J Electron Test (2013) 29:635–646
DOI 10.1007/s10836-013-5405-9

LLVMVF: A Generic Approach for Verification of Multicore Software

Marcelo Sousa · Alper Sen

Received: 26 March 2013 / Accepted: 21 August 2013 / Published online: 7 September 2013

Abstract Proliferation of multicore hardware boosted the need for verification of multicore software that is running on these hardware. Multicore software demands new verification techniques different from the ones used for sequential software. Many optimized compiler frameworks are arising to address the complexities of multicore software. Among these compilers, Low Level Virtual Machine (LLVM) is especially gaining popularity because i) has a universal front-end that allows to read in many different input languages, ii) aggressive optimizations to improve code performance and quality, and iii) a well-defined intermediate bytecode representation, called LLVM IR, that allows a unified intermediate representation. In this work, we present a novel framework, called LLVM Verification Framework (LLVMVF), implemented in a purely functional language for verification of multicore software. To our knowledge, this is the first verification framework using the LLVM bytecode representation for multicore software. We present an SMT-based Bounded Model Checker backend of LLVMVF and perform initial experiments on multicore software using Pthreads library. Furthermore, we compare our results with an existing multicore software verification tool.

Responsible Editor: S. Ray

M. Sousa
Department of Computer Science,
University of Oxford,
Oxford, UK
e-mail: marcelo.sousa@cs.ox.ac.uk

A. Sen (✉)
Department of Computer Engineering,
Bogazici University,
Istanbul, Turkey
e-mail: alper.sen@boun.edu.tr

Keywords Computer-aided verification · Multicore software · Formal verification · Multithreaded programs

1 Introduction

The observation of Moore's Law in the last half century triggered a paradigm shift in mainstream computer architectures. This shift was ignited when the hardware industry realized that the approaches applied for boosting CPU performance reached the physical limits. The solution adopted was the transition from single to multi-processor/multicore 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. This inadequate methodology created a fundamental problem concerning the quality of the existing software with respect to code reliability and efficiency. With the shift to multicore 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.

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 (multicore 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* and in recent years there is a renewed interest in concurrent programming. This interest is driven not only by the evolution of multicore architectures but also the increasing size, complexity, expectations and reliability of software systems.

Formal verification of software is a valuable approach to produce automated analysis and testing tools. The tools are valuable because they simplify the developers' manual quality assurance cycle, that 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 software verification tools focus on sequential software or a specific concurrent model. The increasing usage of concurrent software calls for a general framework for verification of concurrent software. The main goal of our work 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 [29]. Furthermore, we leverage the advantages of Haskell [27], a general purpose strongly-typed functional programming language, and the Utrecht University Attribute Grammar Compiler (uuagc) [38] to perform efficient transformations on our formalizations. Haskell is suitable for prototyping due to its expressive type system that allows the development 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 [24]. 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 input language of SPIN, that is Promela [3], 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. 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, such as Pthreads library, domain specific languages such as SystemC, and message passing interfaces such as MPI [31] or MCAPI [39].

We implemented a multicore software verification framework in a pure functional language. To our knowledge, this is the first verification framework using the LLVM bytecode representation focused on concurrent programs. We present a new LLVM IR formalization using the attribute grammar system, and also an initial application targeting multicore software using Pthreads library, which is the most commonly used multicore software library on multicore systems.

The rest of the paper is organized as follows. Section 2 describes related work in formalization of LLVM IR and SMT-based bounded model checking for LLVM IR. Section 3 details the background on LLVM, bounded model checking, and Pthreads library. In Section 4, we introduce the architecture and details on the Pthreads backend of LLVMVF. Section 5 describes our experimental setup, presents the results of the Pthreads experiments and discusses our results with existing approaches. We conclude with a discussion of our contributions and future work.

2 Related Work

A widely adopted paradigm for modeling concurrent systems is that of interleaving, where non-deterministically a choice is made between concurrently executing threads using a scheduler [2]. An interleaving represents a possible execution of the program where all of the concurrent events are arranged in a linear order. Any change of the active thread in an interleaving is a context switch. Most verification tools use the interleaving model including explicit-state model checkers such as SPIN [25], Java PathFinder [23] as well as symbolic model checkers such as SATABS [11], CBMC [13], ESBMC [15], and LLBMC [30]. Furthermore, memory models play an important role in formal verification of programs. The most commonly used memory model is sequential consistency. We also use the interleaving model and assume sequential consistency in this paper.

Recently, several tools have formalized the LLVM IR language in the context of certified compilers. Vellvm [42] 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 [8], a ANSI-C verified compiler. LLVM M.D. [40] 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 whereas we use the bindings for the LLVM API which is a more reliable solution since the disassembled bytecode may contain errors.

