Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

DivSIM, an Interactive Simulator for LLVM Bitcode

Petr Ročkai · Jiří Barnat

Received: date / Revised version: date

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, DivSIM is compatible with DiVM (DIVINE VM) hypercalls, which makes it possible to load, simulate and analyse counterexamples from an existing model checker, and with abstract bitcode generated by LART (LLVM Abstraction and Refinement Tool), making it suitable for direct analysis of abstract and/or symbolic programs and counterexamples.

Keywords simulation, model checking, counterexample, parallelism, abstract interpretation, symbolic execution, LLVM

Declarations

Funding Not applicable.
Conflict of interest Not applicable.
Availability of data and material Not applicable.
Code availability Yes, under an open-source licence.
The present paper is an extended version of [19].

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 enables those model checkers to work with C and even C++ programs without dealing with their irregularity and complexity. 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 [23] (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.

Faculty of Informatics, Masaryk University
Brno, Czech Republic

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.

1.1 Concurrency and Threads

It is often the case that modern C and C++ programs make use of concurrency and parallel execution. As is widely appreciated, these bring unique challenges not only to development and programming itself, but also to subsequent validation and verification efforts. One of the important tools in our arsenal for dealing with these challenges are rigorous, automated methods for analysis of programs; very often either model checking, or methods based on it.

In fact, what model checkers bring to the table in this regard, when compared to traditional testing, is *universal quantification* over possible executions of the concurrent program. That is, given a model checker, we can ask questions like 'is there an execution of this program where a null pointer is dereferenced?' instead of the more straightforward 'is a null pointer dereferenced in this particular execution?' Moreover, this can be done using a fraction of the resources that would be required to tackle the executions one by one.

However, concurrency brings a whole lot of interaction-at-a-distance effects to software. While powerful automated tools can often uncover unsafe or otherwise unexpected behaviour in concurrent programs, these effects bring the need for detailed yet tractable counterexamples into sharp focus. This is because the behaviours of a concurrent program are even harder to analyze than those for sequential programs, and this carries over to the counterexamples.

This increase in complexity is also closely related to the much harder question that we have asked: if we have a particular (sequential) execution with a null dereference in it, we want to further know where in the program it happened, and perhaps how the program got into that state. We may want to re-trace the decisions the program has made, or inspect the call stack at the point of the crash. But the decisions are explicit and well-understood: which `if` branch was taken? Was the loop executed again, or was it terminated? Each decision corresponds to a syntactic element of the source code.

With concurrency, there is a new type of decision, disconnected from the syntactic structure of the program: as execution flits back and forth between different threads and virtual 'decisions' appear at points in the program that would be otherwise linear. Those are of the form 'do I continue to follow this thread of execution, or do I follow another for a while?'. Of course, this information is present in the model checker output, in the counterexample trace – it tells the user exactly when the focus needs to shift to another thread. However, this also makes following the trace significantly more difficult, and makes interactive counterexample analysis even more important.

1.2 Symbolic and Abstraction-Based Analysis

An important technique for analysing programs is the use of symbolic representation of values: instead of a *concrete* value, such as 7, variables are represented using unevaluated *terms* (expressions), such as $3a + 1$. The free variables (in this case a) in those terms represent inputs to the program (more details relevant for the present paper are given in Section 3.3). This way, many more behaviours can be captured in a single run of a symbolic executor or a symbolic model checker than is possible with explicit methods. In fact, this is exactly analogous to the way concurrency is tackled by automated tools: however, in this case, the universal quantification is for *program input* instead of linearizations of concurrent executions.

Of course, the price is increased computational complexity of the task, and increased conceptual and implementation complexity of the tool. Fortunately, from the point of view of the user, the situation is very different from concurrency – symbolic execution does not, in principle, add complexity to counterexamples. In theory, it is entirely possible to present a counterexample that is indistinguishable from one obtained with explicit methods. In practice, there are technical obstacles that need to be addressed to enable this kind of user experience. We will discuss those issues and how to solve them in Section 4.5.

Another powerful verification technique in widespread use is *abstraction*, often coupled with automated *abstraction refinement* (for instance through the well-known CEGAR process). Again, in theory, the situation with counterexamples and their analysis is unchanged – a counterexample can be presented in essentially the same way like with explicit methods. The technical challenges are somewhat more pronounced though. These will be discussed in more detail in Section 4.4.

In addition to analysing counterexamples, it is sometimes useful to explore symbolic or abstract state spaces directly. This comes with additional challenges

and limitations, though arguably the benefits outweigh the drawbacks. We briefly discuss this mode of operation in Section 3.3.

1.3 Contribution

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-based 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 often miss important features.

From a more theoretical standpoint, the *debug graph* (described in Section 3.5) 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. This, in turn, enables a new architecture based on compiled code, and allows DivSIM to leverage existing debugger-focused infrastructure (debug metadata in particular). As a result, the implementation is especially simple and compact.

