

A Pointcut-based Assertion for High-level Hardware Design

Yusuke Endoh, Takeo Imai, Mikito Iwamasa, and Yoshio Kataoka
System Engineering Laboratory, Corporate R&D Center, Toshiba Corporation
{yusuke.endoh, takeo.imai, mikito.iwamasa, yoshio.kataoka}@toshiba.co.jp

ABSTRACT

Verifying very-large-scale integration (VLSI) circuit design using assertions is becoming more common. Herein, an assertion represents a temporal relationship among circuit events over time using temporal logic expression. On the other hand, “high-level design” has also become more common recently. Instead of conventional hardware description languages (HDLs), VLSI designers introduce C-based design languages. Although both of these trends are fairly effective in VLSI development, there is a big gap between these two approaches. Since conventional assertion languages are for conventional HDLs, they only allow specification of change of variables or low-level events raised along with signals. In high-level design, however, there are various high-level events, such as a method call or a state transition which no conventional assertion language can handle.

This paper proposes a new assertion language extension, namely, a pointcut-based assertion that enhances an existing assertion language to make assertions available even in high-level design. We introduce a pointcut notion to specify various events from low-level signal-related ones to high-level state transition-related ones. To conduct proof-of-concept, we designed and implemented two assertion languages with pointcut-based assertions, called *ASystemC* and *ASpecC*. *ASystemC* uses the pointcut expressions in AspectC++, and the implementation also uses the AspectC++ compiler as its back-end. We present feasibility and preliminary evaluation of our approach with *ASystemC*. *ASpecC* is designed with practical use in mind and based on the *continuation join point model* with slight modification to support hardware-specific matters. We show that the model is useful for making pointcut-based assertion more robust.

1. INTRODUCTION

In recent years, *high-level design* has been attracting attention as a new method to accelerate the development life cycle of hardware design. In high-level design, we describe a system-level model of VLSI design by using not traditional hardware description languages (HDLs) such as VHDL [5] and Verilog HDL [18], but C language-based languages such as SpecC [3] and SystemC [4]. By analyzing and/or simulating the abstract model, circuits under development can be tested at an earlier stage of the design process.

On the other hand, *assertion-based verification* is commonly used in the field of hardware design where an assertion is a logical formula representing circuit properties. Many assertion languages, including Property Specification Language (PSL) [6], are based on temporal logic such as

linear temporal logic (LTL) [15]. In short, an assertion represents an order of events of a circuit. An assertion verifier checks whether assertions hold true by using dynamic simulation and/or static analysis of HDL code, in order to make the code highly dependable.

Our purpose is to enable fast and highly dependable development of hardware design by making assertions available even in high-level design. It is, however, difficult to combine high-level design with assertion-based verification directly since the types of events in high-level design are different from those in the HDL method. HDL code is composed only of wires, registers and their connections. This means that any events in the HDL model are low level or can be monitored as signal variation. Thus, traditional assertion languages are designed only to specify changes of variables. In high-level design, however, many kinds of high-level events that cannot be monitored as changes of variables may be raised. For example, method calls are very often used to represent communication between hardware modules. Therefore, it is of little use to directly combine high-level design with assertion-based verification.

This paper proposes a language extension, *pointcut-based assertion*, that extends assertion languages with pointcut. We consider that almost any kind of high-level event corresponds to join point in aspect-oriented programming. Our pointcut-based assertions are represented as logical formulae consisting of pointcuts instead of specifiers to variable changes. So the pointcuts specify events, including high-level events, and logical formulae specify temporal relationships between the events over time. In this manner, our pointcut-based assertions allow us to combine high-level design and assertion-based verification.

In addition, to confirm the feasibility of our approach, we designed and implemented two assertion languages with pointcut-based assertions, namely, *ASystemC* and *ASpecC*. They work alongside SystemC and SpecC, respectively. *ASystemC* is mainly designed with feasibility test and evaluation of our approach in mind. It uses pointcut of AspectC++, and its implementation translates assertions into aspects of AspectC++. We thus can enjoy full expression of sophisticated pointcuts that AspectC++ provides. On the other hand, *ASpecC* is an experimental testbed for investigation of practical use. It is based on the *continuation join point* [2, 12] with modification for easier handling of proper factors for hardware design or clock. These two languages show that pointcut-based assertions work effectively, and that continuation join point is useful for making pointcut-based assertions more robust.

