Generation of TLM Testbenches Using Mutation Testing

Marcelo Sousa
Dept. of Information Computing Sciences
Utrecht University, Netherlands
marceloabsousa@gmail.com

Alper Sen
Dept. of Computer Engineering
Bogazici University, Turkey
alper.sen@boun.edu.tr

ABSTRACT

Testbench development is a major component of simulation based verification, which is the de-facto verification technique used in the industry. Verification of a TLM design is not complete without a measure of the effectiveness of its testbenches. We devise a coverage driven testbench generation technique where the coverage metric uses that of a fault insertion based approach, namely mutation testing. Mutation testing is a commonly used software testing technique to measure the quality of testbenches. In mutation testing, the goal is to insert mutations (syntactic changes) into the program and check whether the impact of these mutations on the program can be detected by the available testbenches. If the testbenches cannot detect the impact of these inserted mutations then one can potentially add new testbenches to detect these changes. In this work, we automate the process of mutation testing based testbench generation exploiting the properties of concurrent TLM designs. Our framework is novel in that it uses the byte code representation of SystemC TLM models using Low Level Virtual Machine (LLVM) framework. Furthermore, these testbenches can detect concurrency related defects such as deadlocks or race conditions. We perform experiments on TLM models to demonstrate the effectiveness of our test bench generation approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Testing and Debugging**]: Testing tools

Keywords

SystemC, TLM, verification, mutation testing, LLVM

1. INTRODUCTION

The increasing complexity of System-on-Chips (SoCs) exposed limitations of standard Register Transfer Level (RTL) simulations. The main limitation is related to the low level nature of RTL originating simulations with poor performance. Furthermore, the costs of debugging and validation of RTL simulations reached levels unbearable to the industry.

Transaction Level Modeling (TLM) is an approach that overcomes the RTL scalability problem by operating at a higher level of abstraction. The basic idea of TLM is to model hardware components as modules that communicate with *transactions*. Due to its higher abstraction level, TLM allows fast simulations that can be used by designers in an early stage of the design cycle to obtain various performance results and discover potential errors of the real system.

In recent years, SystemC became a de-facto standard for simulation of SoCs using TLM designs. SystemC [1] is an IEEE standard C++ library composed of an event scheduler and constructs to represent the concurrent behavior of hardware. The SystemC TLM [1] standard enables system level verification and debugging as well as hardware/software co-design, architectural exploration, and power/performance analysis.

Automated verification of SystemC TLM designs is valuable since the verification phase can potentially consume a substantial part of the design cycle. Although formal verification of TLM designs are emerging, simulation remains the most popularly used verification technique in the industry. Nevertheless using this approach, verification of a TLM design is not complete without a measure of the effectiveness of its testbenches. Hence, it is crucial to aid hardware designers with methods that automatically generate testbenches based on a coverage criteria to guarantee that the generated testbench is effective with respect to the criteria.

Mutation testing is a commonly used software testing technique to measure the quality of testbenches. Mutation testing consists of inserting errors, known as mutations, into the program and checking if the testbenches detect the inserted mutations. If the testbenches cannot detect the impact of a set of mutations, then one can argue that new testbenches need to be added to detect these changes and improve the overall testbench quality.

In this paper, we devise an automated hybrid coverage driven testbench generation technique based on mutation testing. We base our fault model in the hypothesis that a suitable testbench for a concurrent program should reach the maximum number of synchronization operations possible. We are interested in exploring the effects of mutated synchronization operations in fault identification and resolution.

The problem of whether a test case can reach a point in a program is undecidable [34]. We apply static analysis techniques to generate constraints over the input variables from a bounded version of the original program. Our method exploits the properties of concurrent TLM designs and the deterministic semantics of the SystemC scheduler. From the generated constraints, we obtain concrete input tests using a Satisfiability Modulo Theories (SMT) solver, an efficient constraint-satisfaction problem solver [4]. Finally, we dynamically generate oracles that encapsulate the impact of the mutations based on execution trace comparison. Hence, we achieve testbenches with high ratio of detected mutants.

Our framework is novel in that it uses the byte code representation of SystemC TLM models using the Low Level Virtual Machine (LLVM) framework [22]. LLVM is a compiler framework designed for aggressive multistage optimizations over a well-defined intermediate representation. The main advantage of this approach is that we lift the lack of formal semantics of the high-level language C++ that represents one of the current problems of SystemC TLM design validation [28]. Hence, by using LLVM Intermediate Representation (IR) we leverage its formal semantics to apply formal methods such as SAT/SMT solvers or theorem proving.

We also show with a complete example of our approach that we can detect concurrency related defects such as deadlocks or lost notifies. We perform initial experiments on TLM models to demonstrate the effectiveness of our test bench generation approach.

The rest of the paper is organized as follows. Section 2 describes related work in SystemC mutation testing and test case generation using mutation analysis. Section 3 details the background on mutation testing, SystemC TLM and LLVM. Section 4 describes our mutation framework for SystemC that operates at the LLVM byte code level. Section 5 describes our testbench generation flow with details of our algorithms for input generation. Section 6 presents a dynamic approach for oracle generation and Section 7 describes our experiments. We conclude with a discussion of our current limitations to be left as future work and conclusions.

2. RELATED WORK

Mutation testing is an approach based on software testing to assess the quality of test suites [8, 31, 32]. It has been defined for programming languages such as Java [27, 7], state machines [33] and hardware description languages such as Verilog [15]. A detailed survey on applications of mutation testing can be found in [20].

Mutation testing has been widely applied in exploiting the correlation between the fault model and real faults [3]. The standard mutation testing approach is to inject mutations into the program one at a time, and check whether the test suite can identify the fault introduced. Hence, it helps to strengthen the quality of test suites by providing information related to tests cases that should complement the test suite. Mutation testing has proven more powerful than other coverage criteria such as statement, branch, and all-use dataflow [13, 39, 26].

Standard mutation frameworks implement optimization to reduce the high number of mutants generated. Prior work on mutant equivalence [36] and higher order mutation testing [19] have tackled this problem.

A mutation model for SystemC TLM 2.0 communication interfaces have been defined in [5, 6] and this has been extended to all SystemC concurrency constructs in [37]. Our work implements a mutation framework that supports the latter mutation model. We do not require any modification of the SystemC libraries, hence our mutation framework is scalable and transparent to the user.

