Chapter 6

Isolating Failure Causes

The so far introduced deductive, observational, and inductive techniques already facilitate and accelerate debugging of system models. However, the automatic determination of actual failure causes is not achieved by these techniques. This chapter proposes an experimentation technique that uses a series of experiments in terms of multiple controlled simulation runs of the fully integrated system. There, this debugging technique aims at an automatic and systematic isolation of failure causes (see Figure 6.1).

Isolating the cause of a failure requires to search for cause–effect relationships. Zeller [Zel05] defines this relationship as follows: “A cause is an event preceding another event without which the event in question (the effect) would not have occurred”. Mapped to the debugging context, we could say: A defect causes a certain failure if the removal of this defect would eliminate the failure, as well. The idea is to narrow down the failure-inducing actual defect by comparing two program runs, one run where the effect (the failure) occurs and another program run where the effect does not occur.

The first part of the chapter gives a general overview about failure causes, their search, and their isolation. Then, a methodology for an automatic isolation of actual causes in SystemC design descriptions is introduced.

In the second part of the chapter, the ISOC tool (Isolation Of Failure Causes) is introduced. ISOC automatically narrows down the failure-inducing cause in SystemC designs using the delta debugging algorithm as proposed by Zeller and Hildebrandt [ZH02]. The SystemC scheduler is extended by a record and replay facility to deterministically rerun, formerly recorded, simulation runs. The algorithm is used exemplarily to isolate the minimal difference between two process schedules, one representing a passing (successful) simulation run, and the other resulting in a failing (erroneous) run. A second experiment uses delta debugging to detect the actual failure cause in simulation input data.

1 EXPERIMENTATION TECHNIQUES IN A NUTSHELL

First, this section summarizes some basics of experimentation techniques as introduced by Zeller [Zel05]. Next, the related work section discusses
several approaches that allow to detect typical failure causes in parallel programming languages, e.g. deadlocks, livelocks, or data races.

1.1 Overview

The experiment is a popular instrument in science that is used to systematically reject or confirm a hypothesis. On success, it shows the causality between a cause and the observed effect. A usual procedure is to set up a failing world, where the effect occurs, and an alternate passing world, where the effect does not occur. The experiment tries to find the difference between both worlds which represents the actual failure cause. In contrast to reality that does not allow to repeat the history to explore an alternative course of events, computers facilitate the deterministic replay of an experiment over and over again. So, the alternate world can be explored to check whether cause and effect relate to each other.

One of the most important problems of error search is the identification of the real cause for a failure. So, someone could think that the computer
technology is the overall failure cause because without computers failing programs could not exist. To eliminate such obviously trivial alternatives, Zeller proposes the concept of the closest possible world [Zel05]. Here, the failing and the passing world shall only differ by means of the actual cause. This principle goes back to Ockham’s Razor, which states that if two theories equally explain the same fact, the simpler theory should be picked. Relating to the above example, the unavailability of computers is far away from reality. On the other hand, there is no need, and sometimes no possibility, for a completely correct world but it should be possible to find a closest possible world where the failure does not occur. This difference spawns the search space to detect the actual cause. Concerning computer programs, the actual failure cause can be traced back to different sources such as the program input, the program state, or the program code. The iterative comparison of the closest possible world and the alternative world isolates the actual cause–effect relationship. An extracted failure cause does not necessarily point to the actual bug but gives valuable information where to start debugging. Here, the particular found cause could suggest a fix.

1.2 Related Work

Delta debugging was originally proposed by Zeller and Hildebrandt [ZH02]. They used the algorithm to isolate failure causes in the input of programs, in process schedules of parallel applications, or program code changes. There, a suitable test function and a decomposition strategy between a passing and a failing test case is used to narrow down the difference between both test cases. Misherghi and Su [MS06] suggest an improvement over the general delta debugging algorithm and call it Hierarchical Delta Debugging. Their approach considers the semantics and structure of data to early prune irrelevant parts of the input and to create simpler outputs. Hence, the search space is limited, and thus fewer test cases are needed to isolate the minimal difference between a passing and a failing test case. As result, more complex problems can be handled compared to the original delta debugging algorithm. Both works apply delta debugging in the software domain, while our approach extends the application field to support the debugging of system designs. To our best knowledge, the presented approach is the first work that uses this debugging technique in the SystemC context.

Delta debugging is able to isolate arbitrary failure causes. However, there are specialized approaches to detect or to prevent hard to find errors in parallel programs. Such problems are often caused by a violation of communication constraints or the erroneous synchronization between parallel components.