In the last decade, several tools applied bounded model checking to verify C and C++ programs. The initial tools, CBMC [13] and F-Soft [26] focused on sequential programs. SMT-CBMC [1] proposed a combination of bounded model checking with SMT solvers to use their expressive power. TCBMC [33] and ESBMC [15] apply bounded model checking for C programs using Pthreads. 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 [11] performs verification of multi-threaded software with shared variables with a CEGAR approach on sequential GOTO-programs translated from the original concurrent program [12].

Recently, formal verification tools target intermediate languages. VCC [14] is an assertion verifier for concurrent C programs. VCC uses the SMT solver Z3 to analyze verification conditions (VCs) generated from Boogie. Boogie [4] 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 [30] applies SMT-based bounded model checking for sequential C/C++ programs. LAV [41] combines symbolic execution and SMT based bounded model checking for bug finding in sequential C programs. We developed an initial application of our framework to SystemC test generation [37], which also has a concurrent semantics.

There is a variety of work on verification of shared memory concurrent programs [1, 21, 26, 28, 32, 34, 36]. In order to fully verify a multi-threaded program against a given specification, all possible interleavings must be considered. We use BMC to obtain a finite state program and explore all (finite number of) interleavings present in the bounded program. Similar to ESBMC, we encode all possible interleavings (context-switches) into one single formula and then exploit the high speed of the SMT solvers. There are also other approaches such as context-bounding [19, 32], partial-order reduction [20, 22] to reduce the complexity of verification.

Symbolic execution is also used in bug finding. For example, KLEE [7] is a symbolic execution tool that performs a symbolic path exploration that considers the paths separately, whereas we use BMC, which encodes all paths up to a bounded length in a single formula. Also, KLEE requires manual instrumentation from the user and does not handle concurrent programs.

This is the first time LLVM is used for the verification of multicore (multithreaded) software using Pthreads library, which is the most commonly used multicore software library. Furthermore, we generate SMT formulas in SMT-LIB v.2 format that can be used with many SMT solvers for the purposes of bounded model checking and verification, in general.

3 Background

3.1 Background on LLMV

LLVM is a popular and growing compiler framework that supports aggressive multistage optimizations to overcome known problems of traditional compilation techniques. 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 Fig. 1, 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 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

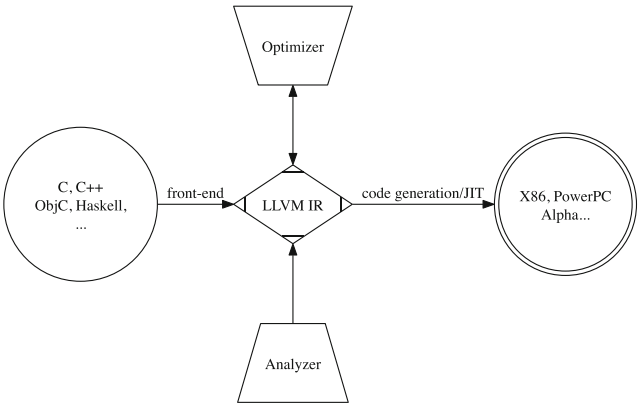

Fig. 1 LLVM architecture

provides code generation for several architectures, and although primarily focused on C and C++, other programming language front-ends have been implemented.

LLVM is used in multiple projects due to its scalability and competitive performance results against industrial and research compilers. 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.

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 Static Single Assignment (SSA) form 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.

In Fig. 2, we describe some of the productions that compose the implemented LLVM IR grammar. We describe a simplified grammar since for our current 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 Fig. 3, *%"class.std :: ios_base :: Init"* is defined as the integer type of one byte. A global variable ⟨global⟩ is represented by an identifier, a type and an optional value if the variable is initialized. Line 2 of Fig. 3, defines "@M1" as a constant with the value "M1". Function definitions are composed of a declaration (identifier ⟨ident⟩, type ⟨ty⟩ and parameters ⟨param⟩*) and a list of basic blocks. Each basic block ⟨bb⟩ is represented by a label identifier, a list of ⟨phi⟩ instructions, a list of instructions and a final terminator ⟨tmn⟩ instruction

$$
\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 * \mid \langle \text{ty} \rangle^* \rightarrow \langle \text{ty} \rangle \mid \{\ \langle \text{ty} \rangle^*\ \}
\end{aligned}
$$

Fig. 2 Abstract LLVM IR grammar

such as an unconditional *branch* (line 4). A *phi* instruction represents a value choice between basic blocks.

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.