Automated test case generation is a natural application of mutation testing. Early research on mutation testing [30] developed methods based on constraint solving to solve the input-mutant reachability problem. In [12], this work is extended with a necessity condition encoding the mutant effects on states of the program. Moreover, [10, 18] have used fault models for test generation.

μTest [14] applies a genetic algorithm to generate unit tests for object-oriented classes in Java based on mutation analysis. The framework generates oracles to increase mutation coverage by comparing execution traces of a test case on a program and its mutants. Our approach focuses on the domain of concurrent programs and uses a byte code representation that supports a wider range of programming languages.

SymBMC [34] uses SMT/bounded model checking approach using mutation analysis to generate test cases from ANSI-C programs. KLEE [9] and KLOVER [24], perform symbolic execution of sequential C and C++ programs represented with the LLVM byte code for automatic test suite generation. Recently, GKLEE [25], a tool based on KLEE, applies concolic execution to CUDA programs for test generation. The tools of the KLEE family require manual instrumentation from the user and all except GKLEE do not handle concurrent programs. Our approach is more transparent to the user and supports a better integration since we do not require any user modification on the source code.

3. BACKGROUND

In this section, we provide background on Mutation Testing, SystemC TLM and LLVM.

3.1 Background on Mutation Testing

Mutant testing is based on a fault model represented as a set of mutation operators. A mutation operator *MO* is a non-deterministic rewrite system.

Example Mutation operator *MO* for *wait(e)*:

$$\text{wait(e)} \rightarrow \text{wait(1,SC\_NS)};$$
$$\text{wait(e)} \rightarrow \text{e.notify()};$$
$$\text{wait(e)} \rightarrow \epsilon;$$

The example above is composed of three rewrite rules. The left hand-side of a rule represents the operation to be mutated, and by definition in a mutation operator, the left hand-side of all rules is the same. The first rule mutates the argument of *wait* to a value of a different type. The second rule applies the dual relation between *wait* and *notify* and the third rule removes the operation.

A mutant P' of a program P is the program generated by one reduction of the mutation operator *MO*. In the previous example, *MO* can potentially generate three mutants. Following the definition of *mutant*, we can define *killed* and *live mutant*.

Definition 1. *A mutant P' of a program P is said to be* killed *by a test t if and only if there are observable differences between $P'(t)$ and $P(t)$.*

Algorithm 1 Mutation Coverage Algorithm

```
Input: a program $P$, a testbench $T$.
Output: mutation coverage.
1: generate a metamutant $MP$ from $P$;
2: for each mutation operator $MO$ in $MP$ do
3:    generate a mutant $P'$ from $MO$ and $P$;
4:    for each test $t \in T$ do
5:       execute $P'$ with $t$;
6:       check if $P'$ is killed by $t$;
7:    end for
8: end for
9: compute mutation coverage;
```

DEFINITION 2. *A mutant P' of a program P is said to be* live *if and only if there is no test t in the testbench T that kills the mutant.*

Using mutation testing we can compute a new coverage metric, *mutation coverage*, that is useful to assess the quality of a testbench.

DEFINITION 3. *The* mutation coverage *of a testbench T for a program P is the ratio of the number of killed mutants to the number of all mutants.*

Mutation coverage is a useful metric since it identifies errors, encoded as mutants, that are not tested in the testbench T. Therefore, mutation coverage of T can be improved by extending the testbench with new test cases that kill live mutants.

Algorithm 1 describes a general mutation coverage algorithm. In line 1, we generate a metamutant MP by inserting mutation operators into P. A metamutant captures all possible mutations and supports a mechanism that allows dynamic activation of one mutation operator. This strategy is popular for mutant testing since it is more efficient than generating a compiled version of the program for each mutation. In line 3, we generate a mutant from the original program using the mutation operator as described above. Then, in line 4, we iterate over the test suite and execute the mutant with every test (line 5) and check if the mutant is killed by the test using Definition 1 (line 6). Finally, we compute the mutation coverage rate using Definition 3. A mutation coverage rate of 1 may no be possible due to equivalent mutants.

The complexity of the algorithm can reach $M \times T \times C_t$, where M is the number of inserted mutations, T is the size of the test suite and C_t is the execution cost of one test. Since the number of mutations can be high, mutation analysis suffers from a scalability problem so mutation coverage can be an expensive metric to compute.

3.2 Background on SystemC TLM

SystemC [1] models hardware components as modules. Modules are composed of processes which encode the concurrent behavior of the component, and ports that are used for interprocess communication. Although processes run concurrently, their execution is sequential. The SystemC scheduler is non-preemptive; hence, a process has to voluntarily yield control for another process to be executed. The scheduler chooses non-deterministically exactly one process at a time to be executed. It implements an event-based simulator similar to VHDL by handling event notifications and managing updates to channels. Hardware parallelism is abstracted with the notion of delta cycle.

SystemC supports method processes and thread processes. A method executes atomically and cannot explicitly suspend itself. The simulator regains control after the entire method has been executed. Threads are run exactly once by the scheduler and are typically enclosed by a loop that keeps them alive for the duration of the simulation. The program-flow control remains with the thread till it explicitly yields by calling $wait()$ or finishing its execution. In the former case, the thread stays in a *wait* state until some event *triggers* it, and it resumes execution from the next statement after *wait*.

Processes are triggered and synchronized with respect to its sensitivity on events. A SystemC event is the occurrence of an *sc_event* notification and happens at a single point in time. An event has no duration or value. Events are controlled via *wait*, *next_trigger* and *notify* functions of the *sc_event* class. A *wait* function changes dynamic sensitivity of a thread process and suspends its execution. For example, $wait(SC\_ZERO\_TIME)$ delays the process by one delta cycle, a process waits on event e with $wait(e)$, and with $wait(e1|e2|e3)$ a process waits on event $e1, e2$, or $e3$.

Events occur explicitly by using the *notify* function, and the scheduler resumes execution of a thread or method process by executing the *trigger* function. For example, $e.notify()$ is called an immediate notification since processes sensitive to event e will run in the current evaluation phase or delta cycle. Using $e.notify(SC\_ZERO\_TIME)$ processes sensitive to event e will run in the evaluation phase of the next delta-cycle. Using $e.notify(t)$ processes sensitive to event e will run during the evaluation phase of some future simulation time.