Finally, in addition to a directly usable tool, we provide an API (application programming interface, based mainly on the debug graph) which can be used for automation, for building extensions and even new tools that make use of the simulator internally. This is discussed in more detail in Section 7.3.

1.4 Paper Structure

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 DivSIM, how the simulator represents the program state and also introduces the *debug graph*. The focus of Section 4 is presentation of the data aspects of a program, while Section 5 is concerned with the program's state space. A discussion of soundness and assumptions made about the simulated system are discussed in Section **??** while Section 7 mentions some of the more important implementation details. Finally, Section **??** gives a summary of an empirical evaluation and Section 9 wraps the paper up. Additional resources are available online.[1]

[1] `https://divine.fi.muni.cz/2021/sim/`

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 [2]. The situation is similar in (linear-time) model checking, where a counterexample trace can often be extracted easily enough, but it may not contain sufficient detail, or conversely, may swamp the user in large amount of irrelevant data [25]. The problem also goes beyond the software realm, as witnessed in, for instance, verification of MATLAB Simulink designs [4].

2.1 Counterexamples and Defect Analysis

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 [3], 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 [2]. An approach to succinctly describe assertion violations (violations of safety properties), based on automated dependency analysis, has also been proposed [5]. Finally, counterexamples from CBMC can be post-processed, in an approach similar to those mentioned above, with a tool called `explain` [9], 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 [6]. Another example of a formal analysis tool with a graphical simulator would be LTSA [17], based on labelled transition systems as its modelling formalism.

Like many verification tools, the `valgrind` [18] runtime program analyser is primarily non-interactive, but it provides an interface to allow interactive exploration of program state upon encountering a problem, based on `gdb` [22].

2.2 LLVM-Based Tools

Our simulator is based on DiVM [20], 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, DivSIM can also transparently work with pure LLVM bitcode.

An important part of the DIVINE [14] ecosystem is the tool LART [13], which implements abstraction as a transformation of LLVM bitcode. It provides multiple abstract domains, though one of them stands out – the domain of free terms. Coupled with an SMT solver, this is how DIVINE implements symbolic model checking (more details are provided in Section 3.3). The transformed bitcode can be directly processed by the simulator without special support, though a few extensions (described in Section 4.5), make its use considerably more comfortable.

Many other automated analysis tools fit into the wider LLVM ecosystem. One of the most prominent examples is KLEE [7], a symbolic executor for LLVM bitcode. Of course, the bitcode which is accepted by KLEE can be also loaded into DivSIM, though in itself, this is only of limited use.

Fortunately, we can do better, with the help of DiOS [21], a small operating system which compiles to the DiVM language. Recently, DiOS has been ported to run on KLEE, using a thin compatibility layer that emulates DiVM hypercalls using a mix of C stubs and built-in functions provided by KLEE. Besides the obvious effect – making it possible to link programs to DiOS and analyse the result with KLEE – this also allows us to generate DivSIM-compatible counterexamples from KLEE (more details are given in Section 5.5).

Symbiotic [8] extends KLEE with program slicing and static analysis. Since it uses KLEE in the backend, the (post-slicing) bitcode can be likewise loaded into our simulator, though we are not aware of anyone attempting to use DiOS with Symbiotic, hence the path to compatibility of counterexamples might be slightly more involved.

While we are not aware of any other tools which would be compatible with DiVM/DivSIM-style counterexamples out of the box, it should be easy, in principle, to adapt most of them, either directly, or by porting DiOS to run on them (which provides additional benefits). Some of the possible candidates are the Vienna Verification Toolkit VVT [10], the statistical model checker Lodin [16], or the stateless model checkers Nidhugg [1] and GenMC [12].

2.3 Comparison to Symbolic Debuggers

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 [22]. Unlike a simulator, which interprets the program, a debugger instead attaches to a standard process executing in its native environment. A more recent example would be `lldb` [15], which works in essentially the same way, but builds on LLVM components.

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 enforce 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, a simulator has a substantial advantage in two additional areas: first, a simulator can very precisely and 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 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 [24], those features are very hard to implement and usually quite limited.

3 LLVM Bitcode

The LLVM bitcode (or intermediate representation) [23] 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 the 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 [20] (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 [21] – the core OS is about 2300 lines of C++, with another 2300 lines of code providing POSIX-compatible file system and socket interfaces. The user-space components (`libc`, `libm`, C++ standard libraries and so on) are largely taken from existing 3rd-party implementations, augmented with fewer than 4000 lines of custom code.

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.

Finally, while the simulator uses DiVM to evaluate program instructions and hence relies on correctness of the implementation, errors in DiOS have a smaller impact. The DiOS code is executed in the virtual machine, and is subject to its error checking: therefore, in this case, the most likely outcome is by far a spurious error which can be analysed using the simulator itself.

