DIVINE: Explicit-State LTL Model Checker
(Competition Contribution)

Vladimír Štill(✉), Petr Ročkai, and Jiří Barnat

Faculty of Informatics, Masaryk Universit, Brno, Czech Republic
xstill@mail.muni.cz, divine@fi.muni.cz

Abstract. DIVINE is an LLVM-based LTL model checker that follows the standard automata-based approach to explicit-state model checking. It aims at verification of unmodified parallel C & C++ programs without inputs. To achieve this DIVINE employs several reduction techniques combined with high-performance parallel and distributed computing.

1 Verification Approach and Software Architecture

As an explicit-state model checker, DIVINE is meant primarily to help detect bugs in multithreaded code [1]. As a matter of fact, the development of multithreaded code suffers from the lack of deterministic testing procedure. Therefore, concurrency related bugs, such as data races, often tend to survive in the code even beyond the release date. DIVINE provides the user with the tool to check all possible relevant executions of multithreaded code. In this way DIVINE may be used to prove the presence or absence of a bug. With this approach DIVINE requires that programs to be verified are closed, i.e. perform no input/output actions.

DIVINE is written in C++. It uses LLVM bitcode as the input formalism. Therefore, it employs Clang to translate input multithreaded C and C++ programs to LLVM bitcode prior verification. See Fig. 1. Thus the core part of DIVINE is a purpose specific LLVM bitcode interpreter. The interpreter allows to completely store and load a state of the program, and is capable of execution of LLVM instructions in order to generate new states. The program is analyzed including all the code that is executed within software libraries which has to be compiled together with the program for verification. In the standard distribution DIVINE provides bitcode with implementations of C and C++ standard libraries and `pthread` threading library.

2 Strengths and Weaknesses

The main strength of DIVINE is its ability to perform a full deterministic verification of closed piece of code. DIVINE can detect a number of issues in the code

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

© Springer-Verlag Berlin Heidelberg 2016
M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 920–922, 2016.
DOI: 10.1007/978-3-662-49674-9_60

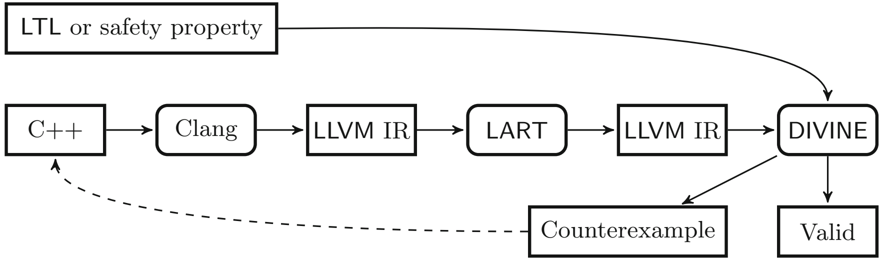

Fig. 1. Verification work-flow. Boxes with rounded corners represent executables.

such as invalid memory access, assertion violation, unhandled exceptions, etc. In addition, DIVINE can verify properties expressed as LTL formulas. Moreover, all the issues discovered can be witnessed with a counterexample.

The LLVM interpreter in DIVINE supports complete instruction set of LLVM bitcode including instructions for exception handling. DIVINE runtime provides almost complete implementation of C and C++ standard libraries and pthread threading library. The LLVM approach has the advantage that the behaviors that are analyzed by DIVINE are quite close to the behaviors that are actually exhibited by the program binary, for example they include most of compiler optimizations. Futhermore, with a proper runtime, DIVINE can handle other languages with LLVM-based compiler.

The ease with which LLVM-bitcode can be transformed allowed us to adapt to specifics of SV-COMP (such as atomic sections) without the need to modify DIVINE at all. For LLVM-to-LLVM transformations DIVINE employs LART— an LLVM transformation platform distributed with DIVINE.

To address the state space explosion problem in terms of both the time and memory, DIVINE offers strong τ-reduction [2], efficient state-compression techniques [3] and also the ability of parallel and distributed-memory processing.

DIVINE requires the program to have finitely many states, however the program need not terminate — there is no need for loop, recursion, or context switch bounding. On the other hand, there are numerous limits of the approach. First of all, DIVINE is purely explicit-state tool, which means that simulating even a single unrestricted 32bit-wide input leads to the 2^{32} wide branching in the state space, making verification of open programs nearly impossible. However, since the nondeterminism in the concurrency category of SV-COMP is fairly limited, DIVINE can tackle most of the benchmarks of this category.

When preparing for SV-COMP, we also run into problems with underspecification of benchmarks — in many benchmarks there is undefined behavior with respect to reads and writes to global variables, which leads to an optimized LLVM bitcode with unexpected behavior. This is, however, not a limitation of DIVINE's approach — it is rather a bug in the benchmarks. To tackle this problem and get expected results we employ LLVM-to-LLVM transformation which adds volatile qualifier to any global variable defined in the benchmark.

3 Tool Setup and Configuration

The web presentation of DIVINE can be found at divine.fi.muni.cz, however, for the purpose of this competition we use not yet released version of DIVINE which can be downloaded from www.fi.muni.cz/~xstill/divine-next, the version used is DIVINE 3.4.1pre. DIVINE can be built on Linux, but it requires the following packages: gcc and g++ at least version 4.9, LLVM and Clang 3.7, and perl 5.

The complete prebuild package can be downloaded from www.fi.muni.cz/~xstill/divine-next/bin/divine-3.4-svcomp.tar.gz. The archive contains DIVINE and LART binaries together with all the necessary dependencies as well as Clang and LLVM `otp` for convenience, therefore, there is no need to install LLVM 3.7 to run DIVINE on Ubuntu 14.04.

Since the build process of C/C++ program for DIVINE has multiple steps, there is a helper script `rundivine` which handles compilation and verification automatically. The usage for SV-COMP is `rundivine <divine-bin-dir> --svcomp --csdr --opt=-Oz <benchmark>`. The meaning of used options is the following: `--svcomp` to run all required LART passes and setup compiler to handle input properly and DIVINE to verify assertions, use only one thread, and use compression; `--csdr` to use the Context-Switch-Directed Reachability algorithm [4]; and `--opt=-Oz` to enable optimizations using LLVM `opt`.

DIVINE will participate in concurrency category, with aforementioned options to the `rundivine` wrapper. The wrapper script for BenchExec is `divine.py`[1].

4 Software Project and Contributors

DIVINE project resides at http://divine.fi.muni.cz. The project was contributed primarily by Petr Ročkai and Vladimír Štill, with a number of other people as contributors. DIVINE is licenced under the 2-clause BSD license.

References

1. Barnat, J., et al.: DiVinE 3.0 – an explicit-state model checker for multithreaded C & C++. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, pp. 863–868. Springer, Heidelberg (2013)
2. Ročkai, P., Barnat, J., Brim, L.: Improved state space reductions for LTL model checking of C and C++ programs. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 1–15. Springer, Heidelberg (2013)
3. Ročkai, P., Štill, V., Barnat, J.: Techniques for memory-efficient model checking of C and C++ code. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 268–282. Springer, Heidelberg (2015)
4. Štill, V., Ročkai, P., Barnat, J.: Context-switch-directed verification in DIVINE. In: Hliněný, P., Dvořák, Z., Jaroš, J., Kofroň, J., Kořenek, J., Matula, P., Pala, K. (eds.) MEMICS 2014. LNCS, vol. 8934, pp. 135–146. Springer, Heidelberg (2014)

[1] github.com/dbeyer/benchexec/blob/master/benchexec/tools/divine.py.