Process synchronization also occurs with the usage of channels, interfaces, and ports. These constructs are the core of SystemC Transaction Level Model (TLM) based methodology [1]. The basic idea of TLM is to model hardware components as modules that communicate with transactions. TLM provides interoperability layer for bus modeling through generic payloads and phases that in turn get used through initiator and target sockets. These sockets can use blocking and non-blocking transport interfaces. Different levels of model abstraction is provided in TLM through different coding styles such as loosely-timed (used by blocking transport interface) and approximately-timed (used by non-blocking transport interface). Loosely-timed is more suitable for software development, whereas approximately-timed is more suitable for performance analysis or architectural exploration. Some of the TLM synchronization functions are *b_transport* (blocking transport), *nb_transport_fw* (non-blocking transport forward path), and *nb_transport_bw* (non-blocking transport backward path). Note that, in SystemC, communication between processes is established either by explicit concurrency functions or by shared variables.

The following displays the steps of the simulation scheduler in more detail. The first phase of a SystemC simulation, the elaboration phase, consists of describing an architecture by registering the processes in the scheduler and defining constructs for module interconnection.

1. *Initialization:* All processes are made executable in an unspecified order.
2. *Evaluate:* Select a ready-to-run process and resume its

execution. This may result in more processes ready for execution in this same phase due to immediate notification. Signals and channels may invoke a request for update in the update phase.

3. Repeat Step 2 until no more processes are ready-to-run.
4. *Update:* Execute all pending update requests due to calls made in Step 2.
5. If Steps 2 or 4 resulted in delta event notifications, go back to Step 2.
6. If there are no more events, simulation is finished for current time.
7. Advance to next simulation time that has pending events. If none, exit simulation.
8. Go back to Step 2.

The SystemC scheduler is not preemptive, that is, a process runs without interruption until it explicitly gives control back with a wait statement. In [17], the authors show that a non-preemptive scheduler introduces implicit atomic sections (a wait-to-wait block in a process) hiding most of the issues regarding concurrent accesses to shared resources.

3.2.1 SystemC Example

Figure 1 illustrates a SystemC design. We will use this example throughout this paper to demonstrate our methodology. The code in Figure 1 declares a SystemC module $M1$ where internal processes, threads $T1$ and $T2$, communicate through a SystemC event e and two shared variables $cs1$ and $cs2$.

In this example, the synchronization of the event e is guarded by two booleans $cs1$ (line 17) and $cs2$ (line 26) assigned in the constructor of $M1$. The values of $cs1$ and $cs2$ are assigned to arguments of the constructor. Therefore, both threads $T1$ and $T2$ are open programs. The input-synchronization dependency of the threads presents an interesting case study since the presence of synchronization errors is not uniquely dependent on the SystemC scheduler. This behavior is similar to synchronization events generated by other processes through interprocess communication. In this paper, we are interested in an automatic method for generation of unit tests that explore different SystemC simulations.

```
 1  SC_MODULE(M1)
 2  {
 3    sc_event e;
 4    bool cs1, cs2;
 5
 6    SC_HAS_PROCESS(M1);
 7
 8    M1(sc_module_name name, bool x,
        bool y)
 9    {
10      SC_THREAD(T1);
11      SC_THREAD(T2);
12      cs1 = x;
13      cs2 = y;
14    }
15
16    void T1() {
17      if(cs1){
18        wait(e);         // Mutation #1
19        cs2=false;
20      }
21      wait(10,SC_NS);  // Mutation #2
22      cs2=true;
23    }
24
25    void T2(){
26      if(cs2){
27        cs1=false;
28        e.notify();      // Mutation #3
29      }
30      wait(10,SC_NS);  // Mutation #4
31      cs1=true;
32    }
33  };
```

Figure 1: A SystemC Design

3.3 Background on LLVM

LLVM is a popular and steadily growing compiler framework that supports aggressive multistage optimizations with competitive results against industrial compilers. The framework was initially designed to be a flexible, well-documented and transparent infrastructure for research projects in the compiler domain [22].

LLVM supports reusable back-end components that simplify the compiler construction process since, out of the box, compiler implementors can reuse several optimizations and analysis already implemented. Hence, the main compiler construction step is the front-end implementation. Moreover, LLVM provides code generation for several architectures, and although primarily focused on C and C++, other programming language front-ends have been implemented.

In LLVM, every optimization or transformation is performed over LLVM Intermediate Representation (IR) code. The LLVM IR language implements an unbounded register machine. The instruction set is composed of RISC-like three address code instructions in Single Static Analysis (SSA) form [35] with high level type information. The SSA representation form and the combination of low-level and high-level information translate in a well-defined and target-independent semantics. Hence, LLVM IR is suitable for analysis and, in theory, is capable of representing 'all' high-level languages cleanly [2, 23].

In Figure 2, we describe some of the productions that compose the implemented LLVM IR grammar. We describe a simplified grammar since we are not interested in attributes such as linkage, section or garbage collection.

An LLVM IR ⟨module⟩ is composed of a list of named types, global variables and functions. A named typed, ⟨nmd-ty⟩, binds an identifier to a type. In line 1 of Figure 3, %"*class.std :: ios_base :: Init*" is defined as the integer type of one byte. A global variable ⟨global⟩ is represented by an identifier, a type and an optional value if the variable is initialized. Line 2 of Figure 3, defines "@M1" as a constant with the value "M1".

Function definitions are composed of a declaration (identifier ⟨ident⟩, type ⟨ty⟩ and parameters ⟨param⟩*) and a list of basic blocks. Each basic block ⟨bb⟩ is represented by a label identifier, a list of ⟨phi⟩ instructions, a list of instructions and a final terminator ⟨tmn⟩ instruction such as an unconditional *branch* (line 4). A *phi* instruction represents a choice between several basic blocks.

