

# A Very Simple L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> 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 5, 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 ?? gives account of previous work. Our new and exciting results are described in Section ??. 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 [3]. 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 optimisation of programs written in different various programming languages. Initially designed for C and C++, LLVM now supports a lot of different high level languages [1].

LLBMC uses the LLVM compiler framework and its intermediate representation LLVM-IR. This makes it possible to use LLBMC on programs that are written in several programming languages, since compiler front-ends for, amongst others, C and C++, are available. LLVM's intermediate representation is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers. Each instruction of LLVM-IR is in static single assignment form (SSA), meaning that each is assigned only once and cannot be assigned again. SSA form helps a lot to simplify the analysis of dependencies among variables. For this reason LLVM-IR is used by LLBMC.

The use of SSA form makes it easier to write optimisations. LLVM compiler framework already provides a good list of optimisation passes. LLBMC internally runs LLVM's `mem2reg` pass in order to convert non-SSA form of LLVM IR into SSA form.

LLVM compiler framework also provide libraries for re-use. LLBMC uses the LLVM libraries to do few transformation( specially for function inlining and loop unrolling).

## 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 theories 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 LLBMC?

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 its 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. That's 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.

<pre> int a[N]; unsigned c; ... c = 0; for(i = 0; i &lt; N; i++)     if(a[i] == 0)         c++; </pre>	$ \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 \\ &\dots \\ &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.

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

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. We will see how llbmc models this idea of Bounded Model Checking. 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.

## 9 LLBMC's Built-In Checks

LLBMC has an extensive set of built-in checks for commonly occurring bugs in C programs and user defined checks (specified via C's `assert` function). Here we provide a brief description of this checks:

- **Arithmetic overflow and underflow:** Arithmetic overflow occurs when the result of a signed or unsigned arithmetic operation cannot be represented with the available number of bits.
- **Logic or arithmetic shift exceeding the bit-width:** Shift operations like  $n \ll l$  is undefined if  $l$  is larger than or equal to the bit-width of  $n$ . LLBMC supports checks for this kind of error by default since this behavior is not expected by most programmers.
- **Memory access at invalid addresses:** An access operation for an object on the heap is only valid if it is completely contained within a block of memory which was previously allocated using `malloc`. LLBMC detects invalid memory accesses on the stack, on the heap, and for global variables.
- **Invalid memory allocation:** Heap memory allocations are considered invalid if a memory block of the requested size can not be allocated.
- **Invalid memory de-allocation:** Only allocated memory is allowed to be de-allocated.
- **Overlapping memory regions in `memcpy`:** `memcpy` is used to copy the content of a block of memory from one location to another. if the source and destination blocks overlap then the result is undefined.
- **Memory leaks:** Memory leaks occur when blocks of memory are allocated, but never de-allocated.
- **User defined assertions:** LLBMC allows user defined properties expressed as `assert` or `assume`.
- **BMC specific assertions:** LLBMC is able to automatically detect insufficient bounds for nested function calls and loop iterations.

## 10 LLBMC approach

Here we will explain how llbmc model the verification problem. Figure 3 shows the overall structure of llbmc approach.

