Data Flow Testing for Virtual Prototypes

Muhammad Hassan[1], Vladimir Herdt[1], Hoang M. Le[1], Mingsong Chen[2], Daniel Große[1,3], Rolf Drechsler[1,3]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
[3]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hassan,vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de ; mschen@sei.ecnu.edu.cn

***Abstract*—Data flow testing (DFT) has been shown to be an effective testing strategy. DFT features a high fault detection rate while avoiding the intense scalability problems to achieve full path coverage. In this paper we propose to apply data flow testing for SystemC virtual prototypes (VPs). Our contribution is twofold: First, we develop a set of SystemC specific coverage criteria for data flow testing. This requires to consider the SystemC semantics of using non-preemptive thread scheduling with shared memory communication and event-based synchronization. Second, we explain how to automatically compute the data flow coverage result for a given VP using a combination of static and dynamic analysis techniques. The coverage result provides clear suggestions for the testing engineer to add new testcases in order to improve the coverage result. Our experimental results on real-world VPs demonstrate the applicability and efficacy of our analysis approach and the SystemC specific coverage criteria to improve the testsuite.**

I. Introduction

The last few years have witnessed fast-growing adoption of *Virtual Prototypes* (VPs) at the abstraction of *Electronic System Level* (ESL). Essentially, a VP is a software simulation model of the entire hardware platform, created by composing models of the individual IP blocks (i.e. Instruction Set Simulators, bus and peripheral models, etc.). For this purpose, the C++-based system modeling language SystemC together with *Transaction Level Modeling* (TLM) techniques [1] are being heavily used in industrial practice. Overall, the adoption of SystemC-based VPs has led to significant improvements on the design and verification of *System-on-Chips* (SoCs). The much earlier availability as well as the significantly faster simulation speed in comparison to the *Register Transfer Level* (RTL) models are among the main benefits of VPs. These enable hardware/software co-design and verification very early in the development flow. Serving as reference for (early) embedded software development and hardware verification, the functional correctness of VPs is very important. Hence, a whole VP as well as its individual components are subjected to rigorous verification.

Despite the recent progress in formal verification of SystemC models (see e.g. [2]–[8]), simulation-based verification is still the method of choice for SystemC-based VPs in industrial practice thanks to its scalability and ease of use. Basically, a set of stimuli is applied to the *Design Under Verification* (DUV; which can be either a whole VP, a set of components or a single component) and for each stimulus, the actual behavior is checked against the expected behavior (e.g. specified by reference outputs or temporal properties). Since a VP is in essence a software model, simulation-based verification for VPs is actually very similar to software testing and therefore, techniques from this domain can be borrowed to ensure a high quality of verification results. For example, high statement coverage of the DUV implied by the set of stimuli (also referred to as *testsuite* in the remainder of the paper to reflect the software testing point of view) is nowadays a minimum requirement.

However, although a necessary step, statement coverage (and also stronger code coverage metrics) have some well-known limitations in their capability to detect bugs as well as to reflect the thoroughness of verification. In the software testing community, a fault-based technique with better bug detection capability, known as mutation analysis [9], [10], has been considered for decades. Essentially, mutation analysis measures the adequacy of the testsuite with respect to its ability to detect a set of injected faults, which are introduced to the program under test by applying small syntactic changes. The key to effectiveness is a fault model that is both simple and representative for typical coding mistakes. The ideas have also been successfully transferred to hardware verification as well as to SystemC. For instance, dedicated fault models for SystemC-based mutation analysis have been proposed in [11]–[13]. Mutation based solutions for software-driven verification have also been presented [14]. Commercial mutation analysis tools with support for SystemC such as Certitude from Synopsys are also available.