Cheung et al. [CS+06] present an approach that dynamically monitors a SystemC simulation and reports a deadlock once it has occurred. A dynamic
synchronization dependency graph is created for this purpose. The vertices represent process dependencies and the directed edges denote synchronization dependencies. During simulation vertices and edges are dynamically added and deleted whenever a process suspends its execution. A recurring loop detection algorithm searches for cyclic dependencies afterwards. The approach has to know all processes that can notify a particular event for what static analysis techniques are used. Due to the nature of static analysis, notifying processes are approximated conservatively, i.e. the real number is typically smaller. A similar deadlock detection approach is proposed for the Metropolis environment [CD+05]. Metropolis is a complete system level design framework with its own model of computation. Both approaches dynamically report a deadlock once it has occurred. This could indicate the failure situation but does not suggest a fix. In contrast, delta debugging reports a minimal difference between a passing and a failing simulation run in a post-mortem analysis. Moreover, due to the generality of delta debugging, the analysis is not limited to specific language constructs used for synchronization as in [CS+06] or [CD+05].

A number of tools, e.g. [SB+97], [PS03], [YRC05], instrument programs to detect race conditions in software programs. Instrumentation allows to evaluate the locking discipline on shared variables. Generally, two approaches are distinguished. Lockset-based tools associate a lock candidate set to record all locks which are used to protect a shared location. When a thread accesses a shared resource, the intersection between the thread locks and the particular location locks may not be empty. A more sophisticated analysis performs a so-called happens-before analysis. The algorithm works with clocks to compare time stamps when a shared location is accessed. There, each thread holds a clock and keeps track of all other thread clocks. Moreover, each shared location stores the time stamps of the last accessing thread. If a thread gains access to a shared location, its time stamp must be higher or equal to the time stamp of previously called threads. Otherwise a race condition has occurred. Modern tools, such as MultiRace [PS03] and RaceTrack [YRC05], combine both approaches. The need for an instrumentation prevents an application of these tools in production systems. Furthermore, dynamic race detectors are not sound, i.e. they only check code that is actually executed which is similar to the delta debugging approach. Finally, all tools utilize implementations to handle the very specific error class of race conditions. On the other hand, delta debugging is a more general approach.

The verification of particular properties of concurrent systems is often done by using formal verification techniques, e.g. to show the absence of deadlocks or data races, e.g. [HJM04], [Sto00], or fairness properties. For this reason, a suitable high-level abstraction of the system has to be available.
Either the abstraction is created from the actual system or it is defined directly from the specification which could be a non-trivial and error-prone task. Such an abstraction suffers from the state explosion problem. Hence, the technique does not scale very well in case of complex concurrent systems but if the algorithm terminates, the results are precise and sound. Our approach is fully based upon simulation, and thus can handle arbitrary complex, real-world system designs.

Several tools use static analysis techniques to detect race conditions or deadlocks in software programs or system designs, e.g. [EA03], [FF01], [SBR02]. As already mentioned in Chapter 3, static analysis relies on conservative approximations which result in possibly numerous reported false positives. To solve this problem, the most sophisticated tools, such as RacerX [EA03], combine an interprocedural dataflow analysis with heuristics, statistical analyses, and ranking techniques. So, the analysis reports highly precise results on large, real-world programs.

2 AUTOMATIC ISOLATION OF FAILURE CAUSES

At first, this section summarizes the requirements for an automated isolation of actual failure causes. Then, the delta debugging approach is described.

2.1 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, also the actual cause, is found. An algorithm implementing the described procedure shall meet the following requirements:

- *Narrowing down strategy*. A strategy is needed that systematically narrows down the difference between a passing and a failing test case.

- *Rating strategy*. An automated test function has to assess a newly created alternate test case whether the failure has disappeared.

Especially the automated test function can be difficult to implement since each class of failure usually requires another test strategy. So, different test data have to be collected and checked such as executed program statements.
2.2 Methodology

A simple and often used debug procedure is simplification. Simplification removes aspects from a failing test case, e.g. lines in the program input, as long as the observed failure disappears. A more efficient approach is delta debugging, abbreviated \textit{dd}, which was proposed by Zeller and Hildebrandt in [ZH02]. It bases on isolation where the passing as well as the failing test case are modified in parallel. Figure 6.2 sketches the general delta debugging algorithm. It calculates the minimal difference between a passing and a failing test case. Whenever a test fails, the failing test case $c_F$ is “reduced”. Additionally, whenever a test passes, the passing test case $c_P$ is “increased”. Hence, the algorithm iteratively narrows down the minimal difference between a passing and a failing run. As result, the \textit{dd} algorithm returns a test case pair $(c_P', c_F')$, where the difference $\Delta = c_F' \setminus c_P'$ is 1-minimal.

\textbf{Definition 13.} A difference between a passing test case $c_P$ and a failing test case $c_F$ is called 1-minimal if this difference is relevant to produce the actual failure. I.e. adding the difference to the passing test case $c_P$ would raise the failure [Zel05].

