Model Checking of C and C++ with DIVINE 4*

Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera,
Henrich Lauko, Jan Mrázek, Petr Ročkai, Vladimír Štill

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

Abstract. The fourth version of the DIVINE model checker provides a modular platform for verification of real-world programs. It is built on an efficient interpreter of LLVM code which, together with a small, verification-oriented operating system and a set of runtime libraries, enables verification of code written in C and C++.

1 Introduction

Building correct software is undoubtedly an important goal for software developers and we firmly believe that formal verification methods can help in this endeavour. In particular, explicit-state model checking promises to put forth a deterministic testing procedure for non-deterministic problems (such as parallel programs or tests which use fault injection). Moreover, it is quite easy to integrate into common test-based workflows. The latest version of DIVINE aims to make good on these promises by providing an efficient and versatile tool for analysis of real-world C and C++ programs.

2 DIVINE 4 Architecture

The most prominent feature of DIVINE 4 is that the runtime environment for the verified program (i.e. support for threads, memory allocation, standard libraries) is not part of the verifier itself, but instead it is split into several components, separated by well-defined interfaces (see Figure 1). The three most important components are: the DIVINE Virtual Machine (DiVM), which is an interpreter of LLVM code and provides basic functionality such as non-determinism and memory management; the DIVINE Operating System (DiOS), which takes care of thread management and scheduling; and finally libraries, which implement standard C, C++ and POSIX APIs. The libraries use syscalls to communicate with DiOS and hypercalls to communicate with DiVM.

The verification core below DiVM is responsible for verification of safety and liveness properties and uses DiVM to generate the state space of the (non-deterministic) program.

* This work has been partially supported by the Czech Science Foundation grant No. 15-08772S and by Red Hat, Inc.

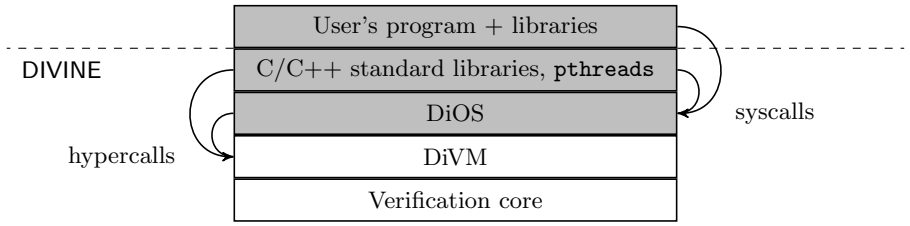

Fig. 1. Overview of the architecture of DIVINE 4. The shaded part consists of LLVM code which is interpreted by DiVM.

2.1 DIVINE Virtual Machine (DiVM)

The basic idea of DiVM is to provide the bare minimum required for efficient model checking of LLVM-based programs. To this end, it executes instructions, manages memory, implements non-deterministic choice, and (with the help of program instrumentation) keeps track of visible actions performed by the program. To the verification core, DiVM provides support for saving and loading snapshots of the program state and for generating successors of a given program state.

DiVM is not intended to execute the user's program in isolation; instead, it is expected that there will be a runtime environment on top of DiVM. This environment is expected to supply a *scheduler*, a procedure invoked by DiVM to explore the successors of a given program state. The scheduler's primary responsibility is thread management. It can, for example, implement asynchronous thread interleaving by managing multiple stacks and non-deterministically choosing which to execute. This design allows DiVM to be small, minimising the space for errors in this crucial part of the verifier. Moreover, it allows for greater flexibility, since it is usually much easier to program for DiVM then to change DiVM itself.

DiVM uses a graph to represent the memory of a program: nodes correspond to memory objects and edges to pointers between these objects. Each program state corresponds to one such graph. When exploring the state space, those graphs are stored, hashed and compared directly (i.e. they are not converted to byte vectors).

More details about DiVM, including an experimental evaluation, can be found in [5].

2.2 DIVINE Operating System (DiOS)

DiOS supplies both a scheduler, which is invoked by DiVM, and a POSIX-like environment for the libraries and the user program. To this end, DiOS exposes a syscall interface to `libc`, in a manner similar to common operating systems. Currently, DiOS supports syscalls which cover an important subset of the POSIX file system interface (provided by the integrated virtual file system). Additional syscalls make thread management and DiOS configuration possible.