Instructions are grouped into binary ⟨bop⟩, bitwise ⟨bwop⟩,

$$
\begin{aligned}
\langle\text{module}\rangle &\models \langle\text{nmd-ty}\rangle^{*}\ \langle\text{global}\rangle^{*}\ \langle\text{fn}\rangle^{*}\\
\langle\text{nmd-ty}\rangle &\models \%\ \langle\text{ident}\rangle\ \langle\text{ty}\rangle\\
\langle\text{global}\rangle &\models @\ \langle\text{ident}\rangle\ \langle\text{ty}\rangle\ \langle\text{val}\rangle?\\
\langle\text{fn}\rangle &\models \text{fun-decl}\ \langle\text{ident}\rangle\ \langle\text{ty}\rangle\ \langle\text{param}\rangle^{*}\\
&\mid\ \text{fun-def}\ \langle\text{ident}\rangle\ \langle\text{ty}\rangle\ \langle\text{param}\rangle^{*}\ \langle\text{bb}\rangle^{+}\\
\langle\text{param}\rangle &\models \langle\text{ident}\rangle\ \langle\text{ty}\rangle\\
\langle\text{bb}\rangle &\models \langle\text{label}\rangle\ \langle\text{phi}\rangle^{*}\ \langle\text{instr}\rangle^{*}\ \langle\text{tmn}\rangle\\
\langle\text{phi}\rangle &\models \text{phi}\ \langle\text{value}\rangle\ \langle\text{ty}\rangle\ (\langle\text{value}\rangle\ \langle\text{label}\rangle)^{+}\\
\langle\text{tmn}\rangle &\models \text{unreachable}\ \mid\ \langle\text{br}\rangle\ \mid\ \langle\text{ret}\rangle\\
\langle\text{instr}\rangle &\models \langle\text{bop}\rangle\ \mid\ \langle\text{bwop}\rangle\ \mid\ \langle\text{vop}\rangle\\
&\mid\ \langle\text{aop}\rangle\ \mid\ \langle\text{mop}\rangle\ \mid\ \langle\text{cop}\rangle\ \mid\ \langle\text{oop}\rangle\\
\langle\text{value}\rangle &\models \langle\text{ident}\rangle\ \mid\ \langle\text{const}\rangle\\
\langle\text{ty}\rangle &\models \text{void}\ \mid\ \text{i}\,\langle\text{int}\rangle\ \mid\ \langle\text{float}\rangle\\
&\mid\ [\ \langle\text{int}\rangle\ \times\ \langle\text{ty}\rangle\ ]\ \mid\ \langle\ \langle\text{int}\rangle\ \times\ \langle\text{ty}\rangle\ \rangle\\
&\mid\ \langle\text{ty}\rangle{*}\ \mid\ \langle\text{ty}\rangle^{*} \rightarrow \langle\text{ty}\rangle\ \mid\ \{\ \langle\text{ty}\rangle^{*}\ \}
\end{aligned}
$$

Figure 2: LLVM IR Grammar

```
%"class.std::ios_base::Init" = type { i8 }
@M1 = linkonce_odr constant [2 x i8] c"M1"
%.0 = phi i8* [%.pre1, %bb1],[%tmp8, %bb4]
br label %bb14
```

Figure 3: LLVM IR example

vector ⟨vop⟩, aggregate ⟨aop⟩, memory access and addressing ⟨mop⟩, conversion ⟨cop⟩ and other ⟨cop⟩ operations.

LLVM IR is equipped with a type system that adds extra expressive power to the language. Every LLVM IR ⟨value⟩, either an identifier ⟨ident⟩ or a constant ⟨const⟩ has a type ⟨ty⟩. LLVM IR supports primitive types: void, integers with a specified bit width i⟨int⟩ or ⟨float⟩; and derived types for arrays, vectors, structs, pointers and functions.

4. MUTATION FRAMEWORK

In this section, we describe our SystemC metamutant generation flow illustrated in Figure 4. It is an instance of our mutation framework that operates at the LLVM IR level, since we can potentially mutate programs of all programming languages supported by LLVM.

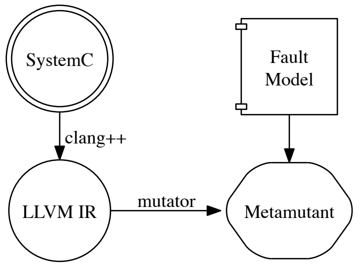

Figure 4: Metamutant Generation Flow

Given a SystemC program, we use clang++ [11], a C++ compiler front-end for LLVM, to generate LLVM IR byte

Algorithm 2 Meta Mutant Generation

```
Input: LLVM IR Module $M$.
Output: Metamutant, Number $c$ of mutation operators.
 1: initialize counter $c$ to 0;
 2: for each function $F$ in $M$ do
 3:    for each basic block $BB$ in $F$ do
 4:       for each instruction $I$ in $BB$ do
 5:          if $isSync(I)$ then
 6:             $I' = mutate(I, c)$;
 7:             increment $c$;
 8:          end if
 9:       end for
10:    end for
11: end for
```

code. Then we apply Algorithm 2 based on a fault model for SystemC through an LLVM pass to generate a metamutant.

Algorithm 2 describes our metamutant generation procedure. In line 1, we initialize the integer c to 0. This integer serves as a counter that represents the number of instructions mutated so far and also is an argument to a new mutate function in our mutation library. The mutate function implements a mutation operator. Then, we iterate over all instructions (lines 2 to 4) and check if the current instruction is a call to a relevant SystemC synchronization function (line 5). In that case, in line 7, we mutate the callee of the instruction to point to our mutation library.

We currently explore a simple fault model that implements mutation operators that remove synchronization functions related to SystemC events and time. Nevertheless, our framework is easily extensible to a complete set of SystemC constructs and richer mutation operators.

```
void T1()
{
   if(cs1){
 // wait(e);
     cs2=false;
   }
   wait(10,SC_NS);
   cs2=true;
}
```

```
void T2()
{
   if(cs2){
     cs1=false;
     e.notify();
   }
   wait(10,SC_NS);
   cs1=true;
}
```

Figure 5: Mutant Example for SystemC Design in Figure 1

Previous work [37] have investigated the relationship between SystemC TLM synchronization operations and common concurrent error patterns such as deadlocks, lost notifies or data races. Figure 5 shows an example of a mutant generated from the SystemC design in Figure 1. In this case, *Mutation #1* was enabled and the mutation operator removed the call to $wait(e)$. Independent of the scheduling order, this mutant has a lost notify in thread $T2$ if the value of $cs2$ is *true*. This information is useful for testers in fault identification and resolution.

