Guided Lightweight Software Test Qualification for IP Integration using Virtual Prototypes

Daniel Große[1,2] Hoang M. Le[1] Muhammad Hassan[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{grosse,hle,hassan,drechsle}@informatik.uni-bremen.de

Abstract—**Software-Driven Verification (SDV) has the promise to significantly reduce the overall time and effort for the task of IP integration and verification. With the help of SystemC Virtual Prototypes (VPs), SW tests to verify the (new) integrated IP blocks and the HW/SW integration can be developed in an early design stage and reused in the subsequent steps. However, the crucial question regarding the quality of these tests has not been considered so far. For this purpose, we propose in this paper a novel quality-driven methodology based on mutation analysis. By elevating the main concepts of mutation-based qualification to the context of SDV, our methodology is capable to detect serious quality issues in the SW tests. At its heart is a novel consistency analysis, that measures the coverage of the IP in HW/SW co-simulation in a lightweight fashion and relates this coverage to the SW test results to provide clear feedback on how to further improve the quality of tests. We provide two case studies on real-world VPs and SW tests to demonstrate the applicability and efficacy of our methodology.**

I. Introduction

The emergence of *Virtual Prototypes* (VPs) at the abstraction of *Electronic System Level* (ESL) has modernized the design and verification of *System-on-Chips* (SoCs) in many ways. In industrial practice, the C++-based system modeling language SystemC [1], [2] together with *Transaction Level Modeling* (TLM) techniques are being heavily used to create VPs. The much earlier availability as well as the significantly faster simulation speed in comparison to RTL are among the main benefits of SystemC-based VPs. These enable hardware/-software co-design and verification very early in the design flow, and in particular, the approach of *Software-Driven Verification* (SDV) proposed in [3]. Essentially, software tests are developed for functional verification of the (new) integrated IP blocks and the HW/SW integration. The tests are typically written in C and run on a processor core of the VP. The key benefit of SDV is that the tests can be reused along all following design phases, i.e. in RTL simulation, emulation, FPGA prototyping, and even the silicon. This is very valuable as IP integration is becoming more and more a bottleneck for today's high-performance SoCs that typically include multiple processor cores and hundreds of IP blocks.

To reap the most benefit from SDV, the quality of the software tests is crucial, as low-quality software tests could miss serious integration issues. However, to the best of our knowledge, the qualification of software tests with the particular focus on IP integration has not been considered so far.

In this paper we propose a novel guided approach to evaluate and improve software tests developed for integration verification of an IP block. Our approach is based on mutation analysis. In the software testing community, mutation analysis as proposed in [4], [5] has been considered for decades as a fault-based technique (see also Section II for more details). Essentially, it is checked whether the tests are capable of detecting (killing) the deviating behavior of a syntactically correct but modified program (a mutant). The ideas have also been transferred to hardware verification and are referred there as *functional testbench qualification* [6], [7] (see also Section II). In this context, the three main tasks of qualification are distinguished: 1) activate, i.e. stimuli have to be provided to activate the mutation; 2) propagate, i.e. the effect of the mutation has to propagate to an observable point; and 3) detect, i.e. the testbench must detect functional mismatches between the original design and the mutated one. However, these qualification tasks give very little information about the nature of mismatches in the compared designs.

Our paper makes a two-fold contribution to enable qualification of software tests for verification of IP integration. First, we define the main tasks activation, propagation and detection in the context of SystemC VP-based IP integration. Building on that, our main contribution goes one step further to provide a complete methodology to *guide* the verification engineer in improving the software test quality. If the mutation in the IP block is not killed by the software tests, the engineer wants to know the reason and improve the tests. For this problem we propose a novel *consistency analysis* that relates the mutation results with the coverage results of the original (not mutated) IP block verification and provides a guided solution: If they are inconsistent the methodology gives clear hints when and for which mutants to add more tests and when to use a more powerful coverage model. Following the proposed methodology, a big jump in quality of the SW test suite can be achieved in consecutive iterations while using different variants of well-known code coverage models which can be very easily measured.

II. Related Work

Mutation analysis and mutation testing has been intensively investigated for software testing [8], [9]. Also for the hardware domain approaches have been proposed which apply mutation analysis to the standard HDLs, see e.g. [10], [11]. The focus of [10] is on qualification of the stimuli and then improving the

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, and University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

validation data. The approach of [11] generates high coverage input vectors for RTL designs by measuring branch coverage controlled via mutated guards during symbolic simulation.

Mutation analysis has also been considered for system level design, in particular for SystemC TLM. Dedicated fault models used for mutation analysis have been proposed in [12], [13]. Automatic fault localization employing mutations has been presented in [14]. The work in [15] addressed the concurrency-oriented verification of SystemC designs based on mutation analysis. In [16] some mutation operators targeting concurrency constructs and synchronization in SystemC are proposed. In [17] mutation operators for IP-XACT electronic component descriptions have been introduced.

In [18] functional qualification for SystemC TLM models has been introduced to measure the quality of functional verification. However, it does not target a SDV setting. Several methods have been developed for automatically generating simulation data, for a comprehensive overview also addressing software see [19]. The approach presented in [20] considers the problem of automatic simulation data generation targeting HDL mutation faults. It defines a cost function for directing search heuristics on the test input space. For doing this the authors employ a CDFG structure which allows to see the fault propagation progress.

Closest to our approach is [21]. This paper proposed a new metric for functional testbench qualification which targets the functional qualification aspects of propagation and detection. For this task the paper also relates coverage results with reactions from checkers. However, all these works qualify (and improve) TLM testbenches which significantly differs from software-driven verification for IP integration on a VP.

III. SW Test Qualification Methodology

In this section we present the proposed methodology which qualifies software tests developed for verification of IP integration in VPs with the help of our guidance mechanism. At first, the setting when verifying IP integration is described. Next, the core of the proposed methodology is introduced, i.e. the consistency analysis of coverage measurement and software test result wrt. a mutation. Then, the overall methodology is presented. Finally, easy-to-grasp examples are provided to demonstrate the different steps of the methodology.

