DIVINE: Extended Compilation-based Symbolic Verification[⋆]
(Competition Contribution)

Zuzana Baranová, Lukáš Korenčik, Henrich Lauko, Adam Matoušek, Petr Ročkai and Vladimír Štill

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

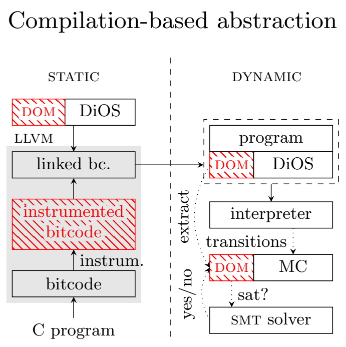
Abstract. DIVINE is an explicit-state model checker based on the LLVM framework with a focus on real-world C and C++ programs. Verification in DIVINE covers a wide variety of aspects of these languages like concurrency, memory safety, verification of programs with exceptions as well as programs that interact with an operating system, including file system and POSIX syscalls. Furthermore, these programs usually interact with their environment and obtain nondeterministic values. In DIVINE, we tackle data nondeterminism via symbolic computation that is instrumented into the original program on the level of LLVM bitcode. Using this approach, we limit modifications of DIVINE's internal interpreter which can remain purely explicit.
Up till now, DIVINE was able to instrument an abstraction of scalar integers. This year we have enriched DIVINE's abilities by instrumentation of floating-point values abstraction. Moreover, we present instrumentation for array abstractions, which allows DIVINE to allocate symbolically large arrays and compete on a broader set of benchmarks.

1 Verification Approach and Software Architecture

DIVINE is designed to separate the responsibilities of verification to autonomous modules [4]. We distinguish main components as a virtual machine, i.e. LLVM interpreter [4], state-space exploration algorithms (i.e., model-checker MC), static analysis together with instrumentation of LLVM [3], and support layer for the program environment in the form of standard C snd C++ libraries and verification dedicated operating system – DiOS [5]. DiOS is a lightweight POSIX system, whose primary responsibility is to take care of the scheduling of threads or processes.

To handle non-deterministic values, DIVINE employs SMT-based symbolic representation. It would be possible to maintain and manipulate symbolic data directly in the interpreter. However, in order to not complicate its internals,

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



Example of instruction replacement

```
%z = add %x %y
```

```
%z = call @sym_add(%x, %y)
```

```
term sym_add(value x, value y) {
  if (is_constant(x))
    x = smt::lift(x);
  if (is_constant(y))
    y = smt::lift(y);
  return smt::add(x, y);
}
```

Fig. 1. Instrumentation replaces instructions that operate on nondeterministic values by abstract instructions (e.g., `add` → `sym_add`). The abstract domain is provided as a C++ library (DOM). The domain implements abstract operations – lifting of constants and manipulation with SMT representation.

we shift responsibility for symbolic values to the verified program itself. This is performed using an LLVM-to-LLVM transformation in the following steps:

1. A *dataflow analysis* determines which LLVM instructions come into contact with non-deterministic values.
2. All affected instructions are replaced by their abstract counterparts[1].
3. Bitcode is linked with the implementation of symbolic operations.

This approach is actually more general and allows us to change the domain of scalar values by providing an implementation of an arbitrary *abstract domain* in step 3. To perform a bit-precise verification, we have chosen to use a *term domain* that represents data by SMT formulae in bit-vector theory. In this domain operations manipulate formulae representations of data, alternatively extend a global path condition by program constraints (see Figure 1).

The instrumented program is executed on an explicit virtual machine, while symbolic values are maintained in the program's memory. The exploration algorithm has to extract symbolic formulae from the program's memory in order to decide the feasibility of the current state and equality with the previously visited states. These queries are decided by an external SMT solver (either Z3 [1] or STP [2] depending on the used theory).

Floating-point values: To support abstraction of nondeterministic floating-point values, we have extended the term domain by implementation of symbolic counterparts to LLVM instructions on floating-point values. Moreover, we abstract calls to standard library math functions whenever an equivalent operation in SMT theory exists (e.g., `fabs` – computes absolute value). In this

[1] The abstract operation takes care of lifting possibly concrete arguments into a symbolic representation, and calls the implementation of symbolic operation on lifted arguments.

way, we keep more semantic information about operations for an SMT solver. Since DIVINE's primary SMT solver is STP, which does not support a floating-point theory, we switch to Z3 solver to answer floating-point theory queries.

Array abstraction: In compilation-based abstraction, it would be convenient to instrument abstraction of arrays in a similar manner as scalar values. However, abstract arrays behave differently to scalars. The main difference is that arrays may contain values from other abstract domains (either scalars or arrays). Consequently, the array-manipulating operations (`load`, `store`, `getelementptr`) need to operate on multiple domains. To capture this behavior we introduce *domain categories*:

1. *Scalar domains* defines arithmetic and relational operations on scalar values, and respect value semantics (i.e., each operation creates a new value).
2. *Aggregate domains* abstract program objects and allows for parametrization of the domain of values kept in the aggregate.

Moreover, in comparison to scalar values that respect value semantics, array values need to work as pointers to a shared object, i.e., each reference to an array in the program consists of a pointer to a shared symbolic array and a symbolic offset to that array. Hence updates of an array also become visible to its other references.

Two abstract domains parametrize our presented *array domain*. The first domain is used for the representation of array content, and the second is used to represent offset to the array, as well as the size of the array.

For the competition run, we have used the *term domain* to represent array content as well as its offsets. This allows us to utilize the SMT theory of arrays with the already used representation of scalar values. Moreover, this representation also simplifies transitions between aggregate and scalar domains since loads from symbolic arrays are already in the term domain. Since SMT array theory has strongly typed arrays, we instantiate all of them as byte arrays. Consequently, a `load` of a larger object performs multiple loads, which are concatenated to the resulting value. Similarly, a `store` performs multiple stores for each byte of the storing object.

2 Strengths and Weaknesses

Besides the limitation of interpreter modifications, the advantage of static transformation is that it is performed only once on the bitcode, and the subsequent verification with possible refinement can be executed without repeated transformation.

Even though the array domain allows us to perform analysis on array manipulating code, we reach limits of the symbolic representation on benchmarks that iterate over a symbolic variable (e.g., size of a symbolic array). Such benchmarks result in possibly infinite loop unrolling. In the future, we intend to parametrize the array domain by bounded abstract domains (e.g., congruence domains).

3 Tool Setup and Configuration

The verifier archive can be found on the SV-COMP 2020 page[2] under the name DIVINE. 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/2020/sv-comp`.

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, the `divine-svc` wrapper handles additional settings.[3] The only option used for DIVINE is `--32` for 32 bit categories. The wrapper sets DIVINE options based on the property file and the benchmark. In particular, DIVINE enables symbolic mode if any nondeterminism 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`. DIVINE is open source software distributed under the ISC license. Active contributors to the tool are listed as authors of this paper.

References

1. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
2. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, page 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.
3. Henrich Lauko, Petr Ročkai, and Jiří Barnat. Symbolic computation via program transformation. *Theoretical Aspects of Computing – ICTAC 2018*, 2018.
4. 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.
5. Petr Ročkai, Zuzana Baranová, Jan Mrázek, Katarína Kejstová, and Jiří Barnat. Reproducible execution of posix programs with dios. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 333–349, Cham, 2019. Springer International Publishing.

[2] `https://sv-comp.sosy-lab.org/2020/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`.