Marcelo André Barbosa de Sousa

# A Framework for Formal Verification of Concurrent Software

– MSc Thesis –
ICA-3535592

August 31, 2012

Utrecht Universiteit, Netherlands
Boğaziçi Üniversitesi, Türkiye

*Vague and nebulous is the beginning of all things, but not their end.*

Kahlil Gibran

**Summary.** The widespread usage of concurrent software boosted the need for verification tools to help designers and implementors in the overall software engineering process. Formal verification techniques aim at producing automated tools to reduce the quality assurance stage. Currently, most of the verification approaches have only been applied to sequential software or partially to concurrent software. We present a novel framework implemented in a pure functional language for verification of commonly used concurrent mechanisms. To our knowledge, this is the first verification framework using the Low Level Virtual Machine (LLVM) byte code representation focused on concurrent programs and using a generic representation of a concurrent model. In this thesis, we focus on test-suite development as the main application of the framework. We devise two orthogonal approaches for automated test-suite generation: a SMT-based bounded model checker for Pthread programs and a mutation testing driven SMT-based static analysis procedure for SystemC TLM models. We perform initial experiments on SystemC designs and Pthread programs with multiple threads and mutex locking operations to demonstrate the effectiveness of our framework.

*I would like to thank Prof. Alper for his constant guidance and support to freely explore my research ideas, and Prof. Wishnu for his helpful insights. I dedicate this thesis to my family for their unconditional support.*

# Contents

**1 Introduction** . . . 1
1.1 Motivation . . . 2
1.2 Research Contributions . . . 3
1.3 Roadmap . . . 3

**2 Background** . . . 4
2.1 Formal Program Verification . . . 4
2.2 Bounded Model Checking . . . 6
2.3 Low-Level Virtual Machine . . . 8
2.4 Shared Memory Concurrency . . . 11
2.5 Mutation Testing . . . 15

**3 Related Work** . . . 18
3.1 Formalization of LLVM IR . . . 18
3.2 Model Checking . . . 18
3.3 Mutation Testing . . . 19
3.4 Automated Test-Case Generation . . . 20

**4 LLVM Verification Framework** . . . 21
4.1 Architecture . . . 21
4.2 Front-End . . . 22
4.2.1 Extraction . . . 22
4.2.2 Abstraction . . . 23
4.3 Back-End . . . 24
4.3.1 SMT-based Bounded Model Checker . . . 24
4.3.2 Mutation Framework . . . 29
4.3.3 Generation of SystemC TLM Testbenches Using Mutation Testing . . . 31

**5 Experiments** . . . 38

**6 Conclusion** . . . 42

**References** . . . 45

# 1 Introduction

> *Software gets slower faster than hardware gets faster.*
>
> Niklaus Wirth

The observation of Moore's Law [62] in the last half century triggered a paradigm shift in mainstream computer architectures. This shift ignited when the hardware industry realized that the approaches applied for boosting CPU performance reached the physical limits [68]. The solution adopted was the transition from single to multi-processor architectures. From the hardware perspective, this transition seems to scale with the increasing size and complexity of hardware designs. However, for various reasons the software industry still does not leverage the potential CPU speedups.

In the past, software developers methodologically relied on hardware advances to optimize their programs [77]. This inadequate methodology created a fundamental problem concerning the quality of the existent software with respect to code reliability and efficiency. With the shift to multi-core architectures, programming language designers developed specific concurrent programming languages and a wide spectrum of mechanisms to support concurrency. However, in practice, software developers still relied on operating systems and compilers to optimize their programs with efficient scheduling and parallelization techniques. Hence, we observed that concurrent programming was deceivingly ubiquitous in software engineering.

The skepticism shared among software developers concerning concurrent programming can be pinpointed to the idea that *developing concurrent programs is hard*. This idea results from the inherent non-deterministic nature of concurrent programs. The conceptual gap between standard sequential or object-oriented programming and concurrent programming affects all major phases of the software development process: design, implementation and validation.

The design and implementation of concurrent software is harder because developers need to decide which concurrent model and synchronization strategy is appropriate. These decisions are not trivial considering the wide range of programming languages that support concurrency, and the time required to develop a deep understanding of the concurrent and synchronization constructs. Optimization of concurrent software requires extra effort, since the notion of atomicity depends on the underlying memory model and compiler technology used. Testing and debugging concurrent programs is particularly problematic because of their non-deterministic

nature. In practice, there is a rupture of the traditional testing and debug tools since developers cannot rely on a single execution of their test suites and erroneous test cases might not be reproducible.

Despite the skepticism, concurrent programming has long been claimed as *the way of the future* [78] and in recent years there is a renewed interest in concurrent programming. This interest is driven not only by the evolution of hardware architectures but also the increasing size, complexity, expectations and reliability of software systems [40].

## 1.1 Motivation

Formal verification of software is a valuable approach to produce automated analysis and testing tools [51]. The tools are valuable because they simplify the developers manual quality assurance cycle, which represents a major portion of software development. Automated test-suite generation techniques and techniques to strengthen the quality of existing test-suites for concurrent programs are valuable for the industry to lower the costs of the quality assurance cycle. Moreover, research in this direction provides insights on the behavior of concurrent software and allows the identification of common error patterns in the industry.

Following the software industry trend, currently most of the verification tools focused on sequential software or a specific concurrent model [27]. The continuous widespread of concurrent software calls for a general framework for verification of concurrent software. The main goal is a transparent and scalable infrastructure for verification of several concurrent mechanisms. Hence, we design the infrastructure to verify programs represented at the compiler intermediate language, namely the Low-Level Virtual Machine (LLVM) Intermediate Representation (IR) level [52, 2, 53]. Furthermore, we leverage the advantages of Haskell [50], a general purpose strongly-typed functional programming language, and the Utrecht University Attribute Grammar Compiler (uuagc) [79] to perform efficient transformations on our formalizations. Haskell is suitable for prototyping due to its expressive type system that allows the implementation of reliable and scalable solutions. Moreover, it is suitable for program analysis and verification due to its ease of equational reasoning, a feature of pure functional languages.

The motivation for a new infrastructure is based on two main observations. The first observation is that a common practice in formal verification approaches, such as model checking is to reuse existing frameworks. For example, in the past several tools reused the explicit-state model checker SPIN [45]. This approach reduces the overall implementation effort for verification of a domain specific language because of the numerous optimizations already implemented in SPIN. However, it still requires a considerable amount of work to implement a front-end that translates the native language to Promela models [5], and in general, there is no guarantee that the translation process is reliable. Moreover, users need to understand the translation process and the verification results provided by SPIN. These limitations apply to other existing verification frameworks.

The second observation is that most of the verification tools focus on a specific concurrent domain. With the popularity of LLVM and since LLVM IR is designed to be a universal compiler intermediate representation, a verification framework operating at the LLVM IR level is applicable for concurrent programs represented in

the programming languages supported by LLVM. Furthermore, LLVM IR is a suitable language for verification since it is a well-defined language that considerably eases a logical encoding and closely reassemble the actual executed programs [61]. We are interested in the identification and formalization of an abstract concurrent model based on LLVM IR and apply several verification techniques over that model. Therefore, achieving a framework that could verify different shared-memory concurrent libraries, e.g. Pthread, domain specific languages, e.g. SystemC, and message passing interfaces, e.g. MPI [63] or MCAPI [80].

## 1.2 Research Contributions

We present a novel framework implemented in a pure functional language for verification of commonly used concurrent mechanisms. To our knowledge, this is the first verification framework using the LLVM byte code representation focused on concurrent programs using a generic representation of a concurrent model. The extension of the framework to a particular concurrent model is accomplished with an interpretation of the concurrent constructs. We present a new LLVM IR formalization using the attribute grammar system, and also an initial formalization of an abstract concurrency model.

We devise two applications for our framework: a SMT-based bounded model checker [8] for Pthread [64] programs that is able to produce counter examples for assertion violations in the original program; and a mutation testing driven SMT-based static analysis procedure that generates testbenches for SystemC TLM models [1] with high mutation coverage ratios.

## 1.3 Roadmap

The remaining of this thesis is organized as follows. Chapter 2 provides background on formal verification with focus on SMT-based bounded model checking, the LLVM framework, the two concurrent domains of research: Pthread and SystemC, and mutation testing. Chapter 3 describes related work in formalization of LLVM IR, SMT-based bounded model checking for LLVM IR, SystemC mutation testing and test case generation using mutation analysis. In Chapter 4, we introduce the architecture and applications of our verification framework *llvmvf* (LLVM Verification Framework). Chapter 5, describes our experimental setup and discusses the results of the experiments with the Pthread library and SystemC. We conclude with discussion of our current limitations, optimizations and extensions to be left as future work.

# 2

# Background

*Precise language is not the problem. Clear language is the problem.*

Richard Feynman

This chapter provides background on Formal Verification (Section 2.1), Bounded Model Checking (Section 2.2), LLVM (Section 2.3), shared-memory concurrent models (Section 2.4) and Mutation Testing (Section 2.5).

## 2.1 Formal Program Verification

The roots of formal program verification, also referenced as formal methods, are tied to the early beginnings of computer science. The fundamental idea behind formal methods is that programs can be viewed as mathematical objects with a well-defined behavior. Therefore, using mathematical logic we can reason about program correctness [33]. Program correctness is the main motivation for verification, but is a stand-alone general concept. The interpretation of correctness for a particular program is captured in its specification. Whether the formalism used in the specification is appropriate, or the specification truly captures the informal notion of correctness, are relevant issues that affect the verification approach.