3.2 Concurrent Program Model

A concurrent program consists of a finite set of threads $T_0, \ldots, T_k$ communicating via shared variables. Threads are allowed to fork other threads in a bounded manner, and the total number of threads is finite. We represent a concurrent program using a concurrent control flow graph (CCFG), similar to [36]. A CCFG $G = (V, E)$ consists of a set of vertices V and a set of edges E. We use two special types of vertices fork and join to model thread creation and thread join, respectively. A thread T_i corresponds to a sub-graph (V_i, E_i) of the CCFG, where V_i consists of nodes representing program locations as well as statements in thread T_i and E_i consists of edges representing the thread program order at the bytecode level. We assume that N_i contains unique entry and exit nodes of T_i and there are no cycles in the

```
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. 3 LLVM IR example

control flow graph. For each T_i (other then the main thread) the entry node has a single incoming edge from a fork node and the exit node has a single outgoing edge to a join node. The transition relation of a thread is composed of executions of statements at nodes in the control flow graph. The transition relation of the CCFG is obtained from transitions of all the threads and will be detailed below.

3.3 Bounded Model Checking

In general, the problem of verifying two-threaded programs (with unbounded stacks) is undecidable [35]. Bounded Model Checking (BMC) [5], which is a symbolic model checking approach, limits all program runs and data structures to finite ones, thereby achieving decidability. This is accomplished by analyzing only bounded program runs. The bound is typically imposed by restricting the number of nested function calls and loop iterations that are allowed. By considering only finite program runs, affected data structures and context-switches also become finite. Function inlining and loop unrolling then results in one function that is subject to verification. We use LLVM "decidability" transformations similar to LLBMC [30] and LAV [41], described in Section 4.1, to obtain a bounded program from LLVM-IR representations. Using BMC, we can check the violations of safety properties such as assertions, reachability of certain program statements or race conditions.

More specifically, Bounded Model Checking (BMC) [5] originated as an alternative to symbolic model checking using Binary Decision Diagrams (BDD) [6] 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. SAT/SMT-based BMC generates a first-order propositional formula Ψ given a transition system M, a bound k, and a property ϕ according to the formula:

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

The general BMC formula above encodes the entire model M as a set of transitions ($T_i(M)$) 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. In this work, we exploit the optimization advances of SMT solvers to tackle the verification of multi-threaded software.

3.4 Background on Pthreads Library

We develop algorithms for the verification of multithreaded programs written using the Pthreads library. The Pthreads library 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. Moreover, the library does not specify 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.

3.4.1 Pthread Example

Figure 4 presents a Pthreads example from [15]. 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 work.

```
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. 4 Pthread example

4 LLVMVF: LLVM Verification Framework

The architecture of LLVMVF is designed to create a compact model of the bytecode from the original high level program. In Fig. 5, we present several phases of our framework. The input of our flow is a LLVM bytecode file corresponding to the high-level program under verification. For example, in the case of C/C++ we can use a compiler from the clang family [10] to obtain the bytecode. Since the implementation of C++ libraries is heavily based on template programming, applying various optimizations to obtain more compact bytecode modules removes some overhead from the infrastructure as described below.

4.1 Simplification

We start, phase (a), by using the optimizer to transform the current bytecode into a form that is suitable for our analysis and specifically to obtain a bounded program similar to transformations in LLBMC [30] and LAV [41]. In total, we apply 18 LLVM passes over the input bytecode file. The LLVM Pass Framework is an infrastructure to structurally implement bytecode 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 bytecode that has no cycles in the basic block graph. Simplification transformations aim at simplifying our formalization. We reduce the bytecode size eliminating LLVM IR constructs not supported by our analysis such as invoke or switch instructions. Furthermore, we lift stack operations by promoting the stack to registers and use an LLVM pass to name all nameless identifiers.

4.2 Extraction

In phase (b) of Fig. 5, we obtain a formal LLVM IR model in Haskell by using a binding mechanism. Specifically, the extraction function uses a double binding schema to call an LLVM API function from Haskell and follows the abstract grammar in Fig. 2. An alternative solution to formalize LLVM IR in Haskell would be to create a parser from the disassembler output. Our approach is more reliable because the output from the disassembler might contain errors.

We now give an example of model extraction. To call the LLVM API function *getModuleIdentifier* that retrieves the name of the bytecode module:

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

First, we create the C binding:

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

Then, we create 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.

4.3 Abstraction

In phase (c) of Fig. 5, 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 threads. To extend the framework to a new concurrent mechanism or language, the only component that a user has to implement is the refinement from Haskell model of LLVM IR to the abstract concurrent model.

The LLVM IR Haskell model extracted implicitly represents a concurrent control flow graph through function calls to concurrent libraries such as Pthreads. We model concurrency by interleaving as described above and refine the LLVM IR model as a concurrent model with explicit scheduler where only one thread can execute at a time.

