A Simulator for **LLVM** Bitcode⋆

Petr Ročkai and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai,barnat}@fi.muni.cz

Abstract. In this paper, we introduce an interactive simulator for programs in the form of **LLVM** bitcode. The main features of the simulator include precise control over thread scheduling, automatic checkpoints and reverse stepping, support for source-level information about functions and variables in C and C++ programs and structured heap visualisation. Additionally, the simulator is compatible with DiVM (DIVINE VM) hypercalls, which makes it possible to load, simulate and analyse counterexamples from an existing model checker.

1   Introduction

Verification tools are increasingly adopting **LLVM** bitcode as their input language of choice. A frequent reason for implementing **LLVM**-based model checkers (and other analysis tools) is that they can leverage existing compiler front ends, CLang in particular. This in turn makes it possible to use those model checkers on C and even C++ programs without dealing with the irregularity and complexity of these programming languages. Clearly, this tremendously improves the usefulness of any such tool, since C and C++ are widespread implementation languages, and implementation-level model checking is naturally desirable for many reasons.

An additional benefit of the standardisation around the **LLVM** IR [9] (intermediate representation) is that an ecosystem of tools is emerging, where those tools can cooperate through the common input format. Analysis and model checking tools can be used to ascertain correctness of the program with respect to a specification; however, when they find that there is a violation, printing "property violated" is rarely enough. For the result to be genuinely useful, it must somehow convey *how* the specification is violated to the user, so they can analyse the problem and fix their program. One option is to print a *counterexample trace*, which describes the violating execution of the program. In traditional model checkers, for example, it is often sufficient to provide a textual description of the entire execution, since the input model is usually small and its states and transitions can be described compactly.

More advanced tools, however, provide a *simulator*, an interactive tool for stepping through the counterexample, where the user can highlight and investigate particular sections of the counterexample in more detail, and fast-forward through other, uninteresting parts. A simulator is often also useful as an exploratory tool: the behaviour of the system can be explored by the user, manually navigating through its state space and inspecting variables along the way.

In case of C and C++ programs, it is vitally important that counterexamples can be inspected interactively, since the state of a program is a very complicated structure, often comprising hundreds of kilobytes of structured data. Moreover, violating executions can be quite long, easily hundreds or thousands of distinct states, with non-trivial relationships.

The main contribution of this paper is a reusable simulator for C and C++ code. Since it builds on the LLVM intermediate language, it can be used by multiple different tools which produce counterexamples or otherwise work with LLVM bitcode, and is easily adapted to new high-level languages with LLVM toolchains (like Objective C or Rust). To the best of our knowledge, this work is unique in the sense that no other simulator which would handle C++ programs is available, and simulators which handle C code typically miss important features. Moreover, the simulator is also reusable in the sense that, while originating from the DIVINE tool set, it can be used standalone, or possibly in combination with other analysis and verification tools.

From a more theoretical standpoint, the *debug graph* (described in Section 3.4) represents a new approach to reconciling low-level data as it exists during program runtime, with the high-level structure declared in the source code. Another new idea is to build a simulator based on compiled code (as opposed to interpreting the source code directly) and leveraging existing debugger-focused infrastructure (debug metadata in particular), making the implementation especially simple and compact.

The rest of this paper is structured as follows: in Section 2 we discuss related work and compare our approach to existing tools. Section 3 describes the LLVM bitcode as it is used by the simulator, how the simulator represents the program state and also introduces the *debug graph*. Section 4 mentions some of the more important implementation details. The focus of Section 5 is presentation of the data aspects of a program, while Section 6 concerns the program's state space. Section 7 wraps the paper up. Additional resources (mainly evaluation-related) are available online[1].

2 Related Work

It is a well-established fact that isolating some bad behaviour of a program in a test is, in itself, not sufficient to easily explain the cause of the problem [1]. The situation is similar in model checking, where a counterexample trace can often be extracted easily enough, but it may not contain sufficient detail, or conversely,

---

[1] `https://divine.fi.muni.cz/2017/sim/`

may swamp the user in large amount of irrelevant data [13]. The problem also goes beyond the software realm, as witnessed in, for instance, verification of MATLAB Simulink designs [3].

