Abstract. Bounded model checking (BMC) of C and C++ programs is challenging due to the complex and intricate syntax and semantics of these programming languages. The BMC tool LLBMC presented in this paper thus uses the LLVM compiler framework in order to translate C and C++ programs into LLVM’s intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts as used in C and static/dynamic casts as used in C++. An empirical evaluation shows that LLBMC compares favorable to the related BMC tools CBMC and ESBMC.

1 Introduction

Bounded model checking (BMC) [3], introduced by Biere et al. in 1999, is a popular technique for bug finding and verification of hardware designs that is widely used in an industrial setting. For bug finding of software, BMC of C programs was introduced by Clarke et al. in 2004 [8], and has shown its strength in checking a variety of aspects of embedded and low-level system software (see, e.g., [16,23]). Tools implementing BMC for C programs include CBMC [8] (developed by D. Kröning et al.), F-Soft [15] (developed at NEC Laboratories America), SMT-CBMC [1] (developed by A. Armando et al.), and ESBMC [10] (developed by L. Cordeiro et al.).

To build a BMC tool that supports all language features of a high-level language like C or C++ reliably, including common non-standard extensions that are used by, e.g., the GCC compiler, is a daunting task. This is mostly due to the complex syntax and intricate, sometimes ambiguous, semantics of these languages. The bounded model checker LLBMC presented in this paper therefore performs BMC not on the source code level but on the level of a compiler intermediate representation (IR). This approach offers a range of advantages:

* This work was supported in part by the “Concept for the Future” of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.
The compiler IR possesses a much simpler syntax and semantics than C/C++ and thus eases a logical encoding considerably. Furthermore, most features of C and C++ can be supported without much effort.

The program that is analyzed is much closer to the program that is actually executed on the computer since ambiguities of C/C++’s semantics have already been resolved. Furthermore, it becomes possible to find bugs introduced by the compiler.

In producing the IR, compilers already use program optimizations that can also result in simplified BMC problems.

The use of a compiler IR makes it possible to perform BMC on programs written in a variety of programming languages.

The use of an IR makes LLBMC, to the best of our knowledge, the only BMC tool that can be successfully applied to non-trivial C++ programs (CBMC contains rudimentary support for C++ but failed to analyze nearly all of the over 50 C++ programs we tried it on). A drawback of using an IR is that bugs that are (intuitively) present in the C/C++ program may be “optimized away” by the compiler (but notice that the bugs would then also not occur during execution of the program if the same compiler is used to produce the executable).

Besides using a compiler IR, LLBMC offers the following key features:

**Large Set of Built-In Checks:** LLBMC provides a comprehensive set of built-in checks which are described in detail in Sect. 2.

**Extensive Simplification:** LLBMC uses simplification techniques on different levels. First, using the optimizations of the compiler front-end generates smaller and simpler IR programs. In particular, memory-related compiler optimization techniques (e.g., moving memory operations to registers whenever possible) can simplify the BMC problem significantly. Second, rewriting techniques are used on the logical representation, e.g., to propagate constants or to simplify arithmetical and Boolean expressions.

**Memory is Modeled as a Flat Byte-Array:** This design decision is in contrast to what is implemented in many other tools (e.g., pre-3.9 versions of CBMC [8] or deductive verification tools such as VCC [9]) which use a typed memory model. In a typed memory model, memory is a collection of typed objects rather than a sequence of bytes. Using a byte-array makes it possible to support programs which make use of C’s weak type system, e.g., by converting an `int` to a sequence of `char`s that are written to a file. Another example is the use of a `union` for the conversion between types. Typically, modeling memory on the byte level causes a performance penalty in BMC, but LLBMC uses simplification techniques that, according to an empirical evaluation, compensate for this.

Section 2 recalls BMC of software and discusses the built-in checks of LLBMC. The compiler framework LLVM is briefly introduced in Sect. 3 while Sect. 4

1 Currently, LLBMC does not support floating-point numbers, exception handling and run-time type information (RTTI).