Considering the successful adoption of mutation analysis, it is rather surprising that another effective testing technique know as *Data Flow Testing* (DFT) [15], [16], to the best of our knowledge, has not been yet considered for SystemC. DFT also holds the promise of better bug detection capability. The underlying idea of DFT is that the propagation of (wrong) data is a necessity to reveal bugs: If a line of code produces a wrong value, the execution after that point must include another line of code that uses this erroneous value, otherwise there will be no observable failures. Based on this information, researchers have proposed several data flow adequacy criteria. These criteria require the testsuite to sufficiently exercise the identified *definition-use pairs*, i.e. pairs of definition (a statement where a value is produced) and use (a statement where this value is used). Recent research in DFT focused on

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E, German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, University of Bremens graduate school SyDe, funded by the German Excellence Initiative, National Natural Science Foundation of China (No. 61672230), and German Academic Exchange Service (DAAD) in the PPP 57138060.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

automated test generation for data flow adequacy [17], [18] and on extending DFT to object-oriented programs [19], [20].

In this paper we propose the first DFT approach for SystemC-based VPs, which does not come without challenges. A SystemC DUV is essentially a concurrent program with non-preemptive thread scheduling, shared memory communication and event-based synchronization. This unique combination requires rethinking of the known DFT techniques. To this end, our contribution is twofold: First, we develop a set of SystemC specific coverage criteria for DFT, which takes the non-preemptive context switches and synchronization primitives of SystemC into consideration. Second, we explain how to automatically compute the data flow coverage result for a DUV using a combination of static and dynamic analysis. The coverage result provides clear suggestions for the testing engineer to add new testcases in order to improve the coverage result. Our experimental results on real-world VPs demonstrate the applicability and efficacy of our analysis approach and the SystemC specific coverage criteria to improve the testsuite quality.

II. Preliminaries

A. SystemC Running Example

For brevity, we refrain from giving a proper introduction to SystemC. Instead, we present here an example SystemC program (Fig. 1) that will be used to showcase the main ideas of our approach throughout this paper. The SystemC constructs and semantics necessary to understand the example will be explained as needed. We omitted the SystemC code required for instantiation and binding of components, i.e. the elaboration phase. The example consists of two modules *producer* and *consumer* that communicate through a FIFO. Their behavior is implemented in thread functions (producer: `prod_thread()` Line 37 - consumer: `recv()` Line 53, `filter()` Line 68, `send()` Line 79) registered in the simulation kernel. The behavior of the consumer depends on the input provided by the producer.

The FIFO provides a *write* and a *read* function that adds or removes an element, respectively. The write function is used from the producer thread in Line 41 and the read function from the consumer thread in Line 57. Both functions can potentially suspend the threads execution in case the FIFO is empty on read attempt (Line 16) or full at write attempt (Line 7). The thread becomes runnable again when the awaited event is notified in Line 11 or Line 21, respectively

The consumer module itself consists of three threads. The thread `recv()` is responsible to retrieve the next produced element `x` (Line 58) from the FIFO and transfers it to the send thread for processing. The transfer can happen in two ways: 1) Through the `filter()` thread that applies post-processing and checking (Line 68), or 2) directly without delay to the `send()` thread for high priority items (Line 79). However, the send thread only accepts one fast transfer at a time (Line 80, controlled by the `fast_mode` variable). Therefore, the filtering is always unconditionally initiated (Line 59) as fallback in case the send thread currently does not support fast processing. Finally, the `send()` thread will notify the `recv()` thread (Line 91) to transfer the next element.

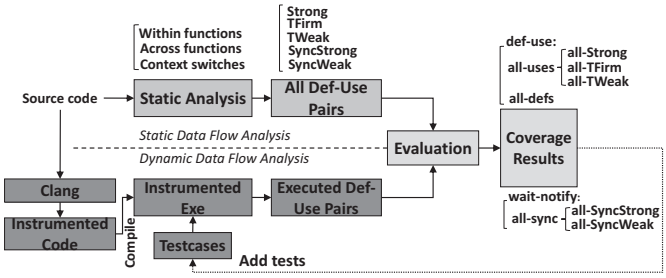

Fig. 2. An overview of our data flow testing approach for SystemC

B. Def-Use Association and Data Flow Testing

A def-use association is an ordered triple (x, d, u) such that d is a statement where variable x is defined and u is a statement where x is used. Furthermore, there is a path in the program from d to u without re-definition of x. For example, consider Fig. 1: variable `fast_mode` is defined in Line 80 and used in Line 82, so it is a def-use association. A def-use association (x, d, u) is exercised by a testcase t, iff execution of t goes through definition d and then use u without re-definition of variable x in-between.