Historically, there were two major periods in formal verification. Until the late 70s, the prevailing paradigm was based on deductive systems such as the successful Floyd-Hoare framework [34, 44]. The standard approach for verification was to use such deduction systems to produce formal proofs of correctness. This approach, known as theorem proving, aimed at a total correctness of the specification in a mathematical constructivist fashion. In practice, the specification was composed of a set of theorems and the verification was an ingenious manual process to produce formal proofs of the theorems.

Although theorem proving allows us to guarantee the reliability of programs, the framework suffers from three major limitations. The first limitation is related to Godel's first incompleteness theorem [38]. The theorem implies that there are sentences in a deductive system that cannot be formally proved. Hence, it restricts the proof domain of theorem proving.

The second limitation is related to the constructivist nature of proofs. Using the theorem proving framework, there are two possible outcomes. In the successful scenario, we obtain formal proofs that mathematically guarantee that the specification is satisfied. However, in case of failure we cannot guarantee that the specification is not satisfied. The lack of information in a failing scenario poses a severe practical limitation to use theorem proving. In practice, we are also interested in identifying which part of the program fails to meet the specification.

The third limitation is the observation that theorem proving does not scale with increasing size and complexity of programs. The main reason for the scalability problem is that manual proofs are laborious, error-prune and require a high level of expertise. The effort to reduce the scalability problem and automatize the proof process originated the development of proof assistants. Generally, a proof assistant is composed of a core and an interface component. The core of a proof assistant is an implementation of a deductive system. The interface guides the user in the proof process by interpreting the current proof state and provide hints. Currently, there are several kinds of proof assistants that implement various deductive systems and provide different user interface approaches [87].

In the late 70s, the seminal paper of Pnueli [70] marked a new period in formal methods. Following the advances in *modal logics*, Pnueli developed Linear Temporal Logic (LTL): a simple and expressive logic to reason about correctness properties of concurrent systems. The flexibility and expressiveness of temporal logic inspired the field of model checking [22, 33]. The approach in model checking is to extract a formal model from the program under verification and check if the specification is satisfied by the model. The framework is valuable because it provides an algorithmic approach for verification, and the specification is composed of temporal properties. This algorithmic approach implies that automated tools can be implemented. Deriving the specification with a temporal logic is a flexible and practical approach, due to the expressive power of temporal logic. Moreover, temporal logics are popular because they are composed of a reduced number of well-defined constructs that can be easily understood.

In general, a model checker execution results in one of three different scenarios. If the execution returns a positive result, we conclude that the model satisfies the specification. However, we can not formally guarantee that the actual program satisfies the specification, because the model might not be semantically equivalent to the program.

If the execution returns a negative result, we can use the model checking algorithm to generate a program trace representing a potential program error. In practice, the main applications of model checking are model falsifiability and automatic test case generation [7].

In the third scenario, the model checker fails. A possible failing scenario is the case where the model checker consumes all available memory. In general, this situation is caused by the state explosion problem. The state explosion problem [83] is an inherent problem in explicit-state model checking that occurs due to the exponential growth of the state space. In particular, the state explosion problem is a major issue in verification of concurrent programs because of the interleaving semantics of such programs. In the last three decades, several algorithms such as abstraction [28], partial order reduction [82] and counterexample guided abstraction refinement [23] were developed and combined with model checking algorithms to tackle the state explosion problem.

In an alternative failing scenario, the model checker issues an execution timeout. Due to the *halting problem*, it is generally undecidable if the model checker will terminate [47]. Moreover, the complexity of standard verification algorithms is NP-complete.

## 2.2 Bounded Model Checking

Bounded Model Checking (BMC) [7, 8] originated as an alternative to symbolic model checking using binary decision diagrams (BBDs) [14] that suffered from a scalability problem considering the increasing trend in complexity and size of hardware systems.

The approach of BMC consists on generating verification conditions that encode the reachability problem of all property violation states in the system up to a given bound. BMC leverages recent optimizations in boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers, that find (if possible) satisfying assignments to a set of constraints, to remain a scalable approach for increasingly complex systems [29]. SAT/SMT-based BMC generates a first-order propositional formula $\Psi$ given a transition system $M$, a bound $k$, and a property $\phi$ according to the formula:

$$\Psi(M, k, \phi) = I(M) \bigwedge_{i \in [0..k-1]} T_i(M) \wedge [\![\neg\phi]\!]_i \tag{2.1}$$

The general BMC formula above encodes the entire model $M$ ($T_i(M)$) at each bound depth constrained by the initial state $I(M)$. BMC encodes the negation of the given property and uses the SAT/SMT solver to determine if the negation of the property is satisfiable, i.e. if there is an assignment to the formula variables such that the formula evaluates to true. Using the dual relationship between satisfiability and validity, BMC proves if the property in the model is invalid.

In practice, BMC is an iterative process that increases the search depth until resource exhaustion or property violation. This approach has been successfully applied for automatic test-case generation of hardware and software systems [7] .

In this work, we exploit the optimization advances of SMT solvers to tackle the infamous state explosion problem in verification of multi-threaded programs.

### The SMT-LIB Language (v.2)

Recent BMC tools owe their success to improvements in SMT solvers. The SMT-LIB language is a part of the SMT-LIB standard for comparison of SMT solvers. Since its initial proposal in 2003, the language suffered several revisions that led to the SMT-LIB v.2. Currently, several SMT solvers are compliant with the standard language such as Boolector [13], MathSAT 5 [39], Yices [32] and Z3 [29], although some of them do not support the latest version of the SMT-LIB language.

The SMT-LIB language encodes logical formulas in a many-sorted first-order logic [26]. It is a strongly-sorted (typed) language, where each well-formed expression has a unique sort and is well-sorted. It supports polymorphic functions such as equality function (=) and new sort definition.

A SMT-LIB program is composed by a non-empty list of S-expressions with optional comments. A S-expression encloses a command in parenthesis. The grammar production for a SMT-LIB command is presented in Figure 2.1.

⟨command⟩ ⊨ set-logic ⟨logic⟩
| declare-fun ⟨symbol⟩ (⟨sort-expr⟩$^*$) ⟨sort-expr⟩
| define-fun ⟨symbol⟩ ((⟨symbol⟩ ⟨sort-expr⟩)$^*$) ⟨sort-expr⟩ ⟨expr⟩
| declare-sort ⟨symbol⟩ ⟨numeral⟩
| define-sort (⟨symbol⟩$^+$) ⟨expr⟩
| assert ⟨expr⟩
| get-assertions
| check-sat
| get-proof
| get-unsat-core
| get-value ⟨expr⟩$^+$
| get-assignment
| push ⟨numeral⟩
| pop ⟨numeral⟩
| get-option ⟨keyword⟩
| set-option ⟨keyword⟩ ⟨attr-value⟩
| get-info ⟨keyword⟩
| set-info ⟨keyword⟩ ⟨attr-value⟩
| exit

**Fig. 2.1:** SMT-LIB commands

SMT-LIB commands have different cardinality, e.g. a valid SMT-LIB program can only have one *set-logic* command, while the command *set-info* can occur multiple times. Moreover, some commands are defined depending on the current context, e.g. the *get-value* command is only valid when the option *produce-models* is set to true and the result from *check-sat* is sat. The commands *declare-sort* and *define-sort* introduce new sorts by respectively, declaring a new sort and introduce a sort synonym.

The grammar for SMT-LIB expression grammar is described in Figure 2.2. A SMT-LIB expression can be a literal, identifier, function application, quantified ∀ and ∃-expressions, and *let*-expressions used to create more compact formulas.

The command *set-logic* initializes the solver with the specified logic. In this thesis we use the QF_AUFBV logic [16], that supports closed quantifier-free formulas over the theory of bit-vectors and bit-vector arrays.

An example of a SMT-LIB program is presented in Figure 2.3. The program encodes the law of excluded middle using the logic of uninterpreted functions (line

$$
\begin{aligned}
\langle\text{expr}\rangle \models\ & \langle\text{literal}\rangle \\
|\ & \langle\text{identifier}\rangle \\
|\ & \langle\text{identifier}\rangle\ \langle\text{expr}\rangle^{+} \\
|\ & \text{forall ( }(\langle\text{symbol}\rangle\ \langle\text{sort}\rangle)^{+})\ \langle\text{expr}\rangle \\
|\ & \text{exists ( }(\langle\text{symbol}\rangle\ \langle\text{sort}\rangle)^{+})\ \langle\text{expr}\rangle \\
|\ & \text{let ( }(\langle\text{symbol}\rangle\ \langle\text{sort}\rangle)^{+})\ \langle\text{expr}\rangle \\
\langle\text{identifier}\rangle \models\ & \langle\text{symbol}\rangle \\
|\ & _\ \langle\text{symbol}\rangle\ \langle\text{numeral}\rangle^{+} \\
|\ & \text{as}\ \langle\text{identifier}\rangle\ \mathit{pnsort} \\
\langle\text{sort-expr}\rangle \models\ & \langle\text{sort}\rangle \\
|\ & \langle\text{symbol}\rangle\ \langle\text{sort-expr}\rangle^{+}
\end{aligned}
$$

**Fig. 2.2:** SMT-LIB expressions

1). In line 2, we declare a boolean value $p$ and in line 3, we assert $p \wedge \neg p$. In line 4, we ask the SMT solver to check the satisfiability of the current stack of assertions and finally in line 5, we terminate the SMT solver. The result of a SMT solver execution is *sat*, satisfiable.

```
(set-logic QF_UF)
(declare-fun p () Bool)
(assert (or p (not p)))
(check-sat)
(exit)
```

**Fig. 2.3:** SMT-Lib Encoding of the Law of excluded middle

## 2.3 Low-Level Virtual Machine

LLVM is a popular and growing compiler framework that supports aggressive multistage optimizations to overcome known problems of traditional compilation techniques [52]. The framework was initially designed to be a flexible, well-documented and transparent infrastructure for research projects in the compiler domain.

