Optimizing and Caching SMT Queries in SymDIVINE

(Competition Contribution)

Jan Mrázek(✉), Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat

Faculty of Informatics, Masaryk University, Brno, Czech Republic
jan.mrazek@mail.muni.cz

Abstract. This paper presents a new version of the tool SymDIVINE, a model-checker for concurrent C/C++ programs. SymDIVINE uses a control-explicit data-symbolic approach to model checking, which allows for the bit-precise verification of programs with inputs, by representing data part of a program state by a first-order bit-vector formula. The new version of the tool employs a refined representation of symbolic states, which allows for efficient caching of SMT queries. Moreover, the new version employs additional simplifications of first-order bit-vector formulas, such as elimination of unconstrained variables from quantified formulas. All changes are documented in detail in the paper.

1 Verification Approach and Software Architecture

SymDIVINE is a model checker that primarily aims for verification of parallel C and C++ programs. In contrast to explicit-state model checker [2], SymDIVINE represents data values symbolically and can therefore handle programs with inputs, which would otherwise cause state-space explosion due to the number of possible input values. In particular, SymDIVINE uses the control-explicit data-symbolic (CEDS) approach to model checking in which control-flow of the program is represented explicitly and values of data structures are represented symbolically [1,7].

We now describe the approach in more detail. In a CEDS model checker, each generated state is a triple that contains a control part (program counter for each thread), explicit data storage, and symbolic data storage. The explicit data storage keeps values of constants and of variables whose values are uniquely determined. The symbolic data storage represents a set of possible values of program variables by a first-order formula in the theory of bit-vectors. To generate the state space, SymDIVINE explores all possible evaluations of the program and tracks the effect of program instructions on the explicit values and on the formula representing the symbolic values. To avoid exploring infeasible paths, an SMT solver is used to check satisfiability of the formula representing the data

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

© Springer-Verlag GmbH Germany 2017
A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 390–393, 2017.
DOI: 10.1007/978-3-662-54580-5_29

values. The current version of SymDIVINE relies on the SMT solver Z3 [6]. For purposes of the competition we used version 4.4.1. Additionally, in order to avoid generating unnecessary thread interleavings, SymDIVINE collapses steps invisible to other threads into a single transition using the τ-reduction algorithm [3].

In addition to verification of safety properties, SymDIVINE also supports verification of properties specified in LTL. To check such properties, SymDIVINE uses standard LTL model checking algorithms based on detection of accepting cycles in the product of the program with the Büchi automaton. However, in order to detect accepting cycles, SymDIVINE has to be able to test states for equality. The equality of states is represented as a quantified bit-vector formula, which is handed to an SMT solver [1]. Use of the state equality test can also reduce the state space, as the same state is represented and explored only once. On the other hand, the equality test requires potentially expensive quantified SMT reasoning.

To increase performance of SymDIVINE, we added several optimizations in the latest version. The first one, state slicing, is a new method of state representation. In this representation, the symbolic part of the state is represented by multiple independent formulas that describe sets of variables that do not affect each other. This allows for more efficient emptiness and equality tests as query results can be cached and smaller queries (related only to the changed program variables) can be issued. Moreover, the issued queries are usually smaller and can, in many cases, be handled by internal SymDIVINE optimizations, like checking for the syntactic equality of formulas, without the need to query the SMT solver. The motivation for state slicing comes from the observation regarding the verified LLVM bitcode. As the LLVM bitcode is in the single static assignment (SSA) form, individual instructions usually affect only a few variables. These local changes are often independent of the rest of the state. This is not just the case for concurrent programs, but also for sequential programs containing repeated function calls or non-trivial loops. We have also implemented caching, which can leverage the decomposition of the issued SMT queries to independent parts.

The second optimization is the integration of formula simplifications based on elimination of unconstrained variables [4,5] (i.e. variables that occur only once in the formula) from quantified bit-vector formulas. The effectivity of such simplifications also follows from the SSA form of LLVM: the formulas generated by SymDIVINE often contain many unconstrained variables. Although the elimination of unconstrained variables in quantifier-free formulas is provided by standard SMT solvers, we have extended the approach to quantified formulas, which is necessary for equality queries generated by SymDIVINE. Therefore, we have implemented our own elimination of unconstrained variables from quantified bit-vector formulas in SymDIVINE.