There are basically two orthogonal approaches that attempt to resolve these problems. One is to locate, or at least narrow down, the error automatically, in the hopes that from such a narrowed-down trace, the user will be able to understand the problem by inspection of the source code. In the domain of software verification, this approach is pursued by many tools: counterexamples for violation of temporal properties, generated by the software model checker SLAM [2], for instance, can be analysed and reduced to only cover a small number of source lines, in which the root cause of the error is most likely to lie [1]. An approach to succinctly describe assertion violations (violations of safety properties), based on automated dependency analysis, has also been proposed [4]. Finally, counterexamples from CBMC can be post-processed, in an approach similar to those mentioned above, with a tool called `explain` [6], in this case based on distance metrics.

Unfortunately, even if the problem area is only a few lines of source code, it can be very hard to understand the dynamic behaviour during the erroneous execution. The problem gets much worse when the program in question is parallel, because reasoning about the behaviour of such programs is much harder than it is in the sequential case.

To make understanding and fixing problems in programs (or complex systems in general) easier, many formal verification tools come equipped with a simulator. For instance the UPPAAL tool for analysis of real-time systems provides an integrated graphical simulator [5]. Another example of a formal analysis tool with a graphical simulator would be LTSA [7], based on labelled transition systems as its modelling formalism.

Like many verification tools, the `valgrind` [8] run-time program analyser is primarily non-interactive, but it provides an interface to allow interactive exploration of program state upon encountering a problem.

Our simulator is based on DiVM [10], an extension of the LLVM language that allows verification and analysis of a wider class of programs (a more detailed description of the DiVM extensions is given in Section 3.1). Since pure LLVM is retained as a subset of the DiVM language, the simulator can also transparently work with pure LLVM bitcode.

Besides its relationship to various simulators for modelling and design languages, a simulator for LLVM bitcode is, through its application to code written in standard programming languages like C, related to standard symbolic debuggers. A ubiquitous example on POSIX systems is `gdb`, the GNU debugger [11]. Unlike a simulator, which interprets the program, a debugger instead attaches to a standard process executing in its native environment.

2.1 Comparison to Symbolic Debuggers

As outlined above, simulators and debuggers substantially differ in their mode of operation and this leads to very different overall trade-offs. For example, a

simulator is much more resilient to memory corruption than a debugger, because the latter has only limited control over the process it is attached to. Both types of tools rely on understanding the execution stack of the program; however, if the program corrupts its execution stack, a debugger must rely on imprecise heuristics to detect this fact and risks providing wrong and possibly misleading information to the user. The simulator can, on the other hand, quite easily prevent such corruption from happening, since it simulates the program at instruction level, and can institute much stricter memory protections.

On the other hand, the situation is reversed when the program interacts with its surroundings through the operating system. In a debugger, such communication comes about transparently from the fact that the program is a standard process in the operating system and has all the standard facilities at its disposal. In a simulator, communication with the operating system must be specifically relayed and due to imperfections in this translation, some programs may misbehave in the simulation.

Finally, the simulator has a substantial advantage in two additional areas: first, a simulator can very precisely and quite comfortably control thread interleaving. This allows analysis of subtle timing-dependent issues in the program. Second, since a simulator has a complete representation of the program's state under its control, it can quite easily move backwards in time or compare variable values from different points in the execution history. While both scheduler locking and reversible debugging exist to a certain degree in traditional debuggers [12], those features are very hard to implement and usually quite limited in their abilities.

3 **LLVM** Bitcode

The LLVM bitcode (or intermediate representation) [9] is an assembly-like language primarily aimed at optimisation and analysis. The idea is that LLVM-based analysis and optimisation code can be shared by many different compilers: a compiler front end builds simple LLVM IR corresponding to its input and delegates all further optimisation and native code generation to a common back end. This architecture is quite common in other compilers: as an example, GCC contains a number of different front ends that share infrastructure and code generation. The major innovation of LLVM is that the language on which all the common middle and back end code operates is exposed and available to 3rd-party tools. It is also quite well documented and LLVM provides stand-alone tools to work with both bitcode and textual form of this intermediate representation.

