The Bounded Model Checker LLBMC

Stephan Falke Florian Merz Carsten Sinz
Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{stephan.falke, florian.merz, carsten.sinz}@kit.edu

***Abstract*—This paper presents LLBMC, a tool for finding bugs and runtime errors in sequential C/C++ programs. LLBMC employs bounded model checking using an SMT-solver for the theory of bitvectors and arrays and thus achieves precision down to the level of single bits. The two main features of LLBMC that distinguish it from other bounded model checking tools for C/C++ are (i) its bit-precise memory model, which makes it possible to support arbitrary type conversions via stores and loads; and (ii) that it operates on a compiler intermediate representation and not directly on the source code.**

I. Introduction

Due to the use of unbounded data structures such as linked lists or trees, property checking of programs is in general undecidable. A promising approach for finding bugs and runtime errors in such programs is *bounded model checking*. While originally introduced in the context of hardware designs, the technique was quickly adapted for C programs by Clarke *et al.* [6]. Nowadays, bounded model checking is routinely used in an industrial setting, both for hardware and for checking a variety of aspects of embedded and low-level systems software.

The main idea of bounded model checking is to only consider finite program runs, thereby obtaining a decidable (but incomplete) method for finding bugs and runtime errors. Finiteness of program runs is achieved by restricting the number of loop iterations and nested function calls that are considered. A tool based on bounded model checking then typically performs loop unrolling and function inlining up to these bounds, resulting in one large function that only admits finite runs. This function is then subject to further analysis.

Writing a program analysis tool that supports all features of a high-level programming language such as C/C++ is a daunting task due to the complex and intricate syntax and semantics that these programming languages employ. Thus, it has recently become more and more popular to analyze programs not on the source code level but on the level of a compiler intermediate representation (IR) instead. This approach has several advantages:

- 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 analyzed is much closer to the program that is actually executed on the computer since semantical ambiguities have already been resolved by the compiler. Furthermore, it becomes possible to analyze programs at various optimization levels offered by the compiler.
- It becomes possible to analyze programs in any language for which a compiler frontend that produces the IR is available.

Concretely, LLBMC operates on LLVM's IR [14], which is an SSA-based abstract assembly language.

LLBMC is fully automatic and requires minimal preparation efforts and user interaction. The tool can help to

- reduce the time and effort needed for software testing,
- improve the quality of software, and
- obtain stable and secure software in shorter time.

Due to its high precision, LLBMC produces almost no false alarms (false positives). Due to the exhaustive list of built-in checks, many common bugs and runtime errors can be detected without relying on user-provided annotations. These built-in checks include:

- Integer overflow and underflow
- Division by zero
- Invalid bit shifts
- Illegal memory access (array index out of bound, illegal pointer access, etc.)
- Invalid `free`, including double `free`
- User-provided assertions (`assert`)

Limitations of LLBMC are the bounded analysis (which makes it incomplete) and program-dependent, sometimes restricted scalability.

The bounded model checker LLBMC is available under a non-commercial (academic) license or under a limited-time evaluation license at http://llbmc.org. The website also contains installation and usage instructions for LLBMC, including a video demonstrating LLBMC being used.

A. Related Tools

In 2004, Clarke *et al.* were the first to describe bounded model checking of C programs [6], resulting in the tool CBMC.

Also in 2004, NEC Laboratories America implemented a bounded model checking approach for C programs in the tool F-Soft as described in [13]. They differentiate their tool from CBMC by applying several static program analysis techniques on the control-flow graph in order to simplify the bounded model checking problem.

In 2009, Armando *et al.* extended CBMC to use SMT-solvers instead of encoding the problem directly into SAT [1]. Results

from that paper clearly show the benefits of using SMT-solvers instead of SAT-solvers as done by both CBMC and F-Soft.

In the same year, Cordeiro *et al.* presented ESBMC [7], which is based on CBMC but uses an SMT-solver instead of a SAT-solver. The main novelty of ESBMC is its added support for finding bugs in multi-threaded software. Recently, support for C++ programs was added to ESBMC [17].

Symbolic execution is a different approach to bug-finding in programs. In contrast to bounded model checking, which encodes all execution paths up to a bounded length in a single formula, symbolic execution performs a symbolic path exploration that considers the paths separately. The constraints obtained for each path are solved using SAT- or SMT-solvers. Recent symbolic execution tools include KLEE [5] for C programs and KLOVER [15], which extends KLEE for C++ programs. Both KLEE and KLOVER operate not on the source code level but on the level of LLVM's IR.

