SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration*

Jan Mrázek, Petr Bauch, Henrich Lauko and Jiří Barnat

Faculty of Informatics, Masaryk University
Botanicka 68a, 602 00 Brno, Czech Republic
{xmrazek7,bauch,xlauko,barnat}@fi.muni.cz

Abstract. We present SymDIVINE: a tool for bit-precise model checking of parallel C and C++ programs. It builds upon LLVM compiler infrastructure, hence, it uses LLVM IR as an input formalism. Internally, SymDIVINE extends the standard explicit-state state space exploration with SMT machinery to handle non-deterministic data values. As such, SymDIVINE is on a halfway between a symbolic executor and an explicit-state model checker. The key differentiating aspect present in SymDIVINE is the ability to decide about equality of two symbolically represented states preventing thus repeated exploration of the state space graph. This is crucially important in particular for verification of parallel programs where the state space graph is full of diamond-shaped subgraphs.

1 Introduction

Automatic program analysis, e.g. detection of use of invalid memory or division by zero, is commonly used by both academia and industry for some time now. On the other hand, automatic program verification has not been widely accepted by the industry and remains mostly exclusively within the academic interest. This situation did not change even after the arrival of modern multi-core CPUs that made the concurrency related problems such as data races quite common, yet difficult to detect and solve by humans — an ideal opportunity for automated formal verification. The reasons for failure are numerous, however, the most tampering factor is the need for remodeling the input program in a modeling language of the model checker [1].

To address this specific issue, we present SymDIVINE– a tool for verification of real parallel C and C++ programs with non-deterministic inputs. The tool is built on top of the LLVM framework in order to avoid the need of modeling and, at the same time, to achieve precise semantics of C and C++ programming languages. SymDIVINE is motivated as an extension of our purely explicit model checker DIVINE [3] that is capable of handling full parallel C/C++ programs without inputs. To properly handle programs with inputs, SymDIVINE relies on Control-Explicit Data-Symbolic approach [2], which we detail below.

* This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

2 Control-Explicit Data-Symbolic Approach

In the standard explicit state model checking, the state space graph of a program is explored by an exhaustive enumeration of its states. SymDIVINE basically follows the same idea, but it employs a *control-explicit data-symbolic* approach to alleviate the state space explosion caused by the non-deterministic input values. While a purely explicit-state model checker has to produce a new state for each and every possible input value, in SymDIVINE a set of states that differ only in data values is represented with a single data structure, the so called *multi-state*. Multi-state is composed of explicit control location and a set of program's memory valuations. The model checking engine in SymDIVINE operates on multi-states, which is the key differentiating factor of SymDIVINE if compared to other, purely explicit approaches. Relying on multi-states is computationally more demanding, but may bring up to exponential time and memory savings. See Figure 1. Moreover, with an equality check for multi-states, we can easily mimic most explicit-state model checking algorithms – from simple reachability of error states to full LTL model checking [6].

2.1 Representation of Multi-States

To perform verification of a program of size that is interesting from an industrial point of view, an efficient data structure for representation of multi-states is needed. While the representation of the explicit control-flow location is straightforward and easy, the representation of the set of program's variable valuations (the symbolic part) is rather challenging. We have tested several different representations during the development of SymDIVINE. Since our aim was to stick with a bit-precise verification, we only considered representations that were suitable for that. In particular, we dealt with binary decision diagrams, integer intervals and SMT formulae [4]. For the rest of the paper, we refer to the symbolic part as *symbolic data*.

In the current version of SymDIVINE, only quantified bit-vector SMT formulae are supported to represent symbolic data. The tool does not support dynamic memory allocation and manipulation at the moment, which makes the representation of symbolic data much simpler. Nevertheless, an unambiguous program variable naming scheme needs to be established so that different local variables with the same name are uniquely identified. Note that identifying variables with the index of a function they belong to, and an offset within the function in the LLVM bitcode is not satisfactory for the purpose of verification. In such scheme, we cannot differentiate the individual instances of the same variable within different function calls during recursion or in the presence of parallel threads. To deal with that we adopted the natural way the program's memory is organized – a stack divided into segments. Each function call made is linked with the corresponding segment on the stack, and so is every live instance of a variable. Therefore, individual instances can be identified by the index of the stack segment and an offset within that segment. Note that the first segment on the stack is reserved for global variables.

```
%a = call i32 @__VERIFIER_nondet_int()
%b = icmp sge i32 %a, 65535
br i1 %b, label %5, label %6
```

The code represents a simple LLVM program, where `a` is initialized with a non-deterministic 32-bit integer, then it is checked whether it is greater or equal to 65535. The result of the check is stored to `b` and used for branching.