From a compiler designer perspective, LLVM offers several advantages. The architecture of the framework, represented in Figure 2.4, was designed to be dependent on the Intermediate Representation (IR) language to ease component reusability. This design decision simplifies the compiler construction process since compiler implementors can reuse LLVM's backend. Using the framework, the main compiler

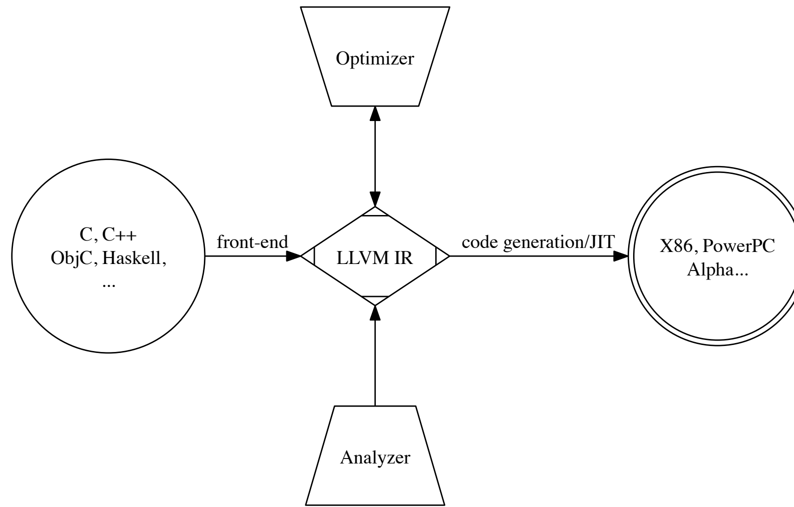


**Fig. 2.4:** LLVM Architecture

construction component is the front-end implementation. With the upward trend of domain specific languages development, LLVM is a valuable framework to implement an efficient compiler with limited resources. Moreover, LLVM provides code generation for several architectures, and although primarily focused on C and C++, other programming languages front-ends have been implemented.

LLVM is used in multiple projects due to its scalability and competitive performance results against industrial and research compilers [58]. Furthermore, there is an active community of users and developers, and a continuous interest of the academic community with hundreds of research papers up to date.

### LLVM Intermediate Representation

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 [74] 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, 53].

In Figure 2.5, we describe some of the productions that compose the implemented LLVM IR grammar. We describe a simplified grammar since for our current

$$
\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^{*} \\
&\quad \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 \\
&\quad \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 \\
&\quad \mid\ [\ \langle\text{int}\rangle\ \times\ \langle\text{ty}\rangle\ ]\ \mid\ \langle\ \langle\text{int}\rangle\ \times\ \langle\text{ty}\rangle\ \rangle \\
&\quad \mid\ \langle\text{ty}\rangle\text{*}\ \mid\ \langle\text{ty}\rangle^{*} \rightarrow \langle\text{ty}\rangle\ \mid\ \{\ \langle\text{ty}\rangle^{*}\ \}
\end{aligned}
$$

**Fig. 2.5:** Abstract LLVM IR Grammar

verification algorithms we are not interested in attributes such as linkage, section or garbage collection.

A 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 2.6, %"$class.std :: ios_base :: Init$" 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 2.6, 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 value choice between basic blocks.

```
1  %"class.std::ios_base::Init" = type { i8 }
2  @M1 = linkonce_odr constant [2 x i8] c"M1"
3  %.0 = phi i8* [%.pre1, %bb1],[%tmp8, %bb4]
4   br label %bb14
```

**Fig. 2.6:** LLVM IR example

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

## 2.4 Shared Memory Concurrency

Concurrency is a property of software systems that leverages the state of the art preemptive operating system schedulers and hardware architectures to speedup the execution of the programs by executing more instructions than sequential systems. Currently there are two predominant models in concurrent systems according to their inherent memory model: shared-memory concurrency and message passing concurrency. Both models have the same expressive power since we can implement a message passing concurrent strategy using shared-memory concurrency and vice versa. In message passing concurrency, the concurrent components, processes or threads, have their own memory and communicate with other processes through messages. Shared-memory concurrency is characterized by shared memory blocks between threads. Although this model arguably requires less memory than message passing, verification of programs that use this model is particularly harder because synchronization errors are more intricate. In this work we focus on two implementations of shared memory concurrency: the Pthread library and SystemC.

### Pthread

The Pthread library [64] is the most popular library for implementation of multi-threaded C/C++ programs. The library provides an extensive API for thread management, scheduling, synchronization, signaling and cancellation.

Table 2.1 summarizes the Pthread library functions.

The Pthread library does not specifies any scheduling algorithm, although it supports functions to change the priority of the processes. The thread scheduling is accomplished by the operating system scheduler, which typically is preemptive, i.e. the scheduler can stop a running thread in any instruction location.

### Pthread Example

Figure 2.7 presents a Pthread example from [27]. The main function (lines 22 to 30) creates two threads $Tx$ and $Ty$ that execute without any synchronization mechanism. In this example, since the function *nondet_uint* can return an integer bigger than 10, if thread $Ty$ executes first and re-assigns the shared variable $x$ to 3, when execution resumes to $Tx$ the assertion will fail. We will use this example to describe our Bounded Model Checker throughout this thesis.

| Construct | Available Functions |
|---|---|
| Management | *pthread_create*, *pthread_join*, *pthread_exit*, *pthread_cancel* |
| Mutex | *pthread_mutex_init*, *pthread_mutex_destroy*, *pthread_mutex_lock*, *pthread_mutex_trylock*, *pthread_mutex_unlock* |
| Conditional Variable | *pthread_cond_init*, *pthread_cond_signal*, *pthread_cond_wait* |

**Table 2.1:** Common Pthread Library Functions

## SystemC

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 standard enables system level verification and debugging as well as hardware/software co-design, architectural exploration, and power/performance analysis. 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 $e3$.

```
1  #define N 10
2
3  int nondet_uint();
4
5  int a[N], i, j=1, x=2;
6
7  void *Tx(void *arg){
8    if(x>2){
9      assert(i>=0 && i<N);
10     a[i]=*((int *)arg);
11   }
12 }
13
14 void *Ty(void *arg){
15   if(x>3)
16     a[j]=*((int *)arg);
17   else{
18     x=3;
19   }
20 }
21
22 int main(){
23   pthread_t id1, id2;
24   int arg1=10, arg2=20;
25
26   i=nondet_uint();
27
28   pthread_create(&id1, NULL, Tx, &arg1);
29   pthread_create(&id2, NULL, Ty, &arg2);
30 }
```

**Fig. 2.7:** Pthread example

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 nonblocking transport interfaces. Different levels of model abstraction are 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). Synchronization is also established through instantiating *sc_semaphore* and *sc_mutex* objects, which provide *wait*, *trywait*, *post* and *lock*, *trylock*, *unlock* functions, respectively. Note that, in SystemC, communication between processes is established either by explicit concurrency functions or by shared variables.

Table 2.2 summarizes SystemC concurrency functions.

| **Construct** | **Available Functions** |
|---|---|
| Event | *notify*, *wait*, *next_trigger* |
| Channel | *read*, *write*, *put*, *get*, *peek*, *nb_put*, *nb_get*, *nb_peek*, *b_transport*, *nb_transport_fw*, *nb_transport_bw* |
| Semaphore | *wait*, *trywait*, *post* |
| Mutex | *lock*, *trylock*, *unlock* |

**Table 2.2:** SystemC Concurrency Functions

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.

In [43], 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.

### SystemC Example

Figure 2.8 illustrates a SystemC design. We will use this example throughout this thesis to demonstrate our second framework application for automated testbench generation using mutation testing. The code in Figure 2.8 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. We are interested in an automatic method for generation of unit tests that explore different SystemC simulations.

## 2.5 Mutation Testing

Mutation testing is a commonly used software testing technique to measure the quality of testbenches [3]. Mutation 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 2.1.* Mutation operator $MO$ for *wait(e)*:

$$\begin{aligned}&\text{wait(e)} \rightarrow \text{wait(1,SC_NS);}\\&\text{wait(e)} \rightarrow \text{e.notify();}\\&\text{wait(e)} \rightarrow \epsilon;\end{aligned}$$

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 2.2.** *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 $P(t)$.*

**Definition 2.3.** *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.

```
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  };
```

**Fig. 2.8:** A SystemC Design

**Definition 2.4.** *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 muta-

**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;

tion. 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 2.2 (line 6). Finally, we compute the mutation coverage rate using Definition 2.4. 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

# Related Work

*To understand a program, you must become both the machine and the program.*

Alan Perlis

In this chapter, we describe previous approaches related to our research questions. Section 3.1 presents previous formalizations of LLVM IR. Section 3.2 provides a general overview on verification of LLVM IR for sequential and concurrent programs focused on bounded model checkers for C and Pthread programs and also previous approaches for verification of SystemC modules. Section 3.3 provides related work in mutation testing and Section 3.4 describes automated test-case generation tools using model checking techniques and mutation testing.

## 3.1 Formalization of LLVM IR

Recently, several tools have formalized the LLVM IR language in the context of certified compilers. Vellvm [88] is a Coq framework to formally prove transformations over LLVM IR programs in a first attempt to a formally verified LLVM compiler. This framework was inspired in CompCert [18], a ANSI-C verified compiler. LLVM M.D. [81] is a compiler research project to detect semantic changes in the input program produced by the optimizer. LLVM M.D. is implemented in Haskell, but their model generation is based on a parser of the LLVM IR language while we use the bindings for the LLVM api which is a more reliable solution since the disassembled byte code is only for human reader purpose and may contain errors.