The efficiency of the \textit{dd} algorithm is closely related to the result of the test function. If all test cases return an unresolved test outcome, the number of tests in \textit{dd} is quadratic to the difference between the input test cases $|c_F \setminus c_P|$ since the algorithm successively increases its granularity. In cases of a defined result for each iteration, i.e. \textit{pass} or \textit{fail}, \textit{dd} has a logarithmic complexity since it behaves like a binary search.

Despite the automated procedure to isolate an actual failure cause, a human user is often more creative during debugging. That means, he possibly finds the failure cause faster. Nevertheless, an automated process is less error prone and systematically tests the complete search space. In practice, especially the writing of a simple but significant test function is a crucial part of the whole approach. This function has to implement much implicit knowledge to efficiently find the failure cause.

2.3 Approach Rating

Although an automatic debugging approach sounds promising, there are a number of limitations and problems:

- \textit{Define the test function.} One of the most crucial parts of the \textit{dd} algorithm is the definition of a proper test function. This function has to be especially adapted to the particular analysis task to check for the
certain failure. There, a too complex implementation could exceed the
effort of a manual debug procedure.

- **Find passing and failing runs.** A successful application of delta
debugging requires two runs, a failing and a passing run. These runs
have to be close enough to be minimized to a single difference. If
both runs are too distinct, the search space can become too large and
the algorithm fails due to complexity reasons.

- **Define a suitable decomposition strategy.** Every application domain
needs to define a suitable decomposition strategy. The designer has to
find a suitable element to calculate the difference between two simu-
lation runs, e.g. lines of the program input, or thread switching times.
The efficient calculation of the difference has an important impact on
the algorithm performance.

---

**Figure 6.2:** General delta debugging algorithm according to [ZH02]

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Require:</strong></td>
<td>set of all test cases $C$, a test function $test$ with $test: 2^C \rightarrow {pass, fail, unresolved}$, a failing test case $c_F$ with $test(c_F) = fail$, a passing test case $c_P$ with $test(c_P) = pass$</td>
</tr>
<tr>
<td><strong>Ensure:</strong></td>
<td>$(c_P', c_F') = dd(c_P, c_F)$ such that $c_P \subseteq c_P' \subseteq c_F'$, $test(c_P') = pass$, $test(c_F') = fail$, and $\Delta = c_F' \setminus c_P'$ is I-minimal</td>
</tr>
</tbody>
</table>

The **Delta Debugging** algorithm is defined as $dd(c_P, c_F) = dd'(c_P, c_F, 2)$

$$
\begin{align*}
(c_P', c_F') & \quad \text{if } |\Delta| = 1 \\
(c_P', c_P' \cup \Delta, 2) & \quad \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c_P' \cup \Delta_i) = fail \\
(c_F' \setminus \Delta, c_F', 2) & \quad \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c_F' \setminus \Delta_i) = pass \\
(c_P' \cup \Delta, c_F', max(n-1, 2)) & \quad \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c_P' \cup \Delta_i) = pass \\
(c_P' \setminus \Delta, max(n-1, 2)) & \quad \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c_P' \cup \Delta_i) = pass \\
(c_P', c_F', min(2n, |\Delta|)) & \quad \text{else if } n \leq |\Delta| \\
(c_P', c_F') & \quad \text{otherwise}
\end{align*}
$$

where $\Delta = c_F' \setminus c_P' = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$ with all $\Delta_i$ pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |\Delta|/n$ holds. The recursion invariant for $dd'$ is $test(c_F') = fail \wedge test(c_P') = pass \wedge n \leq |\Delta|$
- **Check the right failure.** During the delta debugging process many different failing test cases could be artificially created. Hence, the observed failure can differ from the searched failing behavior. To reject these alternate failures, the test function has to evaluate the found failure, exactly. Such a situation could be for instance the simulation time the failure occurred, or a stack trace called if the program had crashed.

- **Locate the defect.** Delta debugging reports only the failure-inducing cause for the observed failure. The algorithm does not point directly to the actual defect. Rather, the result suggests a fix to make the failure disappear. To fix the program, debugging is still needed.

Despite these limitations and problems, delta debugging provides an important aid to automate debugging. It helps the designer to isolate a failure cause after a finite time. Hence, this technique could be a valuable approach to locate hard-to-find defects.

## 3 AUTOMATIC ISOLATION OF FAILURE CAUSES IN SYSTEMC

The ISOC tool integrates the delta debugging algorithm into the SHIELD debugging environment (see Section 3 on page 83). First, ISOC is exemplarily used for the automatic detection of failure-inducing process schedules. The second part describes the application of delta debugging to isolate failure-inducing simulation input in SystemC designs.

