AUTOMATIC DEBUGGING OF SYSTEM-ON-A-CHIP DESIGNS

Frank Rogin[1], Rolf Drechsler[2], Steffen Rülke[1]

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

[2] University of Bremen, Institute of Computer Science, 28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

ABSTRACT

Designing System-on-a-Chip (SoC) using system-level languages is becoming a standard in industry. However, the non-deterministic semantics of such parallel languages could yield failures that are hard to debug. In this paper, we present a new approach that supports automatic debugging of SoC designs written in SystemC using a method that isolates failure-inducing process schedules.

I. INTRODUCTION

Designing SoCs requires new design methods to address corresponding challenges such as an increasing design complexity, or the co-design of hard- and software. System-level design is a promising approach where SystemC is one of the most popular design languages enabling concepts such as object-orientation, high-level modelling, and concurrency. Unfortunately, language features such as multithreading and event-based communication increase program complexity and introduce non-determinism in the system behavior. Thus, debugging SystemC designs can be challenging, in particular in the face of errors such as deadlocks or races.

This paper introduces a new approach that automates the debugging of SoC designs written in SystemC. Here, the *delta debugging* algorithm narrows down the difference between a passing and a failing process schedule. The resulting schedule can be used by the designer to debug the actual failure cause in the design more quickly and systematically.

Delta debugging was developed by Zeller and Hildebrandt [1] for the software domain. They used the algorithm to isolate failure causes in the program input, in process schedules of parallel programs, or due to code changes. Misherghi and Su present Hierarchical Delta Debugging [2] that uses data semantics to handle more complex problems.

A number of specialized approaches detect and debug certain hard to find errors in parallel programs: Cheung et al. present an approach [3] that monitors a SystemC simulation and reports a deadlock based on a synchronization dependency graph. A similar approach is used in the Metropolis environment [4]. Both approaches dynamically report deadlocks while delta debugging performs a post-mortem analysis and suggests a fix.

A number of tools, e.g. [5], [6], [7], instrument software programs to detect race conditions. Lock-set-based tools associate and evaluate lock candidate sets used to protect a shared location. The happens-before analysis algorithm uses clocks to compare time stamps when a shared location is accessed. Code instrumentation hampers an application of proper tools in production systems. Moreover, dynamic race detectors are not sound, i.e. they only check code that is actually executed which is similar to our approach. Finally, these tools are only able to find races while delta debugging is a more general approach.

Based upon an abstraction of the implemented system, formal verification is used to verify particular program properties such as liveness [8], [9]. Due to the state explosion problem, formal techniques do not scale very well for complex concurrent systems but if the analysis terminates, the results are precise and sound. Our approach bases on simulation and can handle complex, real-world system designs.

Several tools use static analysis to detect race conditions or deadlocks in programs, e.g. [10], [11]. Since static analysis relies on conservative approximations, false alarms are a problem for the acceptance of these techniques.

The paper is organized as follows: Section II presents the debugging approach. In Section III its application for SystemC process schedules is detailed while Section IV summarizes experimental results. Finally, Section V concludes the paper.

II. AUTOMATIC DEBUGGING APPROACH

A. Requirements

The basic idea of the presented debug procedure is a systematic test of each difference between a failing and a passing test case, and to check whether the failure still exists. If the failure disappears, the cause for that particular failure is found. An algorithm implementing the described procedure shall meet the following requirements:

- *Narrowing down strategy*. A strategy has to narrow down the difference between passing and failing test cases systematically and efficiently.
- *Rating strategy*. An automated test function has to assess a newly created alternate test case whether the failure has disappeared.

B. Methodology

A simple and often used debug procedure is simplification which removes aspects from a failing test case as long as they are irrelevant to produce the observed failure. A more efficient approach is delta debugging, abbreviated *dd*, (see [1] for details). It bases on isolation where a minimal difference between a passing and a failing test case is calculated. Whenever the new test fails, the failing test case c_F is "reduced" while if the test passes, the passing test case c_P is "increased". Hence, the algorithm narrows down the minimal difference between a passing and failing test case.

Despite the automated procedure to isolate an actual failure cause, a human user is often more creative during debugging, and thus possibly finds the failure cause faster. Nevertheless, an automated procedure is less error prone and systematically tests the complete search space.

III. DEBUGGING OF PROCESS SCHEDULES

The non-deterministic SystemC scheduler can cause many failures that are difficult to debug manually, e.g. deadlocks or data races. To apply delta debugging in the SystemC context, a debugging environment has to provide the following features:

- *Deterministic record/replay*. The SystemC scheduler is extended by a *record/replay* facility for simulation runs that allows to handle process activations in terms of process schedules.
- *Isolating failure causes*. The *dd* algorithm automatically narrows down the failure-inducing minimal difference between a passing and a failing process schedule. The resulting schedule produces a failure iff a particular method or thread process is activated at a specific point in time during simulation.
- *Root-cause analysis*. The system design is debugged while replaying the reported failure-inducing process schedule. Here, the debugging environment presented in [12] aids the designer in locating the failure-causing defect.

