A Very Simple LaTeX 2_ε Template

Vitaly Surazhsky
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

Yossi Gil
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

January 2, 2015

Abstract

This is the paper's abstract ...

1 Introduction

Correctness of computer systems is important in our software dependent society. We depend more and more on this modern computer systems. Such systems consist of complex hardware and software components. So correctness of both hardware and software is important. However, it is much harder to ensure the correctness of the software part of these systems. As the complexity of the software parts are getting more complex than that of the underlying hardware. And manual inspection of complex software is error-prone and costly. Numerous formal tools to find functional design bugs in hardware are available and in wide-spread use. In contrast, the market for software verification tools is still in its infancy. A lot of research in this field is going on. We need highly automated method that provides rigorous guarantee of quality. These methods should be scalable enough to match the enormous complexity of software

systems. Bounded model checking (BMC) of programs is one of such methods used for software verification. In this paper we will explore LLBMC, an implementation of the idea of Bounded model checking.

Outline The remainder of this article is organized as follows. Section 10 gives account of previous work. Our new and exciting results are described in Section 11. Finally, Section 12 gives the conclusions.

2 Verification problem

In industry we see the use of testing to ensure software quality and even to find bugs. However such quality guarantees do not reflect to "correctness" of the system. As we know that ensuring the absence of all errors in a design is usually too expensive. So all testing based approaches e.g. random testing and automated test-case generation are incomplete. The other approach depends on the fact that systems can be viewed as mathematical objects with well-specified behaviour. Or we can specify the system (intended behaviour) using mathematical logic. Then one can reason about whether the system meets its specification or not. This field of study has been active and it is often referred to as formal methods. The *verification problem* is: Given program M and specification h determine whether or not the behaviour of M meets the specification h. The specification is formulated in mathematical logic. The model checking problem is an instance of the verification problem. Model checking provides an automated method for verifying finite state systems and checks it for certain properties. The method is algorithmic and often efficient because the system is finite state. If the check fails, that means the design has a bug. And, most of the time, the model checkers generate a counterexample how the bug is produced so the developer can easily figure out what is the mistake.

3 Bounded Model Checking

Model checking is a method for formally verifying finite-state systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large state-spaces can often

be traversed in minutes.

Bounded Model Checking is first proposed by Biere et. al. in 1999 [2]. It doesn't reduce the complexity of model checking however, it can solve many cases which cannot be solved by BDD (binary decision diagram) based techniques. However, BMC has the disadvantage of not being able to prove the program is valid because of its bounded state structure.

Idea of BMC is searching for a counterexample in the given problem until an execution length n. The BMC problem can be efficiently reduced to a propositional satisfiability problem and can be solved by SAT solvers. Modern SAT solvers can handle propositional satisfiability problems with 6-7 digit variables and sometimes more.

However, as stated previously, BMC approach is used for finding bugs, not for proving the program is valid because of the execution length. A greater execution length might show bugs which cannot be found for smaller execution lengths.

In formally, bounded model checking contains: A transition system, a temporal logic formula and a user supplied bound k. Using these properties, we check if the given formula satisfied in the given bound.

In our work, the program flow is described as transition states for the BMC conversion. The easy transform between BMC and SMT formulas is highly beneficial because of the presence of highly optimised SMT solvers.

4 LLVM

LLVM (Low Level Virtual Machine) is a compiler infrastructure; designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in different various programming languages. Initially designed for C and C++, LLVM now supports a lot of different languages like ActionScript, Ada, D, Fortran, GLSL, Haskell, Java, Julia, Objective-C, Python, Ruby, Rust, Scala.

LLVM supports a language-independent instruction set and type system. [3] Each instruction is in static single assignment form (SSA), meaning that each is assigned only once and cannot be assigned again. This helps simplify the analysis of dependencies among variables. LLVM allows code to be compiled statically, as it is under the traditional GCC system, or left for late-compiling

from the IF to machine code in a just-in-time compiler (JIT) fashion similar to Java. The type system consists of primitive types such as integers or floats and five other derived types: Arrays, vectors, functions, structures and pointers. A type construct is represented by combining these given types in LLVM. For example, a class in C++ can be represented by a combination of structures, functions and arrays of function pointers.

LLVM is used in many areas because of its optimization supporting and competitive performance results. For example, Apple uses LLVM in its Xcode and Cocoa frameworks [10].

LLVM adapted SSA (Static Single Assignment) form. This provides serious advantages for formal conversions. The instruction set composed three address code instructions. Also, the LLVM instructions in Linux can generate CFG (control flow graphs) of the threads and the program which helps our conversion. Because LLVM-IR is represented in a similar format with 3 address assembly instructions, the adaptation period for unfamiliar users should not take very long, considering the developer already had some experience with Assembly.

