Rheinisch-Westfälische Technische Hochschule Aachen
Research Group Computer Science 2
Prof. Dr. Jürgen Giesl

Seminar: Verification Techniques in WS 2014/15

Finding Hard Bugs in C with Bounded Model Checking

Meshkatul Anwer

Matrikelnummer 299508

Datum des Vortrages

Supervisor: Cornelius Aschermann

1 Introduction

Correctness of computer systems is important in our software dependent society. We depend more and more on these 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 is 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 is going on in this field. We need a highly automated method that provides a 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 2 gives an overview of the basic notions. Section 3 provides an overview of LLBMC tool. Section 5 gives a simple example. Section 7 gives an overview of LLBMC approach. Finally, Section 10 gives the remarks and conclusions.

2 Background

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 the *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 C and specification P determine whether or not the behaviour of C meets the specification P [15]. We need an automatic procedure which will use concrete mathematical reasoning to report design bug.

Software Model Checking

Model checking is an algorithmic method for determining if model of a system satisfies a correctness specification [9], [10]. A model of a program consists of *states* and *transitions*, and a specification

or *property* is a logical formula. A state of the program is an evaluation of the program counter, the values of variables and the configuration of the stack and the heap. Transitions describe how a program evolves from one state to another. Model checking algorithms exhaustively examine the reachable states of a program. This procedure is guaranteed to terminate if the state space is finite. If a state violating a correctness property is found, a *counterexample* an execution trace demonstrating the error) is produced. Model checking tools verify partial specifications, usually classified as *safety* or *liveness* properties. Where, safety properties express the unreachability of bad states, such as those in which an assertion violation, null pointer dereference or buffer overflow has occurred. Liveness properties express that something good eventually happens, like the condition that requests must be served eventually, or that a program must eventually terminate [11].

Bounded Model Checking

Model checking works only when the system has finite states. Which is not always true for the case of program or software system. Software systems or programs are inherently very complex and often exhibit non-terminating behaviour. Here comes Bounded Model Checking, the most commonly applied formal verification techniques in the semiconductor industry. The technique owes this success to the impressive capacity of propositional SAT solvers.

Bounded Model Checking is first proposed by Biere et. al. in 1999 [2] as as a complementary technique to BDD-based unbounded model checking. In BMC, the design under verification is unwound k times together with a property to form a propositional formula, which is then passed to the SAT solver. The formula is satisfiable if and only if there is a trace of length k that refutes the property. The technique is inconclusive if the formula is unsatisfiable, as there may be counterexamples longer than k steps. Nevertheless, the technique is successful, as many bugs have been identified that would otherwise have gone unnoticed. Later we will see how LLBMC implements this idea.

Satisfiability Modulo Theories

The original idea of SAT-based Bounded Model Checking (BMC) was to encode the bounded behaviours of the system as propositinal formula. The approach is ok as long as we are dealing with finite state systems (e.g. hardware circuits). When we try to apply the idea of BMC to software we are faced with new chalenges, as most programs are inherently infinite-state and new, non trivial issues such as the handling of (recursive) function calls and the modeling of complex data structures must be properly addressed. Instead of encoding the program into a propositional formula, we encode it into a quantifierfree formula. Then the formula is checked for satisfiability w.r.t. some given *background theory*. Later we weill see an example of such formula. According to [3], LLBMC uses an state-of-the-art SMT solver (Boolector or STP) to perform the satisfiability checking. This approach of enconding lead to considerably more compact formulae when arrays are involved in the

input program. The SMT solver also allows us to perform the satisfiability checking w.r.t. some combination of two (or more) decidable theories (say $\mathcal{T}_1 \cup \mathcal{T}_2$). In the context of programe verfication the following first order theories are interested, 1. *the theory of linear arithmetics*, 2. *the theory of lists*, 3. *the theory of arrays* , and 4. *the theory of bit-vectors*. LLBMC passes the final formula to the SMT solver (Boolector or STP) for the logic of bit-vectors and arrays. Both *the theory of bit-vector* and *the theory of arrays* are available as build-in theories in these two state-of-the-art SMT solver.

LLVM

LLVM (Low Level Virtual Machine) [14] 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.

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 converts 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).

3 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.

4 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.