Figure 6 shows the concurrent control flow graph for the Pthread program in Fig. 4. The diamond node represents thread forking. Each node is annotated with a program counter and the actions. For Pthread programs, the abstraction algorithm traverses the main function

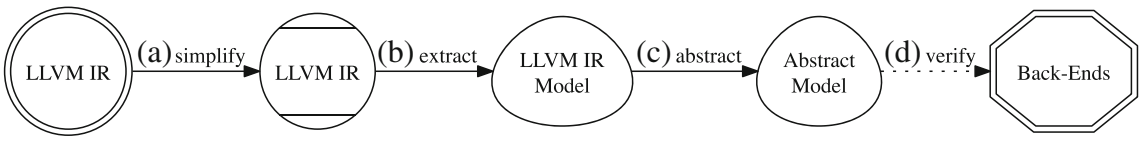

Fig. 5 LLVMVF architecture

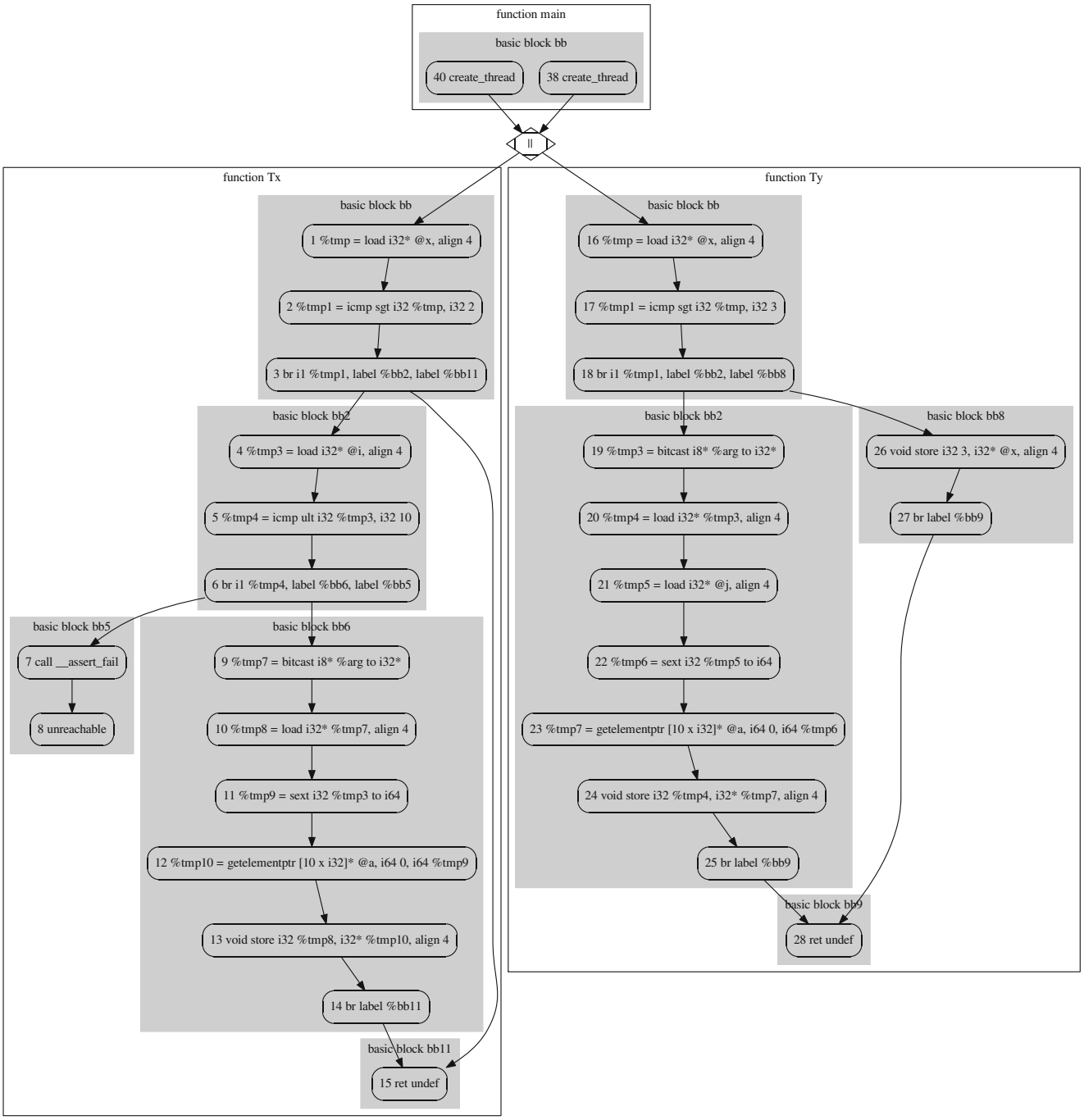

Fig. 6 Concurrent control flow graph for example 4

calculating the intra-procedural control flow and searching for thread fork/join 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))
```

At the end of the abstraction phase, our model is ready to be used by the particular backend implementation. In this work, we devise an SMT-based Bounded Model Checking

technique for Pthreads programs as the backend implementation. Earlier, we described a backend for automated test generation of SystemC designs [37].

4.4 SMT Encoding and Verification

In phase (d) of Fig. 5, we obtain an encoding of the concurrent model in SMT-LIB v.2 format that can be used with several SMT solvers. The SMT-LIB language encodes logical formulas in a many-sorted first-order logic. 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. Since we use the SMT-LIB with bitvectors and arrays, many LLVM constructs can be easily encoded including types, global variables, and instructions. Once the program is encoded then it is given to an SMT solver for verification.

As described in Section 3.3, given a transition system M, a bound k and a property ϕ, an SMT-based BMC generates the formula:

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

Although we have a partial implementation for a linear encoding of LTL properties, we currently focus on verifying user assert statements. Therefore, we are able to statically determine the error states since they are calls to the function *__assert_fail* and reduce the formula above to a reachability problem:

$$\Psi(M,k) = I(M) \wedge \left(\bigwedge_{i\in[0..k-1]} scheduler_i \wedge T_i(M_{threads}) \right) \wedge error \tag{2}$$

$$error = \bigvee_{i\in[0..k-1]} \overbrace{\bigvee_{f\in F(M)} \bigvee_{j\in[0..tc-1]} \Pi_j pc_i = f}^{\text{Current state is a fail state}} \tag{3}$$

In Formula 2, we explicitly encode a scheduler. Furthermore, in Formula 3, tc is the number of threads and $\Pi_j pc_i$ is the program counter in bound i of the thread Π_j. Intuitively, formula 2 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.

Next, we will show how we encode Ψ by describing encodings for types, global variables, thread instructions, the scheduler, the initial state, and the error state. Then we will show an overall encoding example.

4.4.1 Encoding Types

We now describe our encoding of all the types used in the program. We define the type *iN* as a new sort *IN* that is a synonym for a $BitVector$ of size N, except for *i*1 that is mapped to *Bool*. For example, the LLVM type *i*8 is encoded as:

```
(define-sort I8 () (_ BitVec 8))
```

An LLVM array $(Array\ n\ ty)$ is encoded as:

(define−sort $Arraynty$ () ($Array$ (_ BitVec $bsize(n)$) $sort(ty)$))

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 48 elements of type *i*8 is encoded as:

```
(define-sort Array48I8 () (Array (_ BitVec 6)
    I8))
```

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:

```
(declare-sort Pair 2)
(define-sort union.pthread_attr_t () (Pair I64
     Array48I8))
```

Encoding for the remaining LLVM types is not supported yet.

4.4.2 Encoding Global Variables

We first 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 program counter where an initial assignment was performed. By default, this program counter is 0. For example, the global declaration:

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

is encoded as:

```
(declare-fun x () I32)
(declare-fun lx () I32)
(assert(and(= x (_ bv2 32))(= lx (_ bv0 32))))
```

4.4.3 Encoding Instructions

The encoding function 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.

Note that, all encoding examples below consider the encoding for the first step (bound 0) of BMC, unless otherwise specified.

To encode memory access instructions related to global variables (such as variable x) at step N of BMC, we introduce a fresh variable *xN*, for each store of the global

variable, and another fresh variable $px(N+1)$ that represents the program counter of the last store to the variable. The variable $px(N+1)$ will be useful in load instructions of the next BMC step (step $N+1$), whereas variable *pxN* is used in the current step. In Fig. 4, two threads *Tx* and *Ty* are active and for simplicity we assume that they modify global variables *x* and *i* only. In Fig. 6, the *store* instruction at pc = 26, (that is, $x = 3$ at line 18 of Fig. 4)

```
void store i32 3, i32* @x, align 4
```

is encoded as follows:

```
(and (and (and Ty0
(and (= Typc0 (_ bv26 32))
(and (= x0 (_ bv3 32)) (and (= pi1 pi0) (= px1
     (_ bv26 32))))))
(= Typc1 (_ bv27 32)))
(= Txpc1 Txpc0))
```

This instruction is active when thread Ty is chosen by the scheduler at bound 0, that is, Ty0 is true. Thread Ty goes from pc = 26 to pc = 27, that is, Typc0 = 26 and Typc1 = 27, while thread Tx does not change its pc, Txpc1 = Txpc0. Since this instruction is a store of *x*, we create fresh variables *x0, px1* and assign 3 to *x0* and the value of the last store program counter, that is 26, to *px1*. Similarly, since variable *i* is not modified, the last store program counter of *i*, that is $pi1$, is the same as the current program counter value, $pi0$.

The encoding of the *load* instruction at pc = 1 (that is, read of x at line 8 of Fig. 4)

```
%tmp = load i32* @x, align 4
```

is encoded as follows:

```
(and (and (and Tx0
(and (= Txpc0 (_ bv1 32))
(and
(or (and (= px0 lx) (= Txtmp x)) (and (= px0 (
    _ bv26 32)) (= Txtmp x0)))
(and (= pi1 pi0) (= px1 px0)))))
(= Txpc1 (_ bv2 32)))
(= Typc1 Typc0))
```

This instruction is active when thread Tx is chosen by the scheduler at bound 0, that is, Tx0 is true. Thread Tx goes from pc = 1 to pc = 2, that is, Txpc0 = 1 and Txpc1 = 2, while thread Ty does not change its pc. In this formula, we assign a value to the local variable *tmp* of *Tx*, that is, *Txtmp*, depending on the value of *px0*, which is the variable that represents the program counter of the last store. In this case, *px0* is either *lx*, which keeps track of the program counter for the global variable initialization of x, hence $Txtmp = x$ or *px0* is 26 that corresponds to the store by *Ty* shown above and $Txtmp = x0$. This approach can also be used to encode *phi* instructions.

For encoding a conditional branch instruction, we generate two alternatives. If the condition is true then the program counter is updated with the program counter of the true branch, otherwise with the false branch. In Fig. 6, the *branch* instruction at pc = 3 updates pc to either 4 or 15, that is,

```
br i1 %tmp1, label %bb2, label %bb11
```

is encoded as follows:

```
(and (and (and Tx0
(and (= Txpc0 (_ bv3 32)) (and (= pi1 pi0) (=
    px1 px0))))
(or (and Txtmp1 (= Txpc1 (_ bv4 32))) (and (
    not Txtmp1) (= Txpc1 (_ bv15 32)))))
(= Typc1 Typc0))
```

4.4.4 Encoding Scheduler

Finally, we explicitly add an encoding for the *scheduler* where at each bound depth a single thread is nondeterministically chosen for execution while other threads do not make any progress. This captures modeling concurrency by interleaving and allows us to cover all possible interleavings during verification. The encoding of the scheduler for bound, N = 0, is as follows.

```
(or (and Tx0 (not Ty0)) (and Ty0 (not Tx0)))
```

4.4.5 Overall Encoding

We now show the overall encoding for our SMT based BMC using Formula 2 when bound $k = 1$. For ease of reading we show the formula below,

$$\Psi(M,k) = I(M) \wedge \Big(\bigwedge_{i\in[0..k-1]} scheduler_i \wedge T_i(M_{threads})\Big) \wedge error$$

The initial state $I(M)$ contains encoding of types, global variables as well as the initializations of variables and program counters. We show the encoding for $I(M)$ as follows:

```
// Encoding of types
// Encoding of global variables
// Initializations
(and (and (= Txpc0 (_ bv1 32)) (= Typc0 (_
    bv16 32)))(and (= pi0 li7) (= px0 lx)))
```

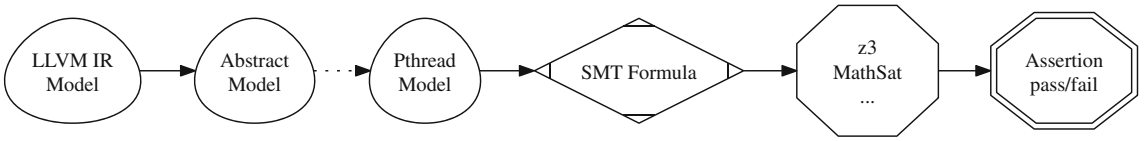

Fig. 7 Verification flow in LLVMVF

Table 1 SMT-based BMC results for the last bound using LLVMVF

Benchmark	# Lines	# Threads	Bound	Result	Z3 Time (s)	MathSat5 Time (s)
account_bad	111	3	30	sat	1.4	0.97
account_ok	111	3	50	unsat	87	49
lazy01_bad	86	3	15	sat	0.5	0.4
simple	90	2	11	sat	0.5	0.36
phase01_bad	74	2	10	sat	0.3	0.27
deadlock01_bad	77	2	50	sat	1.2	0.9
stateful01_unsafe	82	2	26	sat	3	1
carter01_unsafe	80	2	9	sat	0.24	0.28

The error state *error* refers to the program counter 7, so it is encoded as

```
(or (= Txpc0 (_ bv7 32))(= Typc0 (_ bv7 32)))
```

We encode the scheduler $scheduler_0$ for $k = 1$ as