5. TESTBENCH GENERATION FLOW

We explore the dependency relation between input variables and synchronization operations following the hypothesis that a suitable testbench for a concurrent program should reach all possible synchronization operations. Although the reachability problem for a general program is undecidable, we use a bounded version of the original program to partially solve this problem. Therefore, we only support SystemC designs that have free variables in at least one of their processes and support a mechanism such that testing different values for those free variables is possible. Also, note that although in this section we motivate our techniques through SystemC models our results are applicable to SystemC TLM models as well.

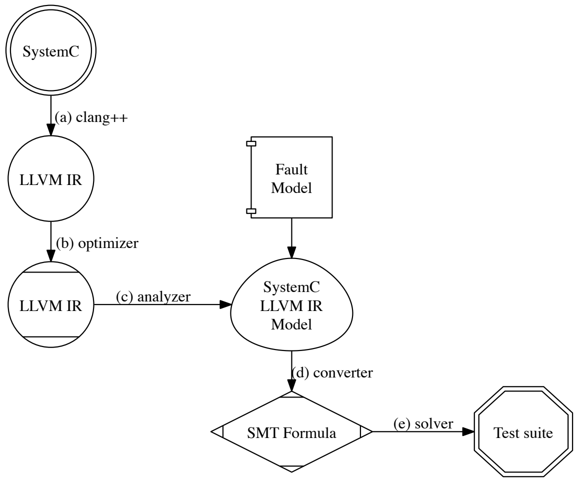

Figure 6: Testbench Generation Flow

In Figure 6, we present our automatic test generation flow. Given a SystemC design we start by generating LLVM byte code with clang++. Since the implementation of SystemC and other C++ libraries is heavily based on template programming, we apply various optimizations to obtain more compact byte code modules.

In phase (b), we use the optimizer to transform the current byte code into a form that is suitable for our analysis. In total, we apply 18 LLVM passes (optimizations, transformations) achieving considerable byte code reduction. For the design in Figure 1, we achieve a 80% reduction between the byte code bounded version and an unoptimized version. We describe some of these passes now. We divide the set of transformations applied into decidability and simplification categories. Decidability transformations aim at generating a bounded version of the program such that the reachability problem becomes decidable. For the purpose of our analysis we want to obtain byte code that has no cycles in the basic block graph. Simplification transformations aim at simplifying our formalization. We further reduce the byte code size eliminating LLVM IR constructs not supported by our analysis such as invoke or switch instructions. Also, we leverage the LLVM IR unbounded number of registers to promote the stack to registers and we name all nameless identifiers.

In phase (c), we extract a SystemC model that contains information about the architecture and the behavior of the processes. We describe this method in detail in Section 5.1.

In Section 5.2, we describe our conversion function from a SystemC model to an SMT formula, phase (d), exploiting the deterministic implementation of the SystemC scheduler. Finally, in phase (e), described in Section 5.3, we use an SMT solver to generate a test case.

5.1 SystemC Model Extraction

We have formalized the LLVM IR language with the Utrecht Attribute Grammar System (uuagc) [38]. Our representation of LLVM IR is similar to the one in Figure 2. We use a binding library for Haskell [21], a general purely functional language, to retrieve information from LLVM byte code modules. Our main motivation for this approach is a combination of the advantages of Haskell as a prototyping language with an expressive type system and equational reasoning, and the similarities between LLVM IR and functional languages. *uuagc* is a framework suitable for analysis of functional programming languages that generates optimized Haskell programs.

```
int sc_main(int argc , char *argv[])
{
  // Initialize cs1 and cs2 with argv

  M1 M1("M1", cs1, cs2);
  sc_start();

  return 0;
}
```

Figure 7: *sc_main* of SystemC design in Figure 1

In Figure 7, we present a simple example of a simulation model for the SystemC design in Figure 1. In *sc_main*, we start by assigning $cs1$ and $cs2$ with the user input arguments. Then, we create an instance of the SystemC module $M1$ with $cs1$ and $cs2$. Finally, we start the simulation with a call to *sc_start*.

```
 1
 2 %struct.M1 = type { %"class.sc_core::sc_module", %"
        class.sc_core::sc_event", i8, i8 }
 3  @.str = private unnamed_addr constant [3 x i8] c"M1\00"
        , align 1
 4  define i32 @sc_main(i32 %argc, i8** nocapture %argv)
        uwtable {
 5    ; ...
 6    %M1 = alloca %struct.M1, align 8
 7    %tmp1 = alloca %"class.sc_core::sc_module_name",
          align 8
 8    ; %tmp5 ~= argv[1], %tmp9 ~= argv[2]
 9    call void @_ZN7sc_core14sc_module_nameC1EPKc(%"
          class.sc_core::sc_module_name"* %tmp1, i8*
          getelementptr inbounds ([3 x i8]* @.str, i64 0,
          i64 0))
10    call void @_ZN2M1C2EN7sc_core14sc_module_nameEbb(%
          struct.M1* %M1, %"class.sc_core::sc_module_name"
          * undef, i1 zeroext %tmp5, i1 zeroext %tmp9)
11    call void @_ZN7sc_core14sc_module_nameD1Ev(%"
          class.sc_core::sc_module_name"* %tmp1)
12    call void @_ZN7sc_core8sc_startEv()
13
14  ; ...
15  }
```

Figure 8: LLVM IR of *sc_main*

Figure 8 lists the LLVM byte code relevant to *sc_main*, and in Figure 9 we present its formalization as Haskell *data*. The [ ] notation is the Haskell representation for lists. The model represented in Figure 9 is a simplified, structured and readable version of the actual LLVM byte code of Figure 8.

