Debug Patterns for Efficient High-level SystemC Debugging*

Frank Rogin, Erhard Fehlauer
Fraunhofer IIS / EAS Dresden
01069 Dresden, Germany
{frank.rogin|erhard.fehlauer}@eas.iis.fraunhofer.de

Christian Haufe, Sebastian Ohnewald
AMD Saxony LLC & Co. KG
01109 Dresden, Germany
{christian.haufe|sebastian.ohnewald}@amd.com

***Abstract*-This paper proposes debug patterns combined with an intuitive flow to accelerate and simplify the debugging of SystemC designs. A debug pattern provides a formalized procedure to fix a defect (also colloquial *bug*) that is notified by an always recurring failure symptom. It helps to focus the user's attention on a higher level of abstraction joined with minimal learning effort. The presented methodology is based upon a non-intrusive high-level SystemC debugging environment and the GNU debugger GDB. The usability of each pattern is demonstrated by practical examples.**

I. Introduction

Innovative System-on-Chip (SoC) designs require new approaches to address the associated challenges such as an increasing design complexity, or the codesign of hard- and software. System level design methodologies are a promising approach where many design languages and environments have been proposed within recent years, e.g. [8], [9]. SystemC [2] is one of the most popular system level design languages enabling concepts such as object-orientation, high-level modelling, and concurrency. Unfortunately, language features such as multithreading and event-based communication increase the program complexity and introduce nondeterminism in the system behavior. Thus, debugging a SystemC design can be challenging, in particular due to the possible occurrence of deadlocks or race conditions. Studies revealed that today often more than 50% of design time is spent verifying a complex design (meaning to detect, understand, locate, and correct errors [7]). To shorten this verification gap, we developed an approach to debug a SystemC design at a higher level of abstraction working with signals, ports, events, and processes [4]. Thus, the developer gets quick and concise insight into the static structure and dynamic behavior of the design without the burden of gaining a detailed knowledge of the SystemC simulation kernel or operating only at the comparatively low-level of standard C++ debuggers.

In this paper, we take a step forward and propose a superimposed methodology to further accelerate and simplify the debugging process. We gathered typical SystemC defects occurring in daily work at AMD to provide a catalog of so called *debug patterns* which are based upon our high-level debugging solution. A debug pattern is a guidance to fix a defect of an already known defect class. Thus, common sources of errors can be systematically excluded.

This paper is organized as follows: Section II discusses other approaches. Section III describes the different debug levels and propose a flow to apply our debug patterns. Section IV introduces the high-level debugging environment and how it supports the user at the different debug levels. In Section V we introduce the debug patterns found and exemplarily describe two of them in more detail. Section VI concludes the paper.

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

II. Related Work

Several existing approaches also propose patterns to automate debugging. The debugging environment MAD [11] is based on event graphs which are constructed from recorded event traces of parallel program runs. A subsequent graph analysis automatically detects errors and anomalies. MAD allows specifying of communication patterns which define the expected behavior of the program. These patterns are checked against the event graph and the results help the user focus his attention on the most critical places in the graph. A quite similar approach of pattern-oriented debugging was proposed by the TAU programming analysis environment [6] which uses the event-based debugger Ariadne [5]. Ariadne matches user-specified models of intended program behavior against the actual program behavior captured in event traces.

MAD and TAU environments focus on the interprocess communication of massively parallel programs operating on monitored traces where the entire debugging process is done in a post-processing step. In contrast, we do not monitor and evaluate traces but provide debugging support directly at run-time especially tailored to SystemC needs. While both approaches rely upon proprietary debug tools, our solution is implemented on top of the Open Source debugger GDB [1].