From a language viewpoint, LLVM IR is in a partial SSA form (single static assignment) with explicit basic blocks. Each basic block is made up of instructions, the last of which is a *terminator*. The terminator instruction encodes relationships between basic blocks, which form an explicit control flow graph. An example of a terminator instruction would be a conditional or an unconditional branch or a `ret`. Such instructions either transfer control to another basic block of the same function or stop execution of the function altogether.

Besides explicit control flow, LLVM also strives to make much of the data flow explicit, taking advantage of partial SSA for this reason. It is, in general, impossible to convert entire programs to a full SSA form; however, especially within a single function, it is possible to convert a significant portion of code. The SSA-form values are called *registers* in LLVM and only a few instructions can "lift" values from memory into registers and put them back again (most importantly `load` and `store`, respectively, plus a handful of atomic memory access instructions).

From the point of view of a simulator, memory and registers are somewhat distinct entities, both of which can hold values. Memory is completely unstructured at the LLVM level, the only assumption is that it is byte-addressed (endianity of multi-byte values is configurable, but uniform). Traditional C stack is, however, not required. Instead, all "local" memory is obtained via a special instruction, `alloca`, and treated like any other memory (memory obtained by `alloca` is assumed to be freed automatically when the function that requested the memory exits, via `ret` or any other way, e.g. due to stack unwinding during an exception propagation). Therefore, a C-style stack is a legitimate way to implement `alloca`, but not the most convenient in a simulator (for more details on how memory is handled in our simulator, see Section 3.2).

3.1 Verification Extensions

Unfortunately, LLVM bitcode alone is not sufficiently expressive to describe real programs: most importantly, it is not possible to encode interaction with the operating system into LLVM instructions. When LLVM is used as an intermediate step in a compiler, the lowest level of the user side of the system call mechanism is usually provided as an external, platform-specific function with a standard C calling convention. This function is usually implemented in the platform's assembly language. The system call interface, in turn, serves as a gateway between the program and the operating system, unlocking OS-specific functionality to the program. An important point is that the gateway function itself cannot be implemented in portable LLVM. Moreover, while large portions of the kernel are often implemented in C or a similar portable language, they are also tightly coupled to the underlying hardware platform.

The language of "real" programs is, therefore, LLVM enriched with system calls, which are provided by the operating system kernel. For verification purposes, however, this language is quite unsuitable: the list of system calls is long (well over 100 functions on many systems) and exposes implementation details of the particular kernel. Moreover, re-implementing a complete operating system inside every LLVM analysis tool is wasteful. To reduce this problem, a much smaller set of requisite primitives was proposed in [10] (henceforth, we will refer to this enriched language as DiVM). Since for model checking and simulation purposes, the program needs to be isolated from the outside world, we can skip most of the complexity of an operating system kernel – communication with hardware in particular. Therefore, it is possible to implement a small, isolated operating system in the DiVM language alone. One such operating system is

DiOS – the core OS is about 1500 lines of C++, with additional 5000 lines of code providing POSIX-compatible file system and socket interfaces.

Thanks to its support for the DiVM language, our simulator can transparently load programs which are linked to DiOS and its `libc` implementation. Since a program compiled into the DiVM language is fully isolated from any environment effects, it can be simulated just like a pure LLVM program could be.

3.2   Program Memory

Internally, the simulator uses DiVM to evaluate LLVM bitcode, and therefore, how memory is represented in the simulator is directly inherited from DiVM. This means that we can take advantage of the fact that DiVM tracks each object stored in memory separately, and also keeps track of relationships (pointers) between such objects.[2] This way, the simulator precisely knows which words stored in memory are pointers and the exact bounds of each object in memory.

Moreover, DiVM can efficiently store multiple snapshots of the entire address space of the program, both in terms of space (most of the actual storage is shared between such snapshots) and time (taking a snapshot needs time roughly proportional to the total size of modified objects since the last snapshot). Once a snapshot is taken, it is preserved unmodified, regardless of the future behaviour of the program (that is, it becomes persistent).

The execution stack of a LLVM program consists of activation frames, one for each active procedure call. In DiVM, activation frames are separate memory objects. Moreover, each memory-stored local variable (i.e. those represented by `alloca` instructions) is again represented by a distinct memory object. Each frame object contains 2 pointers in its header (one points at the currently executing instruction, the other to the parent frame). Besides the header, the rest of the object is split into *slots*, where each slot corresponds to a single LLVM *register*. The correspondence between slots and LLVM registers is maintained by DiVM and is available to the simulator.