The *sc_main* byte code starts by allocating a struct $M1$ (line 6). The definition of the named type *struct.*$M1$ is listed in line 2. Note that it contains information about the class variables of $M1$. In line 9, there is a call to the constructor of *sc_module_name*. Then, in line 10, an instance of $M1$ is defined with a call to the constructor. Finally, *sc_start* is called (line 12) after the call for the destructor of *sc_module_name* (line 11).

```
Module
  [NmdTy (Local ”struct.M1”) TyStruct (Local ”struct.M1”
        ) [SyscMod, SyscEv, (TyI 8), (TyI 8)]]
  [Global (Global ”.str”) (TyArray 3 (TyI 8)) (Const (
        CString ”M1\00”))]
  [FunctionDef (Global ”sc_main”) (TyI 32) [Param (Local
        ”argc”) (TyI 32), Param (Local ”argv”) (TyPtr (
        TyPtr (TyI 8)))]
   [BasicBlock
    [Alloca (Local ”M1”) (TyStruct (Local ”struct.M1”)
         ...)
    ,Alloca (Local ”tmp1”) (TyStruct (Local ”class.
         sc_core::sc_module_name”) ...)
    ,...
    ,Call TyVoid (Global ”sc_core::sc_module_name::
         sc_module_name”) [Ident (Local ”tmp1”), Ident (
         Global ”.str”), Const (CInt (TyI 64) 0), Const
         (CInt (TyI 64) 0)]
    ,Call TyVoid (Global ”M1::M1”) [Ident (Local ”M1”),
         Const UndefC, Ident (Local ”tmp5”), Ident (
         Local ”tmp5”)]
    ,Call TyVoid (Global ”sc_core::sc_module_name::~
         sc_module_name()”) [Ident (Local ”tmp1”)]
    ,Call TyVoid (Global ”sc_core::sc_start()”) []
    ]
   ]
  ]
```

Figure 9: *sc_main* Model

We apply static analysis techniques to analyze the elaboration phase of a SystemC simulation model. Our method iterates over all instructions in the *sc_main* function and checks for calls to *sc_module* constructors until it reaches the call to *sc_start*. In case of a call to a *sc_module* constructor we retrieve the module information. This function analyzes the named type to retrieve the class variables and inspects the constructor to gather information about the module processes and communication.

```
SysC
  [ScMod (Id ”M1”)
         (Vars [ScEvent, TyI 8, TyI 8])
         (ScProcs [ScThread (Id ”T1”)
                     (..)
                   ,ScThread (Id ”T2”)
                     (..)
                   ]
         )
  ]
```

Figure 10: SystemC Model in Haskell

We generate a SystemC model with the architecture as in Figure 10. The model is represented by a list of modules. Each module is composed of an identifier, a list of class variables and a list of processes that respect the same declaration order as in the original constructor. A process is either a thread or a method. They are represented by an identifier and the LLVM byte code model of their function bodies.

5.2 SystemC Model Encoding

In this section, we present an encoding function to generate an SMT closed quantifier-free formula with bit-vectors from a SystemC model. Given a mutated instruction i in a basic block β_i, and a SystemC model M, as the one in Figure 10, we generate the following SMT formula:

$$\Psi(i,M) = \overbrace{\otimes(i,M)}^{\text{inter } \beta_i \text{ analysis}} \;\wedge\; \underbrace{\oplus(i,M)}_{\text{intra } \beta_i \text{ analysis}} \tag{1}$$

Formula 1 encodes the reachability problem for LLVM IR with concurrent constructs. We model this problem as a conjunction of the predicates given by operators $\otimes$ (otimes) and $\oplus$ (oplus).

$\otimes$ encodes the constraints related to local identifiers to reach the basic block of instruction i and is formally defined as follow.

$$\otimes(i,M) = \bigvee_{\beta' \in \Pi(\beta_i)} \odot(\llbracket \tau_{\beta'},\ \beta_i \rrbracket,\ \Pi(\tau_{\beta'}),\ M) \wedge \overbrace{\otimes(\tau_{\beta'},\ M)}^{\text{recursion}} \tag{2}$$

To reach the mutated basic block β_i, we iterate over its direct basic block predecessors and for each basic block the operator $\odot$ (odot) encodes the reachability constraints. $\odot$ performs an intra basic block backwards analysis with an initial predicate that represents the reachability path to β_i. The initial predicate is given by $\llbracket \tau_{\beta'},\ \beta_i \rrbracket$ (Formula 3), where $\tau_{\beta'}$ is the terminator instruction of the basic block β'.

Finally, we reach the entry basic block by recursion, and guarantee termination based on the assumption that there are no cycles in the basic block graph.

Note that Π is a polymorphic function; $\Pi(\beta_i)$ computes the direct predecessor set of basic blocks and $\Pi(\tau_{\beta'})$ computes the predecessor set of instructions.

$$\llbracket \tau_{\beta'},\ \beta_i \rrbracket = \begin{cases} true, & (\tau_{\beta'} = \text{br } \beta_\kappa) \wedge \beta_i \equiv \beta_\kappa \\ \nu \equiv 1, & (\tau_{\beta'} = \text{br } \nu, \beta_t\ \beta_f) \wedge \beta_i \equiv \beta_t \\ \nu \equiv 0, & (\tau_{\beta'} = \text{br } \nu, \beta_t\ \beta_f) \wedge \beta_i \equiv \beta_f \\ false, & \text{otherwise} \end{cases} \tag{3}$$

$\odot$ is defined based on set induction. The base case, Formula 4, is the head of the basic block and since there are no more instructions to interpret, we return the predicate. In the induction step, Formula 5, we interpret the instruction ϵ with the current predicate ψ. The interpretation function $\llbracket\ \rrbracket$ encodes the instruction in a simple flat memory model supported by SMT solvers and applies logical conjunction with the current predicate. We have extended the encoding provided in [29] to support synchronization operations. In that case, we apply function Ψ to retrieve the SMT formula generated. Since such operations are also mutated, we can use memoisation techniques to optimize our implementation.

$$\odot(\psi,\ \emptyset,\ M) = \psi \tag{4}$$

$$\odot(\psi,\ \epsilon \cup \Upsilon,\ M) = \odot(\llbracket \epsilon,\ \psi,\ M \rrbracket,\ \Upsilon,\ M) \tag{5}$$

When reasoning about sequential programs at the LLVM byte code level we can assume reachability of any instruction in a basic block if we reach the entry point of the basic block. However, in a concurrent program reachability from the head of the basic block to the current mutated instruction is not guaranteed. The operator $\oplus$ (oplus) defined in Formula 6, encodes the constraints for the current thread to be enabled (operator $\uplus_i$, uplus) and also considers previous deadlock scenarios in the mutated basic block β_i (operator $\uplus$). The function $\Lambda(i)$ returns the set of predecessor operations in the basic block that may lead to a deadlock situation.

