Extending DIVINE with Symbolic Verification using SMT* **
(Competition Contribution)

Henrich Lauko***, Vladimír Štill, Petr Ročkai and Jiří Barnat

Faculty of Informatics, Masaryk University, Brno, Czech Republic
divine@fi.muni.cz

Abstract. DIVINE is an LLVM-based verification tool focusing on analysis of real-world C and C++ programs. Such programs often interact with their environment, for example via inputs from users or network. When these programs are analyzed, it is desirable that the verification tool can deal with inputs symbolically and analyze runs for all inputs. In DIVINE, it is now possible to deal with input data via symbolic computation instrumented into the original program at the level of LLVM bitcode. Such an instrumented program maintains symbolic values internally and operates directly on them. Instrumentation allows us to enhance the tool with support for symbolic data without substantial modifications of the tool itself. Namely, this competition contribution uses SMT formulae for representation of input data.

1 Verification Approach and Software Architecture

DIVINE is an explicit-state model checker primarily designed to detect bugs in multithreaded programs [5]. Testing of multithreaded programs is a known hard problem because of nondeterminism in the execution caused by thread interleavings. To deal with control flow nondeterminism, DIVINE exhaustively explores all relevant executions of the multithreaded program. Unfortunately, this explicit approach fails to deal with data nondeterminism caused by communication with the environment. In order to verify a program with inputs, DIVINE would need to examine all the possible inputs of the program. This would cause enormous state-space explosion and would be unmanageable in reasonable time and space.

The traditional way to cope with input values during verification is to represent them symbolically – i.e., to perform symbolic execution on the program. In DIVINE it would be sufficient to extend the LLVM interpreter to work with input values symbolically and adapt the exploration algorithm to work with symbolic states, similarly as other tools do [1, 4, 2]. However, this would make the core

* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S and by Red Hat, Inc.

** The final publication is available at Springer via https://doi.org/10.1007/978-3-030-17502-3_14

*** SV-COMP jury member: xlauko@mail.muni.cz

of the verification procedure more complicated and possibly slow it down, introduce bugs and/or reduce maintainability and extensibility. Hence, we have decided to shift the responsibility for symbolic values from the verifier to the verified program [3]. Instead of (re-)interpreting instructions symbolically, we translate symbolic instructions into equivalent explicit code which performs the computation symbolically. The transformation performs a dependency analysis on symbolic values of the program and translates symbolic instructions. By providing a set of symbolic operations as a library, we obtain a program that manipulates symbolic values. The method is further described in Figure 1.

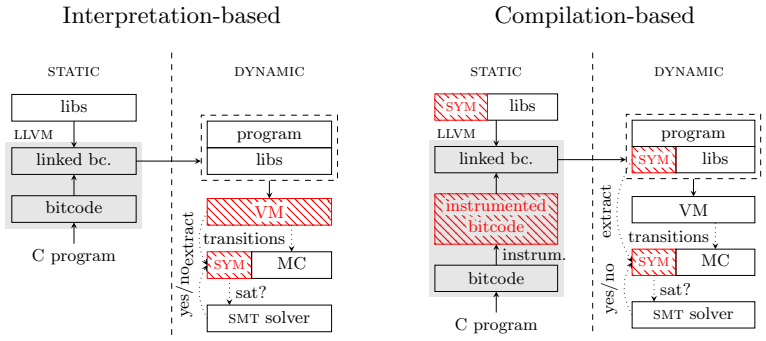

Fig. 1. Comparison of interpretation-based approach and compilation-based approaches. All manipulations of symbolic values are denoted by red color. In both cases, the virtual machine generates transitions in the state space and passes them to the model checker (MC), which performs safety analysis. In the compilation-based approach, symbolic operations are instrumented into the program, while in the interpretation-based one, they are the responsibility of the VM.

In order to maintain efficiency we do not transform the entire program, but only the parts that might come into contact with symbolic values. As shown in Figure 2, the program is analyzed starting from `input` points, and all downstream operations are augmented (`s_add`, `s_eq`), but concrete computation remains unchanged (`fun`). The transformed program uses a special operation called `lift`, which takes a concrete value and returns a symbolic one. The result of lifting `*` represents an arbitrary input value.