Data flow testing tries to maximize the exercised def-use associations. Essentially, it works by refining the testsuite by adding testcases until the coverage criteria are met or testing resources are exhausted. This requires to detect def-use associations and measure the data flow coverage of the current testsuite. How to do this for SystemC and thereby taking SystemC specifics into account is shown in the following. Please note that we will use the term data flow association as a generalization of a def-use association to avoid confusion. The reason is that we define a SystemC specific wait-notify association, which is also a data flow association.

III. Data Flow Testing for SystemC

A. Overview

An overview of our data flow testing approach for SystemC is shown in Fig. 2. Essentially, our approach combines a static and dynamic analysis to fully automatically compute a SystemC specific data flow coverage result.

The *static analysis* (upper half of Fig. 2) identifies the set of all data flow associations. Our static analysis computes an over-approximation of all def-use associations and thus also contains *infeasible* associations, i.e. associations that cannot be exercised no matter which input is applied. A precise computation of all feasible associations requires heavy use of formal verification techniques and therefore is not practical due to scalability issues. To guide testcase selection, associations are classified into different disjoint groups based on the likeliness of being infeasible. Please note that the static analysis needs to be only run once at the beginning on the source code.

The *dynamic analysis* (lower half of Fig. 2) detects which data flow associations have been exercised by the current testsuite. It works by instrumenting the SystemC source file to log relevant runtime information. The instrumented source file is then compiled with a standard C++ compiler and executed for every testcase. The resulting logs are analyzed and combined to obtain the set of exercised data flow associations.

```
1  struct fifo : public sc_channel {
2    fifo(sc_module_name name)
3      : sc_channel(name), num_elements(0), first(0) {}
4
5    void write(char c) {
6      if (num_elements == max)
7        wait(read_event);
8
9      data[(first + num_elements) % max] = c;
10     ++num_elements;
11     write_event.notify();
12   }
13
14   void read(char &c){
15     if (num_elements == 0)
16       wait(write_event);
17
18     c = data[first];
19     --num_elements;
20     first = (first + 1) % max;
21     read_event.notify();
22   }
23
24 private:
25   enum e { max = 10 };
26   char data[max];
27   int num_elements, first;
28   sc_event write_event, read_event;
29 };
30
31
32 SC_MODULE(producer) {
33   SC_CTOR(producer) {
34     SC_THREAD(prod_thread);
35   }
36
37   void prod_thread() {
38     wait(0); // start together with the consumer
39     const char *str = "SystemC Example"; // input to
           the design, can influence consumer behavior
40     while (*str)
41       out->write(*str++); // out is bound to fifo
             instance
42   }
43
44   sc_port<fifo> out;
45 };
46
47
48 SC_MODULE(consumer) {
49   SC_CTOR(consumer) {
50     SC_THREAD(recv); SC_THREAD(send);
51     SC_THREAD(filter);
52   }
53
54   void recv() {
55     wait(0, SC_NS); // ensure send and filter are run
           first
56     char c;
57     while (true) {
58       in->read(c); // in is bound to fifo instance
59       x = c;
60       filter_event.notify(1, SC_NS);
61       if (x < 10) {
62         // high priortiy data handled immediately
63         send_fast_event.notify();
64       }
65       wait(recv_event);
66     }
67   }
68
69   void filter() {
70     while (true) {
71       wait(filter_event);
72       if (x < 0)
73         x = 0;
74       if (x > 126)
75         x = 126;
76       send_regular_event.notify(1, SC_NS);
77     }
78   }
79
80   void send() {
81     bool fast_mode = true;
82     while (true) {
83       if (fast_mode) {
84         wait(send_regular_event | send_fast_event);
85         fast_mode = false;
86       } else {
87         wait(send_regular_event);
88         fast_mode = true;
89       }
90       assert (x >= 0);
91       cout << x << endl;
92       recv_event.notify();
93     }
94   }
95
96   sc_port<fifo> in;
97 private:
98   int x;
99   sc_event recv_event, filter_event,
         send_regular_event, send_fast_event;
100 };
```