Fig. 1 depicts the particular debug process.

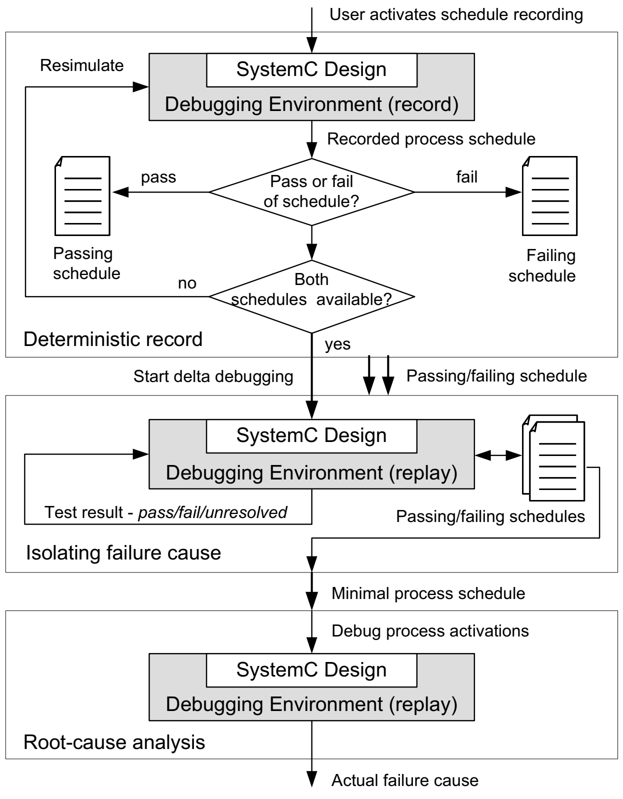
User activates schedule recording
Resimulate
SystemC Design
Debugging Environment (record)
Recorded process schedule
pass
Pass or fail of schedule?
fail
Passing schedule
Failing schedule
no
Both schedules available?
Deterministic record
yes
Start delta debugging
Passing/failing schedule
SystemC Design
Debugging Environment (replay)
Test result - pass/fail/unresolved
Passing/failing schedules
Isolating failure cause
Minimal process schedule
Debug process activations
SystemC Design
Debugging Environment (replay)
Root-cause analysis
Actual failure cause

Figure 1: Isolating failure-inducing process schedules

First, the simulation is run in the record mode to capture a process schedule that is subsequently tested for pass or fail. As soon as a passing and a failing schedule are available, the failure-inducing difference is isolated between them using *dd*. After analysis, the reported minimal process schedule is used to debug the erroneous simulation.

A. Deterministic record/replay facility

A SystemC design executes all its method and thread processes in a non-preemptive fashion. Each process activation is determined by the execution logic and is represented by a particular point in time.

Definition 1. *A tuple* (t_i, δ_n) *with the simulation time* t_i *with* $0 \le t_i \le t_{end}$ *and the* n^{th} *delta cycle* δ_n *with* $n \ge 1$ *is called an activation time point. T is the set of all activation time points.*

Definition 2. *Let T be the set of all activation time points for a SystemC design D. A 4-tuple* $S = (P, T, \pi, f)$ *is called process schedule of D with*

- P *: the finite set of instantiated method and thread processes in D,*
- T *: the finite set of activation time points for D,*
- π *: the sequence* $\pi = (t_0, \delta_1), \dots, (t_{end}, \delta_{end})$ *of activation time points, and*

- *f : the function $f : (t_i, \delta_n) \rightarrow P^*$ assigning each activation time point a number of processes to be consecutively activated at this time point.*

If the record mode is enabled during simulation, the debugging environment records all activated processes at the different activation time points until the specified end time t_{end} is reached.

A recorded process schedule S is replayed during the activated replay mode in five phases:

1. *Initialization phase.* This phase initializes all processes in P and sets the simulation time t_i to the first recorded activation time point in π, i.e. (t_i, δ_n) with $i = 0$ and $n = 1$.
2. *Evaluation phase*. All processes $p \in f(t_i, \delta_n)$ are executed in the recorded order.
3. *Update phase*. This phase performs needed channel updates to propagate data created by previously activated processes.
4. *Delta notification phase*. After delta notifications have been processed, the next element of π is retrieved. In case, processes become active at (t_i, δ_{n+1}), n is set to $n = n + 1$ and step 2 is executed, again. Otherwise step 5 is processed.
5. *Timed notification phase*. The timed notifications are processed. If π is finished, the simulation stops. Otherwise step 2 is called with the current element of π.