## 3.2 Model Checking

In the last decade, several tools applied bounded model checking to verify C and C++ programs. The initial tools, CBMC [24] and F-Soft [46] focused on sequential programs. SMT-CBMC [4] proposed a combination of bounded model checking with SMT solvers to use their expressive power. TCBMC [71] and ESBMC [27] apply bounded model checking for Pthread C programs. TCBMC is limited to concurrent

programs with two threads. ESBMC reuses CBMC front-end to generate verification conditions and supports several encoding approaches to produce a boolean formula. SATABS [21] performs verification of multi-threaded software with shared variables with a CEGAR approach on sequential GOTO-programs translated from the original concurrent program [23].

Recently, formal verification tools target intermediate languages. VCC [25] is an assertion verifier for concurrent C programs. VCC uses the SMT solver z3 to analyze verification conditions (VCs) generated from Boogie. Boogie [6] produces VCs for programs represented in an intermediate verification language also called Boogie that is previously translated from high-level languages such as C, C# or Spec#. Concerning LLVM IR, LLBMC [61] applies SMT based bounded model checking for sequential C/C++ programs. LAV [84] combines symbolic execution and SMT based bounded model checking for bug finding in sequential C programs. Our backwards static analysis reachability method for SystemC is a symbolic execution approach similar to Dijkstra weakest pre-condition formulation [31]. A formal weakest pre-conditional model to reason about assembly language programs can be found in [54].

Concerning SystemC, PinaVM [60] is a SystemC front-end that translates LLVM IR code into a usable intermediate format for several verification tools. We believe that our approach is more flexible since we do not have to maintain another intermediate representation. Moreover, we do not introduce overhead and potential implementation bugs with the translation process. Kratos [19] is a model checker for SystemC that uses abstraction techniques to perform concurrent or sequential analysis over GOTO-programs translated from SystemC designs. Scoot [9] is a static analyzer for SystemC that extracts a multipurpose semantic model. Because of the dynamic nature of the SystemC library implementation, Scoot requires library modifications for correct usage.

A common disadvantage of model checkers that operate at the level of LLVM IR is the fact that they depend on a specific version of LLVM which reduces the usability of the tool. Several tools have installation problems and furthermore the user might be interested in verifying a feature of a newer version of LLVM IR. SMT-based bounded model checkers also use the API of the SMT solvers. Therefore, they depend on older versions of the SMT solver that do not leverage newer optimizations. Our approach uses the current version of LLVM and our installation process does not require a specific package of LLVM. Moreover, we generate SMT-LIB v.2 language programs that can be used with all SMT solvers that are compliant with the STM-LIB standard. Hence, we can compare performance of SMT solvers in our benchmarks.

## 3.3 Mutation Testing

Mutation testing is an approach based on software testing to assess the quality of test suites [15, 66, 67]. It has been defined for programming languages such as Java [59, 12], state machines [69] and hardware description languages such as Verilog [41]. A detailed survey on applications of mutation testing can be found in [49].

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 [35, 85, 57].

Standard mutation frameworks implement optimization to reduce the high number of mutants generated. Prior work on mutant equivalence [75] and higher order mutation testing [48] have tackled this problem.

A mutation model for SystemC TLM 2.0 communication interfaces have been defined in [10, 11] and this has been extended to all SystemC concurrency constructs in [76]. 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.

## 3.4 Automated Test-Case Generation

Automated test case generation is a natural application of mutation testing. Early research on mutation testing [65] developed methods based on constraint solving to solve the input-mutant reachability problem. In [30], this work is extended with a necessity condition encoding the mutant effects on states of the program.

$\mu$Test [36] 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 [73] uses SMT/bounded model checking approach using mutation analysis to generate test cases from ANSI-C programs. KLEE [17] and KLOVER [55], perform symbolic execution of sequential C and C++ programs represented with the LLVM byte code for automatic test suite generation. Recently, GKLEE [56], 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.

# 4

# LLVM Verification Framework

> *One day Chao-Chou fell down in the snow, and called out: "Help me! Help Me!" A monk came and lay down beside him. Chao-Chou got up and went away.*
>
> Zen Kōan

In this chapter, we present our LLVM verification framework, *llvmvf*. Section 4.1 describes the design goals and architecture of the framework. Section 4.2 provides implementation details on the front-end of the framework to formalize LLVM IR and the extraction of an abstract concurrent model under verification. In Section 4.3, we detail the three main applications of *llvmvf*: an SMT-based Bounded Model Checker (4.3.1), a Mutation Testing framework (4.3.2) and a technique for generation of SystemC TLM testbenches using Mutation Testing (4.3.3).

## 4.1 Architecture

The architecture of *llvmvf* is designed to create a compact model of the byte code from the original high level program. In Figure 4.1, we present the several phases of our framework. The input of our flow is a LLVM byte code file correspondent from the high-level program under verification. For example, in the case of C/C++ we can use a compiler from the clang family [20]. Since the implementation of SystemC and other C++ libraries is heavily based on template programming, applying various optimizations to obtain more compact byte code modules removes some overhead from the infrastructure.

We start, phase (a), by using the optimizer to transform the current byte code into a form that is suitable for our analysis. In total, we apply 18 LLVM passes over the input byte code file. The LLVM Pass Framework is an infrastructure to structurally implement byte-code traversals at different levels of abstraction for compiler transformations/optimizations and analysis. 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 sequential programs. For the purpose of our analysis we want to obtain byte code that has no cycles in the basic block graph. Simplification

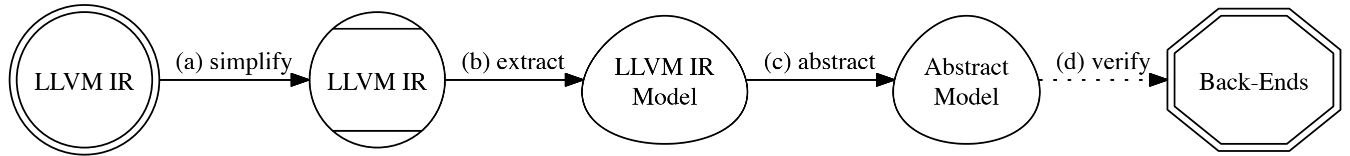


**Fig. 4.1:** *llvmvf* Architecture

transformations aim at simplifying our formalization. We reduce the byte code size eliminating LLVM IR constructs not supported by our analysis such as invoke or switch instructions. Furthermore, we lift stack operations by promote the stack to registers and use an LLVM pass to name all nameless identifiers.

## 4.2 Front-End

In phase (b) of Figure 4.1, we use a binding mechanism to extract a LLVM IR model that follows the abstract grammar in Figure 2.5 (4.2.1). The model generated by the extraction function is refined into an abstract concurrent model that contains information about the architecture and the behavior of the processes (4.2.2). To extend the framework to a new concurrent mechanism or language, the only component that a user has to implement is the refinement from LLVM IR to the abstract concurrent model.

### 4.2.1 Extraction

The extraction function uses a double binding schema to call a LLVM API function from Haskell as Figure 4.2 illustrates.

An alternative solution to formalize LLVM IR in Haskell would be to create a parser from the dissembler output. Our approach is more reliable because the output from the dissembler might contain errors but we experienced a scalability problem from the Haskell package *llvm-base* already developed. This problem arose because of API changes in LLVM with newer versions and it required the implementation of a complete set of bindings both on the Haskell and *c* binding libraries.

For example, to call the LLVM API function *getModuleIdentifier* that retrieves the name of the byte code module:

```
const std::string &getModuleIdentifier() const {
  return ModuleID;
}
```

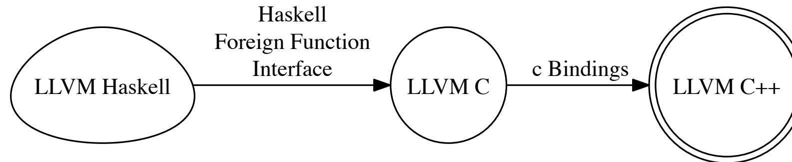


**Fig. 4.2:** Calling the LLVM API in Haskell.

First we need to create the c binding:

```
const char *LLVMGetModuleIdentifier(LLVMModuleRef M){
  return unwrap(M)->getModuleIdentifier().c_str();
}
```

And finally, the Haskell binding:

```
foreign import ccall unsafe "LLVMGetModuleIdentifier"
    getModuleIdentifier :: ModuleRef -> IO CString
```

We have extended the functionality of the current Haskell and C binding infrastructure with about 25 LLVM API function calls.

### 4.2.2 Abstraction

The LLVM IR model extracted implicitly represents a concurrent model through function calls to concurrent libraries such as Pthread and the SystemC kernel. We refine the LLVM IR model as a synchronous model of concurrent program [37]. A synchronous model is a concurrency model with explicit scheduler where only one thread can execute at a time. We use an Haskell type class to denote the class of synchronous concurrent models:

```
class SCModel t where
  model :: Module -> Model t
```

The *model* function transforms a LLVM model *Module* into an abstract *Model* presented in Figure 4.3.

We record the main function in the *Model* to be able to map arguments to the threads parameters. A *Process* is a model of the original LLVM *Function* with concurrent productions such as *CreateThread* or *MutexLock*. Each instruction is labelled with an unique attribute representing the program counter.

A model represents a concurrent control flow graph. A concurrent control flow graph is an annotated control flow graph with scheduling information such as thread creation.