### 3.1 Debugging Process Schedules

Exploiting the range of description capabilities in parallel, multithreaded SDLs such as SystemC complicates debugging of therewith described system models. Especially, the nondeterministic SystemC scheduler can cause many failures that are difficult to debug manually such as deadlocks, or race conditions. A recording of process activations in a process schedule forms one basis to enable an automatic localization of deadlocks or races in SystemC designs. Hence, the failure-inducing difference between two schedules can be narrowed down, automatically. Therefore, ISOC provides a set of features:

- **Deterministic record/replay.** By using the non-intrusive implementation approach of SHIELD (see Section 3.5 on page 93), ISOC extends the SystemC scheduler by a record and replay facility for simulation
runs. Hence, process activations in terms of process schedules can be handled. Here, ISOC provides the debugging command `ptrace` (see Table 6.1) which enables schedule recording and deterministic replaying.

- **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 if and only if a particular method or thread process is activated at a specific point in time during simulation. ISOC starts the `dd` algorithm by calling the `ldl` command (see Table 6.1).

- **Root-cause analysis.** The system design is debugged while replaying the reported failure-inducing process schedule. Here, the system-level debugging features of SHIELD assists the designer in locating the failure-causing defect.

In Figure 6.3 the particular debug process in ISOC is shown. If the user has detected a failure, SHIELD provides a certain debug strategy in terms of the `dd` algorithm.

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, ISOC proposes to narrow down the failure-inducing difference between both schedules using `dd`. After analysis, the minimal difference is reported and the associated process schedule is used to debug the erroneous simulation. During simulation, all debugging features of SHIELD are available to locate the failure-causing defect.
3.1.1 Deterministic Record/Replay Facility

A SystemC design executes all its method and thread processes in a non-preemptive fashion (see Section 1.2 on page 15). Each process activation is determined by the execution logic and is represented by a particular point in time.

**Definition 14.** Let $D$ be a SystemC design. A tuple $(t_i, \delta_n)$ with the simulation time $t_i$ with $0 \leq t_i \leq t_{end}$ and the $n$th delta cycle $\delta_n$ with $n \geq 1$ is called an activation time point. $X$ is the set of all activation time points of $D$. 

![Figure 6.3: Isolation of failure-inducing process schedules](image)

**Figure 6.3:** Isolation of failure-inducing process schedules
Definition 15. Let $X$ be the set of all activation time points of a SystemC design $D$. A 4-tuple $\Psi = (I, X, \pi, f)$ is called process schedule of a concrete simulation run of $D$ with

- $I$: the finite set of instantiated method and thread processes in $D$,
- $X$: the finite set of activation time points of $D$,
- $\pi$: a sequence $\pi = (t_0, \delta_1), \ldots, (t_{\text{end}}, \delta_{\text{end}})$ of activation time points, and
- $f$: the function $f : (t_i, \delta_n) \rightarrow I^*$ assigning each activation time point a number of processes to be consecutively activated (process activations) at this time point.

If the record mode is enabled during simulation, ISOC records a process schedule until the specified end time $t_{\text{end}}$ is reached.

Example 23. Figure 6.4 shows an extract of a process schedule recorded for the “deadlock” example presented on page 161.

A previously recorded process schedule $\Psi = (I, X, \pi, f)$ is replayed during the activated replay mode in five phases:

1. Initialization phase. This phase initializes all processes in $I$ and sets the simulation time to the first recorded activation time point in $\pi$, i.e. $(t_i, \delta_n)$ with $i = 0$ and $n = 1$.
2. Evaluation phase. All processes $x \in f(t_i, \delta_n)$ 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 $\pi$ is retrieved. In case, processes become active at $(t_i, \delta_m)$ with $m > n$, step 2 is called with $(t_i, \delta_m)$.

5. **Timed notification phase.** If processes become active at $(t_k, \delta_m)$ with $k > i$ and $m > n$, the timed notifications are processed. Moreover, the simulation time is advanced. Then, step 2 is called with $(t_k, \delta_m)$. The simulation stops if there is no further element in $\pi$.

The $dd$ algorithm generates virtual new process schedules without any knowledge of the SystemC simulation and program semantics. So, the consistency of the generated schedule has to be checked for validity. A schedule is discarded and marked with an unresolved simulation outcome, if

- a process is activated twice at $(t_i, \delta_n)$ without becoming runnable in that delta cycle a second time,
- the schedule file specifies the execution of a process at $(t_i, \delta_n)$ that was already activated at $(t_i, \delta_{n-m})$ although its execution was suspended by a timed waiting statement,
- the simulation logic determines a process that becomes ready to run at $(t_i, \delta_n)$ through immediate event notifications but the recorded schedule does not contain a proper activation, or
- a process suspends its execution at $(t_i, \delta_n)$ and is waiting for a certain event but the process schedule instructs its activation at $(t_{i+k}, \delta_{n+m})$ without the particular event notification has been occurred.