A. Setting of IP Integration Verification

In a software-driven verification environment with the task of verifying the integration of new IP, the test creator typically writes a sequence of tests which form the test suite. These tests interact step-by-step with the IP block and they are self-checking, i.e. the results of interactions with the IP are checked within each test e.g. by using C assertions. Ideally, the test suite should examine the IP thoroughly, otherwise integration issues could be missed.

As a prerequisite for our methodology, we assume that the tests already achieve a high statement coverage of the IP block. We believe this assumption is fair due to the following reasons. In practice very often statement coverage of the IP block is measured to ensure that each statement has been exercised, at least by one test. High statement coverage is a positive indicator of the quality of the tests.

It can be achieved quickly by taking the IP block without any mutation and writing a test suite sufficient to trigger higher number of statements and branches. By not focusing on any particular area rather going throughout the IP features briefly can prove helpful in maintaining high testing productivity. The strategy is not to have 100% coverage initially, but to have maximum coverage with minimum efforts. If the coverage is low initially, more tests should be added by the test suite creator, either manually or by employing an automated test generator (which is out of scope of this paper).

However, statement coverage (and also stronger code coverage metrics) have severe limitations regarding whether the desired behavior has really been checked. Furthermore, there is typically a point of diminishing returns, i.e. after a high coverage, for example 90%, is achieved, it is very difficult to increase it further. When that happens, the effort should be shifted to a more sophisticated (but still lightweight) qualification methodology. We propose such a methodology for software test qualification. But before we present the overall methodology, we introduce the core of our methodology – consistency analysis – in the following section.

B. Consistency Analysis

Let us just for a moment assume a single mutation is considered only. Then, the consistency analysis includes the following 4 main steps:

1) Generate coverage report for original IP running current test suite
2) Mutate IP using a fault model
3) Generate coverage report for mutated IP
4) Analyze consistency of coverage results and software test result

In the subsequent sections, we detail the major aspects of each step. Furthermore, we describe the relation to the main qualification tasks (activate, propagate, detect).

1) Fault Model: Mutation of IP Block: When mutating the new integrated TLM IP block, mutations are only performed in its SystemC/C++ code that has been marked as covered in the code coverage report, i.e. mutations will never be done in dead code (such mutations cannot be activated, hence their simulation is a waste of time). Essentially, this gives us the *activation* of the mutation since we know based on the coverage that the mutated statement is reachable by at least one test. At this point, the importance of the prerequisite for high code coverage can be emphasized. Because otherwise, mutations could only be applied to a small portion of code limiting its effectiveness significantly.

Since we are "looking" from the software test perspective, mutations that affect the functional behavior of the IP block are the most interesting. Mutations that affects the TLM commutation, for example, modifying register addresses or holding off responses, are for the most part detected by simple checks that are present in SW tests (e.g. write some value to a register, then check if a read from the register returns the same value). Furthermore, restricting mutation operators to a small

TABLE I
CONSISTENCY ANALYSIS RESULTS FOR A MUTATION

Cat.	Coverage (propagate)	Result of SW test (detect)	Consistent	Interpretation
C1	Fluctuate	Fail	yes	Adequate test
C2	Fluctuate	Pass	no	Weak detection by SW test
C3	Stable	Fail	no	Propagation path missing
C4	Stable	Pass	yes	Propagation/detection problem

TABLE II
SUMMARY OF SYSTEMC-SPECIFIC MUTATION OPERATORS

Operator	Original	Mutant
Modify	wait (200)	wait (200/2)
Remove	wait (200)	–
Replace	wait (200)	wait ()
Exchange	trywait ()	wait ()
	wait (200)	notify ()
	wait (event1)	wait (event2)

number of really relevant classes reduces the overall mutation effort.

Therefore, as a fault model we target common modeling mistakes in the functionality of a SystemC TLM IP. These include both the sequential and concurrent aspects. For sequential modeling faults, we adopt the comprehensive set of mutation operators as proposed in [22], where 77 C/C++ mutation operators are explained. The mutation operators are categorized in four domains: statement mutations, operator mutations, variable mutations and constant mutations. They are primarily based on the `competent programmer` hypothesis, i.e. faults are syntactically small and only few keystrokes away from original program. Since the IP has already passed the initial SW test suite with a high statement coverage, this hypothesis is also plausible in our setting. Furthermore, our set of mutation operators is extended by SystemC-specific mutation operators as proposed in [16]. These operators target TLM communication and synchronization with a particular focus on concurrency constructs. They are summarized in Table II (for the details we refer to reader to [16]).

2) Coverage of Mutated IP: Measuring code coverage of the mutated IP is straight-forward. Note that code coverage allows to observe the *propagation*. This is similar to but simpler than CFG/DFG-based propagation detection, which requires more complex source code analyses. In a perfect setting of course, a propagation monitor would be used which checks at the boundary that the mutation leads to a difference. However, the definition of boundary is not obvious. Also, for such a propagation monitor, detailed knowledge of the IP would be required, e.g. to specify corresponding SystemVerilog Assertions (SVA) properties. Furthermore, the "natural" boundaries for the monitor might be only observable at the IP level but not from the perspective of SDV, hence additional effort might be required to lift these to the software level. Before such effort becomes necessary, we propose to consider code coverage as a lightweight alternative for observing propagation.

3) Consistency Analysis: Comparison of Coverage Result and Software Test Result: In this section, first the principles of the consistency analysis are introduced. Then, it is shown how to measure the quality of the software tests in form of a consistency score.