Since the *dd* algorithm generates virtual new process schedules without any knowledge of the SystemC simulation and program semantics, the schedule consistency has to be checked for validity. A schedule is discarded (*unresolved test result*), if

- a process is activated twice at (t_i, δ_n) without being notified during a delta notification in t_i,
- the schedule file specifies the execution of a process at (t_i, δ_n) that was already activated at (t_i, δ_{n-m}) and where its execution was suspended by a timed waiting statement,
- the simulation logic determines a process that becomes ready to run at (t_i, δ_n) through immediate event notification but the recorded schedule does not contain a proper activation, or
- a process suspends its execution at (t_i, δ_n) and is waiting for a certain event but the process schedule instructs its activation at (t_{i+k}, δ_{n+m}) without the particular event notification has been occurred.

B. Isolating Failure Causes

The implementation of the *dd* algorithm is straightforward (see [1]). Solely, the calculation of the difference Δ between a passing process schedule $c_P = S_P = (P, T_P, \pi_P, f_P)$ and a failing schedule $c_F = S_F = (P, T_F, \pi_F, f_F)$ needs to be defined. To do this, each process activation in f_P and f_F is assigned a unique slot number s_k. The n^{th} activation of a process $p \in P$, written p^n, is for instance $p^n = s_m$ in c_P and $p^n = s_k$ in c_F. So, the difference between c_F and c_P for p^n is calculated by $\Delta_P[s_m] = s_k - s_m$, so that $c_P \cup \Delta_P = c_F$. The reduction step $c_F \setminus \Delta_F$ is implemented by a "reversed delta" Δ_F which is calculated by $\Delta_F[s_k] = s_m - s_k$, so that $c_F \cup \Delta_F = c_P$. To mix both schedules, they have to be possibly aligned using dummy slots that are filled with virtual process activations.

Table 1: Illustrating the first steps of the *dd* algorithm

s_k	c_P	c_F	Δ_F	$\Delta_{F,1}$	$c_F \cup \Delta_{F,1}$
0	$(0, 1) \rightarrow A$	$(0, 1) \rightarrow B$	1	1	$(0, 1) \rightarrow A$
1	$(0, 1) \rightarrow B$	$(0, 1) \rightarrow A$	-1	-1	$(0, 1) \rightarrow B$
2	$(0, 1) \rightarrow A$	$(5, 3) \rightarrow B$	1	0	$(5, 3) \rightarrow B$
3	$(5, 3) \rightarrow B$	$[(5, 4) \rightarrow A]$	-1	0	$[(5, 4) \rightarrow A]$
4	$(5, 3) \rightarrow A$	$[(5, 4) \rightarrow A]$	0	0	$[(5, 4) \rightarrow A]$

The *dd* algorithm divides Δ_P and Δ_F into n disjoint parts with $\Delta_P = \Delta_{P,1} \cup \ldots \cup \Delta_{P,n}$ and $\Delta_F = \Delta_{F,1} \cup \ldots \cup \Delta_{F,n}$. Hence, new schedules are iteratively created by $c_P \cup \Delta_{P,i}$ and $c_F \cup \Delta_{F,i}$ where the new target slot of a process activation is calculated by counting back the particular slot difference. The activation time point (t_i, δ_n) of a process is taken from the target schedule, i.e. c_P in case of $c_F \cup \Delta_{F,i}$ and c_F in case of $c_P \cup \Delta_{P,i}$. Then, the new schedule is checked for consistency and is simulated to check for pass or fail.

Example 1. *Table 1 shows the calculation of Δ_F (column 4) as described in the paragraph above for two process schedules c_P and c_F (column 2 and 3). Using the first delta part $\Delta_{F,1}$ generates a new schedule (column 6) which is simulated afterwards.*

C. Root-Cause Analysis

Finally, the reported minimized process schedule is replayed. The system-level debugging features presented in [12] allow to debug the location of the defect that caused the actual failure using the reported minimal process schedule.