### 3.1.2 Isolating Failure Causes

The implementation of the $dd$ algorithm as shown in Figure 6.2 is straightforward. Solely, the calculation of the difference $\Delta$ between a passing process schedule $c_p = \Psi_p = (I, X_p, \pi_p, f_p)$ and a failing schedule $c_F = \Psi_F = (I, X_F, \pi_F, f_F)$ needs to be defined. To do this, each process activation in $f_p$ and $f_F$ is assigned a unique slot number $s_k$. These numbers are used to calculate the $\Delta$ between the $n$th activation of a process $x \in I$, written $x^n$, in both process schedules. Assuming $x^n = s_m$ in $c_p$ and $x^n = s_k$ in $c_F$. The difference between $c_F$ and $c_p$ for $x^n$ is calculated by $\Delta_p[s_m] = s_k - s_m$, so that $c_p \cup \Delta_p = c_F$. The reduction step $c_F \setminus \Delta_F$ of the $dd$ algorithm is implemented by a “reversed delta” $\Delta_F$ 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. These slots are filled with virtual, never executed, process activations balancing the number of activated processes.
Chapter 6 Isolating Failure Causes

The \( dd \) algorithm divides \( \Delta_P \) and \( \Delta_F \) into \( n \) disjoint parts with
\[
\Delta_P = \Delta_{P,1} \cup \Delta_{P,2} \cup ... \cup \Delta_{P,n} \quad \text{and} \quad \Delta_F = \Delta_{F,1} \cup \Delta_{F,2} \cup ... \cup \Delta_{F,n}.
\]
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 \( (t_i, \delta_n) \) 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 \( c_P \cup \Delta_{P,i} \). Due to the partial delta calculation using \( \Delta_{P,i} \) or \( \Delta_{F,i} \), it could happen that a calculated target slot is already occupied by an unmoved slot entry. In that case, a temporary slot is provided. If the actual slot will be freed due to a further delta calculation, the temporal slot entry becomes the permanent entry. Then, the new schedule is checked for consistency and is simulated to check for pass or fail. Afterwards, the procedure is started again and ends if the difference is 1-minimal.

**Example 24.** Figure 6.5 sketches a simple producer–consumer application. The producer component creates a stream of data and sends it to the consumer (SC_THREAD p.C). A thread process at the consumer side (SC_THREAD c.G) receives the data and subsumes them into blocks of a fixed size. After one block is completed, it is sent to a processing unit. Data processing is implemented using two levels, i.e. Processing Level A (SC_THREAD c.A) and Processing Level B (SC_THREAD c.B). A wrong synchronization of the processing threads yields to incompletely processed data in some cases. So, delta debugging is used to locate the defect. Table 6.2 shows the calculation of \( \Delta_F \) (column 4) for the two recorded passing and failing process schedules \( c_P \) and \( c_F \) (columns 2 and 3). Both process schedules are represented by a sequence of activation time points and the assigned processes according to Definition 15. Since the failing schedule misses two process activations, two dummy slots are added. Using the first delta part \( \Delta_{F,1} \) (column 5) generates a new process schedule (column 7) that is simulated (column 8). Here, the dummy slots are left out from any simulation. Due to an erroneous simulation logic the algorithm will proceed with \( \Delta_{F,2} \).

![Figure 6.5: Simple producer–consumer application](image-url)
Besides the delta calculation, a test function \textit{test} has to be implemented. This function corresponds to the particular analysis task, e.g. deadlock detection or data race isolation. ISOC supplies a test function template that has to be implemented by the designer according to the design analysis problem.

### 3.1.3 Root-Cause Analysis

Finally, the reported minimized process schedule is replayed using ISOC. The system-level debugging features of SHIELD (see Section 3.3 on page 86) allow to debug the location of the defect that has caused the actual failure. There, simulation time and delta cycle information in the schedule file provides valuable information for defect localization.

### 3.2 Debugging Program Input

Delta debugging can be also used to automatically isolate a certain piece of SystemC simulation input yielding an erroneous simulation behavior. The objective is to find the minimal difference between two input data sets. The minimal difference could be a single character (set), or a particular line of input. As basis for the realization an existing Python implementation of the \textit{dd} algorithm.