Fig. 1. SystemC example

In the next step, both static and dynamic analysis results are evaluated and combined to obtain a coverage result. Essentially, the result shows which data flow associations have been exercised by at least one testcase and which have been completely missed. An association can be missed due to two reasons: 1) The testsuite is insufficient to cover it. In this case a new testcase needs to be added. 2) The association is infeasible, i.e. there is no possible input that will cover it. In this case it can be ignored.

Our classification system, that ranks associations according to their likeliness of being infeasible, allows the testing engineer to focus his efforts on promising testcases to efficiently improve the coverage result. Please note that we do not yet consider automated test generation to exercise a specific data flow association in this work, as it will exceed the scope of this paper. Automated test generation is left for future work.

In the following we describe our classification system and the coverage result in more detail and demonstrate both using the running example.

B. Classification of Data Flow Associations

Our classification system attempts to preserve scalability of the data flow testing approach and at the same time provide meaningful results and suggestions to guide the testcase generation. We define *five SystemC specific classifications*: *Strong, TFirm, TWeak, SyncStrong*, and *SyncWeak*.

The first three (details see below) extend the classical notion of data flow testing that reason about variable definition and use. Therefore, these fall into the def-use association category. These classifications especially deal with the non-preemptive threads of the SystemC simulation (and hence the T in TWeak and TFirm).

The last two classifications are used to classify event-based synchronization of SystemC by means of the wait/notify function. This can also be considered as a data flow relation. The wait can be considered a definition which suspends the active thread, while the notify is considered a use. However, to avoid confusion we introduce a new data flow association called *wait-notify association* for these synchronization related flows.

1) Def-Use Associations: Our static analysis reports the following def-use associations (x, d, u): There is a static path from d to u in the program without re-definition of x in-between. Please note, there is a static path from every context switch statement from one thread to the start of a transition of every other thread. A transition starts at the beginning of a thread and right after a context switch.

Based on this general observation we define three classifications for def-use associations (x, d, u). In this context we define a *du-path* as a static path between d and u without re-definition of x:

- *Strong* : a) Every *du-path* is without context switch, or b) x is a thread local variable, or c) d is the only definition of x, i.e. x is a constant.
- *TFirm* : At least one *du-path* is without context switch and at least one *du-path* with context switch.
- *TWeak* : Every *du-path* contains a context switch.

Since a local thread variable cannot be re-defined on context switch, the def-use associations is considered Strong. This refinement of def-use associations provides clear guidelines for testcase selection. In general one should focus on Strong and TFirm associations first. In both cases there exists at least one (static) path without context switch in-between the definition and use.

2) Wait-Notify Associations: Similarly to def-use associations, we define a wait-notify association as an ordered triple (e, w, n) where w contains a wait of event e and n contains a notify of event e, and there exists a path in the program such that w is notified from n.

This definition does not require a notify to happen after the wait during execution. This is due to timed notification, where the notify statement only schedules the notification to happen at a later point in time. Such a timed notification is much more difficult to handle precisely using static information. Furthermore, the notification can be canceled, e.g. by issuing a new notification of event e. Immediate notifications on the other hand more directly resemble the classical data flow relation as the notification happens when the notify statement is executed. Events are inherently not thread local and there is always a context switch between a wait and a notify, which makes static analysis more difficult.