$$\oplus(i,M) = \uplus_i(i,M) \bigwedge_{i' \in \Lambda(i)} \uplus(i',M) \tag{6}$$

We exploit the deterministic implementation of the SystemC scheduler to fix a scheduling order. In a SystemC model M, we define a poset $\prec_{proc}$ composed of the processes in the model. We build this relation based on the declaration order in the original program. For the SystemC design in Figure 1, $T1 < T2$.

$$\uplus_i(i, M) = \begin{cases} true & , T_i \text{ minimal } \prec_{proc} \\ \bigvee_{T<T_i} \Psi(i^{-1}, M) & , \text{ otherwise} \end{cases} \tag{7}$$

Operator $\uplus_i$ in Formula 7, checks if the thread executing instruction i, T_i is the minimal element of $\prec_{proc}$. In that case we can assume that T_i will be the first thread to be executed. To guarantee reachability of i if T_i is not the minimal element of $\prec_{proc}$, at least one of the previously executed threads has to reach an instruction i^{-1} that gives control to the scheduler. The operator $\uplus$ in Formula 8, encodes the constraints for previous i^{-1} instructions in the basic block.

$$\uplus(i, M) = \bigvee_{T<T_i} \Psi(i^{-1}, M) \tag{8}$$

5.3 Testbench Generation

We pass the generated SMT formula to a solver to generate a set of assignments. It is not guaranteed that the formula generated will be satisfiable. This means that the mutated instruction might be dead or that for that scheduling a deadlock occurs.

In the final step of test generation we map the thread local identifiers to user input arguments of *sc_main*. The method described in Section 5.2 can be applied to the constructor instruction of the SystemC module. If the formula contains are free variables, the SMT solver will choose a default value according to its type.

5.3.1 SystemC Model Encoding Example

We demonstrate our method with the mutant of Figure 5. In Figure 11, we provide the relevant LLVM byte code instructions of thread $T1$.

```
1   bb:
2     %tmp = alloca %"class.sc_core::sc_time", align 8
3     %tmp1 = getelementptr inbounds %struct.M1* %this, i64
            0, i32 2
4     %tmp2 = load i8* %tmp1, align 1, !tbaa !6, !range !7
5     %tmp3 = icmp eq i8 %tmp2, 0
6      br i1 %tmp3, label %._crit_edge, label %bb4
7
8   bb4:                                        ; preds = %
          bb
9     %tmp5 = getelementptr inbounds %struct.M1* %this, i64
            0, i32 1
10    %tmp6 = getelementptr inbounds %struct.M1* %this, i64
            0, i32 0, i32 0, i32 1
11    %tmp7 = load %"class.sc_core::sc_simcontext"** %tmp6,
            align 8, !tbaa !0
12     call void
            @_ZN7sc_core4waitERKNS_8sc_event_13sc_simcontextE
            (%"class.sc_core::sc_event"* %tmp5, %"
            class.sc_core::sc_simcontext"* %tmp7)
```

Figure 11: LLVM byte code for *T1*

In this example, the mutated instruction i is the $wait(e)$ in line 12, and we use the extracted model M in Figure 10.

$$\Psi(\text{wait(e)}, M) = \otimes(\text{wait(e)}, M) \wedge \oplus(\text{wait(e)}, M)$$

Since the basic block predecessor of *bb4* is the entry basic block *bb*, the operator $\otimes$ is reduced to:

$$\otimes(\text{wait(e)}, M) = \odot([\![\tau_{\text{bb}},\ \text{bb4}]\!],\ \Pi(\tau_{\text{bb}}),\ M)$$

Then, we use the interpretation function in Formula 3 to further reduce $\otimes$ to:

$$\odot(\text{tmp3} = 0,\ \Upsilon,\ M) = \text{struct.M1[2]} \neq 0$$

Υ is the set of instructions from lines 5 to 2.

The $\oplus$ operator is reduced to *true* since $T1$ is the minimal element of M_T and there is no more *call wait* instructions before line 12. The simplified SMT expression generated is:

$$\Psi(\text{wait(e)}, M) = \text{struct.M1[2]} \neq 0 \wedge \text{true}$$

Table 1 shows a test suite for M using four mutants. In this case, we achieve total input coverage. Note that some formulas can generate the same test case. This information could be used as a heuristic to reduce the number of mutants.

M#	Simplified SMT Formula	Input
M1	argv[1] = t	(t,f)
M2	argv[1] = f $\vee$ (argv[1] = t $\wedge$ argv[2] = t)	(f,f)
M3	argv[2] = t	(f,t)
M4	argv[1] = t $\wedge$ argv[2] = t	(t,t)

Table 1: Test suite generated for SystemC in Figure 1

6. ORACLE GENERATION

A test oracle is a mechanism that determines the program correctness with respect to a test [40]. We modify this notion of test oracle for our mutation testing based technique as follows. Using the testbenches generated with our method, we produce oracles that represent the differences between a program and its mutant by instrumenting the program with observers for all variables. The observers dump an internal state triggered by any synchronization operation. Our state representation is the thread identifier and the values of all variables in scope. We use execution trace comparison as observable differences to obtain a complete definition of *killed mutant*. Note that an assertion could also be an oracle for our technique.

Table 2 lists two execution traces generated when executing the original design in Figure 1 and Mutant #1 with the input generated (t,f) from Table 1. Both the original and mutant program start executing $T1$ with the initial input (t,f). The original program will reach $wait(e)$ and suspend itself. Then $T2$ will not call notify since $cs2$ is false and wait 10 simulation time units. Since $T1$ is waiting for the notification of the event, the scheduler will update the time to enable $T2$, which will then execute $cs1 = true$ and terminate leading to a deadlock. Mutant #1 starts executing $T1$ and skips the operation $wait(e)$. The second line of the trace corresponds to the dump of the assignment $cs2 = false$. Then both threads will wait 10 simulation time units. The fourth line of the trace corresponds to the assignment $cs2 = true$ and finally, $T2$ will reach the assignment $cs1 = true$.

Oracles provide useful information to find synchronization errors. Table 3 lists all the schedulings for the test suite generated in Table 1. We can use the execution traces for

Original Trace	Mutant #1 Trace
T1:(t,f)	T1:(t,f)
T2:(t,f)	T2:(t,f)
Time Elapse	Time Elapse
T2:(t,f)	T1:(t,t)
Deadlock	T2:(t,t)

Table 2: Execution Trace Comparison.

fault localization to explain why the program deadlocks in the original version and not the mutant.