#### Table 6.2: Illustrating the first steps of the \textit{dd} algorithm

<table>
<thead>
<tr>
<th>$s_k$</th>
<th>$c_p$</th>
<th>$c_F$</th>
<th>$\Delta_F$</th>
<th>$\Delta_{F,1}$</th>
<th>$\Delta_{F,2}$</th>
<th>$c_F \cup \Delta_{F,1}$</th>
<th>$c_{\text{new}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
</tr>
<tr>
<td>1</td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
</tr>
<tr>
<td>2</td>
<td>((0, 1), c.B)</td>
<td>((0, 1), c.A)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>((0, 1), c.B)</td>
<td>((0, 1), c.B)</td>
</tr>
<tr>
<td>3</td>
<td>((0, 1), c.A)</td>
<td>((0, 1), c.B)</td>
<td>–1</td>
<td>–1</td>
<td></td>
<td>((0, 1), c.A)</td>
<td>((0, 1), c.A)</td>
</tr>
<tr>
<td>4</td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
</tr>
<tr>
<td>5</td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
</tr>
<tr>
<td>6</td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), p.C)</td>
<td>((0, 1), p.C)</td>
</tr>
<tr>
<td>7</td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
<td>0</td>
<td></td>
<td></td>
<td>((0, 1), c.G)</td>
<td>((0, 1), c.G)</td>
</tr>
<tr>
<td>8</td>
<td>((0, 1), c.A)</td>
<td>((0, 1), c.B)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>((0, 1), c.A)</td>
<td>((0, 1), c.A)</td>
</tr>
<tr>
<td>9</td>
<td>((0, 1), c.B)</td>
<td>((0, 1), c.A)</td>
<td>–1</td>
<td>–1</td>
<td></td>
<td>((0, 1), c.B)</td>
<td>((0, 1), c.B)</td>
</tr>
<tr>
<td>10</td>
<td>((0, 1), c.A)</td>
<td>((20, 3), c.B)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>((20, 3), c.B)</td>
<td>((20, 3), c.B)</td>
</tr>
<tr>
<td>11</td>
<td>((20, 3), c.B)</td>
<td>[((20, 4), c.A)]</td>
<td>–1</td>
<td>–1</td>
<td></td>
<td>[((20, 4), c.A)]</td>
<td>[((20, 4), c.A)]</td>
</tr>
<tr>
<td>12</td>
<td>((20, 3), c.A)</td>
<td>[((20, 4), c.A)]</td>
<td>0</td>
<td></td>
<td></td>
<td>[((20, 4), c.A)]</td>
<td>[((20, 4), c.A)]</td>
</tr>
</tbody>
</table>
algorithm is used [Zel08]. This implementation was modified to determine the
difference between two text files on a line-by-line basis instead of a decompo-
sition of the input into single characters. The test function is implemented by
analyzing properties generated by DIANOSIS (see Section 3 on page 116). Hence,
missing or additional properties make the difference between a passing or a
failing test. Figure 6.6 depicts the described analysis flow.

4 EXPERIMENTAL RESULTS

The following section summarizes the experimental results while applying
the dd algorithm to different SystemC designs. First, delta debugging is used
to automatically detect a failure-inducing instruction sequence in erroneous
SIMD programs. Second, two experiments demonstrate the usage of ISOC to
narrow down failure-inducing process schedules.

4.1 SIMD Data Transfer Example Continued

In Chapter 5 1,009 randomly created programs were simulated on the
SIMD data transfer example to produce 1,009 simulation traces. Thirteen
of these programs have shown a difference in the number of generated
properties. Precisely, the difference concerned the handshake property
core.ld_req → core.st_req. As already shown in Chapter 5, this find-
ing has indicated an erroneous handling of the instruction sequence “multiply–
subtraction instruction followed by a maximum calculation”. There, the error
was manually detected by a comparison of the commonalities between all 13
programs. This manual procedure is a very tedious and error-prone.
By using delta debugging, the failure-inducing instruction sequence can be isolated automatically. Here, the analysis flow from Figure 6.6 is applied. The test function checks whether the missing handshake property could be found or not. The dd algorithm is started with the failing program while the empty program is assumed to pass. As result, the algorithm reports the minimal difference between a passing and a failing SIMD program in terms of a single instruction. So, either the multiply–subtraction or the maximum calculation instruction has to be removed to make the failure disappear, e.g.:

Process program tmp/prog_116
Output: Error detected: [(191, 'maxu R15, R15, R1\n')]
Done. See tmp/prog_116.delta.log1 and tmp/prog_116.delta.log2 for details

Process program tmp/prog_204
Output: Error detected: [(21, 'mulsubss R7, R0, R11, R3\n')]
Done. See tmp/prog_204.delta.log1 and tmp/prog_204.delta.log2 for details

Using an automated approach allows a much faster error location. The interpretation of the analysis output of all 13 programs suggested that either one of the reported instructions or their combination result in the observed wrong simulation behavior.

Figure 6.7 depicts the dd algorithm while narrowing down the failure-inducing program statement in program 433. After 80 tests, 904 initial differences are reduced to a minimal difference.
Table 6.3: *dd* algorithm applied on 13 erroneous programs