A recent tool that combines features of symbolic execution and bounded model checking and targets C programs is LAV [21]. Like KLEE, KLOVER, and LLBMC, the tool LAV also operates on the level of LLVM's IR.

II. A Brief Overview of LLBMC

The overall approach of LLBMC is summarized in Fig. 1, where further details can be found in [16], [20], [10].

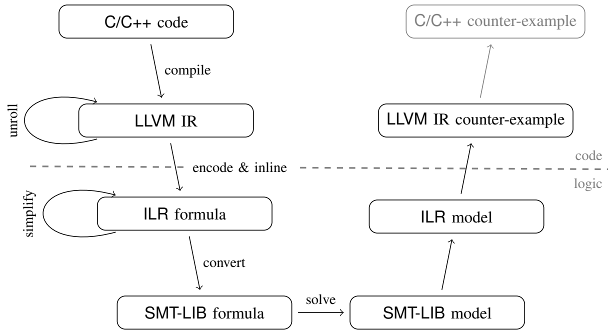

Fig. 1: Overview of LLBMC's approach.

The C/C++ program under investigation is first compiled into LLVM's IR using the existing compiler (frontend) clang. The unrolling of loops up to a user-provided or automatically determined bound is then performed as a transformation of the IR program using LLVM's comprehensive transformation framework. Subsequently, the IR program is encoded into LLBMC's intermediate logical representation ILR, which extends the logic QF_ABV of SMT-LIB [2] by various constructs for encoding the built-in checks of LLBMC and by high-level constructs for manipulating arrays [11], [9]. In contrast to loop unrolling, function inlining is done on-the-fly during encoding and not as a transformation of the IR program.[1]

Next, the ILR formula is simplified using an extensive set of rewrite rules. These rewrite rules are sufficient to discharge many "easy" proof obligations before invocation of an SMT-solver. Finally, the ILR formula is lowered into an SMT-LIB formula by expanding any remaining constructs related to the built-in checks. This SMT-LIB formula is then solved using the SMT-solver STP [12].

[1]In the future, we are planning to also perform the unrolling of loops on-the-fly during encoding.

If the SMT-LIB formula is satisfiable, then each model corresponds to a found bug or runtime error. A model of the SMT-LIB formula is then converted into a model of the ILR formula which is subsequently used in order to construct a counter-example (error trace) on the level of LLVM's IR within LLBMC. We are currently working on extending LLBMC to construct counter-examples on the level of C/C++. This construction relies on the debug information that is inserted by clang into the LLVM IR program.

An important feature that sets LLBMC apart from the tools surveyed in Sect. I-A is its bit-precise unified memory model for heap, stack, and global variables. LLBMC is thus able to find hard-to-detect memory access errors like heap or stack buffer overflows. Details on this can be found in [19], [8].

III. Usage Scenarios

This section presents two usage scenarios of LLBMC. In the first scenario, it is shown how user-defined assertions can be checked. In the process of doing this, the bug-finding capabilities of LLBMC using the built-in checks are demonstrated as well. The second usage scenario demonstrates how LLBMC can be used in order to check the equivalence of two functions.

A. Checking Assertions

Consider the simple C function shown in Fig. 2a which is supposed to compute the absolute value of an int. For ensuring correctness, the user wants to check that the return value is always non-negative. First, the C program is converted into LLVM's IR using

```
clang -c -emit-llvm -g abs.c -o abs.bc
```

which produces the file abs.bc containing the binary bitcode representation of LLVM's IR. Next, LLBMC is run on this bitcode file using

```
llbmc abs.bc
```

This results in the output given in Fig. 2b (on a 32-bit machine). The abs function thus contains an overflow bug which occurs when INT_MIN is passed to the function since -INT_MIN cannot be represented using 32 bits.