In [12], the importance of having knowledge of likely defects is underlined. The authors introduce the notion of a *stereotyped bug* which describes the fact that the same couple "symptom(s) + bug" has appeared several times. A symptom can be an I/O discrepancy, a trace, or a program state. We utilize similar symptoms to select a proper debug pattern. The paper mentions different tools that perform a recognition of stereotyped bugs by defining appropriate debug procedures. But most of these tools are aimed at logic programming systems where the applied techniques cannot be adopted directly to an object-oriented programming language like SystemC.

The authors in [13] present some loose hints and generally

accepted approaches to find defects faster such as using the *divide-and-conquer* approach to isolate the point of failure, or evaluating logged trace data. Instead, our patterns are presented in a formalized way especially aiming at the SystemC debugging needs. Our environment guides the user through the debug process and supports him by partially suggesting applicable patterns.

III. Debugging At Different Levels

A. Debug levels

Due to the introduction of debug patterns and our concept of a high-level SystemC debugging extension we propose different levels and dedicated methods to locate and correct defects in SystemC designs (Fig. 1).

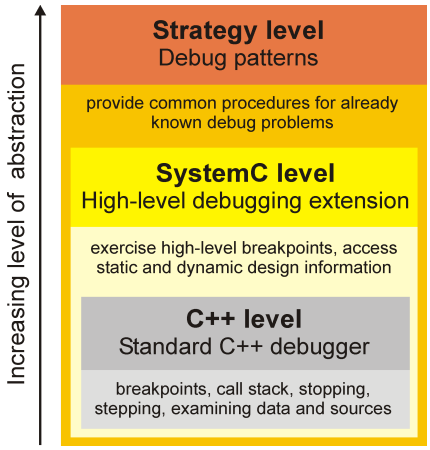

Fig. 1. Debugging at different levels

C++ level. At this level standard C++ debuggers are applied to analyze C++ functions and variables during simulation run. Unfortunately, they do not understand specific SystemC constructs. Instead, a C++ debugger offers different capabilities to investigate low-level program details which usually include setting breakpoints, examining the call stack or data, and stepping or stopping the program execution.

SystemC level. At this level a high-level debugging environment [4] enables the user to debug SystemC programs at system level. It operates at the level of signals, events, processes, and modules where the environment offers high-level breakpoints and allows to access static and dynamic simulation information.

Strategy level. At the highest level several debug patterns guide the debugging process. That means, when a familiar failure symptom occurs (e.g. the program seems to run forever, or outputs unexpected and wrong values), the user gets some help in form of a guide on how to locate a specific defect and how to fix it.

B. Proposed debug flow

Based on the debug levels of Section A we propose a three-level strategy to find and correct defects in a SystemC design (Fig. 2). A defect causes the simulation to behave in an unexpected erroneous way, which is represented by several error cases:

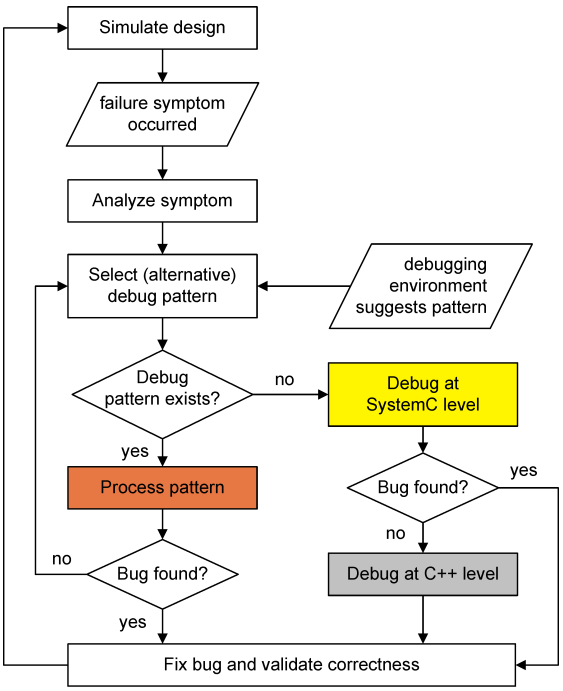

Fig. 2. Proposed debug flow

- program crashes,
- abnormal long running simulations,
- waveform mismatches,
- unexpected debug messages, or
- self-detected errors through checkers or assertions.