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).

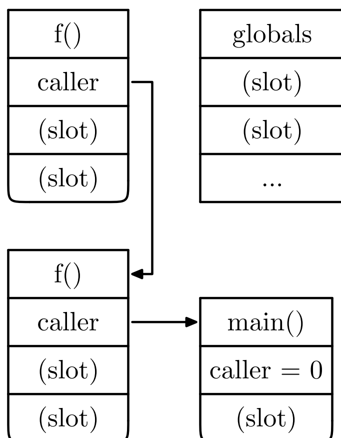

Fig. 1: Execution stack and global variables.

The execution stack of an 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*. An example stack structure is shown in Figure 1. 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 Abstraction and Symbolic Values

As we have briefly mentioned in Section 2.2, it is possible to perform symbolic and abstract model checking with DiVM by using a separate tool called LART [13] (LLVM Abstraction and Refinement Tool) to transform the input bitcode. LART itself performs abstraction in the following way:

1. some of the values in the input program are marked as *inputs*, either manually (by calling a special-purpose C function to obtain the value), indirectly (via DiOS system calls) or automatically (by an under-approximation refinement tool),
2. using a lightweight dataflow analysis, LART then computes which instructions in the program may come into contact with abstract values and instruments them to perform abstract operations, if needed,
3. DiOS provides two libraries, LAVA (Library of Abstract Values) and LAMP (LART Metadomain Package), which together supply the definitions (implementation) of the abstract operations inserted in the previous step.

For traditional abstract domains (interval arithmetic, congruence classes and so on), this is all that needs to be done – the transformed bitcode can be directly processed by all DiVM-based tools (including DivSIM) without any special support for abstraction.

[2] How this is achieved is described in more detail in [20].

Of course, if presented with an abstract version of a program, a model checker may find a counterexample which then turns out to be infeasible: for this reason, it is desirable to process such counterexamples to either validate them, or if they are indeed infeasible, to refine the abstraction as needed. Likewise, by interactively exploring the abstract state space, it may be possible to reach abstract states which do not appear in any concrete executions. In this case, it is up to the user to be careful. However, they need to be given sufficient information about the abstract values which appear in the program – how DivSIM does this is further discussed in Section 4.4.

As we have hinted at in Section 2.2, symbolic model checking in DIVINE is implemented on top of this abstraction mechanism. This is achieved by using a specific abstract domain, the *term domain*, in which values are unevaluated terms with free variables. However, the abstract domain itself does not constrain the execution of the program at all – all behaviours involving such abstract values become possible. To restrict the program to behaviours which actually exist in the concrete program, the term domain maintains a set of *constraints*, updated every time a conditional branch which depends on a symbolic value is executed (these constraints are also known as a *path condition*).

Since operations on terms are implemented in the program itself (by the LAVA library, which is simply linked into the program under test with the rest of DiOS), these constraints are simply an object in the memory of the program. When executed in symbolic mode, the model checker will extract these constraints from program memory, convert them into an SMT query and present it to a suitable solver. If the constraints in a particular program state turn out to be unsatisfiable, the execution can be abandoned. Therefore, infeasible counterexamples are ruled out. The consequences for DivSIM are further discussed in Section 4.5.

3.4 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. 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.

The situation is analogous in LLVM-based tools. 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 2).

3.5 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 4). The nodes in the debug graph may be further structured: they have *attributes* (atomic properties, such as an integer or a 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 represent 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 the memory it represents (it may, however, point back at the same object it is embedded in). An example debug graph is shown in Figure 3.

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.

```c
struct Point { float x, y; };

struct Circle
{
        Point center;
        float radius;
};

Circle c = {
  .center = { .x = 0, .y = 0.5 },
  .radius = 7
};
```

```
binary: 000000000000003f0000e040
.center:
    type:    Point
    .x:
        type: float
        value: 0
    .y:
        type: float
        value: 0.5
.radius:
    type:    float
    value:   7
```

Fig. 2: An example C `struct` type and the corresponding representations: binary and structured (the latter is only possible with debug metadata).

