 5): transmitter, receiver, modem interface, baudrate generator and interrupt controller. This structure is reflected in the VHDL source code, which consists of the files `uart16550_baudgen`, `uart16550_intgen`, `uart16550_modem`, `uart16550_rx` and `uart16550_tx`. In addition, the file `uart16550_top` contains the top-level description of the circuit, i.e., its external bus interfaces.

Each of the VHDL source files contains an entity part describing the components' interface and an architecture part consisting of several parallel processes implementing the required functionality.

RESET handling

One common problem of digital circuits is the handling of RESET conditions, in which the component is to be initialised. This code is usually scattered over the various components of the design, each process that includes state, that has to be reset asynchronously has the RESET signal in its sensitivity list, see fig. 6. Overall, there are 14 processes in the UART's source code that are sensitive to the RESET signal contained in the source spread over five source code files.

The code implementing the various reset functions can be factored out into an aspect. Fig. 7 shows a hypothetical VHDL AOP extension. Here, the pointcut describing the affected code positions has to match all functions that are sensitive to a reset. Since a process is started when *any* of the

[4] One important repository for open source hardware is http://www.opencores.org

[5] Universal Asynchronous Receiver and Transmitter

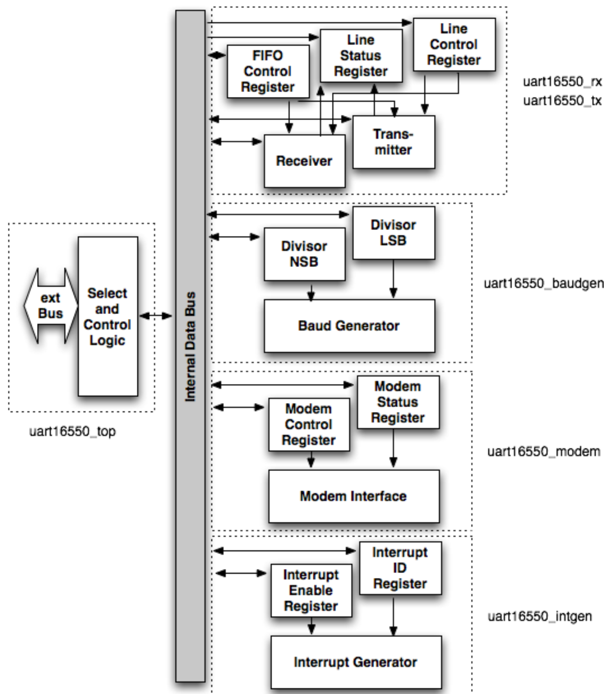

Figure 5: 16550 UART Structure

signals in its sensitivity list change, this pointcut includes all of the functions that include RESET in their sensitivity list. The corresponding advice function can then implement the reset of all signals.

System Clock Handling

Another crosscutting concern in the UART code is clock handling. There are eight different processes that depend on the system clock (SCLK). The processes are distributed over the source files for the receiver, transmitter, and the top level design file. In each of these processes, the level sensitivity of SCLK is checked via

```
if (SCLK'event and SCLK = '1') then
   ...
end if
```

Here, one possible application for aspects is to change the clock base by introducing a local signal and creating a clock divider. When implementing this additional functionality, code at all of join-points, the eight processes that include the SCLK signal in their sensitivity list, is affected.

5. RELATED WORK

The earliest mention of aspect-oriented approaches in hardware design can be found in Kiczales' original paper on aspect-oriented programming[12]. Here, he discusses the Ptolemy project[7], which studies modeling, simulation, and design of concurrent, real-time, embedded systems with a focus on assembly of concurrent components. Interaction between heterogeneous components in Ptolemy is handled with concepts that are similar to aspects.