<table>
<thead>
<tr>
<th>Program (# program lines)</th>
<th>Initial difference</th>
<th>Overall test runs</th>
<th>Passed tests</th>
<th>Failed tests</th>
<th>Unresolved tests</th>
<th>Analysis timea</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>329</td>
<td>151</td>
<td>4</td>
<td>6</td>
<td>141</td>
<td>12.5 s</td>
</tr>
<tr>
<td>204</td>
<td>406</td>
<td>340</td>
<td>3</td>
<td>21</td>
<td>316</td>
<td>32.4 s</td>
</tr>
<tr>
<td>208</td>
<td>400</td>
<td>479</td>
<td>3</td>
<td>47</td>
<td>429</td>
<td>47.0 s</td>
</tr>
<tr>
<td>433</td>
<td>904</td>
<td>80</td>
<td>4</td>
<td>3</td>
<td>73</td>
<td>9.3 s</td>
</tr>
<tr>
<td>440</td>
<td>391</td>
<td>27</td>
<td>5</td>
<td>2</td>
<td>20</td>
<td>3.9 s</td>
</tr>
<tr>
<td>548</td>
<td>623</td>
<td>312</td>
<td>4</td>
<td>9</td>
<td>299</td>
<td>28.0 s</td>
</tr>
<tr>
<td>669</td>
<td>726</td>
<td>984</td>
<td>17</td>
<td>103</td>
<td>864</td>
<td>1 m 51.8 s</td>
</tr>
<tr>
<td>755</td>
<td>329</td>
<td>202</td>
<td>3</td>
<td>13</td>
<td>186</td>
<td>17.2 s</td>
</tr>
<tr>
<td>787</td>
<td>281</td>
<td>414</td>
<td>3</td>
<td>37</td>
<td>374</td>
<td>38.9 s</td>
</tr>
<tr>
<td>788</td>
<td>752</td>
<td>1,198</td>
<td>19</td>
<td>138</td>
<td>1,041</td>
<td>2 m 22.2 s</td>
</tr>
<tr>
<td>856</td>
<td>705</td>
<td>1,076</td>
<td>10</td>
<td>97</td>
<td>969</td>
<td>2 m 3.0 s</td>
</tr>
<tr>
<td>861</td>
<td>403</td>
<td>398</td>
<td>3</td>
<td>37</td>
<td>358</td>
<td>36.8 s</td>
</tr>
<tr>
<td>981</td>
<td>938</td>
<td>258</td>
<td>3</td>
<td>6</td>
<td>249</td>
<td>24.0 s</td>
</tr>
</tbody>
</table>

a. Test system: AMD Opteron™ 248 processor @2200 MHz, 3 GB RAM

Table 6.3 summarizes the number of required test runs to isolate the failure-inducing instruction sequence in each of the 13 erroneous programs. The *dd* algorithm requires 27 to 1,198 runs while this number does not correlate with the length of the input program, i.e. the initial difference. This discrepancy results from the very different number of unresolved tests while checking new programs. New programs are created by mixing the current failing and passing program. There, a different quantity of spurious programs is generated that violate the program logic. The violated program logic produces an unresolved test outcome. The analysis times, that are reported in Table 6.3, show the efficiency of an automatic approach. In such a short time a manual debug procedure would not be possible, in general. After the failure-causing input is known, the example is debugged using the SHIELD debugging environment. As a result, the designer identifies a wrong synchronization of the data transfer in case of a combination of both instructions.
This section summarizes the experimental results concerning the isolation of failure-inducing process schedules (see Section 3.1 on page 150). Two synthetic SystemC designs were analyzed, i.e. a simple producer–consumer design and a deadlock example. Both examples randomly, but definitely, produce various erroneous situations.

### 4.2 Producer–Consumer Application

In general, a SystemC design has to be independent of any process execution order. In the producer–consumer application from Example 24, a wrong synchronization could lead to wrong simulation results. Hence, transmitted data are incompletely processed. Here, the nondeterminism at the delta cycle level causes a serious problem: If the data processing thread c.A is initially executed before thread c.B, the initial event notification runs into the void since events are not persistent in SystemC. The wrong simulation outcome is used to write a proper test function for the dd algorithm. Following the flow in Figure 6.3, a passing and a failing run are generated at first. After 21 test runs, the dd algorithm has been isolated the failure-inducing minimal difference between both recorded process schedules. Figure 6.8 depicts the dd analysis result. The marker line indicates the (first) difference between both process schedules. Hence, the user knows that the failure is caused by a changed activation sequence of the threads c.A and c.B. Using the reported failing schedule allows to replay and debug the erroneous simulation in ISOC. Debugging shows that the event used for thread communication is sent by thread c.A but runs into the void since there is no process waiting for it, yet.

---

**Figure 6.8: dd algorithm analysis result for Example 24**
4.2.2 Deadlock Example

Figure 6.9 sketches the general architecture of a SystemC design definitely producing a deadlock after a random time. The example is based on a SystemC Training Course [IIS]. Four types of threads, i.e. A, B, C, and D, try to acquire four different resource types, i.e. 1 to 4. The design provides two instances of resource 2 and one instance of all other resources. During simulation each thread instance tries to lock two particular resource instances one after another to start “working”. If the thread gets the first resource, it tries to lock a second resource, immediately. In case a lock does not succeed, the thread waits for a random time and tries a relock. If both resources could be successfully locked, the thread “works” a random time and releases the resources afterwards. Since the thread keeps the first resource locked, while it is waiting for the availability of the second one, a deadlock will randomly occur.