Different results are possible for an injected mutation when analyzing the consistency of the coverage result and the software test result. The possible results are summarized in Table I. Column *Category* assigns a number to each of the four possible categories. The second column *Coverage* lists whether the coverage of the original IP block when running the test suite (Step 1 of overall consistency analysis as described in Section III-B) in comparison with the coverage results (Step 3) changes or not. In case of a difference this is labeled as *fluctuate*, if it remains unchanged it is labeled as *stable*. Column *SW Test Results* shows whether the execution of the software test suite resulted in *fail* or *pass*. The next column *Consistent* defines whether the comparison outcome of the coverage result and the software result is consistent or not. In the last column *Interpretation* - a short intuitive explanation is given. A more detailed explanation is provided in the following:

C1: If the coverage fluctuates and the SW test fails, their behavior is consistent since the propagation has been recognized by code coverage and due to the mutation at least one test fails as expected. As a consequence, for the current mutation the tests are adequate.

C2: If the coverage fluctuates and the SW test passes, the situation is inconsistent. The propagation is recognized (manifesting in change of coverage) but the software tests unexpectedly pass. Hence, the detection is weak, meaning that a test should be added to improve the test suite.

C3: If the coverage is stable and the SW test fails, again the situation is inconsistent. The reason for inconsistency is that code coverage could not recognize the propagation path. This inconsistency is not considered harmful because the mutant still gets killed. Hence, a C3 mutant does not require an action on its own. Instead, C4 category is consulted.

C4: If the coverage is stable and the SW test passes, the situation is consistent. However, the software tests should not pass when performing a mutation. Different reasons are possible for this scenario, so we have to deal with both the propagation and detection problem. We know that the propagation is problematic, specially if C3 mutants are also present. A potential solution is the use of a stronger coverage metric, for instance branch coverage.

The quality of SW test suite after consistency analysis can be measured with the help of a *consistency score* in a similar manner to the established *mutation score* or *mutation adequacy* [23], [24] as follows:

$$CS = \frac{\#C1}{\#C1 \; + \; \#C2 \; + \#C3 \; + \; \#C4} \tag{1}$$

In Equation 1, Cx is the total number of mutants in category Cx (with $x \in \{1, 2, 3, 4\}$). Please note that in the numerator only C1 is used and not the sum of C1 and C3. The reason is, that only category C1 is consistent *and* has positive interpretation, i.e. the test suite is adequate.

Based on the introduced consistency analysis for a mutation we present our methodology in the next section.

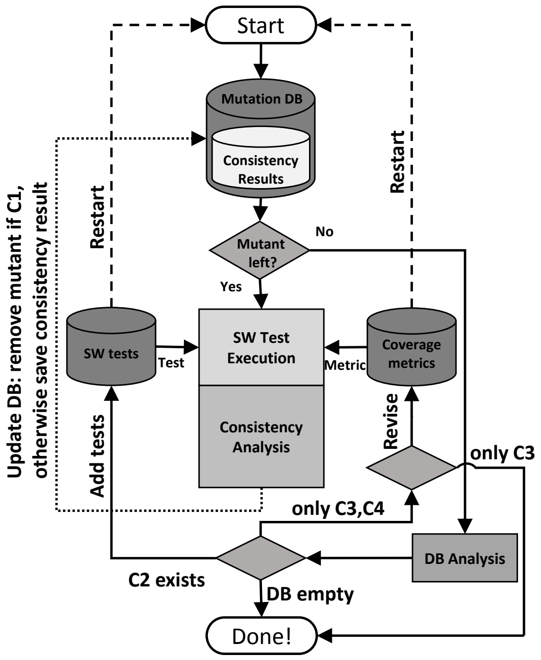


Fig. 1. SW Qualification Methodology

C. *Overall SW Qualification Methodology*

In Fig. 1 the overall methodology is depicted. It starts at the top of the figure with the mutation database containing all possible mutations for the current IP block. The overall goal of the proposed methodology is to finally bring all mutants into the category C1. Generally, several iterations might be needed to achieve this goal. In the first iteration, for each mutant from the database, the current SW tests are executed and the consistency analysis presented in Section III-B is performed. Depending on the returned category, different actions need to be taken to update the database. In case of C1, the test suite is adequate for the mutant, so this mutant is removed. Otherwise, the consistency result for the mutant is saved.

After the first iteration (i.e. no mutant left), the updated database is analyzed. If it is empty, that means every mutant has been in category C1 and has therefore been removed, we are done. If at least one mutant of category C2 can be found, new SW tests must be added to kill the mutant, then a new iteration is started. The last possible outcome of the analysis is that the database contains only mutants of category C3 and C4. If only C3 mutants are alive, the verification engineer can ignore them and consider them as killed (recall the SW test already failed for category C3). But if both categories C3 and C4 are present, then the coverage model should be revised to increase the resolution for propagation. After this a new iteration is started.

D. *Comparison to Classical Mutation Based Qualification*

In comparison to classical mutation based qualification technique, our methodology guides the verification engineer in the correct direction. By limiting the effort of writing new tests to mutants of category C2 only, our methodology ensures that a test is not being written for an equivalent mutant (trying to do so would lead to waste of time and resources). In contrast, the classical technique only gives information on the mutants killed and mutants alive. No information is provided on the nature of mutants. The guidance resulting from the categorization of each mutant through consistency analysis can save a lot of effort and time.

In the next sections we demonstrate the core of our methodology – consistency analysis – for a simple example, and also two case studies for real world examples. They also show how the guidance is provided to make the process easier.

```
void maxIP::find_max() {
 uint32_t a, b, c, d, max;
 while (1) {
   wait(e_signal);
   a = r[SRC_A] ;
   b = r[SRC_B] ;
   c = r[SRC_C] ;
   d = r[SRC_D] ;
   if (a > b)
     if ((a > c) && (a > d) )
       max = a;
     else if ((a > c) && (a < d))
       max = d;
     else if ((a < c) && (a > d))
       max = c;
     else
       if (c > d)
         max = c;
       else
       max = d;
   else
     if ((b > c) && (b > d))
       max = b;
     else if ((b > c) && (b < d))
       max = d;
     else if ((b < c) && (b > d))
       max = c;
     else
     if (c > d)
       max = c;
     else
       max = d;
   r[MAX_VALUE] = max;
 } // end while
} // end find_max()
```

Fig. 2. Code excerpt of IP block maxIP SC_THREAD *find_max*

IV. Consistency Demonstration Example

We use a compact example to demonstrate the ingredients of the methodology.