As in [12] we use such failure symptoms to guide the debugging process. After a detailed analysis of the occurred symptom, the user looks for a suitable debug strategy. Here, the debugging environment can partially suggest matching debug patterns or automate some actions in order to find the probable cause of an observed symptom more quickly. Otherwise the user manually chooses a particular debug pattern. Next, the pattern guides the user to locate the point of failure by applying a specific sequence of high-level SystemC debugging commands. To exactly isolate the point of failure, basic C++ debugger commands can be applied as well. If the defect is successfully identified, the design is fixed and the simulation is restarted. Otherwise another matching debug pattern is chosen. If no known debug pattern is left, the user continues to work at SystemC and C++ level, respectively.

The usage of debug patterns and high-level debugging commands provides comprehensive debugging support. It helps to focus the user's attention on a higher level of abstraction (while ignoring irrelevant design parts) with

- a partially automated process suggesting applicable debug patterns,

- a formalized procedure to fix already known defects using flow charts combined with a scenario-based guidance, and
- an improved debugger usability, particularly becoming familiar with the high-level SystemC debugging features.

Thus, bug fixing is simplified and accelerated. Especially the novice user can exclude and fix many defects before he has to consult an expert. A further advantage is the intuitive partially automated debugging flow resulting in a purposeful proceeding. The user can focus on the underlying cause of a failure instead concentrating on the correct debugger usage or pattern choice. The main disadvantage of the pattern catalog is that it only comprises debug patterns for always recurring SystemC specific failure symptoms most probably caused by the same defect or class of defects. Consequently each problem, in particular the rare and tricky failures, cannot be simply solved using a pattern. Sometimes it only marks the starting point for an advanced debug session.

IV. High-level Debugging Environment

Each debug level (Section III) is supported by specific parts of an integrated and self contained high-level debugging environment.

A. C++ level

At C++ level the Open Source GNU debugger GDB [1] is applied. It provides various features for debugging a SystemC design at C++ level, for example stopping and continuing the simulation, examining source files, local program variables, the memory, or the actual program stack.

B. SystemC level

Studies at AMD indicated several recurring actions to debug a SystemC design. Each action describes steps in the debugger to acquire needed information at system level. Based upon characteristic debug actions, high-level commands were defined, implemented, and integrated in a high-level debugging environment based on GDB [4]. The environment distinguishes two command types: *examination* and *controlling commands* to retrieve static or dynamic simulation information, and to provide high-level breakpoints to reach a point of interest at system level very quickly.

The environment's most important quality feature is the non-intrusive extension of the SystemC kernel v2.0.1 and GDB. To support high-level debugging neither the kernel nor the debugger sources have to be patched. Furthermore, legacy designs or (third-party) intellectual property (IP) blocks have not to be modified.

C. Strategy level

The high-level debugging environment supports the application of debug patterns through various features. Goals are a simplified usage and a widely automated debugging process in order to relieve the user from standard tasks. This section describes two features already available.

Scenario-based guidance. Each pattern can be referenced by a debugging command using the name specified in the pattern catalog (Section V). It initiates a new pattern execution. The user is guided through the debug procedure by calling the command **nps** (next-pattern-step). The following example illustrates this guidance in the case of the CONCURRENCY pattern:

```
(gdb) dp_concurrency
*** CONCURRENCY debug pattern activated
*** find the signal the race problem occurs on
(gdb) nps
*** retrieve the corresponding event
*** using lse_rx "<signal name>"
(gdb) lse_rx "top.rdy_l"
---lse_rx: list all events matching the regex ---
top.rdy_l.m_negedge_event           0x8097858
top.rdy_l.m_posedge_event           0x8097388
top.rdy_l.m_value_changed_event     0x8094360
(gdb) nps
*** choose right event and trace its triggering
*** using dp_sense "<event name>"
(gdb) dp_sense "top.rdy_l.m_negedge_event"
*** restart simulation using <run>
(gdb) run
...
```

The scenario-based guidance significantly enhances the usability of patterns and helps to efficiently apply SystemC high-level debugging features with minimal learning effort.