AOP in SystemC

In [6], AOP is applied to applications developed using SystemC [17] and AspectC++ [15]. Important system aspects like metrics measure, communication and cache policies are modelled. However, synthesizing real hardware from SystemC descriptions is difficult, so this approach is rather more useful in the area of simulation and verification.

```
-- Shift FSM Logic
process (sample, RESET)
begin
    if (RESET = '1') then
        rsr_fsm_state <= Idle_State;
        fsm_count <= "0001";
        par_hold <= '0';
        rx_flags(0) <= '0';
        rsr <= x"00";
    elsif (sample'event and sample = '1') then
        ...
end process;

-- generate a pulse at every rising edge
process (SCLK, RESET)
begin
    if (RESET = '1') then
        baud_old <= '0';
    elsif (SCLK'event and SCLK = '1') then
        ...
end process;

-- Status Register [3:0] - set when modem signals change
process (CLK, RESET)
begin
    if (RESET = '1') then
        status(3 downto 0) <= "0000";
    elsif (CLK'event and CLK = '1') then
       ...
end process;
```

Figure 6: RESET handling

```
pointcut reset: execution(process(...RESET...))

around(reset) {
    if (RESET = '1') then
        -- Shift FSM Logic
        rsr_fsm_state <= Idle_State;
        fsm_count <= "0001";
        par_hold <= '0';
        rx_flags(0) <= '0';
        -- generate a pulse at every rising edge
        baud_old <= '0';
        -- Status Register [3:0]
        -- they are set when modem signals change
        status(3 downto 0) <= "0000";
    else
        proceed();
    end
}
```

Figure 7: RESET advice

ADH

ADH [3] is a domain-specific HDL focusing on control modeling problems as well as the signal and image processing domain. A compiler translates the ADH descriptions into VHDL for synthesis. ADH supports the definition of pointcuts in hardware descriptions and before, after and around advice. However, the nature of the join-points in ADH does not take the inherent parallelism into account, it mostly concentrates on using AOP in sequential control flows for signal and image processing.

The e Verification Language

In [18], aspect-oriented approaches to verification of digital systems using the *e* hardware description language [10] are

described. While the paper gives a good overview of possible causes for crosscutting concerns in hardware description languages, the e language itself is rather limited in its support for AOP. For example, there is no notion of pointcuts, since an advice may only relate to a single join-point. In addition, e is restricted to the verfication of hardware. Due to the lack of quantification, we consider this approach as highly-limited AOP – if at all.

Other developments

In [4], the authors describe a first idea to use AOP to formulate crosscutting concerns in hardware description languages. However, the only example given is the management of distributed clocks in hardware systems; an implementation does not seem to exist.

The options approach [5] is the application of a concept from economy to the problem when to introduce AOP-based constructs in hardware design. However, the AOP concepts this decision relies on is based on the concepts presented in the paragraph above[4], so it should be re-evaluated when a more complete understanding of AOP for hardware design has evolved.

The analysis of related work on aspects in hardware design has shown that there is considerable interest in applying AOP techniques in this domain. However, the existing approaches concentrate on design verification rather than the design process itself, are not verified using real-world code or are only applicable in a domain-specific setting.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed the nature of crosscutting concerns in VHDL-based hardware designs using a real-world code example. This has shown that aspects in hardware exist and that there are cases where an AOP approach to hardware design in VHDL are useful. As a basis for AOP in VHDL, we determined simple join-point types and pointcut descriptions and noted important differences between aspects in programming languages and aspects in HDLs.

The UART description analyzed in section 4 is production quality VHDL code and used in commercial products. While it emulates a complete UART chip, there exist far more complex VHDL descriptions. Further analyses of more complex VHDL models regarding the inherent crosscutting concerns will lead to an improved understanding of the potential for aspect-oriented approaches in VHDL. Possible targets for investigation include open source CPUs like Sun OpenSparc T1 and T2 CPUs[13], the Leon-3[8] SPARC-compatible CPU and the OpenRISC 1200 processor[14].