Figure 4.4 shows the concurrent control flow graph for the Pthread program in Figure 2.7. The diamond node represents the scheduler. Each instruction is annotated with a program counter in the left side. For Pthread programs, the abstraction

```
data Model t = Model { types   :: NamedTypes
                     , globals :: Globals
                     , main    :: Process
                     , procs   :: Processes
                     , sched   :: Scheduler
                     }

data Process   = Process { ident  :: String
                         , unProc :: Function }

data Scheduler = Preemptive
               | NonPreemptive
```

**Fig. 4.3:** Abstract Model Declaration.

algorithm traverses the main function calculating the intra-procedural control flow and searching for scheduling operations. In this example, the LLVM instruction at program counter 38:

```
call i32 @pthread_create(i64* %id1, %union.pthread_attr_t*
    null, i8* (i8*)* @Tx, i8* %tmp1) nounwind
```

Is translated to:

```
Create_Thread "Tx" (Identifier "tmp1" (TyPointer (TyInt 8))
```

## 4.3 Back-End

In this Section we describe phase (d) of Figure 4.1. In this thesis, we devised two applications of *llvmvf*: an SMT-based Bounded Model Checker (4.3.1) for Pthread programs using the abstract model described in the previous section; and a backwards static analysis procedure based on our LLVM IR Mutation Testing framework (4.3.2) for generation of SystemC TLM testbenches framework (4.3.3).

### 4.3.1 SMT-based Bounded Model Checker

The abstract model described in the previous section represents a state transition system. Given a transition system $M$, a bound $k$ and a property $\phi$, a SMT-based BMC generates the formula:

$$\Psi(M,k,\phi) = I(M) \bigwedge_{i\in[0..k-1]} T_i(M) \wedge [\![\neg\phi]\!]_i \tag{4.1}$$

Although we have a partial implementation for a liner encoding of LTL properties, we currently focus on verifying user assertions. Therefore, we are able to statically determining the error states since they are calls to the function *__assert_fail* and reduce Formula 4.1 to a reachability problem:

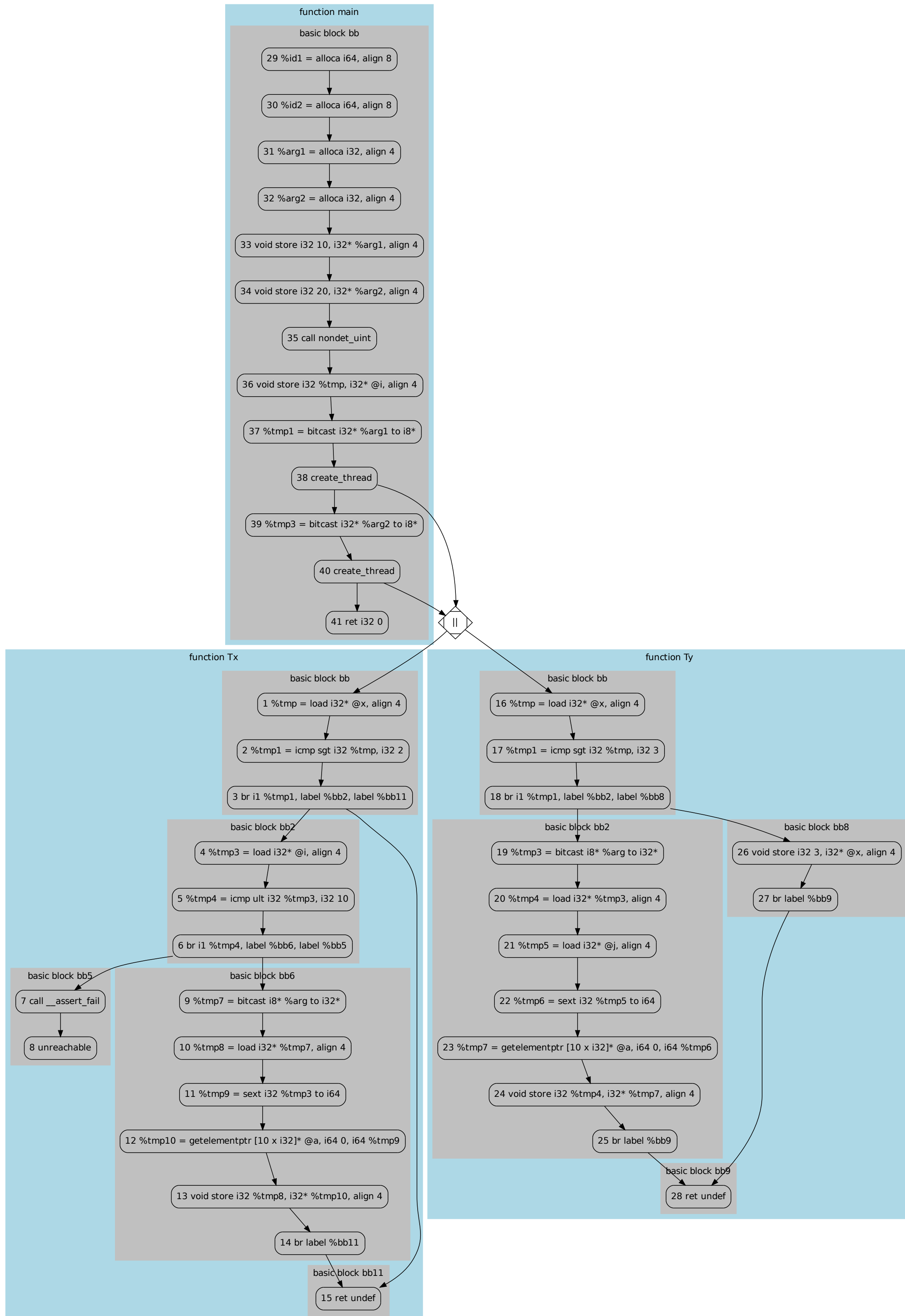


**Fig. 4.4:** Concurrent Control Flow Graph for Example 2.7.

$$\Psi(M,k) = I(M) \bigwedge_{i\in[0..k-1]} T_i(M_{procs}) \wedge \overbrace{\bigvee_{f\in F(M)} \bigvee_{j\in[0..tc-1]} \Pi_j pc_i = f}^{\text{Current state is a fail state}} \quad (4.2)$$

In Formula 4.2, $tc$ is the number of threads and $\Pi_j pc_i$ is the program counter in bound $i$ of the thread $\Pi_j$. The formula encodes a breadth-first search for an error state $f$ in the set of error states of M, $F(M)$, up to the specified bound $k$ in the transition system $M$.

### Model Encoding

Following the Formula 4.2, we start by encoding the initial state of $M$, $I(M)$, as:

$$I(M) = T(M_{types}) \wedge T(M_{globals}) \wedge T(M_{main}) \bigwedge_{pci\in PC_{i_0}(M_{procs})} pc_{\Pi_{i_0}} = pci \quad (4.3)$$

In Formula 4.3:

- $T(M_{types})$ represents the encoding of all the types used in the program;
- $T(M_{globals})$ represents the encoding of all global/shared variables between threads;
- $T(M_{main})$ represents the sequential encoding of the main function;
- $pc_{\Pi_{i_0}} = pci$ assigns a fresh variable that represents the $pc$ of thread $\Pi_i$ in the initial state 0 to the initial program counter of thread $\Pi_i$.

In the encoding $T_i(M_{types})$, we define the type $iN$ as a new sort $IN$ that is a synonym for a $BitVector$ of size N, except for $i1$ that is mapped to $Bool$. For example, the LLVM type $i8$ is encoded as:

$$\text{(define-sort } I8 \text{ () (_ BitVec 8))} \quad (4.4)$$

Arrays and Vectors produce in the same sort:

$$T_i(Array\ n\ ty) = \text{(define-sort } Arraynty \text{ () } (Array \text{ (_ BitVec } bsize(n)) \ sort(ty))) \quad (4.5)$$

The function $sort$ retrieves the sort of the type and the function $bsize$ generates a $BitVector$ of the minimum size required to encode the number of elements in the Array. For example, an Array with 4 elements of type $i8$ is encoded as:

$$\text{(define-sort } Array4I8 \text{ () (Array (_ BitVec 3) } I8)) \quad (4.6)$$

For simplicity, pointer types are defined as the sort of the type they point to. Structures are encoded as pairs of sorts. For example, the type:

```
%union.pthread_attr_t = type { i64, [48 x i8] }
```

is encoded as:

$$\begin{aligned}&\text{(declare-sort } Pair\ 2)\\&\text{(define-sort } union.pthread_attr_t\ ()\ (Pair\ I64\ Array48I8))\end{aligned} \tag{4.7}$$

Encoding for the remaining LLVM types is not supported yet. Note that $\wedge$ is overloaded to concatenate both encodings in $T(M_{types}) \wedge T(M_{globals})$ since $T(M_{types})$ does not generate predicates but SMT-LIB commands.

In the encoding $T_i(M_{globals})$, we declare the global variables and if they are initialized we generate assertions and a predicate *lid*, where *id* is the variable name, that keeps track of the last program counter where an assignment was performed. By default the program counter is 0. The global declaration:

```
@x = global i32 2, align 4
```

is encoded as:

$$\begin{aligned}&\text{(declare-fun } x\ ()\ I32)\\&\text{(declare-fun } lx\ ()\ I32)\\&\text{(assert } (and\ (=\ x\ (_\ bv2\ 32))\ (=\ lx\ (_\ bv0\ 32))))\end{aligned} \tag{4.8}$$

We add extra predicates to $I(M)$ to control possible *store* instructions related to global variables in the main function.