Partially automated process. There are situations where the debugging environment proposes multiple applicable debug patterns using different information sources:

- a reached program/debugger state,
- a performed user action, or
- a logged and processed history of released events or triggered processes.

Aborting the current simulation by pressing Ctrl-C while the debugger seems to hang in one of its processes is a typical example. Here, the environment suggests to use one of the LOCK patterns to find the defect causing the process to hang.

V. Debug Patterns

A. Pattern description

Currently, the catalog consists of seven debug patterns:

- **CONCURRENCY**. This pattern helps to identify signal races caused by nondeterministic process execution.
- **TIMELOCK**. The TIMELOCK pattern helps the user to find defects resulting from infinitely looping processes that cause simulation freezes.
- **DEADLOCK**. When two or more processes are each waiting for another to release a resource, this pattern proposes a procedure to find the deadlock problem.
- **LIVELOCK**. This pattern guides the user to handle livelock problems where two or more processes are working together, that means constantly changing their states, but

never coming to an end.

- **OVERFLOW**. A thread stack overflow causes the simulation to dump a core. The OVERFLOW pattern helps to identify all threads such an overflow has occurred on.
- **LOSTEVENT**. The LOSTEVENT pattern provides a debug procedure to detect events that were missed because of multiple overwriting **notify** calls.
- **PERFORMANCE**. An IP integration yields to an unacceptably slowed simulation. This pattern suggests a procedure to determine the bottleneck probably caused by often released events or multiple activated processes.

Each pattern guides the user step-by-step to identify a defect possibly causing the observed failure symptom. According to [3], we describe each debug pattern in the following common format:

Pattern name
A short and intuitive name for the debug pattern. The second name put in parentheses is used to reference the pattern in the debugging environment.

Motivation
The motivation section consists of a short description why the pattern exists.

Symptoms
This section describes a specific symptom which is typical for the defect the pattern is aimed at.

Assumption
The assumption specifies the problem or conditions (i.e. a specific architecture or object composition in the design) that probably causes the observed symptom.

Participants
Here, all participating high-level debugging commands and their specific responsibilities in the pattern context are described.

Debug procedure
The debug procedure summarizes the steps the user has to follow to pinpoint the defect. Here, well-known flowcharts formally document the required procedure. Each pattern usage is directly supported by the debugging environment.

Example
This section sketches a typical example of how the pattern can help to find and to correct a defect.

Related patterns
Here, other patterns are itemized that match the observed symptom and also could be tried to locate the defect as proposed in the debug flow.

B. Pattern catalog

Due to space limitations of this paper we exemplary discuss the CONCURRENCY and the TIMELOCK pattern only.

CONCURRENCY (dp_concurrency)

Motivation
A typical SystemC design usually consists of different communicating processes that are running in parallel. The execution order of processes is not determined during a simulation delta cycle. This introduces nondeterminism into the program flow which may lead to race conditions.

Symptom
Simulation runs with the same inputs sometimes produce different results (for a specific signal or bus).

Assumption
There are at least two processes concurrently driving data onto that signal or bus which showed different results.