5 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++;
```

$$
\begin{gathered}
c_1 = 0 \wedge \\
c_2 = (a[0] = 0) \ ? \ c_1 + 1 : c1 \wedge \\
c_3 = (a[1] = 0) \ ? \ c_2 + 1 : c2 \wedge \\
... \\
c_{N+1} = (a[N-1] = 0) \ ? \ c_N + 1 : c_N
\end{gathered}
$$

Figure 1: A simple program with loop and static-single-assignment (SSA) form of this program after unwinding its for loop.

Consider the program in the left part of figure **??**. 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.

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

Figure 2: formula after all the rewrite is done.

The ternary operator $c?x : y$ in the equation on the right of figure **??** can be rewritten using a disjunction, as shown in figure **??**. 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. Figures **??**, **??** and the example itself are taken from [12].

6 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: 1. **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. 2. **Logic or arithmetic shift exceeding the bit-width:** Shift operations like $n << l$ 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. 3. **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. 4. **Invalid memory allocation:** Heap memory allocations are considered invalid if a memory block of the requested size can not be allocated. 5. **Invalid memory de-allocation:** Only allocated memory is allowed to be de-allocated. 6. **Overlapping memory regions in `memcpy`:** `memcpy` is used the content of a block of memory from one location to another. if the source and destination blocks overlap then the result is undefined. 7. **Memory leaks:** Memory leaks occur when blocks of memory are allocated, but never de-allocated. 8. **User defined assertions:** LLBMC allows user defined properties expressed as assert or assume. 9. **BMC specific assertions:** LLBMC is able to automatically detect insufficient bounds for nested function calls and loop iterations.

7 LLBMC approach

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

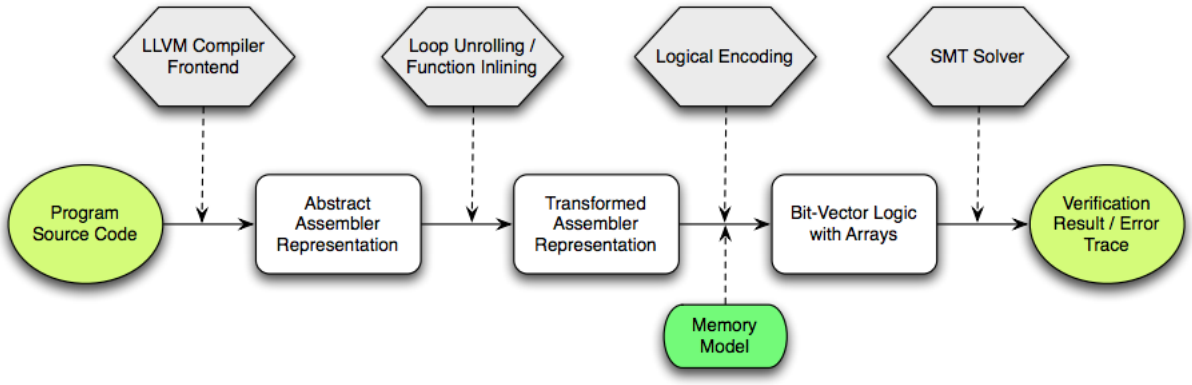

Figure 3: Overview of LLBMC approach. Image is taken from [4].

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 [13].

- *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.

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.

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 )` 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 [4].

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

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

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 [4]. 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 `malloc`s, `read`s and `write`s, 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 [6]. 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.

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.

8 Example

Here we will show an example of the each transformation.

9 Evaluation

LLBMC can be compared with two other BMC tools: the C Bounded Model Checker CBMC [8] and the Efficient SMT-Based 11Context-Bounded Model Checker ESBMC [7]. According to paper [6], an empirical evaluation on a large collection of C programs is conducted. The evaluation has shown

```
union U {
    char c[4];
    struct { int v: 31; int s: 1; } t;
    int i;
};
void __llbmc_main(char n) {
    union U *u; char *p; int i;
    u = malloc(sizeof(union U));
    p = u->c;
    u->t.s = 1;
    u->t.v = 0;
    p[0] = n;
    __llbmc_assert(u -> i == INT_MIN);
}
```

Figure 4: Example C program.

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.

10 Summary and Conclusion

As LLBMC is a bounded model checking tool, it inherits the shortcomings of BMC. However, it is the best technique to find shallow bugs, and it provides a full counterexample trace in case a bug is found. It supports the widest range of program constructions. This includes dynamically allocated data structures. On the other hand, completeness is only obtainable on very 'shallow' programs, i.e., programs without deep loops. In this paper we have explored how LLBMC implements the idea of Model Checking in the context of program verification. We have tried to present an overview on the fundamental ideas and implementation techniques. We hope there will be more and more such tools and will be used by developer communities.

References

[1] Edmund Clarke, Armin Biere, Richard Raimi, Yunshan Zhu. *Bounded Model Checking Using Satisfiability Solving*. Formal Methods in System Design 19.1 (2001): 7-34.