Together, those features of DiVM make it very easy to access the program state in a highly structured fashion. When compared to a traditional debugger, which must work with nearly unstructured memory space, the information our simulator can provide to the user is simultaneously easier to obtain and more detailed and reliable. Finally, since DiVM strictly enforces object boundaries, both the control stack and heap structure in our simulator are very well protected from overflows and other memory corruption bugs in the program. Therefore, the simulated program cannot accidentally destroy information which is vital for the functioning of the simulator, like all too often happens in debuggers.

3.3   Relating Bitcode to Source Code

In native code debuggers, the relationship between the binary and the original source code is often not quite obvious. For this reason, in addition to the executable binary, the compiler emits metadata which describe these relationships.

---

[2] How this is achieved is described in more detail in [10].

For instance, it attaches a source code location (filename and line number) to each machine instruction. This way, when the debugger executes an instruction, it can display the relevant piece of source code. Likewise, it can analyse the execution stack to discover how the currently executing function was called, and display a *backtrace* consisting not only of function names, but also source code lines. This is important whenever a given function contains two similar calls.

```
struct Point { float x, y; };          binary: 000000000000003f0000e040
                                        .center:
struct Circle                               type:    Point
{                                           .x:
        Point center;                           type: float
        float radius;                           value: 0
};                                          .y:
                                                type: float
Circle c = {                                    value: 0.5
  .center = { .x = 0, .y = 0.5 },     .radius:
  .radius = 7                             type:    float
};                                        value:   7
```

Fig. 1. An example C `struct` type and the corresponding representations: binary and structured (the latter is only possible with debug metadata).

The situation is analogous in LLVM-based tools. The compiler front-ends are therefore encouraged to generate *debuginfo metadata* (in a form that reflects the structure of the DWARF debug information format, which is widely used by native source-level debuggers). Besides the vitally important source code locations, the metadata describe local and global variables and their types (including user defined types, like `struct` and `union` types in C). This in turn enables the debugger to display the data in a structured way, resembling the structure which exists in the source code. For example, `struct` types in C have named fields – the debugger can use the debug metadata to discover the relationship between offsets in the binary representation of the value with the source-level field names (an example is shown in Figure 1).

3.4 Debug Graph

The memory graph maintained by DiVM is a good basis for presenting the program state to the user, but on its own is insufficient: the only type information it contains is whether a particular piece of memory holds a pointer or not. Therefore, we overlay another graph structure on top of the memory (heap) graph, with richer type information based on debuginfo metadata (more details on how this graph is computed will be presented in Section 5). The nodes in the debug graph may be further structured: they have *attributes* (atomic properties, such as an integer or floating point value), *components* and *relations*. While both components and relations are again nodes of the graph, they crucially differ in

how they relate to the underlying memory: components of a debug node occupy the same memory as their parent node; for example, a debug node which consists of a `struct` C type will contain a *component* for each field of the `struct`. In contrast, *relations* of a debug node correspond to the pointers embedded in its memory, i.e. a relation always corresponds to a pointer stored in memory (it may, however, point back at the same object it is embedded in).

Since memory objects are *persistent* in DiVM (cf. Section 3.2), so is the debug graph in our simulator. This means that objects (debug nodes) are immutable, i.e. they always come from a *snapshot* of the memory of the program. Since it would be too expensive to make a copy of the entire memory after every instruction, such snapshots are implemented via copy-on-write semantics.

4 Implementation

The ideas presented in this paper are implemented in the simulator component of DIVINE 4, which is available as `divine sim`. All relevant source code is available online[3], under a permissive open source licence. Additional details about the user interface and user interaction in particular can be found in the DIVINE 4 manual[4].

The simulator currently provides a command-driven interface, similar to, for example, `gdb`. The data structures and most of the code, however, are independent of a particular interface. The command-line parser and other interface-specific code entails approximately 800 lines of C++. The interface-agnostic core could be therefore re-used to build a graphical user interface for the simulator.