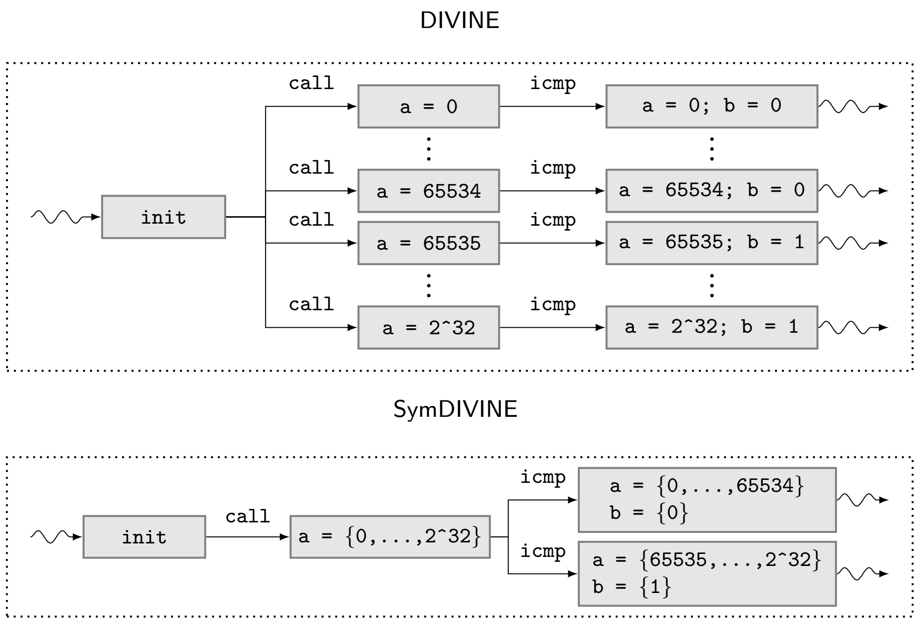

Fig. 1. The figure compares state exploration in the explicit approach of DIVINE and in the control-explicit data-symbolic approach of SymDIVINE on LLVM program example. From `init` state DIVINE explores states for every possible value of `a` (2^{32} values), hence exponentially expands state space. In contrast SymDIVINE approach of symbolic representation generates only two different states. One where the condition on branching ($a \geq 65535$) is satisfied and the other one where the condition is violated.

Another issue the model checker has to deal with is the fact that for some variables, old values have to be remembered. To that end, SymDIVINE maintains the so called *generations* of variables. These are used to keep previous values of variables that have been redefined since the beginning of the execution of the program. Basically, each assignment to a variable generates a new generation of it. Consider, for example, the following C code: `int x = 5; int y = x; x = 42;` after execution of which the model checker have to remember that the variable `y` equals to an old value stored at the variable `x`.

Symbolic data part of a multi-state itself is further structured. In particular, it contains two sections – the so called *path condition* and *definitions*. The *path condition* is a conjunction of formulae that represents a restriction of the data that have been collected during the branching along the path leading to the current location. *Definitions*, on the other hand, are made of a set of formulae in the form *variable* = *expression* that describe internal relations among variables. *Definitions* are produced as a result of an assignment and arithmetic instructions. The structure of symbolic data representation allows for a precise description of what is needed for the model checking, but it lacks the canonical representation. As a matter of fact, the equality of multi-states cannot be performed as a syntax equality, instead, SymDIVINE employs an SMT solver and quantified formulae to check the satisfiability of a path condition and to decide the equality of two multi-states. For more details, we kindly refer to [2].

2.2 State Space Generation

SymDIVINE is built on top of the LLVM framework to simplify interpretation of complicated C/C++ semantics. To generate successors of a given multi-state we have implemented an interpreter of a subset of LLVM instructions. When generating new multi-states, the interpreter first checks if the input multi-state has a satisfiable path condition. If not, no successors are generated and the multi-state is marked as *empty* (invalid state). In the other case, the control location is systematically advanced for one of the threads, the corresponding instruction is executed and a new multi-state is emitted. After that the input multi-state is restored and the procedure is repeated for other threads. In this way, all thread interleavings are generated at the very fine-grained level of individual LLVM instructions. This would result in an enormous state space explosion unless SymDIVINE employed τ-reduction [7] to avoid emitting of invisible multi-states. With τ-reduction the penalty for fine-grained parallelism is eliminated. Moreover, to avoid repeated exploration of already visited states, a set of already seen symbolic data is maintained for each control location. Only new multi-states are further explored. Note that since there is no canonical representation for the symbolic data part, linear search is used to check for the presence of a multi-state in the list. At the moment, SymDIVINE relies on the Z3 SMT solver. To further reduce the length of symbolic data lists associated with the individual control-flow locations, SymDIVINE employs the so called *explication* [2]. If the definition part for a single variable leads to one single possible data value, the variable is removed from the symbolic data part of the multi-state and is treated as a regular explicit value. The process of explication is rather expensive, but it pays off, as it reduces the number of SMT calls made due to the multi-state equality tests.