Our static analysis detects the following wait-notify associations (e, w, n): (1) A timed notify n for event e is executed and the scheduled notification is not canceled until a context switch is executed. And w is a wait for event e. (2) w is a wait for event e in one thread and n is an immediate notify of event e on another thread.

This approximation can be further refined by applying an additional analysis that computes transitions between threads more precisely. However, this approximation is sufficient when dealing with SystemC synchronization primitives. One of the reasons is that often only very few wait-notify statements operate on a single event. Therefore, it is reasonable to assume that most of them are feasible.

This observation motivates to introduce the following two classifications for wait-notify associations (e, w, n):

- *SyncStrong* : Wait w should have a one-to-one relationship with a notify n for an event e.
- *SyncWeak* : Otherwise, i.e. multiple wait or notify statements are available for event e.

C. Coverage Result

Every classification defines a disjoint set of data flow associations. Therefore, we define a coverage criterion for each classification. For instance, the all-Strong criterion is satisfied, iff all data flow associations classified as Strong have been exercised. Criteria for the remaining classifications – i.e. all-TFirm, all-TWeak, all-SyncStrong, and all-SyncWeak – can be defined analogously.

Based on these criteria, we can define specific def-use and wait-notify criteria:

- The all-uses criterion requires that all-Strong, all-TFirm and all-TWeak are satisfied.
- The all-defs criterion requries that for every definition D in the program at least one def-use association (x, d, u) with $D = d$ is exercised.
- The all-sync criterion requires that all-SyncStrong and all-SyncWeak are satisfied.

Finally, the all-data-flow criterion is satisfied iff all-uses and all-sync criteria are satisfied.

While in general satisfying all-data-flow criterion is not practical due to imprecisions in the static analysis, which can result in infeasible data flow associations, it is possible that some of the (sub-)criteria can be fully satisfied – or at least up to a high degree, i.e. 95% of the associations have been exercised. In particular, we expect that all-Strong, all-TFirm and all-SyncStrong to be the primarily focused criteria.

Both Strong and TFirm associations contain a (static) path without re-definition and without context switch between the definition and use. Therefore, we expect that a testcase will exercise them. Otherwise, the definition is dead code or all relevant paths (without re-definition of variable x) between definition and use are infeasible.

Similarly, if the SyncStrong criterion is not satisfied, it means that some notification has never reached a wait, or some wait has not been notified. This implies that some wait/notify statement is essentially unused.

D. Illustration

We have executed the SystemC example in Fig. 1 with four different inputs to gradually increase the data flow coverage. In particular, we used the following inputs for the `str` variable in Line 39: 1) "SystemC Example", 2) "Abc\x7f", 3) "a\tb\xff", and 4) "Test\x80". Table I shows the results, i.e. the statically classified data flow associations, and by which testcase they have been exercised (marked by X). Infeasible data flow associations are marked with '-' for all testcases. The information is grouped in four main columns, and is read from top-to-bottom and then from left-to-right. In this order the Strong, TFirm, TWeak, SyncStrong and SyncWeak associations are listed.

For example consider the Strong def-use association $(num\_elements, 10, 6)$ shown in the first column and exercised by all testcases. This one is Strong because all paths between Lines 10 and 6 are without re-definition of num_elements and are free of context switches. In fact there is only one possible path from Line 10 to Line 6 – first the write function is exited, then the while loop is not finished and so the write function is called again.

For the def-use association $(num\_elements, 10, 9)$, there are two paths between Line 10 and Line 9, due to the branch in Line 6. One involves a context switch and the other path does not. Therefore, this association is TFirm.

The def-use association $(first, 20, 9)$ is TWeak, because the only way to reach the use starting from the definition is through a context switch. This association is only exercised by the first testcase.

TABLE I
DATA FLOW ASSOCIATIONS FOR THE SYSTEMC EXAMPLE IN FIG. 1 SORTED BY CLASSIFICATION.