The rest of the paper elaborates on these topics as follows: Section 2 illustrates high-level design and assertion-based verification with a typical example. In Section 3, we propose a pointcut-based assertion and explain how it solves the difficulty. We then explain design and implementation of ASpecC and ASystemC in Section 4. Section 5 compares our approach with related work. In Section 6, concluding remarks are presented and future work indicated.

2. BACKGROUND

This section clarifies the nature of high-level design by comparing two example modules written in a traditional hardware description language (HDL) and a high-level language. We then illustrate assertion-based verification by showing example assertions for the two models, and finally consider difficulty in directly combining the two methods.

2.1 High-level Design

Figure 1 shows a bus inputter written in Verilog HDL [18]. It defines one block that waits for the order and inputs from the bus.

```

1 always @(posedge clk) begin
2   case(state)
3     'READ_REQ: begin
4       result <= 16'hzzzz;
5       if(must_read) begin
6         bus_addr <= 16'habcd;
7         bus_en <= 1'b1;
8         bus_wr <= 1'b0;
9         state <= 'READ_ACK;
10      end
11     else
12       bus_addr <= 16'hzzzz;
13       bus_en <= 1'bz;
14       bus_wr <= 1'bz;
15     end
16   'READ_ACK: begin
17     result <= bus_data;
18     bus_addr <= 16'hzzzz;
19     bus_en <= 1'bz;
20     bus_wr <= 1'bz;
21     state <= 'READ_REQ;
22   end
23 end
24 end

```

Figure 1: A bus inputter in Verilog HDL

Line 1 starts defining a block that will be triggered whenever the variable `clk` is signaled. This block has two states: `READ_REQ` and `READ_ACK`. When the state is `READ_REQ`, the variable `result` is *not set*¹ (lines 4). Then, if the variable `must_read` is signaled (line 5), the block issues a “read” request by signaling the variables of the bus (lines 6–8), and then the block changes its own state to `READ_ACK`. If not, the block does not set the variables of the bus (lines 11–13) and does not change its own state. When `READ_ACK`, the block saves the response `bus_data` to the variable `result`, then sets “no request” into the bus, and returns its own state to `READ_REQ` (lines 16–21). To summarize; this block reads from the bus whenever the variable `must_read` is signaled.

¹In Verilog HDL, a variable will have a undefined value when multiple modules set the variable at the time. Thus, when a module does not outputs a value to a variable, it must be represented explicitly by writing high impedance (the immediate value `z`).

Note that all states are represented explicitly as the variables (e.g., the state is explicitly represented as `state`).

SpecC [3] can model this more abstractly. Figure 2 illustrates the same bus inputter shown in Figure 1 written in SpecC. This defines one module². First, the main loop (lines

```

1 behavior inputter(i_bus bus,
2                   i_receive must_read);
3 void main(void) {
4   while(true) {
5     must_read.receive();
6     data = bus.read(0xabcd);
7   }
8 }
9 };

```

Figure 2: A bus inputter in SpecC

4–7) calls the method `must_read.receive` (line 5), which may be blocked until the block must start reading. Line 6 issues a read request to the bus and saves the response to the variable `data`. The method call `bus.read` may be blocked too until the data of the bus is available.

Unlike Verilog HDL, a code written in SpecC represents the states and communications as the parts of the code currently being executed, which is a very common manner in software programming.

The model in SpecC is more abstract than the one in Verilog HDL (e.g., clock does not show up at all in SpecC model). Thus, SpecC helps designers to make a lightweight model quickly and accelerates the development life cycle.

2.2 Assertion-based Verification

Assertion-based verification is now commonly used in the field of HDL. An assertion represents a formal specification as a temporal logical formula that represents circuit properties. The verification tool for assertion checks whether all assertions hold true by using dynamic simulation and/or static analysis of the HDL code.

Property Specification Language (PSL) [6] is one of the assertion languages. Assertions in PSL are composed of boolean expressions written in HDL together with temporal operators. In other words, boolean expression is used as the atom of temporal logic and means the event where the expression is evaluated to true.

The temporal operators and sequences describe the relationships between events over time. For example, `always P` is a temporal operator that declares the sub-assertion `P` must hold true at an arbitrary timing, and `eventually! P` declares that there will sure be a future timing at which the specified sub-assertion `P` holds true. These temporal operators are based on temporal logic; see a reference [15] for details.

Figure 3 illustrates an example assertion written in the Verilog flavor of PSL³. This assertion is embedded within target HDL code as a comment⁴. It asserts the following property: the boolean expression `ack==1` must return true

²A module unit in SpecC is called a behavior, which represents a behavior of a concurrent process.