Participants

dp_sense	return all processes triggered by the same event
lpt	review the process sensitivity list
lst	explore the source code of a process
lse_rx	get the correct hierarchical event name

Debug procedure

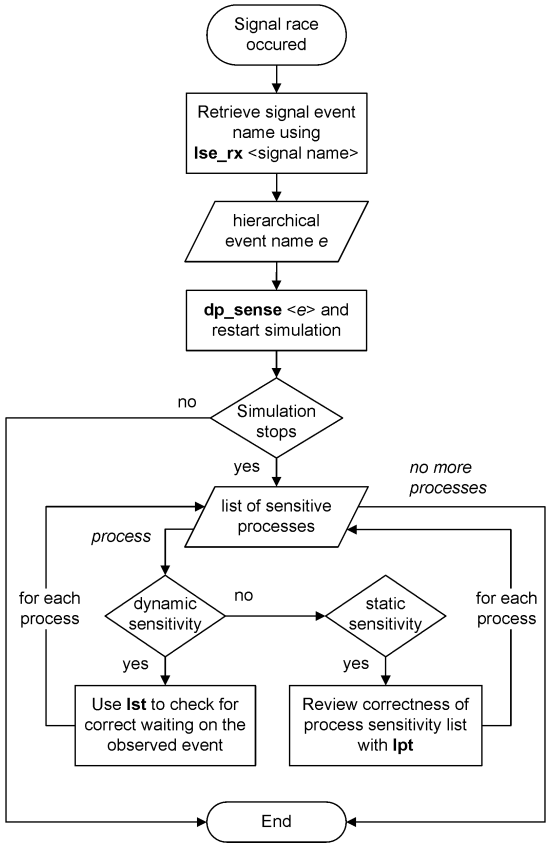

Fig. 3. CONCURRENCY pattern

Example

Problem. Fig. 4 sketches a situation where the two thread

processes **top.bfm._fsm_update** and **top.bfm._fsm_rst** write to the same signal **top.ctrl_w**. Both threads are activated by the ready signal **top.rdy_l**. Since the SystemC simulation kernel does not define a deterministic order of thread activations inside a delta cycle, a race condition can occur on signal **top.ctrl_w**.

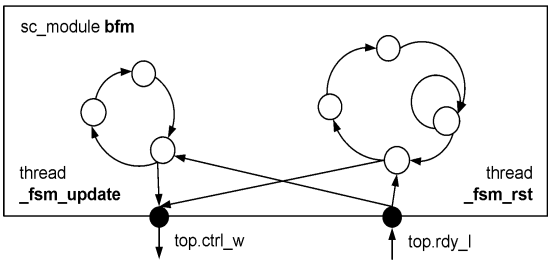

Fig. 4. Exemplary race condition

Debug procedure. Assuming that we already know the corresponding hierarchical event name of the signal causing the problem, we call **dp_sense** with it and restart the simulation:

```
(gdb) dp_sense "top.rdy_l.m_negedge_event"
*** CONCURRENCY debug pattern activated
*** restart simulation using <run>
(gdb) run
```

When the simulation stops **dp_sense** reports two thread processes sensitive to the observed event:

```
*** dp_sense 'top.rdy_l.m_negedge_event'
*** CONCURRENCY between sensitive processes
   top.bfm._fsm_update <static>
   top.bfm._fsm_rst <static>
** check correct static sensitivity using lpt
** check correct dynamic sensitiviy using lst
(gdb)
```

Knowing that thread **_fsm_update** is correctly activated by the ready signal, we investigate the sensitivity list of the statically triggered thread **_fsm_rst** using **lpt**:

```
(gdb) lpt "top.bfm._fsm_rst"
process top.bfm._fsm_rst sensitive to
   <static> top.fsm_rst_l.m_negedge_event
   <static> top.rdy_l.m_negedge_event
   <dynamic> top.write_tx.m_value_changed
```

We figure out that the sensitivity list falsely includes the ready signal which turns out to be an environment defect.

Related patterns
LIVELOCK

TIMELOCK (dp_timelock)

Motivation
Event-based communication between concurrent processes can often lead to a lock condition where a process is caught in an infinite loop and thus never waits for an event.

Symptom
The simulation infinitely loops or at least appears to do so. Additionally, the simulation time does not proceed.