5 SMT-LIB

Satisfiability Modulo Theories (SMT) have the aim of testing logical formulas for if they are satisfiable or not, using one or more theories. An SMT instance is a formula in first order logic, where some function and predicate symbols might have additional interpretations, and SMT is the problem of determining if such a formula is satisfiable. In other words, imagine an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is basically a binary-valued function of non-binary variables [5] [8].

One of the earlier attempts for solving SMT instances was translating them to SAT instances (e.g., a 32-bit integer variable would be encoded by 32 bit variables with appropriate weights) and passing this formula to a Boolean SAT solver because of the existing optimized SAT solvers. This approach, which is referred to as the eager approach, has its merits: by preprocessing the SMT formula into an equivalent Boolean SAT formula we can use existing Boolean SAT solvers "as-is" and leverage their performance and capacity improvements over time. However, the loss of the high-level semantics of the underlying theo-

ries means that the Boolean SAT solver has to work a lot harder than necessary to discover "obvious" facts (such as x + y = y + x for integer addition.) This observation led to the development of a number of SMT solvers that tightly integrate the Boolean reasoning of a DPLL-style search with theory specific solvers (T-solvers) that handle conjunctions (ANDs) of predicates from a given theory. This approach is referred to as the lazy approach [5] [7].

SMT-LIB is a functional language similar to LISP. SMT-LIB was created with the expectation that the availability of common standards and a library of benchmarks would greatly facilitate the evaluation and the comparison of SMT systems, and advance the state of the art in the field in the same way as, for instance, the TPTP library has done for theorem proving, or the SATLIB library has done initially for SAT [6].

Our approach can be summarized as follows: First, converting the multi-threaded code (we are working on C programs at the moment) to LLVM IR format with certain optimizations. Then converting this intermediate code to SMT-LIB format and creating a model of the program in a valid SMT formula. Lastly, giving this model to a SMT checker (MathSAT5) to find if the program has a bug for the given execution length.

6 What is LLMBC?

LLBMC (the low-level bounded model checker) is a static software analysis and verification tool for finding bugs in C (and, to some extent, in C++) programs. It is based on the technique of Bounded Model Checking. It is mainly intended for checking low-level system code. It takes sequential C/C++ programs and finds bugs and runtime errors. It can help software engineers to improve the quality of software and obtain stable and secure programs and reduce the time and effort needed for software testing.

LLBMC is fully automatic and requires minimal preparation efforts and user interaction. it models memory accesses (heap, stack, global variables) with high precision and is thus able to find hard-to-detect memory access errors like heap or stack buffer overflows. LLBMC can also uncover errors due to uninitialized variables or other sources of non-deterministic behaviours e.g. *integer overflow*, *division by zero*, *invalid bit shift*, *illegal memory access (array index out of bound, illegal pointer access, etc.)*, *invalid free*, *double free*, *user-customizable checks.*

We will explain more on these built-in-checks in subsequent sections.

The main limitation of LLBMC is its incompleteness. As it is just an implementation of the idea of BMC, its makes it incomplete due to incompleteness of the bounded analysis. Other limitations are it's high program-dependency and scalability.

7 Why Low-Level

We know that LLBMC takes a program in C code and finally reports bugs. However it does not directly work on source code. Applying BMC for verifying C programs is very hard and comes with many obstacles that have to be tackled. One of the most important differences is that the syntax (and thus semantics) of a high level programming language like C is much more complicated than a hardware description. Issues like memory allocation, (function) pointers, complex data structures, and function calls have to be managed. Further more going from C to C++ introduces more complex issues. Thats why instead of exploring the source code directly, LLBMC makes use of existing compiler technology and performs the analysis on an assembler-like compiler intermediate language. Such an intermediate language offers a much simpler syntax (and semantics), and thus eases a logical encoding of the verification problem considerably. We will describe more on how LLBMC uses LLVM-IR for the analysis. We can summarise the advantages of this approach as,

- The IR has much simpler syntax and semantics than C/C++. This makes it relatively easy to support (nearly) all language features.
- The program that is analysed is much closer to the program that is actually executed on the hardware since semantic ambiguities are resolved by the compiler. Furthermore, it becomes possible to analyze programs at various optimisation levels offered by the compiler.
- It becomes possible to analyse programs in any language for which a compiler frontend that produces the IR is available.