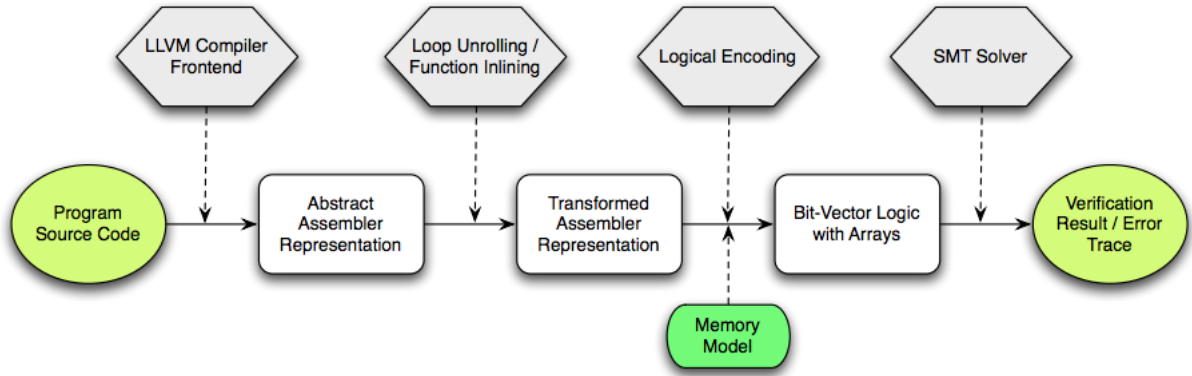


Figure 3: Overview of llbmc approach. Image is taken from [5].

### 10.0.1 LLVM Intermediate Representation

llbmc makes extensive use of llvm compiler framework. it uses the llvm compiler front end to get the program in LLVM abstract assembler from program source code. Since programs in LLVM assembler are already in *static single assignment (SSA)* form, assignments to scalar variables are treated as logical equivalences. However, other LLVM instructions need to be handled. In the context of llbmc the following LLVM instructions are important and we will show how each of these instructions are translated into the logic of bit-vectors and arrays.

- *Three-address-code (TAC)* instructions working on registers or constants.
- *Memory access and allocation* instructions, namely `load`, `store`, `malloc`, and `free`.
- *Address calculations* using `getelementptr`.
- *Branch and phi* instructions.

### 10.0.2 Function inlining and loop unrolling

Before starting the process of logical encoding, the program must be transformed (from llvm-ir to llvm-ir) so that it finishes with a finite run. llbmc uses code provided by the LLVM libraries



to do this function inlining and loop unrolling. The control flow graph get simplified after this transformation. After the loop unrolling the graph becomes directed acyclic graph. LLBMC uses the debug information generated by LLVM to report error trace back to user.

### 10.0.3 Logical Encoding

After function inlining and loop unrolling, llvm program is ready to be encoded into the logic of bit-vectors and arrays. The complete logical encoding is done in several stages. First the initial SMT encoding is done. Basic blocks are ( and their execution conditions) still present in this initial SMT encoding. We will briefly explain how llbmc handles few important llvm instructions.

*Handling of assignments:* Since llvm assembly is in SSA form, assignments are treated as equalities.

*Handling of Three-address-code (TAC) Instructions:* These instructions (like, e.g., add, cmp, mul, sign or zero extension) are supported by SMT solvers. So their translation is straightforward.

*Handling of phi Instructions:* Every phi instruction can be translated into a sequence of ITE (if-then-else) operators, which are also supported by SMT solvers. More on this can be found in [5].

*Handling of The getelementptr Instruction:* Every `getelementptr` instruction is simply transformed into a linear equation as shown bellow.

$$q = \text{getelementptr } p, o_1, \dots, o_k \rightarrow q = p + o_1.s_1 + \dots + o_k.s_k$$

where  $p, q$  are pointers and the  $o_i$ 's are offsets and the constant multiplicands  $s_i$  can be computed easily based on the data type definitions and the type of pointer  $p$ .

*Elimination of Branches:* At this point memory access instructions are need to handled. LLBMC models the memory as an array of bytes. The idea here is to translate llvm memory instructions e.g., `malloc`, `store`, `free`. LLBMC propose SSA form for the memory by introducing an explicit memory state and instructions `read`, `write`, `malloc`, `free`. Memory is accessed using `read` instructions and Memory is changed using `write`, `malloc` and `free` instructions. LLBMC introduce `phi` -instructions for memory states, which serve the same purpose as ordinary `phi` instructions for scalar values. At this point the conversion of memory to SSA form and encoding of `phi` instructions branches are no longer needed.

*Encoding Memory Constraints:* LLBMC provides following built-in memory checks.