The current command interface uses *meta variables* extensively: each such meta variable holds a reference to a single debug node (cf. Section 3.4). There are two basic types of meta variables, *static* and *dynamic*.

Static variables will always point to the same debug node, even as the program executes and the content of its memory changes. Since objects in the DiVM memory are *persistent* (not mutable), this type of variable simply points to such a persistent, immutable object. Static meta variables have names starting with a `#` sign, e.g. `#start`.

Dynamic variables reflect the current state of the program at any given time. The debug nodes referenced by those variables are *refreshed* every time the program mutates its memory, so that they always point to an up-to-date copy of the persistent memory object (in other words, they always refer to the latest memory *snapshot*). Those variables are prefixed with a `$` sign, e.g. `$frame`.

4.1 High-Level Languages

Our simulator design is, to a large degree, independent of the particular high-level language in which the simulated program was developed. The structure

---

[3] `https://divine.fi.muni.cz/download.html`

[4] `https://divine.fi.muni.cz/manual.html`

of the program is described in the debug info metadata in sufficient detail to provide precise and readable information to the user. This is in contrast to tools like `gdb` and `lldb` which mostly rely on evaluating C and/or C++ statements for presenting the program data. That is, the user is allowed to type in a C or C++ expression to be evaluated and the result displayed. The major downside is that if the high-level language support is incomplete (like it is the case with C++ support in `gdb`), it becomes much harder to print certain values without resorting to very low-level means (printing bytes at particular addresses). The debug graph implemented in our simulator (see Section 3.4) is language-neutral, and hence the features derived from this graph are independent of the high-level language as well.

Of course, the amount of implementation work required to support a particular high-level language in a debugger can be prohibitive. We hypothesise that this is the primary reason why interactive simulators (and debuggers in general) are so scarce. Therefore, we consider the debug graph to be an important contribution, since it can be built from LLVM debug info in a comparatively small amount of code, but nonetheless provides a very convenient interface.

5 Working with Data

Providing facilities for inspecting data of the program is one of the main functions of an interactive debugger or a simulator. This data can be presented in different forms and from different starting points. In our simulator, heap memory is structured explicitly as a graph, and we can leverage this to greatly improve presentation of data. An example of such a graph is shown in Figure 2. Each node of the graph corresponds to a single in-memory object, which can have (and often has) additional internal structure. The internal structure reflects the C/C++ type which is deduced from the types of pointers pointing at this particular node.

5.1 Starting Points

In DiVM, there is always a single distinguished object in the heap, from which the entire heap is reachable, including the stacks of all threads and any kernel data structures. This root object is made available to the user as `$state`. In most cases, however, it is better to take a more local view: the currently executing stack frame, for instance, is available as `$frame`. Finally, there is the `$globals` meta variable that holds the debug node associated with the memory object in which all global variables of the currently executing process reside. Objects which represent stack frames and global variables consist of *slots* which in turn contain values. Values in those slots either correspond to current values of local source-level variables, or contain pointers to variables held in memory. In both cases, a *component* debug node is created, based on debug information generated by the compiler. These components then form a basis for presenting the data to the user.

5.2 Typing the Heap

In all cases, the type information available for the starting point is used to derive type information for the entire heap reachable from that starting point. For frames, we can deduce which function the frame belongs to, and obtain information about the frame layout used by that function. That is, for each LLVM register, we obtain a corresponding C type, which is usually either a primitive type or a pointer. If the type is a pointer and it is not null or otherwise invalid, there is an edge in the graph of the heap corresponding to this pointer. The object at the other end of the edge is then assigned the *base type* of the pointer, that is, type of a value obtained by dereferencing the pointer. This procedure is then repeated recursively until all objects where type information exists are assigned a type.