```
define void @__llbmc_main(i8 %n) {
entry:
  %0 = call i8* @malloc(i32 4)      ; u = malloc(sizeof(
  %1 = bitcast i8* %0 to i32*       ;              union U));
  store i32 -2147483648, i32* %1    ; u->t.s = 1; u->t.v = 0;
  store i8 %n, i8* %0               ; p[0] = n;
  %2 = load i32* %1                 ; u->i
  %3 = icmp eq i32 %2, -2147483648 ; == INT_MIN ?
  %4 = zext i1 %3 to i32
  call void @ llbmc assert(i32 %4)
  ret void
}
```

Figure 5: Example C program is converted into the given LLVM-IR program by the C front-end llvm-gcc.

[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu. *Symbolic Model Checking without BDDs.* Springer Berlin Heidelberg, 1999.

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

[4] Stephan Falke, Florian Merz, Carsten Sinz. *A Precise Memory Model for Low-Level Bounded Model Checking.* Proceedings of the 5th international conference on Systems software verification. USENIX Association, 2010.

[5] Stephan Falke, Florian Merz, Carsten Sinz. *A Theory of C-Style Memory Allocation.* Proc. SMT (2011): 71-80.

[6] Stephan Falke, Florian Merz, Carsten Sinz. *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR* Verified Software: Theories, Tools, Experiments. Springer Berlin Heidelberg, 2012. 146-161.

[7] Lucas Cordeiro, Bernd Fischer, Joao Marques Silva. *SMT-based bounded model checking for embedded ANSI-C software* Software Engineering, IEEE Transactions on 38.4 (2012): 957-974.

[8] Edmund Clarke, Daniel Kroening, Flavio Lerda. *A tool for checking ANSI-C programs* Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2004. 168-176.

[9] Edmund M. Clarke, E Allen Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*. Springer Berlin Heidelberg, 1982.

[10] Jean-Pierre Queille, Joseph Sifakis. *Specification and verification of concurrent systems in CE-SAR*. International Symposium on Programming, 1982 - Springer.

[11] Vijay D'Silva, Daniel Kroening, Georg Weissenbacher. *A Survey of Automated Techniques for Formal Software Verification*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27.7 (2008): 1165-1178.

[12] Daniel Kroening, Ofer Strichman. *Decision Procedures An Algorithmic Point of View*. Berlin: Springer, 2008. Print. ISBN 978-3-540-74104-6

[13] Daniel Kroening, Ofer Strichman. *Decision Procedures An Algorithmic Point of View*. Chapter 6 and 7, Berlin: Springer, 2008. Print. ISBN 978-3-540-74104-6

[14] Chris Lattner, Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. `http://llvm.org`

[15] E. Allen Emerson *The beginning of model checking: A personal perspective*. 25 Years of Model Checking. Springer Berlin Heidelberg, 2008. 27-45.

```
i8 %n = nondef()
i8* %0 = nondef()
heap %1 = malloc(%initialHeap, %0, i32__4)
bool %2 = valid malloc(%initialHeap, %0, i32__4)
assert(%2,"valid malloc")
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32__2147483648)
bool %5 = valid access(%1, %3, i32__4)
bool %6 = and(%2, %5)
bool %7 = not(%2)
bool %8 = or(%7, %6)
assert(%8,"valid store")
mem %9 = store(%4, %0, %n)
bool %10 = valid access(%1, %0, i32__1)
bool %11 = and(%6, %10)
bool %12 = not(%6)
bool %13 = or(%12, %11)
assert(%13,"valid store")
i32 %14 = load(%9, %3)
bool %15 = valid access(%1, %3, i32__4)
bool %16 = and(%11, %15)
bool %17 = not(%11)
bool %18 = or(%17, %16)
assert ( valid load , %18)
bool %19 = compare(EQ,%14,i32__2147483648)
bool %20 = and(%16, %19)
bool %21 = not(%16)
bool %22 = or(%21, %20)
assert(%22,"custom")
```

Figure 6: ILR formula obtained for the LLVM-IR program from Fig. 1.

```
i8 %n = nondef()
i8* %0 = nondef()
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32__2147483648)
mem %9 = store(%4, %0, %n)
i32 %14 = load(%9, %3)
bool %19 = compare(EQ, %14, i32__2147483648)
assert(%19, "custom")
```

Figure 7: ILR formula obtained by simplifying the ILR formula from Fig. 3.