8 Software Bounded Model Checking

Here we briefly describe the main ideas of bounded model checking for software especially for programs written in C/C++. Generally programs are composed of data structures such as linked lists or trees. The unbounded nature of such data structures may give rise to infinite program runs. Which is very common in all kind of systems e.g. in reactive or interactive systems. Property checking of such programs is in general undecidable. Here bounded model checking provides a solution. BMC limits such program runs to finite ones, thereby achieving decidability. The bound is imposed by restricting the number of nested function calls and loop iterations that are considered. BMC performs function inlining and loop unrolling (up to these bounds), resulting in one large function that is then subject to further analysis. Since the analysis is done on the finite run of the program, it can only look for bugs up to specific depth. This is good enough for many applications e.g. embedded systems.

Here we will see an example of loop unrolling for a simple program. We assume that properties for the program are given by assertions from the user. And we need to verify that these properties holds for the program where the loop iteration is bounded by number k. The states that the program can reach within this bound are represented symbolically by a formula, together with the negation of the given condition. Then this combined formula is feed to SAT solver. If the formula is satisfiable, then there exists a path in the program that violates the property.

```
int a[N]; unsigned c;
  ...

  c = 0;
  for(i = 0; i < N; i++)
    if(a[i] == 0)
      c++;
```

$$
\begin{aligned}
& c_1 = 0 \ \wedge \\
& c_2 = (a[0] = 0)\,?\,c_1 + 1 \,:\, c_1 \ \wedge \\
& c_3 = (a[1] = 0)\,?\,c_2 + 1 \,:\, c_2 \ \wedge \\
& \cdots \\
& c_{N+1} = (a[N-1] = 0)\,?\,c_N + 1 : c_N
\end{aligned}
$$

Figure 1: A simple dummy picture to demonstrate how to include graphics into your report.

Consider the program in the left part of figure 1. The number of paths through this program is exponential in N, as each of the $a[i]$ elements can

be either zero or nonzero. Despite the exponential number of paths through the program, its states can be encoded with a formula of size linear in N, as demonstrated in the right part of the figure. This formula encodes the states of the original program on its left, using the static-single-assignment (SSA) form.

$$
\begin{aligned}
& c_1 = 0 \wedge \\
& ((a[0] = 0 \wedge c_2 = c_1 + 1) \vee (a[0] \neq 0 \wedge c_2 = c_1)) \wedge \\
& ((a[1] = 0 \wedge c_3 = c_2 + 1) \vee (a[1] \neq 0 \wedge c_3 = c_2)) \wedge \\
& \ldots \\
& ((a[N-1] = 0 \wedge c_{N+1} = c_N + 1) \vee (a[N-1] \neq 0 \wedge c_{N+1} = c_N)) \; .
\end{aligned}
$$

Figure 2: formula after all the rewrite is donw.

The ternary operator $c?x : y$ in the equation on the right of figure 1 can be rewritten using a disjunction, as shown in figure 2. After the rewrite we have the formula in DNF. Now to verify that some assertion provided by user holds at a specific, we add a constraint corresponding to the negation of this assertion. For example, to prove that at the end of the program $c \leq N$, we need to conjoin formula with $(c_{N+1} > N)$. Now we can test this combined formula for satisfiability.

In this example we have shown how the idea works on the source code level. But LLMBC works on LLVM-IR, so the implementation could be different but the original idea is same. LLBMC uses the LLVM libraries to do Function inlining and loop unrolling. Properties of a program are typically expressed as pre-condition($assume$) or post-conditions($assert$). Where assume states a pre-condition that is assumed to hold at its location and assert states a post-condition that is to be checked at its location. The program $Prog$ is correct if

$$Prog \wedge \bigwedge assume \Rightarrow \bigwedge assert$$

is valid. This check is decided using SAT or SMT solver. To report error traces back to the user, llbmc uses debug information generated by LLVM.

9 Logical Encoding

We now describe how an LLVM program can be transformed into a formula. After parsing the LLVM-IR program, a number of transformations are applied to it. e.g., loops are unrolled and functions are inlined a fixed number of times

and the control flow graph is simplified. This transformed program is then converted into ILR(internal logical representation). ILR is a representation of a formula in the logic of bit-vectors and arrays with some extensions.

Translation of LLVM?s three-address-code, memory access, address calculation, and bit-level instructions is straightforward, since these instructions are part of the theory of bit-vectors and arrays?or can easily be encoded into it.

Next the translation of phi instructions. For SMT solvers, a phi expression can be translated into a sequence of ITE (if-then-else) operators. And finally after the rewrite we have the formula in DNF.