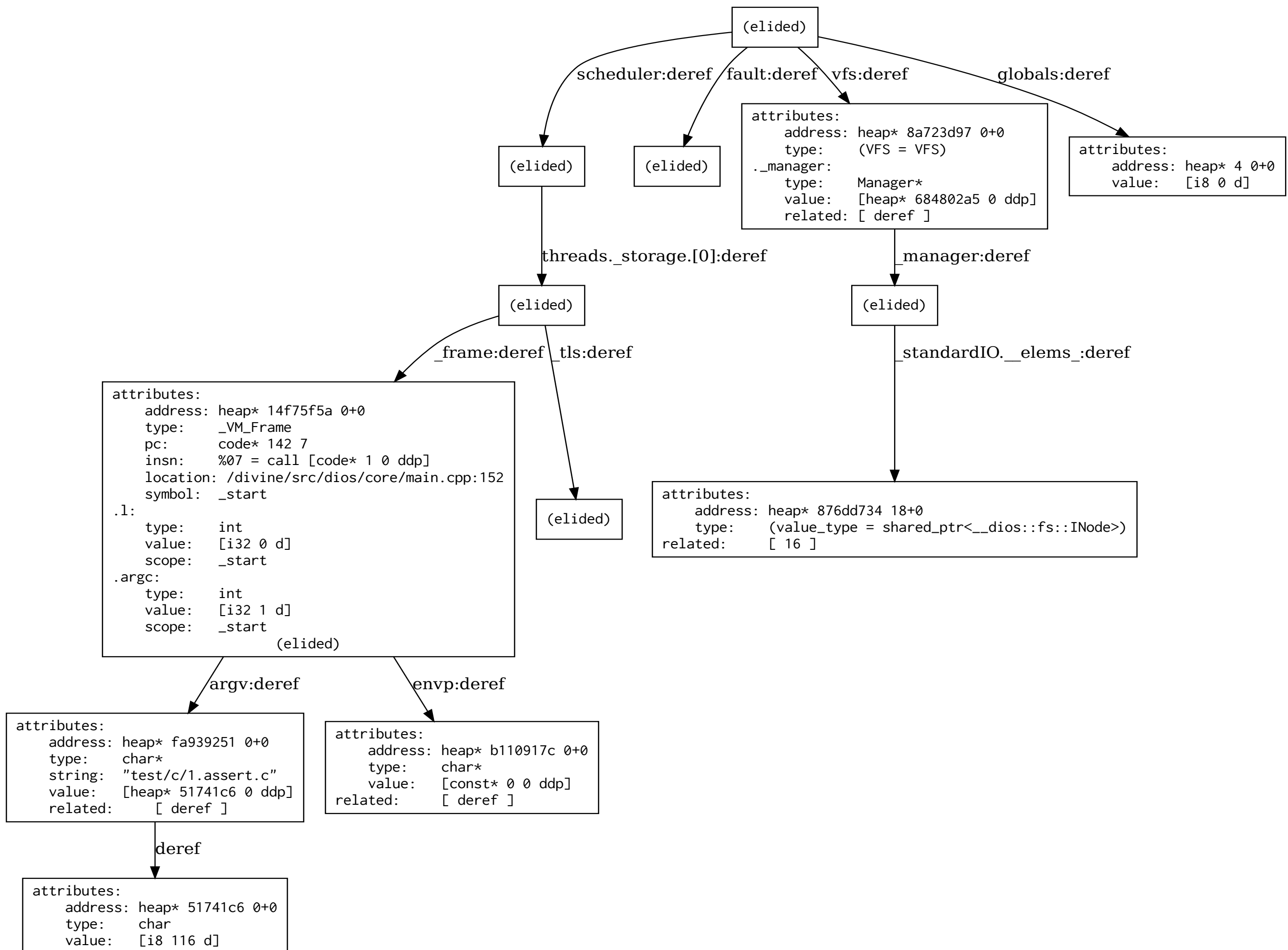

Fig. 3: A debug graph of a simple program. A single memory object may contain multiple *component* debug nodes which are rendered textually. The arrows correspond to *relations*. The depicted graph was obtained directly from the simulator; the only change was that descriptions of some of the nodes were elided for presentation purposes.

4 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 3. 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.

4.1 Starting Points

For certain memory objects, the type information is directly encoded in the metadata generated by the compiler and does not need to be inferred via pointers. Such objects are the starting point of the type assignment process by which the *debug graph* is obtained.

In principle, there are 2 types of such objects: *activation frames* and *globals*. Both consist of *slots* which in turn contain values. Values in those slots either correspond to values of (local or global) source-level variables, or contain pointers to variables held elsewhere in memory. In both cases, a *component* debug node is created for each slot, based on the debug information generated by the compiler. These components then form a basis for presenting the data to the user.

Additionally, in DiVM, there is always a single distinguished *root object* in the heap, from which the entire heap is reachable, including the stacks of all threads and any kernel data structures. The address and the C type of this *root object* is also available to the simulator, and is mainly used to discover all the nodes of the 2 above-mentioned types.

4.2 Typing the Heap

In all cases, the type information available for the starting points is used to derive type information for the portion of the 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, the type of the 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 collect all the applicable types and construct a synthetic *union type*, which is assigned to any such ambiguous debug node. This ambiguity might propagate downstream from an affected node, but for most programs, this does not appear to pose a significant problem.