³PSL is specialized to some HDLs, such as VHDL and Verilog HDL. The writing styles in each flavor differ slightly.

⁴PSL also provides a way to write an assertion separately from HDL code.

```
// psl assert always
//      (req==1 -> eventually! ack==1);
```

Figure 3: An example assertion written in PSL

at the future timing whenever the boolean expression `req==1` returns true if it is evaluated. To put it simply, it declares “whenever the variable `req` is signaled, the block must signal the variable `ack` at the future timing.”

Figure 4 shows an assertion for the above-mentioned block in Figure 1. This asserts a property that: whenever `must_read` is signaled, the block must issue a request to input from bus at the next timing. In the assertion, the statement “`must_read` is signaled” is represented as “the value of `must_read` becomes 1”, and the statement “the block issues a request to input from bus” is represented as “the values of `bus_en` and `bus_wr`, respectively, become 1 and 0 simultaneously.”

```
// psl assert always must_read==1 ->
//      next (bus_en==1 && bus_wr==0);
```

Figure 4: An assertion for bus inutter in Figure 1

Note that the above assertions specify events only by boolean expressions.

2.3 Combining Two Methods and its Difficulty

No assertion can be written if events about that the assertion concerns cannot be specified as boolean expression. For example, we cannot assert the same property as Figure 4 for the circuit of Figure 2. The event “`must_read` is signaled” corresponds to “`must_read.receive` returns”, and “the block issues a request to input from bus” corresponds to “`bus.read` starts”. These events actually change no variable and then no boolean expression can specify them.

There is a workaround to solve this problem; to manually insert a dummy assignment where the desired event occurs. In the above situation, an instruction that assigns true to the dummy variable `leave_must_read_received` is inserted just below `must_read.receive()` call. Similarly, assignment to `enter_bus_read_entering` is inserted at just above `bus.read()` call. Figure 5 shows the result. However,

```
behavior inutter(i_bus bus, i_receive must_read) {
void main(void) {
while(true) {
must_read.receive();
leave_must_read_received = true;
enter_bus_read_entering = true;
data = bus.read(0xabcd);
}}};
```

Figure 5: A bus inutter including dummy assignment instructions to monitor events

this workaround causes scattering the fragments of the code in this way which is indeed a cross-cutting concern. It makes

the code difficult to read and maintain and also still breaks modularity not only of the code but of the assertion itself.

3. POINTCUT-BASED ASSERTION

3.1 Overview

We propose *pointcut-based assertion* or a language extension whose assertion is composed of pointcuts instead of boolean expression. In other words, the pointcut is used as the atom of temporal logic. To be precise, not only pointcut but also advice modifier (i.e., `before` or `after`) is needed to specify one instant of an event⁵. The pair allows sampling of high-level events such as method calls and state transition.

In this section, we explain the notation of a pointcut-based assertion by referring to AspectJ [11]-style pointcuts and PSL-style assertions. Figure 6 shows an example of a pointcut-based assertion. It has two pointcuts that, respectively, capture two timings when the function `send_req` starts and when `receive_ack` returns. It thus asserts that at any timing, `receive_ack` must be called and return eventually after `send_req` starts to be called.

```
assert always (
(before():call(void send_req(...))) ->
eventually! (after():call(void receive_ack(...)))
);
```

Figure 6: An example of pointcut-based assertion

3.2 Pointcut-based Assertion for Bus inutter in SpecC

Figure 7 shows an assertion for the bus inutter in SpecC (Figure 2). It checks the same property as the HDL assertion

```
assert always (
(after():call(void must_read.receive(...))) ->
next (before():call(int bus.read(...)))
);
```

Figure 7: A pointcut-based assertion for bus inutter in Figure 2

in Figure 4.

This is because pointcut can capture high-level events as join points such as method calls. Therefore, our assertion language allows assertions to be available in high-level design.

3.3 Compatibility with Traditional Assertion Language

Our pointcut-based assertions subsume traditional ones consisting of boolean expressions. In other words, traditional assertions can be encoded as pointcut-based assertions. For example, an assertion in Figure 8 encodes the assertion in Figure 3. Thus, our language can support traditional assertions by providing syntax sugar.