2.3 State Space Reductions

To be able to verify nontrivial C or C++ programs, DIVINE 4 employs heap symmetry reduction and τ reduction [4]. The latter reduction targets parallel programs and is based on the observation that not all actions performed by a given thread are visible to other threads. These local, invisible actions can be grouped and executed atomically. In DIVINE, actions are considered visible if they access shared memory. As DiVM has no notion of threads, the *shared status* of a memory object is partially maintained by DiOS. However, DiVM transparently handles the propagation of shared status to objects reachable from other shared objects.

It is desirable that threads are only switched at well-defined points in the instruction stream: in particular, this makes counterexamples easier to process. For this reason, DIVINE instruments the program with interrupt points prior to verification. DiVM will then only invoke the scheduler at these explicit interrupt points, and only if the program executed a shared memory access since the previous interrupt.

To further reduce the size of the state space of parallel programs, DIVINE performs heap symmetry reduction. That is, heaps that differ only in concrete values of memory addresses are considered identical for the purpose of verification. On top of that, DIVINE 4 also employs static reductions which modify the LLVM IR. However, it only uses simple transformations which are safe for parallel programs and which cause minimal overhead in DiVM.

2.4 C and C++ Language Support

For practical verification of C and C++ code, it is vital that the verifier has strong support for all language features and for the standard libraries of these languages, allowing the user to verify unmodified code. DIVINE achieves this by integrating ported implementations of existing C and C++ standard libraries. Additionally, an implementation of the POSIX threading API was developed specifically for DIVINE. These libraries together provide full support for C99 and C++14 and their respective standard libraries.

As DiVM executes LLVM instructions and not C or C++ directly, the program needs to be translated to LLVM IR and linked with the aforementioned libraries. This is done by an integrated compiler, based on the clang C/C++ frontend library. How a program is processed by DIVINE is illustrated in Figure 2. The inputs to the build are a C or C++ program and, optionally, a specification of the property to be verified. The program is first compiled and linked with runtime libraries, producing an LLVM IR module. This module is then instrumented to facilitate τ reduction (see Section 2.3) and annotated with metadata required for exception support [6]. The instrumented IR is then passed to the verification algorithm, which uses DiVM to evaluate it. Finally, the verification core (if provided with sufficient resources) either finds an error and produces a counterexample, or concludes that the program is correct.

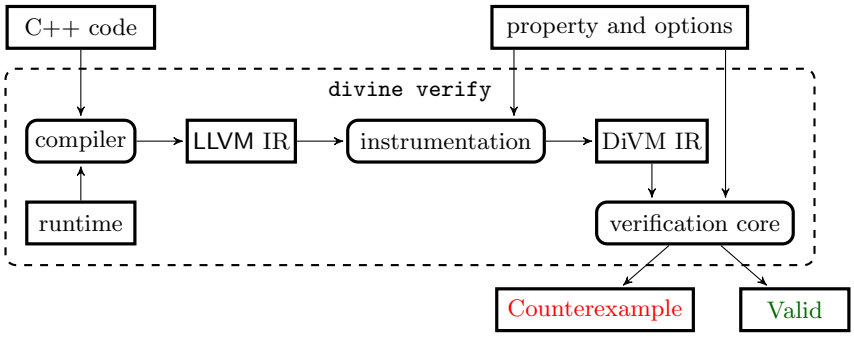

Fig. 2. Verification workflow of the `divine verify` command when it is given a C++ file as an input. Boxes with rounded corners represent stages of input processing.

2.5 Property Specification

DIVINE 4 supports a range of safety properties, namely detection of assertion violations, arithmetic errors, memory errors (e.g. access to freed or otherwise invalid memory), use of uninitialised values in branching, and `pthreads` locking errors.

The libraries shipped with DIVINE can simulate memory allocation failures and spurious wake-ups on `pthreads` conditional variables. The allocation failure simulation can be disabled on DIVINE's command line.

Monitors and Liveness More complex properties can be specified in the form of *monitors*, which are executed synchronously by DiOS every time a visible action (as determined by τ reduction) occurs. This allows such monitors to observe globally visible state changes in the program, and therefore to check global assertions or liveness properties, (using a Büchi accepting condition). Moreover, it is also possible to disallow some runs of the program, i.e. the monitor can specify that the current run of the program should be abandoned and ignored.

In order to check LTL properties in DIVINE, the LTL formula has to be translated to a Büchi automaton encoded as a monitor in C++. This translation can be done automatically by an external tool, `dipot`,[1] which internally uses SPOT [1] to process the LTL formula.