Assumption
Because of design specification knowledge the user suspects one or more processes causing the lock.

Participants

dp_timelock	look for hanging processes manually
lst	explore the source code of a process
dstep	proceed simulation step-wise

Debug procedure

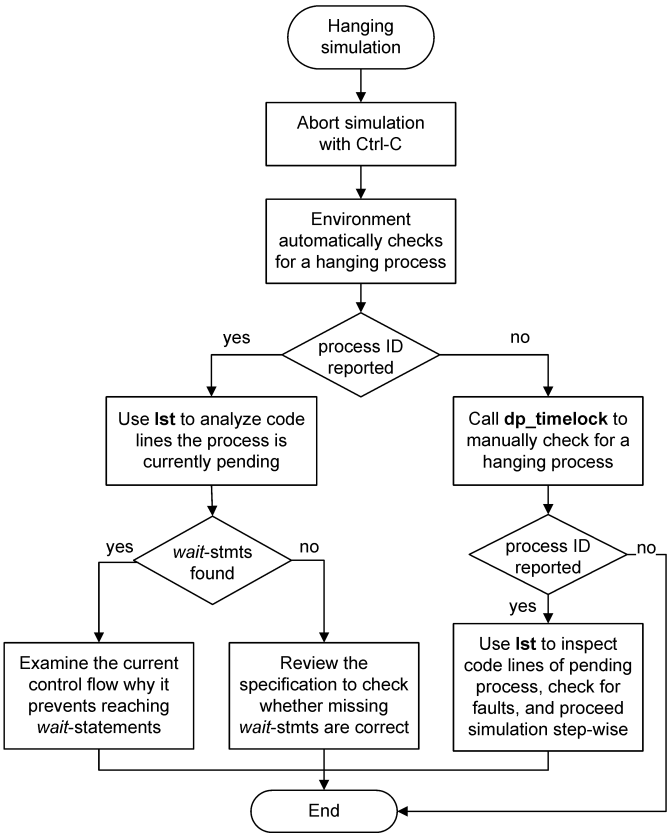

Fig. 5. TIMELOCK pattern

Example

Problem. Fig. 6 illustrates a system where a device (**top.i_device**) communicates with a host (**top.i_host**) over a bus. Device and host exchange data using two signal lines, **top.req_data** and **top.resp_data**. Available data packages are indicated by two ready signals **top.req_ready** and **top.resp_ready**, respectively. If the user models the signal interaction in a faulty way, the simulation can lock.

Debug procedure. The user aborts the simulation by pressing Ctrl-C. Since the debugger does not report any process ID we manually call **dp_timelock** to check for a hanging process:

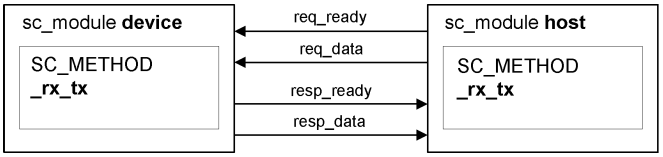

Fig. 6. Exemplary TIMELOCK pattern

```
Program received signal SIGINT, Interrupt.
(gdb) dp_timelock
*** following process seems to hang
   top.i_device._rx_tx
*** TIMELOCK debug pattern activated
*** use lst to examine process source code
```

We investigate the source code of the hanging process to find the root cause of the defect.

```
(gdb) lst "top.i_device._rx_tx"
--lst: list active source of [c]thread/method---
process top.i_device._rx_tx is currently
 at /home/hld/project/tb/src/device.cpp:254
in device::_rx_tx
254   void device::_rx_tx() {
255      process_req_data(req_data.read());
256      resp_data.write(gen_resp_data());
257      resp_ready.write(~resp_ready.read());
258   };
```

At first, we cannot detect any failure. Thus, we proceed the simulation in step-mode using multiple **dstep** commands:

```
(gdb) dstep
*** setting breakpoint(s) at next delta cycle
*** runnable process(es) at delta cycle 3563, @14 ns
   break @ 'host::_rx_tx()' of 'top.i_host._rx_tx'
*** type <continue> to visit breakpoints
(gdb) cont
```

After some more **dstep** commands we see that the processes **top.i_host._rx_tx** and **top.i_device._rx_tx** become mutually active, invoking each other in the same delta cycle. A code review of the affected modules shows a lack of time-consuming statements within these processes.