⁵This can be eliminated by using *continuation join point* [2, 12]-based pointcut, described later in Section 4.2.1.

```

assert always (
  (after():if(req == 1)) ->
  eventually! (before():if(ack == 1))
);

```

Figure 8: A pointcut-based assertion equivalent to the one in Figure 3

Note that the assertion in Figure 8 may make verification slow when simulation is performed. This is because this alone if pointcut may require checking the values of `req` and `ack` at every join point.

4. DESIGN AND IMPLEMENTATION

This section explains the design and implementation of two flavors of a pointcut-based assertion language, namely, `ASystemC` and `ASpecC`.

4.1 `ASystemC`

4.1.1 Design of `ASystemC`

`SystemC` is a high-level design language for developing abstract models of software algorithms, hardware architecture and system-level designs. It is implemented as a C++ template class library. Thus, programs written in `SystemC` can be compiled by any C++ compiler, including `g++`.

`ASystemC` gives `SystemC` a capability of writing and verifying a pointcut-based assertion. It applies `AspectC++`-style pointcuts and `PSL`-style assertions. Figure 9 shows an example of `ASystemC` assertion⁶. It asserts that it must be read eventually if `fifo` is written.

```

always (
  execution("% fifo::write(...)"):before () ->
  eventually! execution("% fifo::read(...)"):before ()
)

```

Figure 9: An example assertion of `ASystemC`

4.1.2 Implementation of `ASystemC`

Our prototype implementation of `ASystemC` consists of three parts: (1) an aspect generator from an assertion, (2) an aspect-oriented compiler, and (3) an LTL checker. We implemented the prototype of parts (1) and (3) in Haskell. We used `ag++` or the `AspectC++` [16] compiler, which is an aspect-oriented extension for C++, as part (2).

Figure 10 summarizes the executing flow of `ASystemC` and compares it with the traditional `SystemC`. Part (1) generates the code of an aspect (Figure 11) from the assertion (Figure 9). The output aspect monitors events specified by any assertion and outputs a log. Part (2) compiles the code of `SystemC` with this aspect. The executable file outputs a sequence of logs when it is run. Part (3) reads the sequence and checks whether the original assertion holds true.

⁶This assertion is written for the sample source code of `SystemC` `simple_fifo.cpp`, which is included with the `SystemC` reference implementation [14].

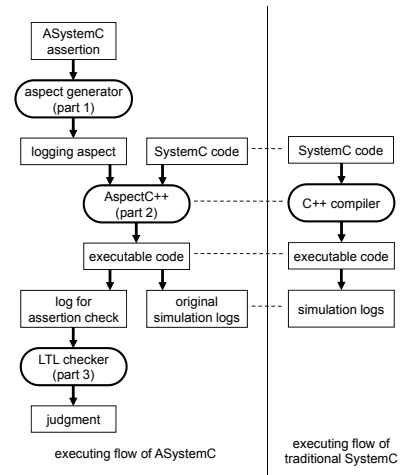


Figure 10: Comparison of executing flows of `ASys-temC` and traditional `SystemC`

```

aspect Foo {
public:
  advice execution("% fifo::write(...)"):before() {
    puts("[[execution(\"% fifo::write(...)\"):before()]]");
  }
  advice execution("% fifo::read(...)"):before() {
    puts("[[execution(\"% fifo::read(...)\"):before()]]");
  }
};

```

Figure 11: A generated aspect from the assertion of Figure 9

4.1.3 Limitation of `ASystemC`

The compilation time of `ASystemC` is longer than that of `SystemC`. We measured them as shown in Table 1⁷. We used Red Hat Enterprise Linux WS release 4 on a machine with four CPUs (Intel(R) Xeon(TM) 2.80GHz) and 8 GB memory to gather the experimental data. We also used `g++` version 3.2.3 and `ag++` version 0.6 (built: Mar. 16, 2006). The target programs are `simple_fifo`, `simple_bus` and `risc_cpu`, which are the sample programs distributed with the `SystemC` reference implementation [14]. Every measurement is performed three times and we calculated the average. We observed the compilation process of `ag++` with option

	code size (lines)	<code>SystemC</code> (sec)	<code>ASystemC</code> (sec)
<code>simple_fifo</code>	166	3.4	10.6
<code>simple_bus</code>	2093	19.0	63.5
<code>risc_cpu</code>	3183	34.2	115.0

Table 1: Comparison of compilation time of `ASys-temC` and `SystemC`

⁷Instead, we could not evaluate the performance of `ASys-temC` yet since we have just applied our prototype to some sample programs in `SystemC` that are sufficiently small to be terminated in an instant.