The encoding of $T(M_{main})$ is given by the formula:

$$T(M_{main}) = \bigwedge_{pc_i \in PC_i(M_{main})} (ppc_i = pc_i) \wedge [\![t_i, next(pc_i)]\!]_{seq} \tag{4.9}$$

In Formula 4.9, $pc_i$ represents the program counter at instruction $i$; $ppc_i$ is a predicate that represents a constraint to enable the effect of the instruction if the current $ppc$ is equal to $pc_i$. The function *next* retrieves the set of program counter successors of $pc_i$. We define the encoding function $[\![_]\!]_{seq}$ for sequential programs as follows:

$$[\![t_i, \emptyset]\!]_{seq} = [\![t_i]\!] \tag{4.10}$$

$$[\![(Br\ c), \{pc_t, pc_f\}]\!]_{seq} = (c \wedge ppc_{i+1} = pc_t) \vee (\neg c \wedge ppc_{i+1} = pc_f) \tag{4.11}$$

$$[\![t_i, \{pc_{i+1}\}]\!]_{seq} = [\![t_i]\!] \wedge ppc_{i+1} = pc_{i+1} \tag{4.12}$$

The first case, Formula 4.10, encodes an instruction that has no successors by simply encoding the instruction without the constraints for the successors. The second case, Formula 4.11, encodes the conditional branch instruction. In this case, we generate two alternatives according to the condition predicate for the true and false branches. In the third case, Formula 4.12, we encode an instruction with one successor. The encoding is a conjunction of the encoding of the instruction and a constraint that determines the next program counter value, $ppc_{i+1}$.

The encoding function $[\![t_i]\!]$ is straightforward for binary, bit-level, casting and address calculation instructions, since these instructions are part of the theory of arrays and fixed-sized bitvectors or can easily be encoded. For example, the LLVM instruction *add* is translated into the instruction *bvadd* in the theory of bitvectors.

The address calculation instruction *getelementptr* is translated into the instruction *select* in the theory of arrays.

The encoding of a set of threads at a given bound depth $i$ is given by the formula:

$$T_i(M_{procs}) = \overbrace{\bigwedge_{j\in[0..tc-2]} \text{xor } \Pi_{i_j} \ \Pi_{i_{j+1}}}^{\text{scheduler}} \wedge \underbrace{\bigvee_{j\in[0..tc-1]} \bigvee_{pc_i\in PC(\Pi_j)} \Pi_{i_j} \wedge (\Pi pc_i = pc_i) \wedge [\![t_i, next(pc_i)]\!]_{con}}_{\text{thread instruction encoding}} \tag{4.13}$$

In Formula 4.13: $\Pi_{i_j}$ is a predicate that represents that thread $\Pi_j$ is enabled in bound $i$. The left sub-formula of the conjunction in Formula 4.13 encodes a preemptive scheduler that at each bound depth $i$ chooses a single thread to execute. The right sub-formula of the conjunction in Formula 4.13 encodes all instructions in every thread. One of the instructions is encoded if the thread that the instruction belongs is chosen and the current location at this bound depth is equal to the current program counter of the thread.

We define the encoding function $[\![_]\!]_{con}$ for concurrent programs as follows:

$$[\![t_i, \emptyset]\!]_{con} = [\![t_i]\!] \tag{4.14}$$

$$[\![(Br\ c), \{pc_t, pc_f\}]\!]_{con} = (c \wedge \Pi_j pc_{i+1} = pc_t) \vee (\neg c \wedge \Pi_j pc_{i+1} = pc_f) \wedge \bigwedge_{g\in[0..tc-1]\wedge j\neq g} \Pi_g pc_{i+1} = \Pi_g pc_i \tag{4.15}$$

$$[\![t_i, \{pc_{i+1}\}]\!]_{con} = [\![t_i]\!] \wedge (\Pi_j pc_{i+1} = pc_{i+1}) \wedge \bigwedge_{g\in[0..tc-1]\wedge j\neq g} \Pi_g pc_{i+1} = \Pi_g pc_i \tag{4.16}$$

The concurrent encoding $[\![_]\!]_{con}$ is an extension to the sequential encoding to handle the various program counters of the threads. To encode memory access instructions related to global variables, we introduce fresh variables to each store of a global variable and a variable associated with the global variable that represents the program counter of the store.

In Figure 4.4, the encoding of the *store* instruction with pc of 26 is:

$$Typc0 = 26 \wedge x0 = 3 \wedge pi1 = pi0 \wedge px1 = 26 \wedge Typc1 = 27 \wedge Txpc1 = Txpc0 \tag{4.17}$$

*pi0* and *pi1* are the predicates related to store instructions of variable *i*. Since this instruction is a store of *x*, we create a new fresh variable *x0* and assign to *px1* the value of the current program counter.

The encoding of the *load* instruction with pc of 1 is:

$$Txpc0 = 1 \wedge ((px0 = lx \wedge Txtmp = x) \vee (px0 = 26 \wedge Txtmp = x0)) \wedge pi1 = pi0 \wedge px1 = px0 \wedge Txpc1 = 2 \wedge Typc1 = Typc0 \tag{4.18}$$

In this formula, we assign a value to *Txtmp* depending on the value of *px0*, the variable that represents the value of the last store. In this case, *px0* is either *lx* which is the program counter for the global variable initialization or *px0* is 26 that corresponds to the store above. This approach can also be used to encode the *phi* instruction.

### Encoding Mutex Locking Operations

We model Pthread mutexes as predicates and for each bound depth we generate fresh predicates that represent the current state of the mutex. We consider the value of the mutex predicate as $false$ if the current state of the mutex is lock and $true$ otherwise. We encode $mutex_lock$ and $mutex_unlock$ as:

$$[\![mutex_lock\ m]\!]_{con} = m_0 \ \wedge \ \neg m_1 \tag{4.19}$$

$$[\![mutex_unlock\ m]\!]_{con} = \neg m_0 \ \wedge \ m_1 \tag{4.20}$$

The complexity of our encoding algorithm is $k \times size(M)$, where $k$ is the bound and $size(M)$ is the number of instructions in model $M$.

### 4.3.2 Mutation Framework

In this section, we describe our mutation framework that operates at the LLVM IR level. Consider the instance of the framework for a SystemC metamutant generation flow that is illustrated in Figure 4.5.

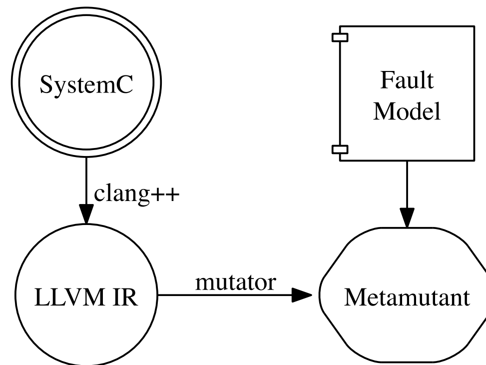


**Fig. 4.5:** Metamutant Generation Flow

Given a SystemC program, we use clang++ [20], a C++ compiler front-end for LLVM, to generate LLVM IR byte 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,

**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**

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;
}
```

**Fig. 4.6:** Mutant Example for SystemC Design in Figure 2.8

Previous work [76] have investigated the relationship between SystemC TLM synchronization operations and common concurrent error patterns such as deadlocks, lost notifies or data races. Figure 4.6 shows an example of a mutant generated from the SystemC design in Figure 2.8. 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.

### 4.3.3 Generation of SystemC TLM Testbenches Using Mutation Testing

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. We are interested 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 approach is applicable to LLVM IR programs in general.

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

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

  return 0;
}
```

**Fig. 4.7:** *sc_main* of SystemC design in Figure 2.8

In Figure 4.7, we present a simple example of a simulation model for the SystemC design in Figure 2.8. 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  %struct.M1 = type { %"class.sc_core::sc_module", %"class.sc_core::sc_event",
       i8, i8 }
2  @.str = private unnamed_addr constant [3 x i8] c"M1\00", align 1
3  define i32 @sc_main(i32 %argc, i8** nocapture %argv) uwtable {
4    ; ...
5    %M1 = alloca %struct.M1, align 8
6    %tmp1 = alloca %"class.sc_core::sc_module_name", align 8
7    ; %tmp5 ~= argv[1], %tmp9 ~= argv[2]
8    call void @_ZN7sc_core14sc_module_nameC1EPKc(%"class.sc_core::sc_module_name
         "* %tmp1, i8* getelementptr inbounds ([3 x i8]* @.str, i64 0, i64 0))
9    call void @_ZN2M1C2EN7sc_core14sc_module_nameEbb(%struct.M1* %M1, %"
         class.sc_core::sc_module_name"* undef, i1 zeroext %tmp5, i1 zeroext %
         tmp9)
10   call void @_ZN7sc_core14sc_module_nameD1Ev(%"class.sc_core::sc_module_name"*
          %tmp1)
11   call void @_ZN7sc_core8sc_startEv()
12
13 ; ...
14 }
```

**Fig. 4.8:** LLVM IR of *sc_main*

Figure 4.8 lists the LLVM byte code relevant to *sc_main*.

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).