A. *IP Block Basic Information*

A code excerpt of a complete SystemC TLM model of the IP block *maxIP* is shown in Fig. 2. The main functionality of this IP is implemented in the SC_THREAD *find_max*. It receives four inputs from its registers (*r[SRC_A], r[SRC_B], r[SRC_C], r[SRC_D]*), finds the maximum value among them, and writes this value back into the register *r[MAX_VALUE]*.

B. *SW Tests*

Listing 1 shows the SW tests for *maxIP*. As can be seen, in each test, four integer values are written into the memory addresses of the registers of *maxIP* (see e.g. Line 11 – Line 14) and then the computed maximum value is read back and compared to the expected value. If the maximum value is correct, the test generates a *success* message (e.g. Line 15 with argument 1 since it is the first test), otherwise a *fail* message is generated (e.g. Line 16 again with argument 1 indicating the test number).

C. *Coverage of SW Tests*

The coverage results for the SW tests of Listing 1 are depicted in Fig. 3 and can be interpreted using Table III. Essentially, each line of Fig. 3 consists of three parts: `Branch Coverage : Line Coverage : src expression`. Lets start with line coverage: As can be seen on the left side of Table III, red color is used to show that the expression has not been hit during execution, and otherwise

```
struct maxIP {
  volatile unsigned int SRC_A; /* 0x00 */
  volatile unsigned int SRC_B; /* 0x04 */
  volatile unsigned int SRC_C; /* 0x08 */
  volatile unsigned int SRC_D; /* 0x0C */
  volatile unsigned int MAX_VALUE;  /* 0x10 */
};
void SW_Test(int addr) {
  struct maxIP *ipBlock = (struct maxIP *) addr;
  /*  Test 1  */
  ipBlock->SRC_A = 5;
  ipBlock->SRC_B = 1;
  ipBlock->SRC_C = 8;
  ipBlock->SRC_D = 4;
  if (ipBlock->MAX_VALUE == 8)  success(1);
  else  fail(1);
  /*  Test 2  */
  ipBlock->SRC_A = 5;
  ipBlock->SRC_B = 1;
  ipBlock->SRC_C = 3;
  ipBlock->SRC_D = 6;
  if (ipBlock->MAX_VALUE == 6)  success(2);
  else  fail(2);
}
```

Listing 1. Consistency example: SW Test

TABLE III
CONSISTENCY EXAMPLE: COVERAGE LEGEND

Branch Coverage			
Line Coverage			
Color	**Line**	**Symbol**	**Description**
Blue	Hit	+	Taken
Red	Not hit	-	Not Taken
		#	Not executed

the color is blue. Moreover, the number of executions is shown as second part in each line of Fig. 3.

For branch coverage, only the source code lines with conditions are relevant. Each condition of a branch in Fig. 3 has a corresponding `[T F]` pair (meaning `[True False]`) shown on the left of each code line. Note that `T` and `F` are replaced with the symbols shown in Table III in column symbol. As can be seen for instance in Line 186 of Fig. 3 it has only one pair, and Line 187 has two pairs, respectively. When the SW tests evaluate a condition with true, the `T` is replaced by a `+` with blue color, and the `F` becomes `-` in color red (so not hit). Similarly, if another SW test evaluates the same condition with a false, the `F` becomes a `+` in blue (cf. Line 187 of Fig. 3 after execution of Test 1 ($5 > 8$ gave false) and Test 2 ($5 > 3$ gave true). This means that the expression has been tested by SW tests for both true and false cases. If the condition is not evaluated in any execution, it is marked with # (see e.g. `c>d` of Line 194 of Fig. 3).

D. Demonstration of Consistency Analysis

When Test 1 is executed, the original IP finds the maximum value as 8, so this single test passes. In the following we show concrete examples for category C2 and C4.

a) C2 Example: Lets now assume, we mutate the IP in Line 191 in Fig. 4 by negating the complete if-condition. Then, running Test 1 this if-condition now results in false instead of true as before and the execution jumps to the next *if*-statement (Line 194) to find the correct answer. Hence, the mutation causes the change of coverage visible at the statement in Line 194 in Fig. 4. Since the maximum of the inputs 5,1,8,4 is still 8 and will be also found by the current mutated IP, the test passes. Hence, we have fluctuating coverage, but passing SW tests, so this example falls in category C2. To solve the problem, the methodology guides the engineer to add a new test to the test suite (Listing 1), e.g. Test 3 as depicted in Listing 2. With this new test suite the error is detected as the mutant calculates the wrong maximum value of 4 for the inputs 2,1,4,5. Therefore, the test fails (Line 7 in Listing 2) and so the considered mutation finally falls in category C1. Evaluating the same IP using classical mutation based qualification technique, an alive mutant will have to be chosen out of many. Hence, our method has reduced the search space.

b) C4 Example: For this example we mutate the original *maxIP* block in Line 189 of Fig. 5 by replacing `&&` with `||` operator. Test 2 with inputs 5,1,3,6 from Listing 1 results in a correct maximum value of 6, with stable line coverage. Hence, this example falls in category C4. Due to the presence