Strong					(x,72,73)			X		**TWeak**					(x,58,89)	X	X	X	X
(c,5,9)	X	X	X	X	(x,58,60)	X	X	X	X	(data,9,18)	X	X	X	X	(x,58,90)	X	X	X	X
(c,57,58)	X	X	X	X						(first,20,9)	X								
(fast_mode,80,82)	X	X	X	X	**TFirm**					(num_elements,10,15)	X				**SyncStrong**				
(fast_mode,84,82)	X	X	X	X	(data,3,18)	-	-	-	-	(num_elements,10,19)	X	X	X	X	(filter_event,70,59)	X	X	X	X
(fast_mode,87,82)	X	X	X	X	(first,3,9)	X	X	X	X	(num_elements,19,6)	-	-	-	-	(read_event,7,21)	X			
(max,25,6)	X	X	X	X	(first,3,18)	X	X	X	X	(num_elements,19,9)	X				(recv_event,64,91)	X	X	X	X
(max,25,9)	X	X	X	X	(first,3,20)	X	X	X	X	(num_elements,19,10)	X				(send_fast_event,83,62)				X
(max,25,20)	X	X	X	X	(first,20,18)	X	X	X	X	(x,72,71)	-	-	-	-	(write_event,16,11)	X	X	X	X
(num_elements,3,6)	X	X	X	X	(first,20,20)	X	X	X	X	(x,72,89)			X						
(num_elements,3,15)	X	X	X	X	(num_elements,3,9)	X	X	X	X	(x,72,90)			X		**SyncWeak**				
(num_elements,10,6)	X	X	X	X	(num_elements,3,10)	X	X	X	X	(x,74,71)	-	-	-	-	(send_regular_event,83,75)	X	X	X	X
(num_elements,19,15)	X	X	X	X	(num_elements,3,19)	-	-	-	-	(x,74,73)	-	-	-	-	(send_regular_event,86,75)	X	X	X	X
(str,39,40)	X	X	X	X	(num_elements,10,9)	X	X	X	X	(x,74,89)		X							
(str,39,41)	X	X	X	X	(num_elements,10,10)	X	X	X	X	(x,74,90)		X							
(str,41,40)	X	X	X	X	(num_elements,19,19)	X	X	X	X	(x,58,71)	X	X	X	X					
(str,41,41)	X	X	X	X						(x,58,73)	X	X	X	X					

The associations involving the `max` variable are classified Strong because it is a constant, so it cannot be redefined. The association $(c, 5, 9)$ is also Strong, even though there are two paths from 5 to 9 and one involves a context switch, because c is thread local – so it cannot be redefined due to the context switch. Most wait-notify associations are SyncStrong because there is only a single wait and notify for the event. The ports *in* and *out* are bound in the elaboration phase, which is not shown in the example. Therefore, no def-use association is reported for them in Table I.

The first testcase already achieves a reasonable data flow coverage. In particular for the FIFO component, which works independent of the actual input. Since the all-Strong and all-TFirm coverage criteria are already satisfied, the next step is to consider the TWeak associations.

The second and third inputs use special characters which are processed separately in the filter thread in Line 71 - Line 74. With the first three inputs, all feasible def-use associations are covered. Therefore, the maximal def-use coverage has been achieved for this example. Please note that full branch and statement coverage has also been achieved with this test set.

However, the coverage with regard to wait-notify associations can still be increased. In particular the association $(send\_fast\_event, 83, 62)$ has not been exercised. The reason is that with the current testsuite all special characters were passed through the filter thread, since the send thread has not been in *fast_mode*, i.e. waiting in Line 86. Therefore, we have added a fourth testcase to exercise its association. This testcase was able to detect a bug in the design, where an invalid character is passed to the send thread.

This example demonstrates that standard def-use coverage criteria alone are not sufficient for extensive testing of SystemC designs. Our proposed SystemC specific wait-notify coverage criteria is important for a high quality testsuite.