Of course, there is a potential for ambiguity: not all C/C++ programs are consistently typed, therefore, multiple edges pointing at a single object can each carry a different type. In this case, we assign the first type which reaches the object – for most programs, this is entirely satisfactory. It is, however, also possible to collect all types and construct a union type out of those.
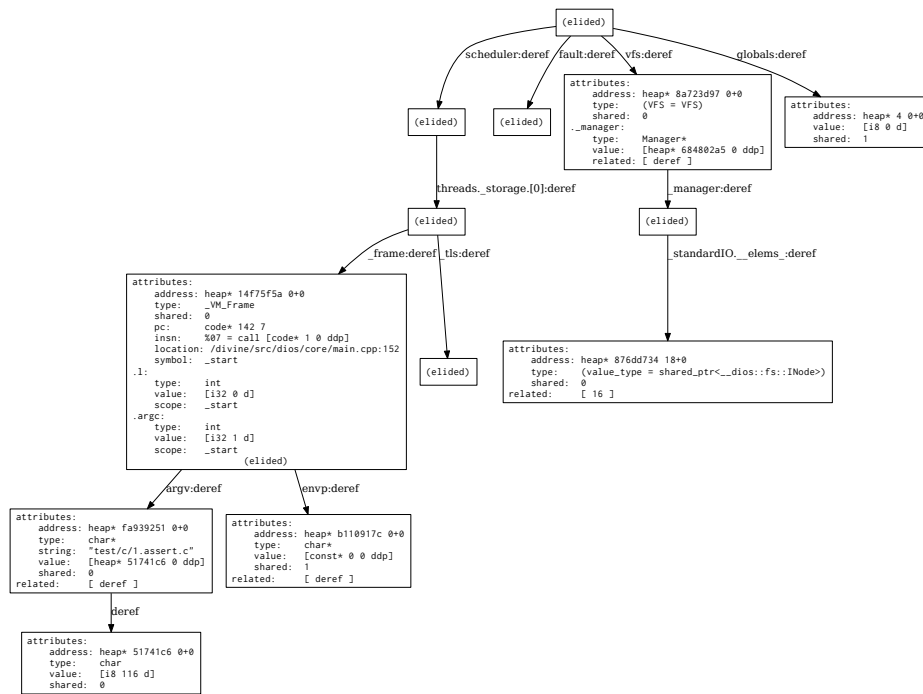

Fig. 2. An example heap structure of a simple program. The depicted graph was obtained directly from the simulator; the only change was that descriptions of some of the nodes were elided for presentation purposes. A memory object may contain multiple debug nodes (components), which are rendered textually.

5.3 Relating Data and Control

The control flow of a C program is reflected in the execution stack and is a part of the program's data. Since C and C++ are lexically scoped languages, the variables that are currently in scope depends on which function (and possibly which block in that function) is currently executing. This is realised by making local variables part of the execution stack: when a function is entered, an *activation frame* (or activation *record*) is pushed onto the execution stack. In a normal execution environment, the frame has space for CPU register spills and for local variables which have their address taken. In DiVM, there are no general-purpose registers as such; instead, LLVM registers are stored inside the frame itself. Any address-taken variables are stored as separate objects (while their address is stored in a register).

Additionally, in a typical implementation of C, the activation frame contains a *return address*, which is a pointer to the `call` instruction that caused the current function to execute. In DiVM, the frame instead contains a *program counter* (in a real CPU, the program counter, also known as instruction pointer, is held in a register). The program counter tells us which function, and which instruction within that function, is currently being executed. Through debug info (cf. Section 3.3), we can tie, to each instruction, the source code location from which it came.

As an example of how this is used in the simulator, if the user requests to list the source code of the currently executed function (using the `source` command), the simulator examines the current active activation frame (held in the `$frame` meta variable) to find the current value of the *program counter*. Then it proceeds to read the corresponding debug info to obtain the source code file name, reads the source file, finds the line corresponding to the program counter and prints the surrounding function (example output is shown in Figure 3).

```
> show $frame                          > source
attributes:                                3 int main()
   address: heap* bf24efc5 0+0             4 {
   shared:  0                      >>      5     assert( 0 );
   pc:      code* 1 0                      6     return 0;
   location: test/c/1.assert.c:5          7 }
   symbol:  main
related:     [ caller ]
```

Fig. 3. An example interaction: listing source code.

6 Navigating the State Space

If we treat the data of a program as a spatial dimension, it is natural, then, to treat the state space – the behaviour of program as it executes – as a time

dimension. Since the state space is a graph, the predecessors of a given state (the path from the initial state to the "current" state – the one that is being examined) constitute the *past* of the computation. The successors, on the other hand, correspond to possible *futures* of the computation (since the behaviour of the program is often non-deterministic; this is because it may depend on unpredictable outcomes from outside its influence, which means there is more than one possible future). In this correspondence of the state-space graph to temporal behaviour of the program, cycles in the state space clearly correspond to behaviours that go on forever.