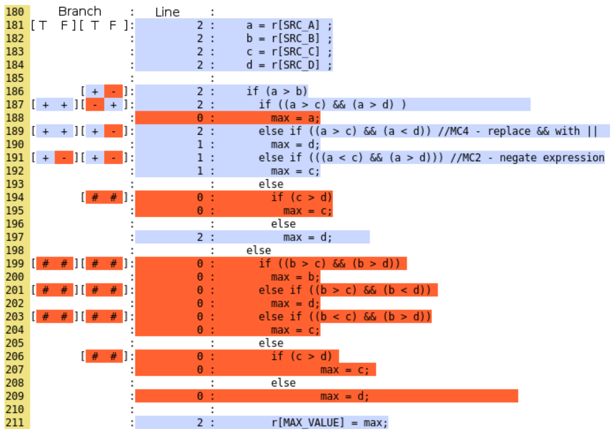

```
180      Branch     :   Line     :
181 [T   F][ T  F ]:          2 :     a = r[SRC_A] ;
182                :          2 :     b = r[SRC_B] ;
183                :          2 :     c = r[SRC_C] ;
184                :          2 :     d = r[SRC_D] ;
185                :            :
186        [ +  - ]:          2 :     if (a > b)
187 [ +  + ][ -  + ]:          2 :       if ((a > c) && (a > d) )
188                :          0 :         max = a;
189 [ +  + ][ +  - ]:          2 :       else if ((a > c) && (a < d)) //MC4 - replace && with ||
190                :          1 :         max = d;
191 [ +  - ][ +  - ]:          1 :       else if (((a < c) && (a > d))) //MC2 - negate expression
192                :          1 :         max = c;
193                :            :       else
194        [ #  # ]:          0 :         if (c > d)
195                :          0 :           max = c;
196                :            :         else
197                :          2 :           max = d;
198                :            :     else
199 [ #  # ][ #  # ]:          0 :       if ((b > c) && (b > d))
200                :          0 :         max = b;
201 [ #  # ][ #  # ]:          0 :       else if ((b > c) && (b < d))
202                :          0 :         max = d;
203 [ #  # ][ #  # ]:          0 :       else if ((b < c) && (b > d))
204                :          0 :         max = c;
205                :            :       else
206        [ #  # ]:          0 :         if (c > d)
207                :          0 :                 max = c;
208                :            :         else
209                :          0 :                 max = d;
210                :            :
211                :          2 :         r[MAX_VALUE] = max;
```

Fig. 3. SC_THREAD find_max w orig. coverage results for Test 1 and Test 2

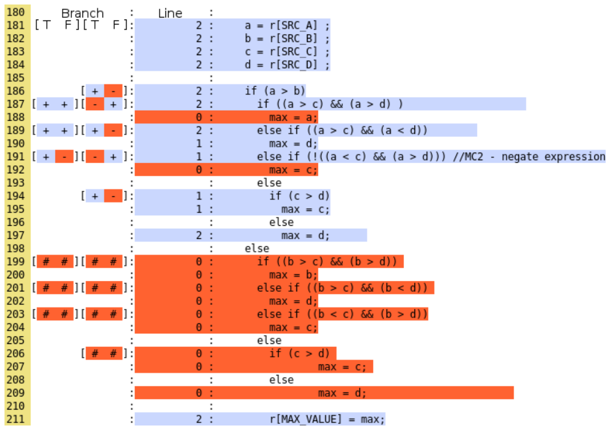

```
180      Branch     :   Line     :
181 [T   F][T   F]:           2 :     a = r[SRC_A] ;
182                :          2 :     b = r[SRC_B] ;
183                :          2 :     c = r[SRC_C] ;
184                :          2 :     d = r[SRC_D] ;
185                :            :
186        [ +  - ]:          2 :     if (a > b)
187 [ +  + ][ -  + ]:          2 :       if ((a > c) && (a > d) )
188                :          0 :         max = a;
189 [ +  + ][ +  - ]:          2 :       else if ((a > c) && (a < d))
190                :          1 :         max = d;
191 [ +  - ][ -  + ]:          1 :       else if (!((a < c) && (a > d))) //MC2 - negate expression
192                :          0 :         max = c;
193                :            :       else
194        [ +  - ]:          1 :         if (c > d)
195                :          1 :           max = c;
196                :            :         else
197                :          2 :           max = d;
198                :            :     else
199 [ #  # ][ #  # ]:          0 :       if ((b > c) && (b > d))
200                :          0 :         max = b;
201 [ #  # ][ #  # ]:          0 :       else if ((b > c) && (b < d))
202                :          0 :         max = d;
203 [ #  # ][ #  # ]:          0 :       else if ((b < c) && (b > d))
204                :          0 :         max = c;
205                :            :       else
206        [ #  # ]:          0 :         if (c > d)
207                :          0 :                 max = c;
208                :            :         else
209                :          0 :                 max = d;
210                :            :
211                :          2 :         r[MAX_VALUE] = max;
```

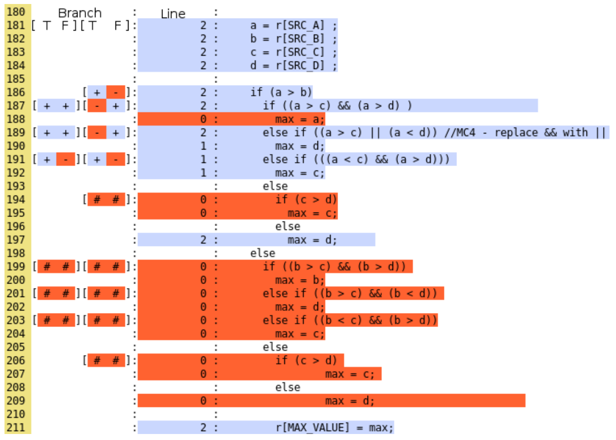
Fig. 4. Consistency ex.: coverage results for C2 example

```
1 /*  Test 3  */
2 ipBlock->SRC_A = 2;
3 ipBlock->SRC_B = 1;
4 ipBlock->SRC_C = 4;
5 ipBlock->SRC_D = 5;
6 if (ipBlock->MAX_VALUE == 5)
       success(3);
7 else  fail(3);
```