IV. IMPLEMENTATION DETAILS

This section describes the static and dynamic analysis of our data flow testing framework in more details. As aforementioned, the static analysis computes an over-approximation of data flow associations classified into disjoint groups. The dynamic analysis then detects which data flow associations really have been exercised by the testsuite.

A. Static Analysis

The static analysis is implemented using the LibTooling library for Clang compiler. Clang generates an *Abstract Syntax Tree* (AST) of the SystemC source code. The AST is parsed to extract the required information to perform static analysis. Then it applies three subsequent analysis steps: 1) Local analysis within every function, 2) Information propagation across function calls, 3) Consideration of the effects of context switches.

B. Dynamic Analysis

The dynamic analysis works in two steps: 1) Instrument the SystemC source code using the Clang compiler framework to log data flow relevant information. Then execute all testcases on the instrumented executable to generate the log. 2) Analyse the log line by line to build the exercised data flow associations. Both steps are described in the following.

1) Source Code Instrumentation: In the first step the SystemC source code is instrumented to log data flow relevant information. Therefore, it is analyzed statement by statement to detect 1) definitions and uses of variables, and 2) wait and notifies of events. For every such detected information a print instruction, which writes to a log, is placed before the statement. Please note, that the order of the print instructions is important in case multiple information are available, e.g. *i++;*, in general the uses are placed before the definitions. For the while loop (and similarly the for loop) print instructions for the loop condition are replicated at the end of the loop since the condition is re-evaluated in every loop step. Therefore, it is not enough to place them only before the while loop.

2) Data Flow Association Construction:

a) Def-Use Associations: The def-use associations can be identified in a straightforward way. We keep a mapping of active definitions. It relates each variable to its last definition, which is updated on re-definition. Whenever a use for a variable is found in the log, the corresponding definition is retrieved from the mapping.

b) Wait-Notify Associations: Detecting wait-notify associations requires additional work due to timed notifications. This decouples the notify statement from the actual event triggering. Therefore, we modified the SystemC kernel to write a log entry whenever an event is triggered.

For example consider the statement sequence *e.notify(1, SC_NS); wait(e);*, where an event e is scheduled for notification and then the thread is suspended to wait for the

notification e. The execution log contains a notify entry for event e followed by a wait entry for e. The actual trigger from the SystemC kernel appears later. In-between can be other log entries. Other threads can also wait for event e. However, a new notification for e will cancel the one before.

Based on this information, we keep a mapping from an event to a set of active wait statements and a mapping to the last scheduled notification. Once the kernel trigger is parsed for an event e, retrieve the last notification n and set of active waits S. Then add a wait-notify association (e, w, n) for every $w \in S$. Immediate notifications are handled in the same manner, the trigger simply appears directly after the notification scheduling in the log.

V. Experimental Results

In this section we present a case study to demonstrate our DFT approach for SystemC. We consider the LEON3-based VP SoCRocket [21] which has been modeled in SystemC TLM and consider one component from it in more detail: the *Interrupt Controller for Multiple Processors* (IRQMP). IRQMP handles the interrupts coming from different connected devices using priority mechanism. The model has I/O wires, register file and APB slave interface. Total of 32 interrupts are supported. When an interrupt arrives, the corresponding bit in the register is set. The IRQMP communicates with connected processors with an interrupt request (*irq_req*) or an acknowledgment (*irq_ack*). When an interrupt request is signaled for a processor, the IRQMP combines the `mask register` and the `pending register` with the `force register` to find the highest priority interrupt. The IRQMP also reads the `broadcast register` before forwarding the request to the processors. If the corresponding bit is set in `broadcast register`, the interrupt is broadcasted to all processors, i.e. written to the `force register` of all connected processors. In this scenario, the IRQMP expects acknowledges from all processors. On the arrival of an interrupt request, if the corresponding bit is not set in the `broadcast register`, it is simply set in the `pending register`. In this scenario, IRQMP expects an acknowledge from any processor.