```
(or (and Tx0 (not Ty0)) (and Ty0 (not Tx0)))
```

The transition relation $T_i(M_{threads})$ is composed of the transition relations for threads *Tx* and *Ty*. We showed instructions for both threads when $k = 1$ above. Similarly, we can obtain the encoding for other bounds.

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.

5 Experiments

In this section, we present our experimental results using LLVMVF. We conducted the experiments using Ubuntu 12.04 on a virtual machine with 2 cores and 365 MB RAM.

In Fig. 7, we illustrate the verification flow backend in LLVMVF. Given a LLVM IR Model, we can instantiate our abstract model to Pthreads and then apply a verification procedure to generate a standard SMT-LIB v.2 formula. Hence, LLVMVF provides an infrastructure for comparison of different SMT tools.

We tested our SMT-based BMC with a set of Pthreads programs based on concurrent benchmarks used in Competition on Software Verification [16] and the simple example described in this paper. All multi-threaded programs have user defined assertions denoting some error conditions.

In Table 1, we present our results. We achieved correct results for all benchmarks. The second column of the table shows the size of the LLVM bytecode file in number of lines and third column represents the number of threads in the program. We run BMC in an iterative fashion with a bound up to 50. The fourth column shows the bound where the formula became satisfiable. We are able to identify potential safe test cases such as *account_ok*. In this case, the generated formula was always unsatisfiable because the program always hit a correct scenario. One of the advantages of generating a SMT-LIB v.2 formula is that we are able to compare different SMT solvers that are compliant with the standard. We present in the sixth and seventh columns the execution time of the last bound for two different SMT solvers, namely Z3 [17] and MathSat5 [9]. Our initial investigation indicates that MathSat5 is more efficient to resolve programs in the theory of bit-vectors.

In Table 2, we present an initial comparison between LLVMVF and ESBMC. ESBMC is a mature BMC tool with several optimizations flags such as options to control the scheduling constraints or to restrict the number of context switches. ESBMC uses CBMC as a front-end but not LLLVM like we do, hence it is not portable to other languages. We ran LLVMVF in a loop incrementing the bound depth until we have a satisfiable assignment or the bound is 50 (unlike above where we list the solver times for the last bound only). We ran ESBMC with the script used during the Competition on Software Verification 2013 with optimizations and a timeout of 450 s.

Table 2 Execution time comparison between LLVMVF and ESBMC

Benchmark	LLVMVF Z3	LLVMVF MathSat5	ESBMC
account_bad	24	11	1.1
account_ok	87	49	TO
lazy01_bad	3.5	3.3	TO
simple	3.7	2.2	1
phase01_bad	1.5	1.5	4.3
deadlock01_bad	1.8	1.9	TO
stateful01_unsafe	36	15	1
carter01_bad	1.3	1.2	1

Our comparison suggests that LLVMVF execution times are similar when the bound depths required for a satisfiable assignment are close.

ESBMC does not achieve a verification result with time-out of 450 s for 3 benchmarks *account_ok*, *lazy01_bad* and *deadlock01_bad*, whereas LLVMVF successfully completes verification for all benchmarks. For benchmarks when ESBMC completes verification, LLVMVF has longer running times. This is because currently we do not support any optimizations in LLVMVF unlike ESBMC.

Note that we can use the program counter to obtain a trace in the counter-example generation given by the SMT solver. We can then use a C-backend for LLVM IR or use the metadata information in a debug compilation to obtain precise information with respect to the original program.

6 Conclusion and Future Work

We present the design and implementation of a framework for verification of multicore software. The goal of this framework is to have a reliable and scalable infrastructure for verification of concurrent programs. Our framework, LLVMVF, is novel in that it operates at the bytecode representation of the programs using the Low Level Virtual Machine (LLVM) framework. Our implementation leverages the advantages of Haskell, a general purpose strongly-typed functional programming language, to achieve a transparent, powerful and scalable framework. We describe a Satisfiability Modulo Theories (SMT) based Bounded Model Checker for Pthreads programs using LLVMVF. We obtained favorable results on benchmarks compared with a previous non-LLVM based multicore software verifier.

The importance of our solution is that it can work at the bytecode level. Therefore, the verification techniques supported by our framework can potentially be used in a wide spectrum of concurrent domains. Also, since we generated standard SMT formulas, SMT-LIB v2, this allowed us to compare performance of different SMT solvers such as MathSat5 and Z3, where MathSat5 had a slight edge over Z3.

In the future, we plan to support more concurrency constructs such as conditional variables to augment the domain of our analysis. Furthermore, we believe that abstraction techniques can reduce the size of our encodings and prior analysis of the source can provide insight to a conservative minimum bound to optimize our iterative approach. Finally, a possible implementation could be to use our framework for test generation in the context of emerging multicore software standards such as MCAPI [18].

Acknowledgments This research was supported by Semiconductor Research Corporation under task 2082.001, Marie Curie European Reintegration Grant within the 7th European Community Framework Programme, BU Research Fund 7223, and the Turkish Academy of Sciences.

References

1. Armando A, Mantovani J, Platania L (2009) Bounded model checking of software using smt solvers instead of sat solvers. Int J Softw Tools Technol Transf 11(1):69–83
2. Baier C, Katoen J-P (2008) Principles of model checking. The MIT Press, Cambridge