--verbose, and then we noticed that parsing systemc.h (the header file of SystemC) was very slow and it was performed twice. However, further investigation is currently under way.

In addition, there are three limitations in the current implementation of ASystemC: (1) multiple assertions may not work since logs of aspects may interleave with one another, (2) the users too easily write a wrong assertion by around advice modifier, (3) set and if cannot be used since AspectC++ does not provide them. Item (1) is our future work. We think the ambiguity of item (2) can be eliminated by continuation join point. We believe that future improvement of AspectC++ will solve item (3).

4.2 ASpecC

4.2.1 Design of ASpecC

SpecC is a C-based language with extension for describing hardware architecture. VisualSpec [8] is a commercial IDE for system-level design, including implementation for SpecC language. Its notable feature is a verifier of PSL-based assertions, as illustrated in Figure 12.

```
note scc_inline_psl={
  "property ReqAck =",
  "  always( req !=0 -> next! ack != 0 );",
  "assert ReqAck;"
};
```

Figure 12: A PSL-based assertion of VisualSpec

We designed ASpecC by extending the VisualSpec’s assertion to support pointcuts. It has two features: (1) it uses *continuation join point*-style pointcut whereas ASystemC uses pointcut of AspectC++, and (2) it modifies the join point model to specialize in hardware design such as clock.

Continuation Join Point Model.

The continuation join point is a finer grained join point than conventional ones such as AspectJ. In the model, two join points are defined at every method call: call join point and receive join point. The former indicates the beginning of the method call and the latter indicates the end. Thus, no advice modifier is needed in the model.

Table 2 summarizes pointcuts of AspectJ, PitJ, which is a language based on continuation join point, and ASpecC. ASpecC provides two pointcuts, `enter` and `leave`, which correspond to call join point and receive join point, respectively. For example, a pointcut `enter(foo)` matches just the beginnings of the call of the method `foo`. In other words, it corresponds to a pair of a normal pointcut `call(foo)` and an advice modifier `before()`. Similarly, a pointcut `leave(foo)` matches join points where the call of the method `foo` ends.

The continuation join point prevents the users from writing a wrong advice modifier such as `around`, and thus it makes a pointcut-based assertion more robust.

Additional Modification for Clock.

The assertion language of VisualSpec is intended to be used with a model that has discrete time information or *clock*. In other words, the programmers specify the variable representing the clock; the verifier evaluates PSL whenever

AspectJ	PitJ	ASpecC
<code>before() : call(foo)</code>	<code>call(foo)</code>	<code>enter(foo)</code>
<code>after() : call(foo)</code>	<code>receive(foo)</code>	<code>leave(foo)</code>

Table 2: Comparison of pointcuts among AspectJ, PitJ and ASpecC

the specified variable is changed. Since this policy is important for hardware design and easy to understand for hardware architects, our current ASpecC follows it.

To provide support, ASpecC slightly modifies the join point model so that a join point represents an instant when the clock is signalled just after the event is raised, instead of the very instant when its event is raised (e.g., when specified method is called). For example, in Figure 14, `enter(foo)` does not match a call itself of the method `foo` but an instant of the next clock change after the call. Finally, Figure 13 shows an ASpecC sample assertion for bus inputter in Figure 2.

```
assert always (
  (leave(void must_read.receive(...)) ->
  next (enter(void bus.read(...)))
);
```

Figure 13: An assertion of ASpecC for bus inputter in Figure 2

Figure 14 summarizes these ASpecC features.

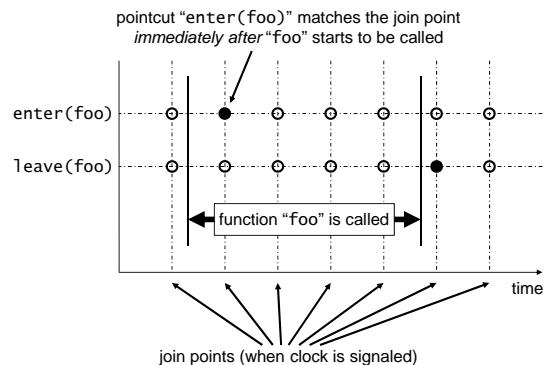


Figure 14: A join point model in ASpecC

4.2.2 Implementation of ASpecC

We implemented ASpecC which consists of the following two parts: (1) preprocessor to allow the original verifier to monitor high-level events, and (2) original verifier with minimal modification.