```
1  Module
2   [NmdTy (Local "struct.M1") TyStruct (Local "struct.M1") [SyscMod, SyscEv, (
        TyI 8), (TyI 8)]]
3   [Global (Global ".str") (TyArray 3 (TyI 8)) (Const (CString "M1\00"))]
4   [FunctionDef (Global "sc_main") (TyI 32) [Param (Local "argc") (TyI 32),
        Param (Local "argv") (TyPtr (TyPtr (TyI 8)))]
5    [BasicBlock
6     [Alloca (Local "M1") (TyStruct (Local "struct.M1") ...)
7     ,Alloca (Local "tmp1") (TyStruct (Local "class.sc_core::sc_module_name")
         ...)
8     ,...
9     ,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)]
10    ,Call TyVoid (Global "M1::M1") [Ident (Local "M1"), Const UndefC, Ident (
         Local "tmp5"), Ident (Local "tmp5")]
11    ,Call TyVoid (Global "sc_core::sc_module_name::~sc_module_name()") [Ident (
         Local "tmp1")]
12    ,Call TyVoid (Global "sc_core::sc_start()") []
13    ]
14   ]
15  ]
```

**Fig. 4.9:** *sc_main* Model

We start by generating a metamutant byte code file of a SystemC design using the mutation framework described previously. Using the front-end of our framework we extract a SystemC model by statically analyze the elaboration phase of a SystemC simulation model. The 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.

## Encoding Reachability Problem

Given a mutated instruction $i$ in a basic block $\beta_i$, and a SystemC model $M$, we generate the following SMT formula in the theory of bit-vectors and arrays:

$$\Psi(i, M) = \overbrace{\otimes(i, M)}^{\text{inter } \beta_i \text{ analysis}} \wedge \underbrace{\oplus(i, M)}_{\text{intra } \beta_i \text{ analysis}} \tag{4.21}$$

Formula 4.21 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.

$$\otimes(i, M) = \bigvee_{\beta' \in \Pi(\beta_i)} \odot([\![\tau_{\beta'},\ \beta_i]\!],\ \Pi(\tau_{\beta'}),\ M) \wedge \overbrace{\otimes(\tau_{\beta'},\ M)}^{\text{recursion}} \tag{4.22}$$

To reach the mutated basic block $\beta_i$, 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 $\beta_i$. The initial predicate is given by $[\![\tau_{\beta'},\ \beta_i]\!]$ (Formula 4.23), where $\tau_{\beta'}$ is the terminator instruction of the basic block $\beta'$.

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

Note that $\Pi$ 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.

$$[\![\tau_{\beta'},\ \beta_i]\!] = \begin{cases} true, & (\tau_{\beta'} = \text{br } \beta_\kappa) \wedge \beta_i \equiv \beta_\kappa \\ \nu \equiv 1, & (\tau_{\beta'} = \text{br } \nu, \beta_t\ \beta_f) \wedge \beta_i \equiv \beta_t \\ \nu \equiv 0, & (\tau_{\beta'} = \text{br } \nu, \beta_t\ \beta_f) \wedge \beta_i \equiv \beta_f \\ false, & \text{otherwise} \end{cases} \tag{4.23}$$

$\odot$ 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 $\epsilon$ with the current predicate $\psi$. The interpretation function $[\![\ ]\!]$ encodes the instruction in a simple flat memory model supported by SMT solvers and applies logical conjunction with the current predicate. If the instruction to be encoded is a mutated instruction, we apply function $\Psi$ to generate the correspondent SMT formula. We can use memoisation techniques to optimize our implementation.

$$\odot(\psi,\ \emptyset,\ M) = \psi \tag{4.24}$$

$$\odot(\psi,\ \epsilon \cup \Upsilon,\ M) = \odot([\![\epsilon,\ \psi,\ M]\!],\ \Upsilon,\ M) \tag{4.25}$$

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 4.26, 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 $\beta_i$ (operator $\uplus$). The function $\Lambda(i)$ returns the set of predecessor operations in the basic block that may lead to a deadlock situation.

$$\oplus(i, M) = \uplus_i(i, M) \bigwedge_{i' \in \Lambda(i)} \uplus(i', M) \tag{4.26}$$

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 2.8, $T1 < T2$.

$$\uplus_i(i, M) = \begin{cases} true & , T_i \text{ minimal } \prec_{proc} \\ \bigvee_{T<T_i} \Psi(i^{-1}, M) & , \text{ otherwise} \end{cases} \tag{4.27}$$

Operator $\uplus_i$ in Formula 4.27, 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 4.28, encodes the constraints for previous $i^{-1}$ instructions in the basic block.

$$\uplus(i, M) = \bigvee_{T<T_i} \Psi(i^{-1}, M) \tag{4.28}$$

### 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*. If the formula contains free variables, the SMT solver will choose a default value according to its type.

We demonstrate our method with the mutant of Figure 4.6. In Figure 4.10, 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)
```

**Fig. 4.10:** 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 $M$ for this example.

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

Since the basic block predecessor of *bb*4 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 4.23 to further reduce $\otimes$ to:

$$\odot(\text{tmp3} = 0,\ \Upsilon,\ M) = \text{struct.M1[2]} \neq 0 \tag{4.29}$$

$\Upsilon$ is the set of instructions from lines 5 to 2. In 4.30, we show the step-by-step analysis execution for Equation 4.29.

$$\begin{aligned}
&\odot(\%\text{tmp3} \equiv 0,\ \Pi(\tau_{\beta'}),\ M)\\
&= \odot([\![\%\text{tmp3} = \ldots,\ \%\text{tmp3} \equiv 0,\ M]\!],\ \Upsilon_1,\ M)\\
&[\![\%\text{tmp3} = \text{icmp eq i8 \%tmp2, 0},\ \%\text{tmp3} \equiv 0,\ M]\!] = \%\text{tmp2} \neq 0\\
&= \odot([\![\%\text{tmp2} = \ldots,\ \%\text{tmp2} \neq 0,\ M]\!],\ \Upsilon_2,\ M)\\
&[\![\%\text{tmp2} = \text{load i8* \%tmp1},\ \%\text{tmp2} \neq 0,\ M]\!] = \%\text{tmp1} \neq 0\\
&= \odot([\![\%\text{tmp1} = \ldots,\ \%\text{tmp1} \neq 0,\ M]\!],\ \Upsilon_3,\ M)\\
&[\![\%\text{tmp1} = \text{getelementptr \%struct.M1* \%this, i64 0, i32 2},\\
&\quad \%\text{tmp2} \neq 0,\ M]\!] = \%\text{struct.M1[2]} \neq 0\\
&= \odot([\![\%\text{tmp} = \ldots,\ \%\text{struct.M1[2]} \neq 0,\ M]\!],\ \emptyset,\ M)\\
&[\![\%\text{tmp} = \text{alloca \%class.sc_core::sc_time, align 8},\\
&\quad \%\text{struct.M1[2]} \neq 0,\ M]\!] = \%\text{struct.M1[2]} \neq 0\\
&= \%\text{struct.M1[2]} \neq 0
\end{aligned} \tag{4.30}$$

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:

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

$$\oplus(\text{wait(e)}, M) = \uplus_i(\text{wait(e)}, M) \bigwedge_{i' \in \Lambda(\text{wait(e)})} \uplus(i', M) \tag{4.31}$$

$$\begin{aligned}
\Lambda(i) &= \emptyset\\
\uplus_i(i, M) &= \text{true}\\
T_i &= \text{T1} \prec M\\
\oplus(i, M) &= \text{true}
\end{aligned} \tag{4.32}$$

Table 4.1 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 4.1:** Test suite generated for SystemC in Figure 2.8

### Oracle Generation

A test oracle is a mechanism that determines the program correctness with respect to a test [86]. 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 4.2 lists two execution traces generated when executing the original design in Figure 2.8 and Mutant #1 with the input generated (t,f) from Table 4.1. 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 $cs1 = true$.

| 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 4.2:** Execution Trace Comparison.

Oracles provide useful information to find synchronization errors. Table 4.3 lists all the schedulings for the test suite generated in Table 4.1. We can use the execution traces for 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 4.3:** Schedulings. (TE = time elapse, DL = deadlock, LN = lost notify)

# 5

# Experiments

*It is better to have an approximate answer to the right question than an exact answer to the wrong one.*

John Tukey

In this Chapter, we present initial experimental results of the applications described in the Chapter 4. We conducted the experiments using Ubuntu 11.10 in a quad-core Intel Xeon Processor E5520 with 32 GB of RAM.

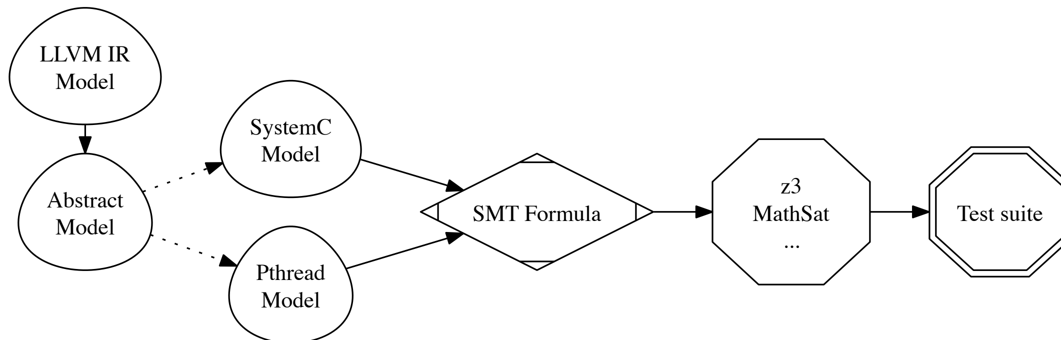


**Fig. 5.1:** Automated Test Generation Flow in *llvmvf*.

In Figure 5.1, we illustrate the test generation flow backend in *llvmvf*. Given a LLVM IR Model, we can instantiate our abstract model to SystemC or Pthread and then apply a verification procedure to generate a SMT formula. Hence, *llvmvf* provides an infrastructure for comparison of different verification methods. Since the

strategies described previously are orthogonal, in the future we can use the SMT-based BMC to generate SystemC testbenches and compare the performance of our BMC and our backwards analysis algorithm.