- *Valid read/writes* (i.e. only to allocated memory).
- *Valid frees* (i.e. `free` is only called for a block of allocated memory).
- *No double frees* (i.e. no memory block is `free`'d twice).
- *No memory leaks* (i.e. all allocated heap memory is de-allocated when the program ends).

llbmc introduces few predicates ( and needed formulas) on the memory state: `valid_mem_access`, `deallocated`, `non_overlap`, `malloc_assumption`, `valid_free`, `no_memory_leaks`. The definitions of these memory consistency predicates can be found in [5]. Here we will see how these predicates helps to generate memory consistency constraints for *valid read/writes* checks. The idea here is to add appropriate guards before/after the memory access instructions. These guards are taken from the defined memory consistency predicates. For the *Valid read/writes* check, llbmc first add an `assume(malloc_assumption( m', p, s )` statement after each `m' = malloc(m, p, s)` instruction. Then llbmc add `assert(valid_mem_access(m, p, s))` before each `m' = write(m, p, x)` and `x = read(m, p)` instruction, where `s` is the size in bytes of the data to be read or written. Having processed all `mallocs`, `reads` and `writes`, the `malloc` and `free` instructions can be removed. After this stage of transformation, llbmc uses the defined formulas to translate the predicates into the logic of bit-vectors and arrays.

*Adding other checks to the ILR formula:* This initial ILR formula is annotated with LLBMC's other built-in checks. Most of these checks are supported by a predicate that is part of ILR. Then, an instruction that can possibly overflow is guarded by assertion that no overflow occurs and so on.

*Simplification of the ILR formula:* LLBMC uses term rewriting in order to simplify the ILR formula before passing it to an SMT solver. LLBMC implements approximately 150 (conditional) rewrite rules [7]. As all ILR's predicates for built-in checks are not supported by SMT-solver, those predicates are expanded. e.g. checks for logic and arithmetic shift which can easily be encoded in bit-vector logic. After expanding the predicates for the built-in checks and rewrite-based simplifications of the formula, it is ready to be passed to the SMT solver.

#### 10.0.4 Counterexample generation

The simplified ILR formula is then passed to an SMT solver. If the formula is satisfiable, any satisfying assignment corresponds to a bug in the program. By mapping ILR variables to the corresponding instructions in the LLVM-IR program and simulating execution with these values, a trace of the LLVM-IR program that exhibits the bug can be obtained.

## 11 Evaluation

LLBMC can be compared with two other BMC tools: the C Bounded Model Checker CBMC [9] and the Efficient SMT-Based 11Context-Bounded Model Checker ESBMC [8]. According to paper [7], an empirical evaluation on a large collection of C programs is conducted. The evaluation has shown that LLBMC compares favourably to CBMC and ESBMC, both in run-time and in number of found bugs. Furthermore, LLBMC has successfully been used on over 50 non-trivial C++ programs.

## 12 Conclusions

We worked hard, and achieved very little.

## References

- [1] Chris Lattner, Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. <http://llvm.org>
- [2] Edmund Clarke, Armin Biere, Richard Raimi and Yunshan Zhu. *Bounded Model Checking Using Satisfiability Solving*.
- [3] Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu. *Symbolic Model Checking without BDDs*.
- [4] Stephan Falke, Florian Merz, Carsten Sinz. *The Bounded Model Checker LLBMC*. <http://llbmc.org>
- [5] Stephan Falke, Florian Merz and Carsten Sinz. *A Precise Memory Model for Low-Level Bounded Model Checking*. <http://llbmc.org>.
- [6] Stephan Falke, Florian Merz and Carsten Sinz. *A Theory of C-Style Memory Allocation*. <http://llbmc.org/files/papers/SMT11.pdf>
- [7] Stephan Falke, Florian Merz and Carsten Sinz. *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR\**
- [8] Lucas Cordeiro, Bernd Fischer, Joao Marques-Silva. *SMT-based bounded model checking for embedded ANSI-C software*
- [9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A tool for checking ANSI-C programs*