Listing 2. Consistency ex.: SW Test to kill mutant MC2

```
/*  Test 4  */
ipBlock->SRC_A = 5;
ipBlock->SRC_B = 1;
ipBlock->SRC_C = 8;
ipBlock->SRC_D = 7;
if (ipBlock->MAX_VALUE == 8)
     success(4);
else  fail(4);
```

Listing 3. Consistency ex.: SW Test to kill mutant MC4



```
180         Branch      :    Line    :
181 [ T  F ][T   F ]:          2 :      a = r[SRC_A] ;
182                 :          2 :      b = r[SRC_B] ;
183                 :          2 :      c = r[SRC_C] ;
184                 :          2 :      d = r[SRC_D] ;
185                 :            :
186          [ +  - ]:          2 :      if (a > b)
187 [ +  + ][ -  + ]:          2 :        if ((a > c) && (a > d) )
188                 :          0 :          max = a;
189 [ +  + ][ -  + ]:          2 :        else if ((a > c) || (a < d)) //MC4 - replace && with ||
190                 :          1 :          max = d;
191 [ +  - ][ +  - ]:          1 :        else if (((a < c) && (a > d)))
192                 :          1 :          max = c;
193                 :            :        else
194          [ #  # ]:          0 :          if (c > d)
195                 :          0 :            max = c;
196                 :            :          else
197                 :          2 :            max = d;
198                 :            :      else
199 [ #  # ][ #  # ]:          0 :        if ((b > c) && (b > d))
200                 :          0 :          max = b;
201 [ #  # ][ #  # ]:          0 :        else if ((b > c) && (b < d))
202                 :          0 :          max = d;
203 [ #  # ][ #  # ]:          0 :        else if ((b < c) && (b > d))
204                 :          0 :          max = c;
205                 :            :        else
206          [ #  # ]:          0 :          if (c > d)
207                 :          0 :                max = c;
208                 :            :          else
209                 :          0 :                max = d;
210                 :            :
211                 :          2 :          r[MAX_VALUE] = max;
```

Fig. 5. Consistency ex.: coverage results for C4 example

of C3 mutants (which are not shown here), we know the propagation is problematic. Changing the coverage metric to branch coverage helps solving the problem. The branch coverage in Line 189 in Fig. 5 ([+ +][- +]) is now different from branch coverage in Line 189 in Fig. 3 ([+ +][+ -]). The C4 mutant therefore now becomes C2. So we have to add another test to the test suite (Listing 1), e.g. Test 4 as depicted in Listing 3. With this new test suite the error is detected as the mutant calculates the wrong maximum value of 4 for the inputs 5,1,8,7. Therefore, the test fails (Line 7 in Listing 3) and so the considered mutation finally falls in category C1.

In the next section the experimental results for our methodology for a real-world VP are given.

V. Experimental Results

This section presents the evaluation of our SW test qualification methodology in a software-driven verification environment. We consider the LEON3-based VP SoCRocket [25] which has been modeled in SystemC TLM. We look at two IP integration scenarios, i.e. the integration of an *Interrupt Controller for Multiple Processors* (IRQMP), and the integration of a *General Purpose Timer* (GPTimer). In the following, we first describe how we automatically generate the mutants and the coverage models used in the case studies. Then, for each IP block, the basics are described before the demonstration of the methodology as well as qualification results are presented.

A. Mutant Generation

The mutants were generated by an in-house tool called *Typhon*. It is a standalone command line tool which generates the mutants from the input SystemC/C++ source files. The underlying infrastructure for Typhon is the LibTooling library of Clang. Clang generates the AST (Abstract Syntax Tree) for the input source files, and *Typhon* takes advantage of that AST to compile new mutants by traversing different required entities. The type of mutants generated can be chosen by the input arguments to the tool. Typhon supports all mutation operators described in Section III-B1.

B. Coverage Models

In addition to *Statement Coverage* (SC) and *Branch Coverage* (BC), we also use their strengthened variants termed as *Differential Statement Coverage (DSC)*, and *Differential Branch Coverage (DBC)*, respectively, in the following. They signify the disturbance created by the mutant in terms of how many times the statement or branch was covered. The disturbance refers to the increase or decrease in coverage counters of statements and branches. It is calculated by taking the difference of coverage counters of original model and mutant reported by the coverage tool LCOV. DSC and DBC are very useful for the elimination of mutants from categories C3 and C4 as these mutants often show stable statement and branch coverage.

C. IRQMP

1) Basics: The first considered IP – IRQMP – processes incoming interrupts from different devices and processors based on priority. It supports 32 interrupt lines numbered from 0 to 31, where line 0 is reserved. Lines 1 to 15 are used for regular interrupts whereas the remaining lines 16 to 31 for extended interrupts. The IRQMP model has a register file, I/O wires and APB slave interface. The register file contains 32-bit processor-specific and configuration registers. When an interrupt is signaled, the corresponding bit is set in the register. This functionality is implemented using the SystemC thread *launch_irq* and callback functions, which are specified for register access (read/write).

The IRQMP interacts with connected processors by sending an interrupt request (*irq_req*) or receiving 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 `broadcast register`, it is simply set in the `pending register`. In this scenario, IRQMP expects an acknowledge from any processor.

2) SW Test Qualification: The initial test suite shipped with the IRQMP IP consists of 60 tests. This test suite has 63% statement coverage of the IP. We add 45 tests to achieve high statement coverage (92%) as required by the methodology. The tool Typhon generates in total 244 mutants.