## SMT-based Bounded Model Checker

We tested our SMT-based BMC with a small set of Pthread programs based on testcases from the benchmark used by ESBMC [27] in their experiments. In Table 5.1, we present our results. Currently, we are restricted to multi-threaded programs that have user assertions and our BMC only supports interleaving semantics and mutex locking operations for Pthreads programs. The second column of the table shows the size of the LLVM byte code file in number of lines and third column represents the number of threads in the program. We run the BMC in a iterative fashion with bound up to 50. The fourth column of the table shows the bound where the formula became satisfiable (fifth column). In the case of *deadlock01_bad*, the generated formula was always unsatisfiable because the program always hit a deadlock scenario. One of the advantages of generating a SMT-LIB v.2 program is that we are able to compare different SMT solvers that are compliant with the standard. We present in the sixth and seventh column the execution time of the last bound for two different SMT solvers, z3 and MathSat 5. Our initial investigation indicates that MathSat 5 is more efficient to resolve programs in the theory of bit-vectors.

| | | | | | z3 | MathSat 5 |
|---|---|---|---|---|---|---|
| **Testcase** | **# Lines** | **# Threads** | **Bound** | **Result** | **Time (s)** | **Time (s)** |
| account_bad | 107 | 2 | 30 | sat | 1.496 | 0.587 |
| simple | 90 | 2 | 11 | sat | 0.290 | 0.090 |
| multiple | 106 | 3 | 15 | sat | 9.830 | 1.709 |
| deadlock01_bad | 77 | 2 | 50 | unsat | 0.750 | 0.528 |
| lazy01_bad | 86 | 3 | 15 | sat | 0.186 | 0.129 |

**Table 5.1:** SMT-based BMC Results

In Table 5.2, we present an initial comparison between llvmvf and esbmc. esbmc is a mature BMC with several optimizations flags such as options to control the scheduling constraints or to restrict the number of context switches. We ran llvmvf in a loop incrementing the bound depth until we have a satisfiable assignment or the bound is 50; and esbmc without optimizations and with a timeout of 30 seconds. Our preliminary comparison suggests that when the bound depth required for a satisfiable assignment or the number of threads is higher, our framework seems to have the similar execution times. In the case of lazy01_bad, esbmc does not achieve a verification error with timeout of 30 seconds. Moreover, running esbmc restricting

the number of context switches to three finds a verification error with at execution time of 3.081 which is still outperformed by our framework. A possible explanation for this situation may be related to the efficient of MathSat over z3 used by esbmc.

| Testcase | llvmvf | esbmc |
|---|---|---|
| account_bad | 12.311 | 9.919 |
| simple | 0.792 | 0.165 |
| multiple | 7.901 | 0.386 |
| deadlock01_bad | 18.143 | TO |
| lazy01_bad | 1.368 | TO |

**Table 5.2:** Execution time (s) comparison between llvmvf and esbmc.

## SystemC Testbench Generation using Mutation Testing

We have conducted initial experiments with 5 SystemC designs. We have modified for our analysis four test cases from the SCRV [42] test suite and implemented an instance of the producer consumer example. In Table 5.3, we present our results. The second and third column of the table represent respectively, the size of the original and optimized LLVM byte code in number of lines. In the fourth column and fifth 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 4.3. Also in the case of *prodcod*, the original program contains redundant synchronization mechanisms, hence mutation these does not produce any observable effect.

| **Design** | **# Lines** | **# Lines (Opt)** | **# Mutants** | **# Test cases** | **MC** |
|---|---|---|---|---|---|
| indexer | 1785 | 1077 | 5 | 3 | 80% |
| sirac | 1980 | 1110 | 9 | 4 | 89% |
| fiveteen | 1986 | 1226 | 4 | 4 | 100% |
| prodcon | 1907 | 1229 | 5 | 3 | 40% |
| srX | 2972 | 1781 | 5 | 2 | 100% |

**Table 5.3:** Mutation Coverage Results

# 6

# Conclusion

> *The most exciting phrase to hear in Science - the one that heralds new discoveries - is not "Eureka!" but "That's funny...".*
>
> Isaac Asimov

In general, the problem of verifying multi-threaded programs is undecidable [72]. Nowadays, the number of approaches that attempt at a partial or complete verification of sequential and concurrent software strongly suggests that formal verification is the holy grail of computer science. The number of approaches result in such a vast number of tools with different capabilities that is not easy to name a new tool with a three letter acronym. Considering the industrial trend of increasingly complex software to meet the higher demands of a global population of consumers, software developers are facing the paradox of choice with respect to which technology to adopt to create faster, scalable and reliable solutions. Since the hardware industry made the decisive factor to accomplish these goals through concurrency, the programming languages that better support mechanisms to safely implement such applications are more likely to strive.

In this thesis, we tackled the problem of finding a general verification solution that can be applied to various programming languages and concurrent models. We mimic the goals of the LLVM framework towards the compiler community and describe the first steps of a verification framework designed to support the state-of-art verification approaches and with an implementation that scalable and reliable. We have introduced a framework that operates at the LLVM byte code level, leveraging its formal semantics. 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 and we introduce two approaches for automated test-case generation: 1) a SMT-based BMC for Pthread programs, and 2) a testbench generation flow for SystemC designs using mutation testing. Our experiments show that we can generate test cases that violate user assertions for Pthread programs and testbenches with high mutant coverage ratio for SystemC designs. We have conducted a preliminary comparison between our SMT-based BMC and esbmc where we were able to outperform esbmc in some scenarios.

In the future, we plan to compare both approaches described for automated test case generation since we can use our SMT-based BMC to encode the reachability

problem of mutated instructions. We plan fully automate our SystemC testbench generation and then to complete our approaches for Pthread and SystemC programs by supporting more concurrent constructs, such as conditional variables in Pthread programs. Furthermore, we want to extend our encoding infrastructure to reduce state space exploration by the SMT solver using abstraction techniques, and investigate completeness thresholds to augment the verification power of our BMC. Finally, we want to further extend our framework as an infrastructure for SMT solvers comparison to understand which SMT solver produces faster results.

# 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. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, Jan. 2009.
5. J. Barnat, L. Brim, and P. Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods Symposium*, volume 7226 of *LNCS*, pages 252–267. Springer, 2012.
6. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.
7. A. Biere. *Bounded Model Checking*, chapter 14, pages 455–481.
8. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.
9. N. Blanc, D. Kroening, and N. Sharygina. Scoot: a tool for the analysis of systemc models. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 467–470, Berlin, Heidelberg, 2008. Springer-Verlag.
10. 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.

11. 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.
12. J. Bradbury, J. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (J2SE 5.0). In *Workshop on Mutation Analysis*, page 11, Nov. 2006.
13. R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.
14. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
15. T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing*, pages 129–148. North-Holland, 1981.
16. C. Tinelli and C. Barrett. The logic of QF_AUFBV, 2010.
17. 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.
18. A. Chlipala. A verified compiler for an impure functional language. *SIGPLAN Not.*, 45(1):93–106, Jan. 2010.
19. A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - a software model checker for systemc. In *CAV*, pages 310–316, 2011.
20. clang: a C language family frontend for LLVM, http://clang.llvm.org/, 2012.
21. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440, pages 570–574, 2005.
22. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
23. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169, London, UK, UK, 2000. Springer-Verlag.
24. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.
25. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.
26. D. R. Cok. The smt-libv2 language and tools: A tutorial. Technical report, GrammaTech, Inc., 2011.
27. L. Cordeiro and B. Fischer. Bounded model checking of multi-threaded software using smt solvers. *CoRR*, abs/1003.3830, 2010.
28. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

29. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
30. R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, Sept. 1991.
31. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
32. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, Aug. 2006.
33. E. A. Emerson. 25 years of model checking. chapter The Beginning of Model Checking: A Personal Perspective, pages 27–45. Springer-Verlag, Berlin, Heidelberg, 2008.
34. R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.
35. 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.
36. 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.
37. M. K. Ganai. Bounded model checking for concurrent systems: Synchronous vs. asynchronous. In *High-Level Verification*, pages 67–95. Springer New York, 2011.
38. K. Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, 1992. Translation B. Meltzer.
39. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.
40. P. Grogono and B. Shearing. Concurrent software engineering: preparing for paradigm shift. In *Proceedings of the 2008 C3S2E conference*, C3S2E '08, pages 99–108, New York, NY, USA, 2008. ACM.
41. 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.
42. 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.
43. 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.
44. C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1967.
45. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
46. F. Ivancic, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3), 2008.
47. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.
48. Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, Oct. 2009.

49. Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, PP(99):1, 2010.
50. S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* http://haskell.org/, September 2002.
51. R. A. Krzysztof and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.
52. C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.
53. C. Lattner and V. Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.
54. W. J. Legato. A weakest precondition model for assembly language programs, unpublished manuscript, 2003.
55. 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.
56. 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.
57. 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.
58. L. D. List. The llvm compiler infrastructure: Llvm users, 2009.
59. 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.
60. K. Marquet and M. Moy. Pinavm: a systemc front-end based on an executable intermediate representation. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 79–88, New York, NY, USA, 2010. ACM.
61. 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.
62. G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, Apr. 1965.
63. MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.
64. F. Mueller. A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, pages 29–41, 1993.
65. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta, USA, 1988.
66. J. Offutt, P. Ammann, and L. Liu. Mutation Testing implements Grammar-Based Testing. In *Workshop on Mutation Analysis, 2006*, pages 12–12, 2006.
67. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal.* Kluwer Academic Publishers, 2001.
68. K. Olukotun and L. Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, Sept. 2005.

69. 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.
70. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
71. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, pages 82–97, 2005.
72. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, Mar. 2000.
73. 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.
74. 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.
75. 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.
76. 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.
77. H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.
78. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.
79. S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.
80. The Multicore Association. Multicore Communications API Working Group, 2012.
81. J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.
82. A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.
83. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.
84. M. Vujošević-Janičić and V. Kuncak. Development and evaluation of lav: an smt-based error finding platform. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 98–113, Berlin, Heidelberg, 2012. Springer-Verlag.

85. P. J. Walsh. *A Measure of Test Case Completeness (software, engineering)*. PhD thesis, State University of New York at Binghamton, Binghamton, NY, USA, 1985.
86. E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.
87. F. Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.
88. J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, Jan. 2012.