In comparison to standard programs, a program with symbolic values might not have deterministic control flow. When a program contains a branch which depends on a symbolic value, both outcomes might be possible.[1] To capture such behavior in the transformed program, we introduce a nondeterministic choice and execute both branches. We take advantage of the fact that DIVINE is already

[1] Given a symbolic value `x` and a branch with condition `x < 5`, the condition can be both true and false. The program makes a nondeterministic choice and extends the path condition with $x < 5$ or $x \geq 5$ respectively.

```
a:int ← input()
b:int ← fun(7)
c:int ← add(a, b)
d:bool ← eq(a, b)
```

```
a:s_int ← lift(*)
b:int   ← fun(7)
c:s_int ← s_add(a, lift(b))
d:s_bool ← s_eq(a, lift(b))
```

Fig. 2. Transformation to the program working with symbolic values (right).

```
a:s_int ← lift(*)
b:s_int ← lift(*)
c:s_int ← s_add(a, b)
if nondet()
  assume(c < 5)
  d:s_int ← s_mul(c, 3)
```

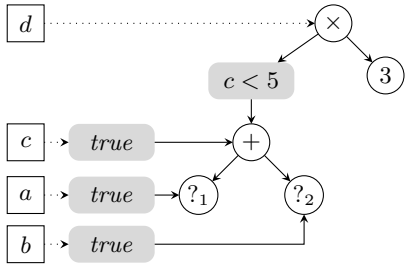

Fig. 3. The transformed program builds term trees that represent symbolic values. The boxes correspond to symbolic variables while the circles are the concrete representation of terms. Question marks denote unconstrained nullary symbols. Gray boxes represent path condition constraints.

capable of handling nondeterminism. Further, in the taken path we constrain values by extending a path condition (see Figure 3).

In the program, symbolic data are represented as term trees – see Figure 3. Exploring the state space, DIVINE extracts term trees from program states in the model checking algorithm, and checks for the feasibility by querying SMT solver (Z3) for satisfiability of extracted path condition. Moreover, DIVINE needs to recognize when it has reached a repeated state. This can not be done by a simple comparison of states, because different symbolic states may represent the same set of concrete states. Hence, to check equality of states, we also utilize the SMT solver. To precisely model program arithmetic, we use the bitvector theory.

2 Strengths and Weaknesses

In comparison to bounded model checkers, DIVINE's strength is sound verification – it explores a whole state-space and uses formulae in bitvector theory to precisely represent symbolic values. However DIVINE produced a few wrong results in the competition, these should not be possible in theory and likely stem from implementation errors in the verification tool.

Our compilation-based approach has allowed us to increase modularity of the tool. It is easy to change the representation of symbolic values, the verification algorithm and even the entire verifier while preserving the transformation. Another upside is that the implementation of symbolic operations is subject to checks performed by the verifier.

On the other hand, the current implementation is only a proof of concept. Our primary goal was to show that a compilation-based approach may compete with interpretation-based approaches even though it increases the size of the verified program and therefore possibly also verification complexity. Currently, the transformation can only handle scalar values, hence verification of programs with symbolic memory is not yet possible.

3 Tool Setup and Configuration

The verifier archive can be found on the SV-COMP 2019 page[2] under the name DIVINE-SMT. In case the binary distribution does not work on your system, we also provide a source distribution and build instructions at `https://divine.fi.muni.cz/2019/sv-comp-smt`.

It is usually sufficient to run divine as follows: `divine check --symbolic --svcomp TESTCASE.c`. This command runs DIVINE with the SMT-based representation of symbolic data described in this paper and with SV-COMP-specific instrumentation.

For SV-COMP benchmarks, additional settings are handled by the `divine-svc` wrapper.[3] The only option used for DIVINE-SMT is `--32` for 32 bit categories. The wrapper sets DIVINE options based on the property file and the benchmark. In particular, it enables symbolic mode if any nondetermism is found, sequential mode if no threads are found, and it sets which errors should be reported based on the property file. It also generates witness files. More details can be found on the aforementioned distribution page.

DIVINE participates in all categories, but it can only produce non-unknown results for the error reachability and memory safety categories.

4 Software Project and Contributors

The project home page is `https://divine.fi.muni.cz`. Many people have contributed to DIVINE, including Petr Ročkai, Henrich Lauko and Vladimír Štill. DIVINE is open source software distributed under the ISC license.

References

1. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
2. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Computer Aided Verification*. Springer, 2014.

[2] `https://sv-comp.sosy-lab.org/2019/systems.php`

[3] To be found in the main directory of the binary archive, or in the `tools` directory of the source distribution. Usage: `divine-svc DIVINE_BINARY PROP_FILE [OPTIONS] TESTCASE.c`.

3. Henrich Lauko, Petr Ročkai, and Jiří Barnat. Symbolic computation via program transformation. *Theoretical Aspects of Computing – ICTAC 2018*, 2018.
4. Quoc-Sang Phan, Pasquale Malacaria, and Corina S. Păsăreanu. Concurrent bounded model checking. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, 2015.
5. Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with llvm and graph memory. *Journal of Systems and Software*, 143:1–13, 2018.