4.3 Relating Data and Control

The control flow of a C program is reflected in the execution stack which is part of the program's data. C and C++ are lexically scoped languages: which variables are currently in scope depends on which function (and possibly which block in that function) is currently executing. This is achieved 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 native 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 (and 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. Each instruction can in turn be tied, via debug metadata (cf. Section 3.4), to a particular source code location (a source file and a line number).

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, the simulator examines the current active activation frame to find the current value of the *program counter*. Then it proceeds to read the corresponding debug metadata 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 4).

```
> show $frame
attributes:
    address: heap* bf24efc5 0+0
    shared:  0
    pc:      code* 1 0
    location: test/c/1.assert.c:5
    symbol:  main
related:     [ caller ]

> source
      3 int main()
      4 {
>>    5     assert( 0 );
      6     return 0;
      7 }
```

Fig. 4: An example interaction: listing source code.

4.4 Abstract Values

Debug nodes which correspond to scalars (simple atomic values) have a `value` attribute, which displays either the integer or floating-point value (for base types) or the numeric value of a pointer, along with any tags it might carry. This information is of course very valuable when inspecting the behaviour of a program.

However, as discussed in Section 3.3, DIVINE uses bitcode transformations to implement abstraction, and abstract values are constructed and manipulated by the simulated bitcode like any other value. While this greatly simplifies the implementation of both the model checker and the simulator, it has one very significant downside: without additional support, the debug nodes which correspond to abstract values display, as their `value`, the underlying representation, typically a pointer to some internal data structure. From a user point of view, this is not very useful, and it makes following the computation much harder.

On the other hand, it is undesirable to extend DivSIM or the underlying libraries with the knowledge about every abstract domain that could possibly be used in a program: after all, the user could easily provide their own, without modifying DiVM, DivSIM or LART, simply by implementing the domain operations in C or C++ and linking the definitions to their program. It is therefore important that the code for formatting a user-friendly description of a given abstract value is part of the abstract domain itself.

Fortunately, DiVM provides a mechanism called *debug calls*, which allows the program to execute a function and immediately roll back any changes in the program state. Such debug calls have no observable effect on the later behaviour of the program, and hence do not disturb the simulation. Additionally, the function so invoked has unrestricted access to the program, it may refer to any of its variables (memory) and call any of its functions. In order to be useful, the effect of such a function call must be observable 'on the outside' – this is achieved through the `trace` DiVM hypercall, which provides a one-way channel for the program under test to send data to the VM.

Together, these two mechanisms make it possible to simply add a special abstract operation which produces the human-readable description of the abstract value and uses the `trace` hypercall to send it to the VM. This operation is then invoked as a debug call, which means that abstract values can be formatted whenever DivSIM needs to construct an 'abstract' debug node, without disturbing the simulated program.[3]

The second challenge associated with abstract values is that some conditional branches can, under abstract interpretation, go both ways. Internally, this is realized using the `choose` hypercall, and hence does not need special support in DivSIM – existing mechanisms can be used to instruct the simulator to choose one or the other branch going forward. Of course, if the branch turns out to be uninteresting (or even infeasible) the choice can be easily changed by rewinding the execution and following the other branch. When following a counterexample, the choices which correspond to this counterexample are taken automatically (unless overridden by the user). This is again a consequence of a general mechanism, further discussed in Section 5.5.

4.5 Symbolic Values

As outlined in Section 1.2, it is increasingly common that model checkers use symbolic representation for some (or all) values stored and manipulated by the program under analysis. This allows many behaviours to be examined in a single model checker run, though it does make the analysis more expensive. The consequences for counterexample analysis are comparatively mild, though in general technically challenging.

Section 3.3 then went on to explain that in DIVINE, symbolic model checking is implemented on top of abstraction. Combined with the previous section, this could give an impression that nothing more needs to be done. However, the result would be, again, unsatisfactory: the human-readable description provided by the term domain can be, at best, the term itself, with embedded free variables. This alone is not especially infor-

[3] A minor technical complication arises from the fact that the formatting function needs to be invoked in the context in which the abstract value exists. However, since debug nodes already carry a reference to the snapshot (program state) which they describe and DiVM can cheaply continue execution from an arbitrary snapshot, this is not a serious problem.

mative, though it can be a valuable piece of information in a wider context.

Recall that besides the values of the terms themselves, the term domain maintains a *path condition*, a set of logical constraints on the free variables that appear in the terms. The model checker then uses an SMT solver to decide whether those constraints can be all simultaneously satisfied. However, the solver can perform an additional service: it can compute a *model* of the constraint system: an assignment of concrete values to free variables.

Equipped with this model, it becomes possible to evaluate the terms,[4] by substituting the assigned values for the free variables. Since symbolic values now evaluate to concrete values, these concrete values can be shown in their respective debug nodes. However, this does not solve all the problems. The last remaining issue is that of *model continuity*: if a model is obtained in a particular program state, the same model can be unusable in the next, because additional constraints might have appeared.