In short, the preprocessor automatically performs the second workaround of Section 2.3. The preprocessor first introduces new dummy variables for each pointcut in an assertion. Next, it translates an assertion to conventional VisualSpec’s one by replacing pointcuts with comparison of dummy variables. For example, the assertion in Figure 13 is translated into Figure 15. It finally inserts dummy assign-

```

assert always (
  (leave_must_read_received == true) ->
  next (enter_bus_read_entering == true)
);

```

Figure 15: A translated assertion from Figure 2

ment instruction into SpecC code as shown in Figure 5.

These dummy variables must be reset after each verification step since they will always be true after they are assigned to true once. Thus, we slightly modified the original verifier to initialize all dummy variables.

5. RELATED WORK

In the field of hardware design, Native SystemC Assertion (NSCa) [10] provides SystemVerilog Assertion (SVA) [7]-style assertion for SystemC models. Though SVA has more practical features than PSL (e.g., SVA provides flexible module mechanism for handling assertions), SVA has basically the same ability as PSL because assertions of SVA are also composed of boolean expressions. NSCa provides just the feature to write basic SVA assertion for SystemC model; the assertions in NSCa are also composed of boolean expressions written in SystemC. Thus, NSCa cannot specify high-level events such as method call.

Niemann et al.[13] provided a verification method of SystemC models by using aspect written in AspectC++. They used aspect to log the behavior of a target model, which is similar to our approach, converted the log into the Verilog HDL code and finally checked it with SystemVerilog Assertion. They however did not design assertion language, thus requires writing a logging aspect against every target model.

Some studies [1, 9] introduce aspects into system-level design languages such as SystemC and Jeda language. Their goal is not verification but providing a general aspect framework for debugging, measurement, fault injection, etc.

In aspect-oriented programming studies, Stolz et al.[17] presented runtime verification framework for Java programs. They presented LTL which is over AspectJ pointcuts, not including advice modifiers. Our approach can specify finer events than theirs because we use not only pointcuts but also advice modifiers or use continuation join point.

6. CONCLUSION

We presented a pointcut-based assertion that enhances traditional assertion language to specify high-level events in high-level design. This extension provides verification capability of assertions even in high-level design. In addition, we designed both ASystemC and ASpecC by extending, respectively, SystemC and SpecC with pointcut-based assertions. ASystemC is based on the pointcut expression of AspectC++, and its implementation also uses the AspectC++ compiler as its back-end. We showed the feasibility of our approach and performed preliminary evaluation. ASpecC is based on continuation join point model that allows us to specify finer grained events without advice modifiers. This model makes a pointcut-based assertion more robust. In addition, we slightly modified the join point model to support clock-oriented verification.

Future work will be directed toward evaluating our

method by further experiment with real code. To this end, we intend to improve our implementation to support some features, including multiple assertions and faster compilation. In addition, we will consider more pointcuts such as `args` and `cflow`. We think that they will play an important role in capturing more complex events in high-level design.

7. REFERENCES

- [1] D. Déharbe and S. Medeiros. Aspect-oriented design in SystemC: implementation and applications. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 119–124. ACM, 2006.
- [2] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In *FOAL '06: Proceedings of the Foundations of Aspect-Oriented Languages Workshop at AOSD 2006*, page 1.10, 2006.
- [3] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. 2000.
- [4] T. Grotker. *System Design with SystemC*. 2002.
- [5] IEEE. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*, 2000.
- [6] IEEE. *IEEE 1850 Standard for Property Specification Language (PSL)*, 2005.
- [7] IEEE. *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*, 2005.
- [8] InterDesign Technologies, Inc. Visualspec. <http://www.interdesigntech.co.jp/english/>.
- [9] A. Kasuya. Verification applications of aspect-oriented-programming. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon)*, 2004.
- [10] A. Kasuya and T. Tesfaye. Verification methodologies in a TLM-to-RTL design flow. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 199–204. ACM, 2007.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2001.
- [12] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In *APLAS*, pages 131–147. Springer, 2006.
- [13] B. Niemann and C. Haubelt. Assertion based verification of transaction level models. In *ITG/GI/GMM Workshop*, pages 232–236, 2006.
- [14] Open SystemC Initiative. *SystemC 2.0 Language Reference Manual*. OSCI, 2003.
- [15] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.
- [16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, 2002.
- [17] V. Stolz and E. Bodden. Temporal assertions using AspectJ, 2005.
- [18] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language (4th ed.)*. 1998.