In a standard debugger, time can only flow in one direction, and which of the potential futures is realised can be influenced, but not controlled. In a simulator, however, it is possible to both go backwards in time (rewind the program state to some past configuration) and to pick exactly which future should be explored. Likewise, it is entirely possible to go back in time and select a different future to explore. These possibilities are derived mainly from the persistent and compact memory representation (see Section 3.2).

6.1 Stepping Forward

On the other hand, the state space as explored by model checkers is often too coarse to follow the computation in detail. The states typically correspond to locations where threads interleave or where cycles can potentially form. At this level, the edges in the state space correspond, approximately, to atomic actions in the program. Even in heavily parallel programs, though, such atomic sections will span many instructions and possibly multiple source lines. A simulator which works at the state-space level (as is, for example, the case in DIVINE 3 and in verification-centric tools in general) can only present computation steps at this high level. In many cases, this is inadequate when users rely on the tool to gain precise understanding of a particular problem in the program – that is, the resolution provided by such a tool is insufficient and important details are lost. In contrast, debuggers give the user very precise control over the forward execution of the program, down to stepping one instruction at a time.

Building the simulator on top of DiVM, however, gives us execution control at the level of individual LLVM instructions, analogous to a debugger. Out of single instruction stepping, it is easy to build all the other execution control functionality common in debuggers: source-line stepping – both into and over function calls and various breakpoint types (on a source line or a on a function entry).

6.2 Going Back

Even though the simulator executes instructions individually, it also stores a state whenever a model checker would. For each state (or, more precisely, a debug node corresponding to a state), a new meta variable is created, of the form #n where n is a number. The numbers assigned to states are announced whenever a state is encountered in the simulator (see also Figure 4), so the user

can refer to them later if needed; they can also give names to states of particular interest.

Through this (partially constructed) state space, it is easy enough to implement reversible debugging. The state of DiVM can easily be restored from a memory snapshot, and this functionality is also exposed to the user (as the `rewind` command).

In general, it is impossible to execute individual instructions backwards. However, if we can go back in 'reasonable' increments, such as to individual stored states, it is easy to reach any particular earlier place in the program. Execution happens along the edges of the state-space graph, hence if we can locate the edge that we are interested in, it is then trivial to restore its origin state and single-step the program forward from that point, until the desired location is reached. This is, however, only possible because the execution in DiVM is fully deterministic and not influenced by the environment outside of the simulator. In other words, two ingredients are crucial to obtain simple, reversible debugging: persistence of snapshots and fully deterministic execution.

```
> start                                      > backtrace
# a new program state was stored as #1        address: heap* fa4b97e2 0+0
# active threads: [0:1]                        pc: code* c49 0
# a new program state was stored as #2         location: _PDCLIB/assert.c:21
# active threads: [0:1]                        symbol: _PDCLIB_assert_dios
# executing main at test/c/1.assert.c:5
> stepi                                        address: heap* 96c75834 0+0
  call @_PDCLIB_assert_dios                    pc: code* 1 1
# executing _PDCLIB_assert_dios                location: test/c/1.assert.c:5
#        at _PDCLIB/assert.c:21                symbol: main

                                               address: heap* 797b4e39 0+0
                                               pc: code* 1f4 7
                                               location: dios/core/main.cpp:173
                                               symbol: _start
```

Fig. 4. Left: new states are discovered during execution of a program. Right: displaying a backtrace.

6.3   Inspecting the Stack

As explained in Section 5.3, the control flow of a C program (or, more generally, any LLVM program) is realised as a simple data structure stored in memory along with other data. This data structure often represents the best means for a user to locate themselves within the execution of a program. A so-called *backtrace* (or *stack trace*) is a fundamental program analysis tool. A backtrace lists each activation record in the (reverse) order of activation, and constitutes a description of a location in the computation of the program (an example is shown in Figure 4). Of course, such a description is necessarily incomplete, being much more

concise than the real representation of the program's state. Even some control flow information, such as which iteration of which loop is currently executing, is elided.