To isolate the process which causes the observed deadlock, a test function for the $dd$ algorithm is needed. Here, a resource allocation graph is created from a generated simulation log. Using the graph, a deadlock can be detected, whenever a cycle is found. Since the example can produce various deadlock situations at random, the particular delta cycle and the simulation time of the actual deadlock is saved. Hence, the test function looks only for that particular deadlock situation (see Section 2.3 on page 148). However, a simple test function would recognize any deadlock whenever two threads are waiting for each other.

For test purposes, the example was run several times to create passing and failing process schedules over different simulation time lengths. Table 6.4 summarizes the obtained analysis results running the $dd$ algorithm on the example with a clock period of 1 ns.

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). Due to a violated simulation semantics, unresolved test outcomes (column 6) are produced most frequently. As can be already seen in Table 6.3, the number of needed simulation runs does not correlate to the initial difference between the input schedules. In fact, the number of unresolved test outcomes has a major impact on the performance of the \textit{dd} algorithm (column 7). In the current implementation the \textit{dd} algorithm and the SystemC simulation communicates via files which hampers an efficient error search for long running design simulations. Nevertheless, Table 6.4 demonstrates the applicability of the \textit{dd} algorithm to isolate failure-inducing process activations in SystemC designs systematically in a finite and short time.

<table>
<thead>
<tr>
<th>Simulation time in ns</th>
<th>Initial difference between process activations</th>
<th>Overall test runs</th>
<th>Passed tests</th>
<th>Failed tests</th>
<th>Unresolved tests</th>
<th>Analysis time$^\text{a}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2,820</td>
<td>48</td>
<td>2</td>
<td>17</td>
<td>29</td>
<td>7 s</td>
</tr>
<tr>
<td>100</td>
<td>16,330</td>
<td>158</td>
<td>6</td>
<td>17</td>
<td>135</td>
<td>29 s</td>
</tr>
<tr>
<td>200</td>
<td>51,594</td>
<td>335</td>
<td>6</td>
<td>22</td>
<td>307</td>
<td>68 s</td>
</tr>
<tr>
<td>300</td>
<td>51,972</td>
<td>218</td>
<td>4</td>
<td>33</td>
<td>181</td>
<td>49 s</td>
</tr>
<tr>
<td>400</td>
<td>147,290</td>
<td>825</td>
<td>4</td>
<td>75</td>
<td>746</td>
<td>193 s</td>
</tr>
<tr>
<td>500</td>
<td>155,322</td>
<td>858</td>
<td>4</td>
<td>61</td>
<td>793</td>
<td>230 s</td>
</tr>
<tr>
<td>1,000</td>
<td>1,071,436</td>
<td>363</td>
<td>4</td>
<td>70</td>
<td>289</td>
<td>204 s</td>
</tr>
<tr>
<td>2,000</td>
<td>7,716,550</td>
<td>617</td>
<td>11</td>
<td>202</td>
<td>404</td>
<td>1,803 s</td>
</tr>
</tbody>
</table>

$^\text{a. Test system: Intel Centrino Duo T2400@1830MHz, 1 GB RAM, SystemC Kernel 2.2.0 compiled with gcc-4.2.4}$

5 SUMMARY AND FUTURE WORK

In this chapter an experimentation technique has been proposed that automatically isolates various failure causes in ESL designs written in SystemC. The delta debugging algorithm narrows down failure causes using a series of controlled simulation runs. The algorithm reduces a passing and a failing run to a minimal difference automatically. There, all parts irrelevant for that failure are removed. The reported difference causes the observed failure and suggests a fix to make the failure disappear.
Two use cases of delta debugging demonstrate its applicability for ESL design and emphasize the strength of a combination of different debugging techniques. The first use case shows the isolation of a failure-inducing simulation input. Therefore, property generation as inductive technique has been used to write a proper test function. The second use case extends an observation technique in terms of the SHIELD debugging environment by the ISOC component. ISOC provides a flow to find the minimal difference between two process schedules.

Depending on the application, failures caused by a different execution order of processes can be automatically found. As a basis, the SystemC scheduler was augmented by a deterministic record and replay facility. Herewith, formerly recorded simulation runs can be replayed. Based on two input schedules, ISOC generates alternate process schedules until a minimal difference between the input schedules has been found. Subsequently, the debugging features in SHIELD and the replay facility of ISOC support the designer to locate the actual defect.

Summarized, the presented approach supports an automatic debugging of complex system designs especially in cases where the designer has no clue how to start debugging. An experienced designer sometimes finds the failure cause in a fewer steps. In contrast, the automated approach, although dump, completely and systematically tests the search space and will come up with a result in any case. Three experiments have been documented the efficiency of the $dd$ algorithm. In all examples the failure-causing minimal difference could be narrowed down after a relatively short time. So, the designer gets a result in any case which is (usually) reported in less time than a manual procedure would take.