In this respect, the situation when following a counterexample is again simpler than in the more general ‘free exploration’ mode. For any particular execution, it is always possible to find a single model that works on each of the states along that execution, assuming that we know the entire execution beforehand. But this is exactly the case when analyzing a counterexample: hence, it is possible to compute a single model and use it for all the states along the counterexample, guaranteeing continuity. During free exploration, the user needs to look out for changes in the model along the execution (discontinuities), just as they have to watch out for infeasible paths in an abstract state space.

5 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 the program as it executes – as a time dimension. Since the state space is a graph (cf. Figure 5), 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,[5] 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.

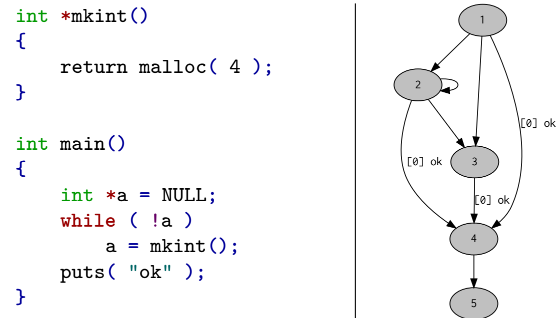

Fig. 5: An example C program and its state space.

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. It is also entirely possible to go back in time and then select a different future to explore. These capabilities are derived mainly from the persistent and compact memory representation (see Section 3.2).

5.1 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 actions will span many instructions and possibly multiple source lines. A simulator which works at this level[6] can only present very coarse computation steps to the user and not seeing the intermediate state of the program can prevent users from relating effects to

[4] There are actually two implementation choices. The evaluation could be handed off to the term domain itself, the same as for formatting abstract values, though in this case it is more complicated, since the model needs to be passed to the evaluation routine. However, since the model checker needs to be able to evaluate the terms anyway to build the SMT query, it seems reasonable to do the evaluation on the DiVM/DivSIM side.

[5] The behaviour of the program may depend on external factors, such as scheduling choices, user inputs, asynchronous events and so on. In DiVM, these all map to the `choose` hypercall.

[6] This is often the case in verification-centric tools, partly because it is a simple implementation strategy that builds on the same primitives as the verification tool itself.

their causes. If the simulator operates with fixed computation steps, the opposite problem can also happen: the user must step through a large number of irrelevant program configurations [11], again frustrating the debugging effort.

In contrast, debuggers give the user very precise control over the forward execution of the program, down to stepping one machine instruction at a time. However, they also make it very easy to fast forward through thousands of lines of code, stopping when a predetermined condition is met, most often a particular source code line is executed (this feature is known as a *breakpoint*).

Building the simulator on top of DiVM, however, gives us execution control at the level of individual LLVM instructions, analogous to a debugger. Building on the instruction stepping mechanism, the simulator also provides all the 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).

5.2 Going Back

In general, it is impossible to execute individual instructions backwards. However, if execution is perfectly repeatable (as it is in a simulator), we can reach any earlier configuration of the program by replaying the current execution from the start and stopping right before the instruction of interest executes.

Additionally, the simulator stores intermediate states (automatically at convenient locations, or at a user request). It is then possible to go back to any such stored state and continue execution from that point. This can make the above-mentioned process considerably more efficient: it is enough to replay execution from the most recent stored state that lies on the current execution path. See Figure 6.

```
> start
 state    #1 [new] -- active threads: [0:0] --
 state    #2 [new] -- active threads: [0:0] --
# executing main at test/c/1.assert.c:5
> stepi
  call @_PDCLIB_assert_dios
# executing _PDCLIB_assert_dios
#         at _PDCLIB/assert.c:21
```