6.4   Thread Interleaving

As mentioned in Section 2.1, a simulator can precisely control thread interleaving. This follows from the fact it is built on the same foundation as a model checker – since a model checker typically explores all possible interleavings, the underlying virtual machine must provide means to switch threads at relevant points. However, many instruction interleavings have equivalent effects, and for this reason, allowing threads to be switched at arbitrary points is wasteful, both in model checking and in simulation. DiVM explicitly marks points in the instruction stream where threads may be switched, and this restriction is carried over to the simulator. These *interrupt points* are inserted in such a manner that all possible behaviours of the program are retained in the state space. From a simulation point of view, the downside is that the interleaving may not be the most intuitive, but the reduction in the number of possible states generally outweighs this, since the user needs to consider fewer runs.

6.5   Simulating Counterexamples

There are two major tasks for the simulator in the context of program analysis and verification. The first is to allow the user to explore program behaviour and read off details about its executions. The other is to support verification tools which provide counterexamples to the user. As detailed in Section 2, it is a difficult task to analyse problem reports from automated analysis and verification tools, and a simulator can be very helpful in this regard. In case of model checkers, the problem report contains an execution *trace*: a step-by-step description of the problematic behaviour. For tools based on DiVM, this trace is simply a list of non-deterministic choices made during the execution of the program (internally, there is only one non-deterministic choice operator and all state-space branching is caused by this operator, including thread interleaving). Since the program is isolated from the environment, this list completely and unambiguously describes its entire execution history. When the DIVINE 4 model checker discovers a problem in the program, it prints such a list of choices, which can then be loaded into the simulator.

   When the simulator loads a trace, it pre-populates the #n-type variables (see Section 6.2) with states along the problematic execution, and "locks" the non-deterministic choices to follow the trace. In this mode, stepping through the program (backwards or forwards) will simply follow the counterexample, unless a particular choice is overridden by the user. In effect, the user will be guided through the faulty behaviour of the program, and can easily move back and forth to locate the cause of the problem (as opposed to the symptom, which is what the model checker reports and may be distinct from the original cause).

7 Conclusion

We have described a novel approach to interactive analysis of real, multi-threaded C and C++ programs. The approach plays an important support role in the wider context of automated verification and, in particular, model checking of software. The simulator naturally supports the compact and universal counterexample format used in DiVM. Compared to earlier tools, DIVINE 4 is substantially more useful in practice, also thanks to the new interactive simulator.[5]

Bibliography

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003. URL `http://doi.acm.org/10.1145/640128.604140`.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, LNCS. Springer, 2004. URL `http://dx.doi.org/10.1007/978-3-540-24756-2_1`.

[3] Jiri Barnat, Jan Beran, Lubos Brim, Tomas Kratochvíla, and Petr Ročkai. Tool chain to support automated formal verification of avionics simulink designs. In *FMICS*, number 7437 in LNCS, pages 78–92. Springer, 2012. URL `http://dx.doi.org/10.1007/978-3-642-32469-7_6`.

[4] Samik Basu, Diptikalyan Saha, and Scott A. Smolka. Getting to the root of the problem: Focus statements for the analysis of counter-examples. 2012.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *SFM*, 2004.

[6] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *CAV*, LNCS, pages 453–456. Springer, 2004.

[7] Jeff Magee. Behavioral analysis of software architectures using LTSA. In *ICSE*, 1999.

[8] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[9] The LLVM Project. LLVM language reference manual, 2016. URL `http://llvm.org/docs/LangRef.html`.

[10] Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with LLVM and graph memory. 2017. URL `https://arxiv.org/abs/1703.05341`. Preliminary version.

[11] Richard Stallman, Roland Pesch, and Stan Shebs. Debugging with gdb. 2010.

[12] Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. Urdb: a universal reversible debugger based on decomposing debugging histories. In *PLOS '11*, 2011.

[13] Willem Visser and Alex Groce. What went wrong: Explaining counterexamples. In *SPIN*, LNCS, pages 121–135. Springer, 2002.

---

[5] Supported by anecdotal evidence from working with students, both individually and in a validation & verification course.