M#	Input	Original Schedule	Mutant Schedule
M1	(t,f)	T1; T2; TE; T2 (DL)	T1; T2; TE; T1; T2
M2	(f,f)	T1; T2; TE; T1; T2	T1; T2; TE; T2
M3	(f,t)	T1; T2; TE; T1; T2 (LN)	T1; T2; TE; T1; T2
M4	(t,t)	T1; T2; T1; TE; T1; T2	T1; T2; TE; T1

Table 3: Schedulings.
(TE = time elapse, DL = deadlock, LN = lost notify)

7. EXPERIMENTS

We have conducted initial experiments with 5 SystemC designs. We have modified for our analysis four test cases from the SCRV [16] test suite and implemented an instance of the producer consumer example. In Table 4, we present our results. The second column of the table represents the size of the optimized LLVM byte code in number of lines. In the third column and fourth columns we present the number of generated mutants and test cases generated, respectively. Note that the number of test cases is smaller or equal to the number of mutants, since we generate at most a single test case for each mutant. The same test case can kill multiple mutants. The last column presents our mutation coverage results. Although we generated test cases that reached all mutations, the mutation coverage scores were not always 100%. The reason is that some mutations do not produce any observable effects in the traces, e.g., Mutation #2 in Table 3. Also in the case of *prodcod*, the original program contains redundant synchronization mechanisms, hence mutation these does not produce any observable effect.

Design	# Lines	# Mutants	# Test cases	MC
indexer	1077	5	3	80%
sirac	1110	9	4	89%
fiveteen	1226	4	4	100%
prodcon	1229	5	3	40%
srX	1781	5	2	100%

Table 4: Mutation Coverage Results

8. CONCLUSIONS AND FUTURE WORK

In this work, we have introduced a complete testbench generation flow for concurrent SystemC programs based on mutation testing. Our framework operates at the LLVM byte code level, leveraging its formal semantics that lifts previous limitations of SystemC analysis. Our implementation operates at the domain of functional languages which are known to provide useful abstraction patterns for program analysis. We generate formal models that have wider applicability in other formal verification fields. Our experiments show that we can automatically generate testbenches with high mutant coverage ratio. In the future, we plan to use the Just In Time (JIT) compiler in LLVM for dynamic compilation of the elaboration phase in SystemC model generation.

9. ACKNOWLEDGMENTS

This work was supported in part by Marie Curie European Reintegration Grant within the 7th European Community Framework Programme, Semiconductor Research Corporation under task 2082.001, Bogazici University Research Fund 5483, and the Turkish Academy of Sciences.

10. REFERENCES

[1] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1 –638, jan 2012.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402 – 411, 2005.

[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 737–797. IOS Press, Amsterdam, The Netherlands, 2009.

[5] N. Bombieri, F. Fummi, and G. Pravadelli. A Mutation Model for the SystemC TLM 2.0 Communication Interfaces. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, pages 396–401. ACM, 2008.

[6] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe. Functional Qualification of TLM Verification. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, pages 190–195. ACM, 2009.

[7] J. Bradbury, J. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (J2SE 5.0). In *Workshop on Mutation Analysis*, page 11, Nov. 2006.

[8] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing*, pages 129–148. North-Holland, 1981.

[9] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[10] D. V. Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. B. Brown. High-level Design

Verification of Microprocessors via Error Modeling. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):581–599, 1998.

[11] clang: a C language family frontend for LLVM, http://clang.llvm.org/, 2012.

[12] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, Sept. 1991.

[13] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness. *Journal of Systems and Software*, 38(3):235–253, September 1997.

[14] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[15] M. Hampton and S. Petithomme. Leveraging a commercial mutation analysis tool for research. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007*, pages 203–209, Sept. 2007.

[16] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz. Full Simulation Coverage for SystemC Transaction-Level Models of Systems-on-a-Chip. *Formal Methods in System Design*, 35(2):152–189, 2009.

[17] C. Helmstetter and O. Ponsini. A Comparison of Two SystemC/TLM Semantics for Formal Verification. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 59–68, 2008.

[18] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Dynamic State Traversal for Sequential Circuit Test Generation. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):548–565, 2000.

[19] Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, Oct. 2009.

[20] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, PP(99):1, 2010.

[21] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* http://haskell.org/, September 2002.

[22] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[23] C. Lattner and V. Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[24] G. Li, I. Ghosh, and S. P. Rajan. Klover: a symbolic execution and automatic test generation tool for c++ programs. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. Gklee: concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, New York, NY, USA, 2012. ACM.

[26] N. Li, U. Praphamontripong, and J. Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 220–229, 2009.

[27] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An Automated Class Mutation System: Research Articles. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[28] F. Maraninchi, M. Moy, J. Cornet, L. Maillet-Contoz, C. Helmstetter, and C. Traulsen. SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation. In IEEE, editor, *2008 Joint IEEE-NEWCAS and TAISA Conference*, page unknown, Montréal, Canada, June 2008.

[29] F. Merz, S. Falke, and C. Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *VSTTE*, pages 146–161, 2012.

[30] J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta, USA, 1988.

[31] J. Offutt, P. Ammann, and L. Liu. Mutation Testing implements Grammar-Based Testing. In *Workshop on Mutation Analysis, 2006*, pages 12–12, 2006.

[32] J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal.* Kluwer Academic Publishers, 2001.

[33] S. P. F. Fabbri, M. Delamaro, J. Maldonado, and P. Masiero. Mutation Analysis Testing for Finite State Machines. In *5th International Symposium on Software Reliability Engineering*, pages 220–229, Nov 1994.

[34] H. Riener, R. Bloem, and G. Fey. Test case generation from mutants using model checking techniques. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 388–397, Washington, DC, USA, 2011. IEEE Computer Society.

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[36] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 69–80, July 2009.

[37] A. Sen. Concurrency-oriented verification and coverage of system-level designs. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):37:1–37:25, Oct. 2011.

[38] S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[39] P. J. Walsh. *A Measure of Test Case Completeness (software, engineering).* PhD thesis, State University of New York at Binghamton, Binghamton, NY, USA, 1985.

[40] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.