The initial testsuite shipped with the IRQMP component consists of 60 tests achieving 63% statement coverage, 74% branch coverage, and 63% data flow coverage: a total number of 571 data flow associations has been computed by our static analysis, and 359 of them have been found to be exercised by the testsuite using our dynamic analysis. Initially 62% Strong, 71% TWeak, and 58% SyncWeak data flow associations are exercised. There is no TFirm association as each instruction has to pass *wait* statement, hence, a context switch is inevitable. The testsuite does not fulfill the all-uses, all-defs, and all-sync criteria. Therefore, the all-data-flow criterion is not satisfied. One interesting point is that there are no SyncStrong wait-notify pairs, instead there are 5 SyncWeak wait-notify pairs wrt. the only *sc_event e_signal*.

To increase the def-use coverage, the uncovered 38% Strong associations are satisfied first by adding additional tests manually. They are followed by 29% TWeak and 42% SyncWeak associations. In total, 54 additional tests are added to the initial testsuite. These new tests increased the overall data flow coverage to 88%, with 96% Strong, 91% TWeak, and 69% SyncWeak data flow associations exercised. After this point, we found it to be very hard to improve these numbers further. This demonstrates the need for future research on automated test generation techniques for SystemC that are capable to derive hard-to-find tests and to prove infeasibility of associations. With the now enhanced testsuite, we also achieve 92% statement coverage and 89% branch coverage. Despite the very high values of these code coverage metrics, they provide no insight into potential synchronization issues. In contrast our developed data flow coverage with regards to wait-notify association shows that in many cases there was no *wait* statement waiting for notify because some other function had already fulfilled the *wait* condition. This can lead to potential problems in integration of the component in a larger system/subsystem, e.g. an incoming interrupt which needs to be handled quickly can be delayed.

VI. Conclusion

In this paper we presented the first DFT approach for SystemC based VPs and SystemC specific coverage criteria. The criteria uses five classifications (Strong, TFirm, TWeak, SyncStrong, SyncWeak) for data flow associations. Furthermore, we explain how to automatically compute the data flow coverage results for a DUV using a combination of static and dynamic analysis. This allows to improve the coverage results by adding tests for uncovered def-use pairs since the users can get useful information. We have demonstrated the applicability in a real world VP showing the results of one IP model.

References

[1] *IEEE Standard SystemC LRM*, IEEE Std. 1666, 2011.
[2] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[3] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.
[4] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.
[5] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–116:6.
[6] V. Herdt, H. M. Le, and R. Drechsler, "Verifying SystemC using stateful symbolic simulation," in *DAC*, 2015, p. 49.
[7] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models – a case study," in *DATE*, 2016, pp. 1160–1163.
[8] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, p. 52.
[9] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, no. 4, pp. 279–290, 1977.
[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.
[11] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *DATE*, 2008, pp. 396–401.
[12] ——, "On the mutation analysis of SystemC TLM-2.0 standard," in *MTV Workshop*, 2009, pp. 32–37.
[13] A. Sen, "Concurrency-oriented verification and coverage of system-level designs," *TODAES*, vol. 16, no. 4, p. 37, 2011.
[14] D. Große, H. M. Le, M. Hassan, and R. Drechsler, "Guided lightweight software test qualification for IP integration using virtual prototypes," in *ICCD*, 2016.
[15] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 347–354, May 1983.
[16] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 367–375, Apr. 1985.
[17] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *ISSRE*, 2013, pp. 370–379.
[18] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *ICSE*, 2015, pp. 654–665.
[19] R. T. Alexander, J. Offutt, and A. Stefik, "Testing coupling relationships in object-oriented programs," *STVR*, vol. 20, no. 4, pp. 291–327, 2010.
[20] G. Denaro, A. Margara, M. Pezzè, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *ICSE*, 2015, pp. 947–958.
[21] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7, available at http://github.com/socrocket.