The results of applying our qualification methodology are shown in Table IV where we report the first 13 iterations. The first row gives the index of the iteration. The second row states the operation done during those iterations, e.g., new tests are added to improve the test suite, or the coverage metric is changed. The third row shows the metric used in the consistency analysis. The last row of Table IV shows the consistency score of the SW test suite calculated by using Equation 1 for each iteration.

a) Handling C2 mutants: The first iteration shows a low consistency score of 0.352. Due to the 14 mutants in category C2, it is clear that more tests need to be added to the suite. In the following, we describe one concrete mutant from category

TABLE IV
IRQMP SW TEST QUALIFICATION RESULTS

Iteration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Operation		**Addition of tests**									**Change of metric**		
Metric	**SC**										**BC**	**DSC**	**DBC**
Category C1	86	89	91	92	93	94	95	97	98	100	113	126	127
Category C2	14	*11*	*9*	*8*	*7*	*6*	*5*	*3*	*2*	*0*	*43*	*98*	*99*
Category C3	27	27	27	27	27	27	27	27	27	27	*14*	*1*	*0*
Category C4	117	117	117	117	117	117	117	117	117	117	*74*	*19*	*18*
Tests	105	106	107	108	109	110	111	112	113	114	114	114	114
Consistency score	0.352	0.365	0.373	0.377	0.381	0.385	0.389	0.398	0.402	0.410	0.463	0.516	0.520

SC: Statement coverage **BC:** Branch coverage
DSC: Differential SC **DBC:** Differential BC

```
1  void Irqmp::incoming_irq(const std::pair<uint32_t, bool> &irq, const
        sc_time &time) {
2   bool t = true;
3   if (!irq.second) {
4   // Return if the value turned to false. Interrupts will not be unset
5   // this way. So we cann simply ignore a false value.
6     return;
7   }
8   for(int32_t line = 0 ; line<32; line++) {
9     if((1 << line) & irq.first) {
10    // Performance counter increase
11      m_irq_counter[line] = m_irq_counter[line] + 1;
12      v::debug << name() << "Interrupt line " << line << " triggered"
             << v::endl;
13      if (!r[BROADCAST].bit_get(line)) {
14        r[IR_PENDING].bit_set(line, t);
15    }
16    if (r[BROADCAST].bit_get(line) && (line < 16)) { // Mutation:
           replace && with ||
17    // set force registers for broadcasted interrupts
18      for (int32_t cpu = 0; cpu < g_ncpu; cpu++) {
19        r[PROC_IR_FORCE(cpu)].bit_set(line, t);
20      forcereg[cpu] |= (t << line);
21      }
22    }
23    }
24  }
25  // Pending and force regs are set now.
26  // To call an explicit launch_irq signal is set here
27  e_signal.notify(2 * clock_cycle);
28  }
```

Listing 4. C2 IRQMP Mutation Example

C2 to demonstrate its fix. An excerpt of the IRQMP model is shown in Listing 4. It shows the implementation of *incoming_irq* which handles the incoming interrupts from different devices. When such an interrupt arrives, the implementation checks the corresponding bit in `broadcast register` and handles the interrupt accordingly as described in the previous section.

When the mutation is performed (see comment in Line 16), the routine registers the interrupt in the `pending register` (Line 14) as well as in `force register` (Line 19). Thus, at least acknowledge from at least one of the connected processors is expected. The SW tests, however, only check whether the interrupt was generated and handled. This is clearly a weakness of the existing tests. After the addition of a test, that checks for `broadcast register` and `pending register`, leads to the SW test failure for this mutant. The mutant can thus according to our methodology be moved from C2 to C1. Moreover, the added test kills two more C2 mutants, resulting in 11 mutants in category C2 as can be seen in Iteration 2 of Table IV. Similarly, more C2 mutants are killed and moved to C1 by adding more tests from Iteration 3 to Iteration 10, where all C2 mutants are eliminated.

The importance of our methodology can be seen by looking at Table IV. Classical analysis has a search space of 131 mutants (#C2 + #C4), whereas, our methodology has a limited search space of only 14 mutants. Hence, the guidance can save a lot of time by focusing the efforts in the right direction.

b) Handling C3 and C4 mutants: Now following the proposed methodology to eliminate the rest of mutants, we need to see both categories C3 and C4. Presence of only C3 mutants requires no action, but presence of C4 mutants in conjunction indicate a problematic propagation. Hence, it is time to strengthen the coverage metrics used by the consistency analysis to eliminate C4 mutants in particular. We first strengthen the coverage metric to *BC* and apply consistency analysis. The results are shown in Table IV as Iteration 11. As can be seen, the strengthening of coverage metric improves propagation, and thus, 43 mutants from C4 are moved into category C2, and additionally, 13 C3 mutations into category C1. In the next iterations, we deviated a bit from the methodology to demonstrate the effect of strengthening the coverage further. We changed the metric to *DSC* and then to *DBC*. The number of mutations in C4 and C3 category went down significantly as expected. The newly identified 99 C2 mutants now require more tests to be added.

Clearly, following the proposed methodology, weaknesses in the test suite can be identified and the test creator gets useful feedback on what to do next to improve the test suite. The improvement is quantified by the consistency score, as a significant jump from 0.352 to 0.520 after 13 iterations can be observed.

D. GPTimer

1) Basics: The second considered IP – GPTimer – implements down-counting timer(s) and generates an interrupt if zero is reached. The IP consists of 7 configurable timers which use ticks from the prescaler unit. The prescaler unit uses system clock as reference clock to decrement its value. The timer can also be configured to be used as a watchdog to prevent any malfunction. All the timers consist of a value register and a reload value register. When zero is reached or reset signal is initiated, the value register is loaded with the value in reload value register, otherwise it is decremented by one in each cycle. The timers are not limited to only 2^{32} value, but can also be executed for a longer duration by chaining them together. This way, the timers decrement when a zero is reached in the previous timer.