IV. EXPERIMENTAL RESULTS

Fig. 2 sketches the general architecture of a SystemC design definitely producing a deadlock after a random time. Four types of threads try to acquire four different resource types. During simulation each thread instance tries to lock two particular resource instances one after another to start "working". In case a lock does not succeed, the thread waits for a random time and tries a relock. Since a

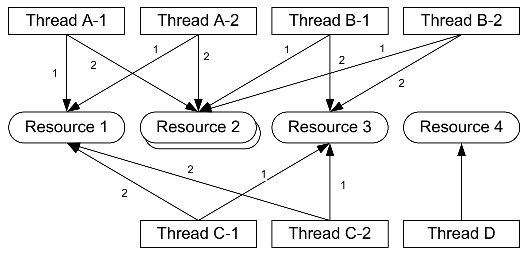

Figure 2: Architecture of the deadlock example

```
...                                  @ 5500 ps
@ 4 ns                               23:
17: C-2                              @ 6 ns
17: D                                25: C-2
17: B-2                              25: A-1
@ 4500 ps                            25: C-1
19:                                  25: B-2
@ 5 ns                               25: B-1
21: D                                ...
```

Figure 3: Extract of an exemplarily recorded process schedule

thread keeps the first resource locked, while it is waiting for the availability of the second one, a deadlock will randomly occur. To test for a deadlock, a resource allocation graph is created from the evaluated simulation log. The graph allows to precisely detect a deadlock whenever a cycle is found. Fig. 3 shows an extract of a recorded process schedule for the SystemC design in Fig. 2.

Table 2 summarizes the analysis results running *dd* on the example with a clock period of 1ns. The second column shows the initial difference between passing and failing process schedules. After constructing a virtual new schedule, it is written into a file and is simulated, afterwards. Either the new schedule results in a passing (column 4) or a failing test outcome (column 5) or it is unresolved (column 6) due to a violated simulation semantics (see Section III.A). As can be seen in Table 2, the number of unresolved test outcomes has a major impact on the algorithm performance (column 7). In the current implementation the *dd* algorithm and the SystemC simulation communicates via files which hampers an efficient error search for long running design simulations. Nevertheless, the experiment demonstrates the applicability of *dd* to isolate failure-inducing process activations in SystemC designs systematically in a finite and short time.

V. Conclusion

We have presented an approach that aids an automatic debugging of complex SoC designs written in SystemC. The approach narrows down the minimal difference between a passing and a failing process schedule using a set of experiments while the reported difference pinpoints to the actual failure cause. The experiments show that after a relatively short time, the designer gets a result in any case. So, a starting point is given especially when the designer has no clue how to start debugging.

Table 2: Running *dd* on the deadlock example

Sim.time in ns	Activations difference	Test runs	Pass tests	Fail tests	Unres. tests	Analysis time[1]
50	2,820	48	2	17	29	11 s
100	16,330	158	6	17	135	40 s
200	51,594	335	6	22	307	90 s
300	51,972	218	4	33	181	65 s
400	147,290	825	4	75	746	249 s
500	155,322	858	4	61	793	282 s
1000	1,071,436	363	4	70	289	228 s
2000	7,716,550	617	11	202	404	1,831 s

1. *Test system*. Intel Centrino Duo T2400@1830 MHz, 1GB RAM

REFERENCES

1. A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure inducing Input", IEEE Transactions on Software Engineering, Vol. 28, No. 2, pp. 183-200, 2002.
2. G. Mishergi and Z. Su, "HDD: Hierarchical delta debugging", Intl. Conference on Software Engineering, pp. 142-151, 2006.
3. E. Cheung, P. Satapathy, Vi Pham, H. Hsieh, and Xi Chen, "Runtime deadlock analysis of SystemC designs", IEEE Intl. HLDVT Workshop, pp. 187-194, 2006.
4. Xi Chen, A. Davare, H. Hsieh, A. Sangiovanni-Vincentelli, and Y. Watanabe, "Simulation based deadlock analysis for system level designs", DAC, pp. 260-265, 2005.
5. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", ACM Transactions on Computer Systems, Volume 15, Issue 4, pp. 391-411, 1997.
6. E. Pozniansky and A. Schuster, "Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs", Principles and Practice of Parallel Programming, pp. 179-190, 2003.
7. Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking", ACM SIGOPS Operating Systems Review, Volume 39, Issue 5, pp. 221-234, 2005.
8. T.A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference", Intl. Conference on Programming Language Design and Implementation, pp. 1-13, 2004.
9. S.D. Stoller, "Model-checking multi-threaded distributed Java programs", International SPIN Workshop on SPIN Model Checking and Software Verification, pp. 224-244, 2000.
10. D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and deadlocks", ACM SIGOPS Operating Systems Review, Volume 37, Issue 5, pp. 237-252, 2003.
11. C. Flanagan and S.N. Freund, "Detecting race conditions in large programs", Workshop on Program Analysis for Software Tools and Engineering, pp. 90-96, 2001.
12. F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An Integrated SystemC Debugging Environment", Embedded Systems Specification and Design Languages: Selected contributions from FDL'07 (E. Villar Editor), Springer, pp. 59-71, 2008.