Related patterns
DEADLOCK, LIVELOCK

VI. Conclusion

Debugging SystemC designs using debug patterns has several advantages. First, the tool usability is improved. The user gets a formalized instruction to efficiently and easily use the debugger, particularly the high-level SystemC debugging feature, to find and to correct errors of already known categories. Thus, especially the novice user can exclude and fix many defects without the need to consult an expert. Second, the tool environment partially automates the debugging process by suggesting applicable patterns. As a consequence, the user can focus on the underlying cause of failure instead of spending too much attention on the correct tool usage. Third, the scenario-based pattern guidance provides a comprehensive debugging support that reduces the learning effort and can be used as a good starting point to learn to debug more complex problems. Finally, the experienced user who does not need pattern guidance anymore has the benefit of the pattern tool support. Here, the partially automated detection of erroneous situations can accelerate the debug process. The usability of debug patterns is demonstrated by examples which are based on practical problems at AMD. To rate the efficiency of our approach Table I compares the effort one has to invest should high-level debugging (HLD) functionality be replicated with GDB commands. Note that this is impossible for half of the functionality. For the other half, the user has to have at least a deep understanding and a detailed knowledge over the SystemC simulation kernel.

TABLE I
Compare High-level Commands and Gdb Debugging Effort

#HLD cmds	#GDB cmds per HLD cmd	Remarks on equivalent GDB functionality
6	1 - 9	fully automatically with canned command sequences
10	dyn./min. 4	partly automatically with manual user intervention
15	∅	additional functionality provided by external helper functions to retrieve, and to store needed information

Future work will improve the tool support using debug patterns, i.e. through a more sophisticated automated detection of erroneous situations. Also, it would be useful to provide additional debugging techniques, i.e. to make it easier to find the root cause of an error where sophisticated methods like a flowback analysis or program slicing technique could be used.

References

[1] GDB home, www.gnu.org/software/gdb, 2006.
[2] Open SystemC Initiative, www.systemc.org, 2006.
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern - Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, 1999.
[4] F. Rogin, E. Fehlauer, S. Rülke, S. Ohnewald, and T. Berndt, "Non-intrusive High-level SystemC Debugging," *In Proceedings of FDL'06*, Sept. 2006.
[5] J. Cuny et al., "The Ariadne Debugger: Scalable Application of Event-based Abstraction," *SIGPLAN Notices*, 28(12):95-95, Dec. 1993.
[6] S. Shende et al., "Event and State-based Debugging in TAU: A Prototype", *In ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.
[7] K.-C. Chen, "Assertion-Based Verification For SoC Designs," *In Proceedings of 5th International Conference on ASIC*, Vol.1, Oct 2003.
[8] F. Balarin et al., "Metropolis: an integrated electronic system design environment," *IEEE Computer*, Vol. 36, No. 4, April 2003.
[9] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic, 2000.
[10] F. Doucet, S. Shukla, and R. Gupta, "Introspection in System-Level Language Frameworks: Meta-level vs. Integrated," *In Proceedings of DATE 03*, March 2003.
[11] D. Kranzlmüller, N. Stankovic, and J. Volkert, "Debugging Parallel Programs with Visual Patterns," *In Proceedings IEEE Symposium on Visual Languages*, 1999.
[12] M. Ducasse, and A.-M. Emde, "A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques," *In Proceedings of the 10th International Conference on Software Engineering*, 1988.
[13] B.W. Kernighan, and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999.