The most complex part is the translation of memory instructions. the special semantics of memory allocation instructions like malloc and free are not handled by provided theory of arrays and bit-vectors τ_{ABV}. However llbmc detects following of memory related bugs:

- load from a non-allocated region of memory
- store to a non-allocated region of memory
- free of a non-allocated region of memory
- free of an already freed region of memory

This problem is fixed by introducing a precise formalisation of malloc and free as an SMT theory τ_H [4]. The special semantics of memory allocation instructions like malloc and free is defined by theory τ_H on top of theory τ_{ABV}. However this theory τ_H is not supported by off-the-shelf SMT solvers. So reduction to theory τ_{ABV} is needed. LLBMC use rewrite rules to simplify the memory formulas into ITE($ifthenelse$) formulas supported by off-the-shelf SMT solvers.

10 Previous work

A much longer LaTeX 2_ε example was written by Gil [1].

11 Results

In this section we describe the results.

Figure 3: A simple dummy picture to demonstrate how to include graphics into your report.

```
int a[N]; unsigned c;
  ...

  c = 0;
  for(i = 0; i < N; i++)
    if(a[i] == 0)
      c++;
```

$$
\begin{aligned}
c_1 &= 0 \ \wedge \\
c_2 &= (a[0] = 0)\,?\,c_1 + 1 \,:\, c_1 \ \wedge \\
c_3 &= (a[1] = 0)\,?\,c_2 + 1 \,:\, c_2 \ \wedge \\
&\cdots \\
c_{N+1} &= (a[N-1] = 0)\,?\,c_N + 1 : c_N
\end{aligned}
$$

Figure 4: A simple dummy picture to demonstrate how to include graphics into your report.

12 Conclusions

We worked hard, and achieved very little.

References

[1] Edmund Clarke, Armin Biere, Richard Raimi and Yunshan Zhu *Bounded Model Checking Using Satisfiability Solving.*

[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu *Symbolic Model Checking without BDDs.*

[3] Stephan Falke, Florian Merz, Carsten Sinz *The Bounded Model Checker LLBMC.* http://llbmc.org

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

Figure 5: Abstract LLVM IR grammar

[4] Stephan Falke, Florian Merz and Carsten Sinz *A Precise Memory Model for Low-Level Bounded Model Checking*. http://llbmc.org

[5] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[6] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891?921, 1905.

[7] Knuth: Computers and Typesetting, `http://www-cs-faculty.stanford.edu/~uno/abcde.html`

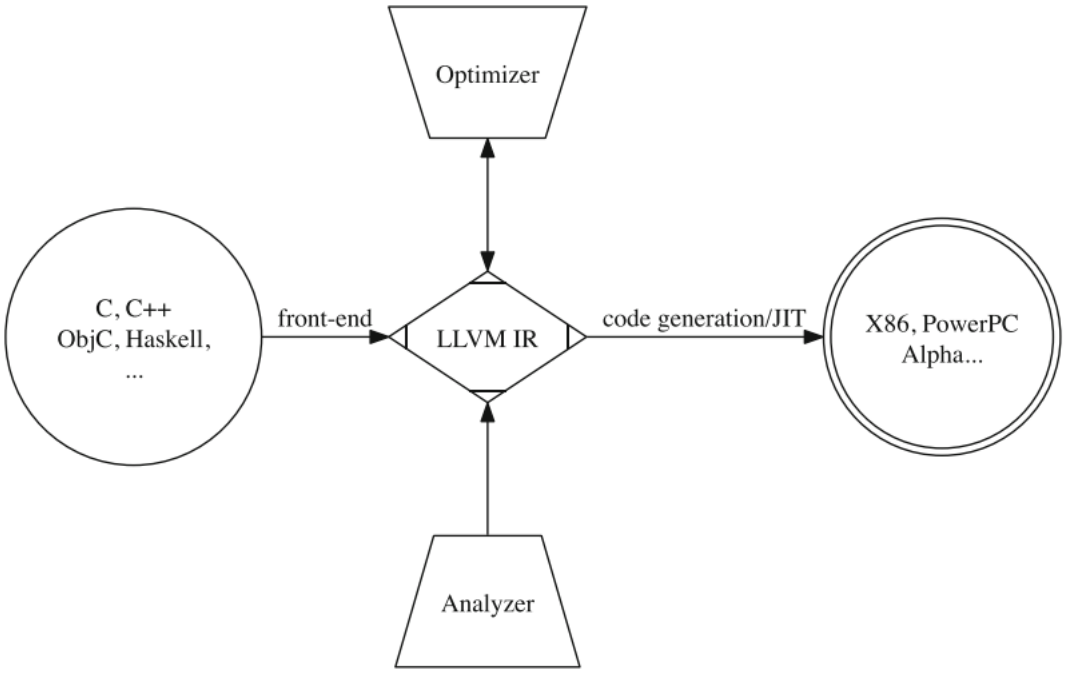

Figure 6: LLVM architecture