In order to not report overflow bugs, LLBMC can be run using the "--no-overflow-checks" command line argument, i.e., using

```
llbmc abs.bc --no-overflow-checks
```

An overflow occurring in the program is then modeled using two's complement arithmetic, just as it is done on most computers. In this case, LLBMC produces the output in Fig. 2c, showing that the user-defined assertion fails if INT_MIN is passed to abs. The reason for this is that -INT_MIN = INT_MIN in two's complement arithmetic, i.e., abs returns a negative number for this input.

```c
#include <assert.h>

int abs(int x)
{
    int ret;

    if (x >= 0) {
        ret = x;
    } else {
        ret = -x;
    }

    assert(ret >= 0);

    return ret;
}
```

(a) C program.

```
Error synopsis:
===============

A signed overflow occurs in
  %sub = sub nsw i32 0, %x, !dbg !17
where
  %x = -2147483648
and the result is
  %sub = -2147483648

Error location:
===============

Source file: abs.c
Line number: 10
Error context:

   5:     int ret;
   6:
   7:     if (x >= 0) {
   8:         ret = x;
   9:     } else {
  10:         ret = -x;
              ^^^^^^^^^

  11:     }

Stack trace:
============

#0 i32 @abs(i32 %x=-2147483648)
```

(b) Result produced by LLBMC.

```
Error synopsis:
===============

Assertion failed.

Error location:
===============

Source file: abs.c
Line number: 13
Error context:

   8:         ret = x;
   9:     } else {
  10:         ret = -x;
  11:     }
  12:
  13:     assert(ret >= 0);
          ^^^^^^^^^^^^^^^^^

  14:

Stack trace:
============

#0 i32 @abs(i32 %x=-2147483648)
```

(c) Result produced by LLBMC using the command line argument --no-overflow-checks.

Fig. 2: Example and output for the "checking assertions" usage scenario.

B. Checking Equivalence

User-defined assertions can also be used in order to verify equivalence of two functions that are supposed to compute the same return values. In order to see how this can be accomplished, consider the C program shown in Fig. 3.

The function popcount is supposed to compute the population count of an unsigned int, i.e., the number of bits that are set to 1 (assuming that an unsigned int consists of 32 bits). In order to verify that this is indeed the case, the driver function __llbmc_main compares the return value of popcount with the return value of a reference implementation.[2] Since x is passed as an argument to the driver function, the assertion fails if and only if popcount and the reference implementation disagree for some input value. LLBMC needs slightly longer than one second in order to show that the assertion never fails, i.e., popcount and the reference implementation return the same result for all 2^{32} possible values of x.

IV. Experiments

A. Comparison with Related Tools

In order to evaluate LLBMC's performance in comparison with other bounded model checking tools, we compared it with

[2]Currently, this driver function needs to be provided by the user. It would be easily possible to generate it automatically, similar to [18].

```c
#include <assert.h>

unsigned int popcount(unsigned int x)
{
    x = x - ((x >> 1 ) & 0x55555555);
    x = (x & 0x33333333) +
        ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x += (x >> 8);
    x += (x >> 16);
    return x & 0x0000003F;
}

unsigned int reference(unsigned int x)
{
    int i, s = 0;
    for(i = 0; i < 32; ++i)
        if(x & (1 << i))
            s++;
    return s;
}

void __llbmc_main(unsigned int x)
{
    assert(popcount(x) == reference(x));
}
```

Fig. 3: Example for the "checking equivalence" usage scenario.

CBMC [6] and ESBMC [7] in [16].[3] The 175 C programs for this comparison were taken from a variety of papers and sources (the benchmark collection is available at http://llbmc.org). The comparison did not include any C++ programs since ESBMC did not support C++ until very recently [17] and CBMC's C++ support is still very rudimentary.

The evaluation was performed on an Intel® Core™ 2 Duo machine with 2.4GHz. For each program, the memory limit was set to 2.5GB and the time limit was set to 15 minutes. The results of the comparison are summarized in Tab. I, while a more thorough discussion can be found in [16].

	S	O	F	I
LLBMC	172	1	0	2
CBMC 3.9	119	8	12	36
ESBMC 1.16	137	8	15	15

TABLE I: Results of the evaluation. "S" denotes the number of successfully solved instances (correctly detected bugs or absence of bugs proved), "O" the number of times the tool ran out of time or memory, "F" the number of failures to handle the program, and "I" the number of incorrect results (i.e., the tool reports a non-existing "bug" or misses a bug).

Note that LLBMC is able to successfully solve (i.e., find bugs or prove absence of bugs) over 25% more benchmarks than the second-best tool in the comparison and has a very low rate of false positives.