As for interpretation of LLVM bitcode, most instructions (including arithmetic instructions) are implemented with the corresponding formula manipulation in the *definitions* section of a multi-state. Branching instructions for a condition φ always produce two succeeding states, where φ is conjuncted with the path condition of the first successor, and $\neg\varphi$ with the condition of the second successor. Function calls result in a initialization of a new stack segment, upon function return, the variables in the stack segment from where the function was called are substituted with the returned values and the corresponding stack segment is deleted. To support parallel C/C++ programs, SymDIVINE contains its own limited implementation of PThread library.

3 Using SymDIVINE

Given a C/C++ program, its verification using SymDIVINE is quite straightforward and simple. We adopted SV-COMP notation [5] to mark a non-deterministic input of the verified program. Using this notation a user can bring the input to the program by calling `__VERIFIER_nondet_{type}` function. We also support input assumptions, atomic sections, and asserts from SV-COMP notation. Beside this denotation of non-deterministic inputs, no other annotation is needed. To verify the annotated program, it has to be first compiled into the LLVM bitcode using Clang. The user can either do it manually with any compiler flags needed, or may use our script `compile_to_bitcode` to compile the program source code with the default compiler flags. After that the user has to choose if the program should be verified for an assertion safety or against an LTL property. To verify the program for the assertion safety, the user has to run `./symdivine reachability {program.ll}`. Optional arguments `verbose` or `vverbose` can be used to track the progress of verification. If there is no run violating any assertion (both C and SV-COMP style) SymDIVINE responds that the model is safe. Otherwise, it outputs a single state in which the assertion is violated.

To verify the program against an LTL property, the user has to run `./symdivine ltl {property} {program.ll}`. The LTL formula is passed as a plain text and is parsed by SymDIVINE internally. The format follows the standard LTL syntax. Atomic propositions are embedded into the formula and are bounded within square brackets. An atomic proposition can refer to any global variable in the verified program or a constant with a given bit-width. Since the support for debugging information in the bit code is not fully implemented yet, the global variables are referred to using their offset in a global segment (this offset can be read in the bitcode file). Note that for a bitcode file, Clang keeps the same order of variables as is the order of the variables in the source file. An example of LTL formula for SymDIVINE is as follows: `!F(G[seg1_off0 = 0(32)])`.

SymDIVINE does not currently support the debug information stored in bit code files, so all counterexamples are in the form of internal representation and with no link to the original source code file. However, since the internal representation follows strict and simple rules and the information obtained from the path condition is clearly readable, it is possible for a user to reconstruct the counterexample simply by following it in the source code file. This is currently the weakest part of SymDIVINE user interface.

4 Conclusions and Future Work

The main advantage of SymDIVINE is the fact that it performs a direct bit-precise verification of C/C++ programs with no need for modeling. Using a bit-vector theory, SymDIVINE can be bit-precise and handle bit-shifts, unsigned overflows, etc. Unlike symbolic executors or bounded model checkers, SymDIVINE also handles programs with infinite behavior, provided that the semi-symbolic state space is finite. The LLVM approach allows us to reflect compiler optimizations and architecture specific issues such as bit-width of variables. With a proper LLVM frontend, SymDIVINE is also applicable to various programming languages.

In the current state SymDIVINE is able to verify pieces of real world code. These pieces are, however, limited by the subset of LLVM instructions that is supported by our tool. The most limiting factor for SymDIVINE is the lack of support for dynamic memory. Besides that, our tool also misses advanced techniques that reduce resource usage and are incorporated within other tools, such as efficient memory compression. Absence of these techniques makes our tool more resource wasteful compared to the others. However, majority of the limitations are purely of technique nature and will be solved in the future. From the conceptional point of view, SymDIVINE approach does not deal well with cycles whose number of iterations depends on an input. SymDIVINE also cannot handle programs that run an infinite number of threads. However, this is not a limiting factor for real world use.

On the other hand, SymDIVINE demonstrates that the *Control-Explicit Data-Symbolic* approach can be used for verification of parallel programs with nondeterministic inputs, and we plan to further support it. SymDIVINE source code can be found at `https://github.com/yaqwsx/SymDIVINE`.

References

1. J. Alglave, A. Donaldson, D. Kroening, and M. Tautschnig. Making Software Verification Tools Really Work. In *ATVA*, pages 28–42, 2011.
2. J. Barnat, P. Bauch, and V. Havel. Model Checking Parallel Programs with Inputs. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 756–759, 2014.
3. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
4. P. Bauch, V. Havel, and J. Barnat. LTL Model Checking of LLVM Bitcode with Symbolic Data. In *MEMICS*, volume 8934 of *LNCS*, pages 47–59. Springer, 2014.
5. D. Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer Berlin Heidelberg, 2015.
6. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
7. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods (NFM 2013)*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.