Fig. 6: Discovery of new states during execution.

```
> backtrace
 address: heap* fa4b97e2 0+0
 pc: code* c49 0
 location: _PDCLIB/assert.c:21
 symbol: _PDCLIB_assert_dios

 address: heap* 96c75834 0+0
 pc: code* 1 1
 location: test/c/1.assert.c:5
 symbol: main

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

Fig. 7: Displaying a backtrace.

5.3 Inspecting the Stack

As explained in Section 4.3, the control flow of a C program (or, more generally, any LLVM program) is tracked by 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[7] (an example is shown in Figure 7).

5.4 Thread Interleaving

As mentioned in Section 2.3, a simulator can precisely control thread interleaving: the underlying virtual machine provides means to switch threads at all relevant points. However, many instruction interleavings have equivalent effects, and for this reason, allowing threads to be switched at arbitrary points would be wasteful. For this reason, DiVM explicitly marks points in the instruction stream where threads may be switched, and this behaviour 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. To further reduce the number of context switches, a model checker may use some form of

[7] This description is necessarily incomplete, being much more concise than the real representation of the program's state. Including additional information improves completeness, but compromises brevity, which is an important strength of this presentation format.

partial order reduction, but this is not necessary in a simulator, since it doesn't need to explore or store the entire state space.

5.5 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 2 types of information:

1. The 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).
2. Which of the *interrupt points* were used in the execution: the model checker may be able to prove that a particular interrupt point is not required, and the simulator needs this information to correctly reproduce the counterexample.

Since the program is isolated from the environment, this list completely and unambiguously describes its entire execution history. When the model checker discovers a problem in the program, it writes this list into a text file, which the simulator can then load along with the program.

When the simulator loads a trace, it locks the outcomes of all 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).

In an ideal world, the model checker or other analysis tool will be directly able to generate a counterexample in the required format. Unfortunately, this is not always the case in practice, and it is not always desirable to extend the analysis tool. An alternative route is then via DiOS [21], a small portable operating system which targets the DiVM language. In this case, the portability refers to the ability of DiOS to run on different tools, including LLVM-based tools which do not directly support the DiVM extensions.

As a proof of concept, DiOS has been ported to the symbolic executor KLEE. This port includes a layer which emulates DiVM hypercalls on KLEE, including `choose`, the non-deterministic choice operator. When KLEE discovers a counterexample, it will, like most tools, print a trace that describes how the execution progressed. The program under test can augment this trace by calling a special built-in function; the strings passed to that function will then appear in the counterexample, in execution order. The KLEE port of DiOS hooks into this mechanism to report the result of every non-deterministic choice, which in turn makes it possible to construct a DivSIM-compatible counterexample, as described above.

Since there are no interrupt points under KLEE (only sequential programs are supported and loops are unfolded), this is all the information that is needed. However, since interrupt points are normally serviced by a DiOS routine, the same mechanism can be used to collect interrupt information (on a system where interrupts are relevant).

6 Correctness

Of course, it is desirable for the simulated execution to correspond to actual execution of the program in its native environment as closely as possible. In the context of verification, there are two classical criteria to consider:

1. Any violation of the specification (i.e. any bug that we care about) that can happen in real execution will be found and reported by the verification tool. This is commonly known as *soundness*.
2. Any violation reported by the tool can be reproduced in real execution (at least in principle: extremely unlikely outcomes are still considered to be a real concern). This is often described as *exactness*. In 'bug-hunting' (falsification) tools, reports of problems that can be never triggered in real executions are known as *false positives*.

For a simulator, the two criteria are important, but not quite satisfactory in themselves: we are interested in execution history, as much as the outcome. This history includes all user-observable properties of the program, like organisation of memory, values of variables at any given point in execution, etc. Unfortunately, there is no obvious, accessible criterion which would quantify how similar the real and the simulated execution traces

are. Clearly, there are many programs that are indistinguishable from each other based solely on the above 2 criteria but still produce very different computation histories. We will discuss this problem in more detail in Section 6.2.

6.1 Soundness and Exactness

Except for possible implementation bugs, the simulator itself does not introduce any new sources of unsoundness (or inexactness). Instead, it inherits the properties of DiVM (and to a lesser extent, DiOS).

Regarding the soundness and exactness of DiVM, we make the following observations:

1. The most obvious source of possible problems is that DiVM does not interpret the machine code that will be executed natively: there is an additional translation step from the LLVM IR to native code. Any difference in behaviour of the program introduced at this stage is out of reach of DiVM, and hence also of DivSIM.
   Perhaps less obviously, this also covers different understanding of the semantics of LLVM instructions, which are only documented informally and hence subject to interpretation. If the code generator (which translates LLVM to machine code) interprets an instruction differently from DiVM, the simulated behaviour may be different from the real behaviour.
2. Undefined behaviour in C (or C++) programs may cause the compiler to significantly change the behaviour of the program. On one hand, DiVM works with this altered program and will detect anomalies caused by such compiler-induced changes. Unfortunately, such changes can also affect consistency checks performed by the program itself, e.g. assertions. In this sense, an actual safety violation may remain unreported because the assertion itself was removed or altered by the compiler.
3. The FPU modelled by DiVM is limited and programs which rely on non-default rounding modes or FPU exceptions are not handled soundly.