B. Participation in SV-COMP

In order to assess LLBMC's ability to operate on real-life programs, the tool participated in the 1st and 2nd International Competition on Software Verification (SV-COMP 2012 [3], [20] held at TACAS 2012 and SV-COMP 2013 [4], [10] held at TACAS 2013). Overall, LLBMC performed very well:

- In SV-COMP 2012, the participating tools were evaluated on 277 C programs in 6 categories. The size of the C programs ranged from 20 LOC to 182 kLOC. LLBMC participated in 5 categories and won the gold medal in the category "DeviceDrivers" and the silver medal in the category "HeapManipulation".
- In SV-COMP 2013, the participating tools were evaluated on 2315 C programs in 10 categories. The size of the C programs ranged from 13 LOC to 182 kLOC. LLBMC participated in 7 categories and won a total of 6 medals: gold in the categories "BitVectors" and "Loops" and silver in the categories "FeatureChecks", "HeapManipulation", "MemorySafety", and "ProductLines".

C. MPEG2 Case Study

For a further experiment, we tested LLBMC on the MPEG2 decoder of the MPEG Software Simulation Group available from http://www.mpeg.org. The source code of this decoder consists of more than 10 kLOC of C code. We checked the inverse fast discrete cosine transformation (IDCT) of this decoder with LLBMC and discovered a buffer overflow bug in function idctcol within 8 seconds. This bug may be exploited by passing a specially prepared video to the decoder.

[3]The bounded model checking tools F-Soft [13] and SMT-CBMC [1] mentioned in Sect. I-A are not publicly available.

V. Conclusions

In this paper, we have presented the bounded model checker LLBMC, a tool for finding bugs and runtime errors in sequential C/C++ programs. We have given a brief overview of LLBMC's approach. Using two representative usage scenarios, we have shown how LLBMC can be used by software engineers in order to improve the quality of software and obtain stable and secure programs. Finally, we have reported on experiments that demonstrate the performance of the tool.

References

[1] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *STTT*, vol. 11, no. 1, pp. 69–83, 2009.

[2] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," 2012. [Online]. Available: http://www.smt-lib.org

[3] D. Beyer, "Competition on software verification (SV-COMP)," in *TACAS 2012*, ser. LNCS, vol. 7214, 2012, pp. 504–524.

[4] ——, "Second competition on software verification (summary of SV-COMP 2013)," in *TACAS 2013*, ser. LNCS, vol. 7795, 2013, pp. 594–609.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI 2008*, 2008, pp. 209–224.

[6] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS 2004*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[7] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," in *ASE 2009*, 2009, pp. 137–148.

[8] S. Falke, F. Merz, and C. Sinz, "A theory of C-style memory allocation," in *SMT 2011*, 2011, pp. 71–80.

[9] ——, "Extending the theory of arrays: memset, memcpy, and beyond," in *VSTTE 2013*, ser. LNCS, 2013, to appear.

[10] ——, "LLBMC: Improved bounded model checking of C programs using LLVM (competition contribution)," in *TACAS 2013*, ser. LNCS, vol. 7795, 2013, pp. 623–626.

[11] S. Falke, C. Sinz, and F. Merz, "A theory of arrays with set and copy operations (extended abstract)," in *SMT 2012*, 2012, pp. 97–106.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV 2007*, ser. LNCS, vol. 4590, 2007, pp. 519–531.

[13] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *TCS*, vol. 404, no. 3, pp. 256–274, 2008.

[14] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO 2004*, 2004, pp. 75–88.

[15] G. Li, I. Ghosh, and S. Rajan, "KLOVER: A symbolic execution and automatic test generation tool for C++ programs," in *CAV 2011*, ser. LNCS, vol. 6806, 2011, pp. 609–615.

[16] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," in *VSTTE 2012*, ser. LNCS, vol. 7152, 2012, pp. 146–161.

[17] M. Ramalho, M. Freitas, F. Sousa, H. Marques, L. Cordeiro, and B. Fischer, "SMT-based bounded model checking of C++ programs," in *ECBS 2013*, 2013, pp. 147–156.

[18] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *CAV 2011*, ser. LNCS, vol. 6806, 2011, pp. 669–685.

[19] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *SSV 2010*, 2010.

[20] C. Sinz, F. Merz, and S. Falke, "LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution)," in *TACAS 2012*, ser. LNCS, vol. 7214, 2012, pp. 542–544.

[21] M. Vujošević-Janičić and V. Kuncak, "Development and evaluation of LAV: An SMT-based error finding platform," in *VSTTE 2012*, ser. LNCS, vol. 7152, 2012, pp. 98–113.