TABLE V
GPTIMER SW TEST QUALIFICATION RESULTS

Iteration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operation		**Addition of tests**					**Change of metric**			
Metric	**SC**						**BC**	**DSC**	**DBC**	**PC**
Category C1	89	91	92	93	94	95	131	139	140	182
Category C2	6	*4*	*3*	*2*	*1*	*0*	*0*	*0*	*2*	*70*
Category C3	112	112	112	112	112	112	*76*	*68*	*67*	*25*
Category C4	101	101	101	101	101	101	101	101	*99*	*31*
Tests	14	15	16	17	18	19	19	19	19	19
Consistency Score	0.289	0.295	0.299	0.302	0.305	0.308	0.425	0.451	0.455	0.590

SC:	Statement coverage	**BC:**	Branch coverage	**PC:**	Path coverage
DSC:	Differential SC	**DBC:**	Differential BC		

2) SW Test Qualification: The test suite shipped with the IP and is used as basis for our SW test qualification methodology consists of 14 tests initially.

For this IP, we report the results of the first 10 iterations as shown in Table V. The terminologies used in Table V are the same as used in Table IV. The first iteration shows the distribution of mutants after *SC* analysis. Out of 308 mutants, 89 fall in category C1 initially and do not require any additional processing, whereas 219 mutants require additional work. Again, as expected, from iteration 2 to 6, the number of C2 mutants decrease as new tests are added to the suite. The total number of tests increase from 14 to 19. In Iteration 6, all mutants in C2 have been killed.

Therefore, following the methodology, we strengthen the coverage metric from *SC* to *BC* to try to eliminate category C4 and C3 mutants. The propagation is not observed for C4 mutants, but it can be observed for C3 mutants, as this moves 36 mutants from category C3 to C1. Changing the metric to *DBC* successfully detected two C2 mutants at the end of Iteration 9, where the consistency score has significantly increased from 0.289 to 0.455. There are still a big number of mutants in category C4 and C3 (99 and 67, respectively). We decided to change the metric to path coverage to eliminate the mutants. As can be seen, the numbers reduced significantly from 99 to 31 (for C4) and from 67 to 25 (for C3), respectively.

In summary, the test creator can use our proposed methodology to strengthen the SW test suite as he can now identify the weaknesses clearly. The strength of SW test suite is shown in last row of Table V as consistency score. It can be observed that the consistency score increased from 0.289 to 0.590 after 10 iterations.

VI. LIMITATIONS OF METHODOLOGY

Since our SW test qualification methodology is based on mutation analysis, it inherits the same limitations. Mutation analysis is computationally expensive as the program has to be executed several times. Various approaches are available to reduce this cost like selective mutation [26], weak mutation [27], and separate compilation [28] to name a few. Our methodology is also dependent on this factor as new tests are added and coverage metrics are changed during the iterations to kill the mutants.

It can be partially solved by configuring the program to be instrumented with all the known coverage metrics, and later only doing the comparison to generate results.

VII. CONCLUSION

In this paper we proposed a methodology for SW test qualification of IP integration in a software-driven verification flow. Our methodology is based on mutation analysis an we have shown how to define the main tasks of functional qualification (activate, propagate, detect) in the context of SW test based IP verification. Furthermore, our qualification methodology also relates the coverage results and the SW test results wrt. the original and mutated IP block. This allows to improve the tests since the user gets information whether for instance a new test is required or the coverage model should be strengthened. We have demonstrated the applicability in a real world VP showing the integration of two IP blocks.

REFERENCES

[1] *IEEE Standard SystemC LRM*, IEEE Std. 1666, 2011.
[2] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[3] R. Leupers, F. Schirrmeister, G. Martin, T. Kogel, R. Plyaskin, A. Herkersdorf, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.
[4] 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.
[5] R. G. Hamlet, "Testing programs with the aid of a compiler," *TSE*, no. 4, pp. 279–290, 1977.
[6] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *MUTATION*, 2007, pp. 203–209.
[7] Synopsys, "Certitude," 2015, https://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/certitude-ds.aspx.
[8] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *TSE*, vol. 32, no. 9, pp. 733–752, Sep. 2006.
[9] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.
[10] Y. Serrestou, V. Beroulle, and C. Robach, "Functional verification of RTL designs driven by mutation testing metrics," in *DSD*, 2007, pp. 222–227.
[11] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *DATE*, 2011, pp. 1596–1601.
[12] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *DATE*, 2008, pp. 396–401.
[13] ——, "On the mutation analysis of SystemC TLM-2.0 standard," in *MTV Workshop*, 2009, pp. 32–37.
[14] H. M. Le, D. Große, and R. Drechsler, "Automatic TLM fault localization for SystemC," *TCAD*, vol. 31, no. 8, pp. 1249–1262, Aug. 2012.
[15] A. Sen, "Concurrency-oriented verification and coverage of system-level designs," *TODAES*, vol. 16, no. 4, p. 37, 2011.
[16] ——, "Mutation operators for concurrent SystemC designs," in *MTV Workshop*, 2009, pp. 27–31.
[17] T. Xie, W. Müller, and F. Letombe, "IP-XACT based system level mutation testing," in *HLDVT*, 2011, pp. 65–71.
[18] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, "Functional qualification of TLM verification," in *DATE*, 2009, pp. 190–195.
[19] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, 2016.
[20] T. Xie, W. Müller, and F. Letombe, "HDL-mutation based simulation data generation by propagation guided search," in *DSD*, 2011, pp. 608–615.
[21] H. Kai, P. Zhu, R. Yan, and X. Yan, "Functional testbench qualification by mutation analysis." *VLSI Design*, vol. 2015, pp. 256 474:1–256 474:9, 2015.
[22] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Purdue University, Tech. Rep., 1989.
[23] T. A. Budd, R. J. Lipton, R. DeMillo, and F. Sayward, "The design of a prototype mutation system for program testing," in *AFIPS*, 1978, pp. 623–627.
[24] B. J. Grun, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *ICSTW*, 2009, pp. 192–199.
[25] 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.
[26] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *STVR*, vol. 9, no. 4, pp. 205–232, 1999.
[27] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis*, 1988, pp. 152–158.
[28] C. Byoungju and A. P. Mathur, "High-performance mutation testing," *Journal of Systems and Software*, vol. 20, no. 2, pp. 135–152, 1993.