2.6 Interactive Program Simulator

Model checkers traditionally provide the user with a counterexample: a trace from the initial state to the error state. However, with real-world programs, the presentation of this trace is critical: the user needs to be able to understand the complex data structures which are part of the state, and how they evolve along the error trace. To help with this task, DIVINE now contains an interactive

[1] available from `https://github.com/xlauko/dipot`

```cpp
#include <cstdio>
#include <cassert>
void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```

Fig. 3. Example C++ code which creates an array `x` of size 4 (on the stack) and then, in function `foo`, writes into this array. `foo` does, however, attempt to write one element past the array, which would normally overwrite the next entry on the stack but not cause an immediate program failure. In DIVINE, this error is detected and reported.

simulator that can be used to perform the steps of the program which led to the error and to inspect values of program variables at any point in the execution [3].

2.7 Major Changes Compared to DIVINE 3

Compared to DIVINE 3, the new version comes with several improvements. From architectural point of view, the most important changes are the introduction of DiVM and DiOS and the graph-based representation of program memory [5]. From user perspective, the most important changes include better support for C++ and its libraries, an improved compilation process which makes it easier to compile C and C++ programs into LLVM IR, an interactive simulator of counterexamples, and support for simulation of POSIX-compatible file system operations.

3 Using DIVINE

DIVINE is freely available online[2], including source code, a user manual, and examples which demonstrate the most important features of DIVINE. In addition to the source code, it is also possible to download a pre-built binary for 64bit Linux, or a virtual machine image with DIVINE installed (available in 2 formats, OVA for VirtualBox, and VDI for QEMU and other hypervisors). If you choose to build DIVINE from source code, please refer to the user manual[3] for details.

Consider the code from Figure 3, saved in file `test.cpp`. Assuming DIVINE is installed[4] the code can be verified by simply executing `divine verify test.cpp`. DIVINE will report an invalid write right after the end of the `x` array. You can observe that the output of `printf` is present in the `error trace` part of DIVINE's output. Moreover, toward the end, the output includes stack traces of all running threads. In this case, there are two threads, the main thread of the program and a kernel thread, in which the fault handler is being executed.

[2] `https://divine.fi.muni.cz/2017/divine4/`
[3] `https://divine.fi.muni.cz/manual.html`
[4] the binary has to be in a directory which is listed in the `PATH` environment variable

If we wanted to inspect the error state in more detail, we could use DIVINE's simulator. First, we need a way to identify the error state: the counterexample contains a line which reads `choices made: 0^182`; the sequence of numbers after the colon is a sequence of non-deterministic choices made by DIVINE. We can now run `divine sim test.cpp` and execute `trace 0^182` (replacing the sequence of choices with the ones actually produced by `divine`). This makes the simulator stop after the last non-deterministic choice before the error. The actual location of the error can be inspected by executing `stepa`, which tells DIVINE to perform a single atomic step (unless an error occurs, in which case it stops as soon as the error is reported). Now the frame of the error handler can be inspected, although it is more useful to move to the frame which caused the error by executing the `up` command. At this point, local variables can be inspected by using `show`.

Please consult the user manual for more detailed information on using DIVINE. Additionally, `divine help` and the `help` command in the simulator provide short descriptions of all available commands and switches.

4 Conclusion and Future Work

In this paper, we have introduced DIVINE 4, a versatile explicit-state model checker for C and C++ programs, which can handle real-world code using an efficient LLVM interpreter which has strong support for state space reductions. The analysed programs can make use of the full C99 and C++14 standards, including the standard libraries.

In the future, we would like to take advantage of the new program representation and versatility of DiVM to extend DIVINE with support for programs with significant data non-determinism, taking advantage of abstract and/or symbolic data representation, building on ideas introduced in SymDIVINE [2].

References

1. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, volume 9938 of *LNCS*, pages 122–129. Springer, October 2016.
2. J. Mrázek, P. Bauch, H. Lauko, and J. Barnat. SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration. In *SPIN*, pages 208–213. Springer, 2016.
3. P. Ročkai and J. Barnat. A Simulator for LLVM Bitcode. *Preliminary version, ArXiv: 1704.05551*, 2017.
4. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NFM*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
5. P. Ročkai, V. Štill, I. Černá, and J. Barnat. DiVM: Model Checking with LLVM and Graph Memory. *Preliminary version, ArXiv: 1703.05341*, 2017.
6. V. Štill, P. Ročkai, and J. Barnat. Using Off-the-Shelf Exception Support Components in C++ Verification. *Preliminary version, ArXiv: 1703.02394*, 2017.