Of further interest will be the analysis of complete systems-on-chip (SOCs), which not only include a CPU core, but also required peripherals like bus controllers, communication cores and video generators. Due to the increased complexity of the systems, we expect a high number of crosscutting concerns to show up.

One interesting alternative approach to implementing aspects in hardware descriptions on the VHDL layer lies in the combination of SystemC and AspectC++. This is currently only used to implement AOP for verification and testing of circuits, since compiling SystemC descriptions into hardware is still an open research topic. Using future SystemC-to-VHDL compilers, AOP-based hardware designs could also be implemented in SystemC. However, the SystemC developer would have to execute great care not to introduce non-synthesizable constructs by using aspects for an otherwise synthesizable (i.e., translatable to synthesizable VHDL) SystemC code base.

7. REFERENCES

[1] R. Åberg, J. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur, *On the Automatic Evolution of an OS Kernel Using Temporal Logic and AOP*, Proc. of the IEEE Conf. on Automated Software Engineering, Montreal, Canada, IEEE, 2003, pp. 196–204.

[2] P. Ashenden, *The VHDL Cookbook*, University of Adelaide, 1 ed., July 1990.

[3] A. Bainbridge-Smith and S.-H. Park, *ADH: An Aspect Described Hardware Programming Language*, Proc. of the International Conference on Field-Programmable Technology, Singagore, IEEE, 2005, pp. 283–284.

[4] P. Burapathana, P. Pitsatorn, and B. Sowanwanichkul, *Applying Aspect-Oriented Concept to Sequential Logic Design*, Prof. of ISNG, IEEE, 2005, pp. 63–68.

[5] S. Chaiworawitgul and D. Sutivong, *The Options Approach to Hardware Design Decision: Switching from Object-Oriented to Aspect-Oriented Concepts*, Engineering Management Conference, IEEE, 2006, pp. 204–208.

[6] D. Déharbe and S. Medeiros, *Aspect-Oriented Design in SystemC: Implementation and Applications*, Proc. of SBCCI '06 (New York, NY, USA), ACM, 2006, pp. 119–124.

[7] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, *Taming heterogeneity—the Ptolemy approach*, IEEE Special Issue on Modeling and Design of Embedded Software, IEEE, 2002.

[8] J. Gaisler, *A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture*, Proc. of DSN, IEEE, 2002, pp. 409–415.

[9] E. Hilsdale, *Aspect-Oriented Programming with AspectJ*, TOOLS (39), IEEE, 2001, p. 368.

[10] Y. Hollander, M. Morley, and A. Noy, *The e Language: A Fresh Separation of Concerns*, Proc. of TOOLS '01 (Washington, DC, USA), IEEE, 2001, pp. 41–50.

[11] QuickLogic Inc., *Application Note 69: Open Source 16550 UART Core*, 2002.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-Marc Loingtier, and J. Irwin, *Aspect-Oriented Programming*, Proc. of ECOOP, 1997, pp. 220–242.

[13] P. Kongetira, K. Aingaran, and K. Olukotun, *Niagara: A 32-way multithreaded sparc processor*, IEEE Micro **25** (2005), no. 2, 21–29.

[14] OpenCores.org, *OpenRISC 1200 Specification*, 2006.

[15] W. Schröder-Preikschat, D. Lohmann, F. Scheler, W. Gilani, and O. Spinczyk, *Static and Dynamic Weaving in System Software with AspectC++*, Proc. of HICSS, Kauai, HI, USA, IEEE, 2006.

[16] The Design Automation Standards Committee of the IEEE, *IEEE Standard 1076-1993: VHDL*, IEEE, 1993.

[17] The IEEE Computer Society, *IEEE Standard 1666-2005: SystemC*, IEEE, 2006.

[18] M. Vax, *Conservative Aspect-Oriented Programming with the e Language*, Proc. of AOSD '07 (New York, NY, USA), ACM, 2007, pp. 149–160.