From the implementation point of view, SymDIVINE can be seen as three components – an LLVM interpreter, a state representation and an exploration algorithm. The algorithm uses the interpreter to produce successors of each state and uses emptiness and equality tests provided by the state representation to detect empty (unreachable) or already visited states. An overview of this architecture can be seen in Fig. 1. In the picture, the SMT *store* refers to

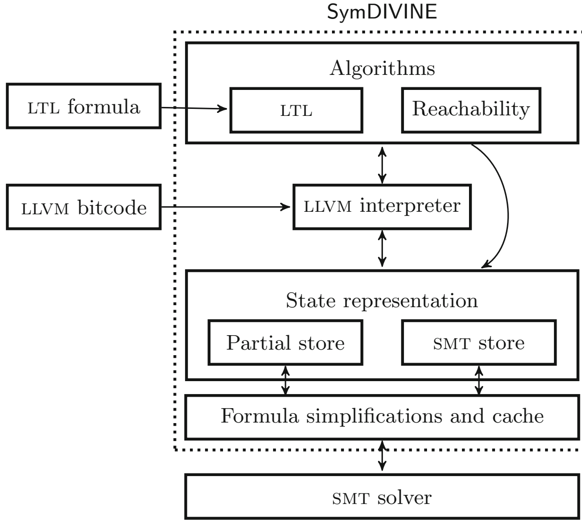

Fig. 1. High-level overview of the SymDIVINE architecture. Nested boxes correspond to interfaces and their concrete implementations.

the original storage of states and the *partial store* refers to the newly implemented storage using state slicing. Both storages are available and users can use whichever they prefer. The entire tool is written in C++ and leverages the LLVM framework. Thanks to the well-defined interface, each of the three main components is easily interchangeable.

2 Strengths and Weaknesses

The main strength of the approach is its universality: although it is aimed at parallel programs, SymDIVINE is applicable to all competition categories except termination, heap manipulation and overflows. SymDIVINE can also verify programs in multiple programming languages, as it uses the LLVM bitcode as the input format.

SymDIVINE is also precise: it can find every race condition in the program regardless of the necessary number of context switches, and thanks to the symbolic representation in the bit-vector theory, the verification is also bit-precise. Moreover, unlike symbolic execution or bounded model checkers, SymDIVINE also handles programs with infinite behaviour provided that their state space is finite. The usage of the LLVM infrastructure allows to precisely capture compiler optimizations and architecture-specific issues such as the bit width of variables.

On the other hand, the approach does not deal well with loops with number of iterations dependent on an input. In the worst-case scenario, SymDIVINE unrolls the cycle completely, resulting in an enormous state space. SymDIVINE also cannot handle programs that spawn an infinite number of threads or allocate memory from the heap. Support for other SMT solvers is not currently implemented in SymDIVINE.

3 Tool Setup and Configuration

In order to run SymDIVINE, `libboost-graph`, `Z3` and `clang-3.5` have to be installed. If LTL model checking is requested, `ltl2tgba` is also required.

A prebuilt package of the tool (version 0.5) can be downloaded from a GitHub release[1]. The archive contains binaries for SymDIVINE and also a run script that eases the process of verification by automatically compiling C/C++ files to the LLVM bitcode. To verify a C program, run `run_symdivine <symdivine_dir> [options] <benchmark>`, where `<symdivine_dir>` is a directory in which the SymDIVINE executable is located. All available options can be listed by using the switch `--help`. We decided to opt-out from categories Arrays, BitVectors, Heap Data Structures and Floats. The tool should be run with options `--fix_volatile --fix_inline --silent -Os`.

4 Software Project and Contributors

SymDIVINE source code can be found on GitHub[2] under the MIT License. The tool is developed at the Faculty of Informatics, Masaryk University, and includes contributions by the authors of this paper, Petr Bauch, and Vojtěch Havel.

References

1. Barnat, J., Bauch, P., Havel, V.: Model checking parallel programs with inputs. In: PDP, pp. 756–759 (2014)
2. Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., Weiser, J.: DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 863–868. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_60
3. Barnat, J., Brim, L., Ročkai, P.: Towards LTL model checking of unmodified thread-based C & C++ programs. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 252–266. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28891-3_25
4. Brummayer, R.: Efficient SMT solving for bit vectors and the extensional theory of arrays. Ph.D. thesis, Johannes Kepler University of Linz (2010)
5. Bruttomesso, R.: RTL verification: from SAT to SMT(BV). Ph.D. thesis, University of Trento (2008)
6. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
7. Mrázek, J., Bauch, P., Lauko, H., Barnat, J.: SymDIVINE: tool for control-explicit data-symbolic state space exploration. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 208–213. Springer, Cham (2016). doi:10.1007/978-3-319-32582-8_14

[1] https://github.com/yaqwsx/SymDIVINE/releases/download/v0.5/symdivine.zip.
[2] https://github.com/yaqwsx/SymDIVINE.