3. Barnat J, Brim L, Ročkai P (2012) Towards LTL model checking of unmodified thread-based C & C++ programs. In: NASA formal methods symposium, volume 7226 of LNCS, pp 252–267
4. Barnett M, Chang B-YE, DeLine R, Jacobs B, Leino KRM (2006) Boogie: a modular reusable verifier for object-oriented programs. In: Proceedings of the 4th international conference on formal methods for components and objects, pp 364–387
5. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without bdds. In: Proceedings of the 5th international conference on tools and algorithms for construction and analysis of systems, pp 193–207
6. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691
7. Cadar C, Dunbar D, Engler DR (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp 209–224
8. Chlipala A (2010) A verified compiler for an impure functional language. SIGPLAN Not 45(1):93–106
9. Cimatti A, Griggio A, Schaafsma B, Sebastiani R (2013) The MathSAT5 SMT solver. In: Proceedings of the international conference on tools and algorithms for construction and analysis of systems (TACAS), number 7795 in Lecture Notes in Computer Science
10. clang: a C language family frontend for LLVM, http://clang.llvm.org/ (2012)
11. Clarke E, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS 2005), pp 570–574
12. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Proceedings of the 12th international conference on computer aided verification, CAV '00, pp 154–169
13. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ansi-c programs. In: Proceedings of the international conference on tools and algorithms for construction and analysis of systems (TACAS), pp 168–176
14. Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S (2009) VCC: a practical system for verifying concurrent C. In: Proceedings of the 22nd international conference on theorem proving in higher order logics, pp 23–42
15. Cordeiro L, Fischer B (2010) Bounded model checking of multi-threaded software using smt solvers. In: Presentation-only paper in 8th international workshop on satisfiability modulo theories (SMT) at FLoC, Edinburgh, Scotland
16. Competition on software verification, http://sv-comp.sosy-lab.org/ (2013)

17. de Moura LM, Bjørner N (2008) Z3: an efficient smt solver. In: Proceedings of the international conference on tools and algorithms for construction and analysis of systems (TACAS), pp 337–340
18. Deniz E, Sen A, Holt J (2012) Verification and coverage of message passing multicore applications. ACM Trans Des Autom Electron Syst 17(3):1–31
19. Emmi M, Qadeer S, Rakamarić Z (2011) Delay-bounded scheduling. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages
20. Flanagan C, Godefroid P (2005) Dynamic partial-order reduction for model checking software. In: Principles of programming languages'05, pp 110–121
21. Ganai MK, Gupta A (2008) Efficient modeling of concurrent systems in bmc. In: Proceedings of the 15th international workshop on model checking software
22. Godefroid P, Wolper P (1991) A partial approach to model checking. In: Proceedings of the 6th IEEE symposium on logic in computer science, pp 406–415
23. Havelund K, Pressburger T (2000) Model checking Java programs using Java PathFinder. Int J Softw Tools Technol Transf 2(4):366–381
24. Holzmann G (2003) Spin model checker, the: primer and reference manual, 1st edn. Addison-Wesley Professional
25. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
26. Ivancic F, Yang Z, Ganai M, Gupta A, Ashar P (2008) Efficient sat-based bounded model checking for software verification. Theoret Comput Sci 404(3):256–274
27. Jones SP (ed) (2002) Haskell 98 language and libraries: the revised report. http://haskell.org/
28. Kahlon V, Gupta A, Sinha N (2006) Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Proceedings of the 18th international conference on computer aided verification
29. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04)
30. Merz F, Falke S, Sinz C (2012) LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: VSTTE, pp 146–161
31. MPI Forum (2009) MPI: a message-passing interface standard. Version 2.2
32. Qadeer S, Rehof J (2005) Context-bounded model checking of concurrent software. In: Proceedings of tools and algorithms for the construction and analysis of systems
33. Rabinovitz I, Grumberg O (2005) Bounded model checking of concurrent programs. In: Proceedings of the international conference on computer-aided verification (CAV), pp 319–325
34. Rabinovitz I, Grumberg O (2005) Bounded model checking of concurrent programs. In: CAV, pp 82–97
35. Ramalingam G (2000) Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans Program Lang Syst 22(2):416–430
36. Sinha N, Wang C (2011) On interference abstractions. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages
37. Sousa M, Sen A (2012) Generation of TLM testbenches using mutation testing. In: Proceedings of international conference on hardware/software codesign and system synthesis (CODES/ISSS)
38. Swierstra SD, Alcocer PRA, Saraiva J (1998) Designing and implementing combinator languages. In: Advanced functional programming, pp 150–206
39. The Multicore Association (2012) Multicore communications API working group
40. Tristan J-B, Govereau P, Morrisett G (2011) Evaluating value-graph translation validation for llvm. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, pp 295–305
41. Vujošević-Janičić M, Kuncak V (2012) Development and evaluation of LAV: an SMT-based error finding platform. In: Proceedings of the 4th international conference on verified software: theories, tools, experiments, pp 98–113
42. Zhao J, Nagarakatte S, Martin MM, Zdancewic S (2012) Formalizing the llvm intermediate representation for verified program transformations. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, pp 427–440

Alper Sen is an Associate Professor of Computer Engineering at Bogazici Unviersity, Istanbul, Turkey.

Marcelo Sousa is a Ph.D. student in the Department of Computer Science at Oxford University, UK.