On the other hand, the following issues may appear as concerning, but are in fact addressed either directly by DiVM or in the wider ecosystem. These results apply more or less transparently to DivSIM.

4. While DiVM itself does not handle relaxed memory models, DiOS implements an emulation thereof based on reorder buffers (with user-selectable depth). The bitcode loader can be instructed to automatically translate memory access instructions to use this emulation layer. The current implementation provides the most common weak models (TSO and x86).
5. Program behaviour may depend on numeric values of pointers, e.g. because they are used as keys in search trees or hash tables, or due to program bugs. Optionally, DiVM can report any such behaviours as errors, and DiOS provides an abstract domain (in the sense of Section 4.4) which models numeric values of pointers symbolically. Using the latter, the full state space, including dependencies on numeric pointer values, can be explored faithfully.
6. The optimizer may change the behaviour of the program. Except as indicated in points 1 and 2 above, this is not a major concern, since the bulk of the optimisation is performed as an LLVM → LLVM transformation and the post-optimisation bitcode can be loaded into DiVM (and hence into the simulator), at the expense of some user comfort.

While strictly speaking, DiOS is an optional component (the environment can be modelled by the user in its entirety, if desired), it is almost always used, even if the program under test does not use POSIX APIs: the standard C library is also provided by DiOS, including fundamental primitives like `malloc` and `free`. Like with DiVM, the main source of problems is the interpretation of the relevant informal specifications (ISO C, POSIX). Fundamentally, it is impossible to guarantee that the behaviour of DiOS will match any particular native implementation. To minimize the risks, DiOS takes the following approach:

1. Perform strict validation of inputs and report any violations of documented preconditions as errors. While this is a form of inexactness (programs that work correctly when executed in a particular native environment may be rejected), this approach prevents much worse problems with unsoundness (the same program would not work correctly in a different native environment).
   Examples of such precondition checks are `free` (accepts only pointers which were returned by `malloc` and not yet freed – in this case, DiOS relies on DiVM to keep track of pointer validity), overlap checks in `memcpy` and `strcpy`, consistency checks in the `pthread` API, etc.
2. Non-deterministically simulate possible error conditions. This is again a possible source of inexactness, especially in cases where a particular failure is ruled out by external guarantees (e.g. the system is designed in such a way that there is always sufficient memory and hence `malloc` never fails). Conversely,

incomplete coverage of error conditions is a possible source of unsoundness.

Especially the second category is rather open-ended, and is not covered completely. In this sense, DiOS is unsound: for instance, in native execution, an arbitrary signal may arrive at any time, the content of the file system may change in arbitrary ways at arbitrary times, the system clock may move arbitrarily during execution, and so on. Many of these cases are either not covered at all (e.g. arbitrary incoming signals: those would need to be modelled explicitly by the user) or are not enabled by default due to the high rate of false positives (or rather true positives that are nonetheless not interesting to most users and most programs, e.g. arbitrary clock jumps or spontaneous file system changes).

6.2 Execution History

As mentioned earlier, soundness and exactness are not the only criteria that affect usability (and usefulness) of a simulator. Since we are not aware of any formal mechanism which could usefully model the correspondence between the real and the simulated execution, we will limit ourselves to informal observations about the possible problems and their impact.

6.2.1 Commutativity

Some operations in the program are commutative, and the outcome does not depend on their ordering. In sequential programs, this is mostly a hypothetical problem, since the main source of operation re-ordering is the compiler, which is the same for both the simulator and for real execution. Some care must be taken to use the same compiler flags for best results (or even better, generate the native code from the same bitcode).

In concurrent programs, the situation is nearly reversed: a huge number of commutations (between different threads) can happen in both real execution and in the simulation. Trying to match them 1:1 is both hopeless and unhelpful – in this case, the soundness and exactness criteria are the best we can hope for (i.e. the simulated execution has *some* result-equivalent counterpart in the real program, and vice versa). Informally, this alone should give a good approximation of execution history, since an assertion can be placed at any point of the program and if the simulator is sound and exact, the assertion will be reachable iff it is reachable in real execution. Unfortunately, adding the assertion can, in principle, change the trace arbitrarily. In practice, it will almost always only cause a small local perturbation.

6.2.2 Execution Model

The instruction set of the simulator (LLVM) is different from the instruction set used in the real execution (x86_64, arm64, etc.). Since the latter is obtained from the former, and LLVM is already quite granular, there is a close correspondence between the two programs. However, we again only have the high-level guarantees about behaviour correspondence. The code generator (the component responsible for translating LLVM instructions into native machine instructions) can, and does, perform additional optimisations, which include instruction reordering, splitting, and combining. Those effects can be observed by the user, though they are unlikely to be important in most cases.

The machine model of LLVM (and consequently of the simulator) is different than the one used in real execution: all contemporary CPUs are register machines with a fixed set of registers. This means that values need to be moved in and out of registers to perform operations on them, even in cases where the original LLVM bitcode does not indicate such movement of values. Like with the above, this is usually of little practical consequence to the user.

Finally, register spills can also interfere with concurrency, where a spill (not present in the LLVM code) can be observed by another thread. If soundness is assumed, the corresponding trace in the simulator will always end with an error before the discrepancy can be observed.[8] The soundness is, however, dependent on the behaviour of the code generator: we cannot rule out that a code generator will emit incorrect machine code (one with observable spills) from correct LLVM bitcode (where the affected memory location is updated and accessed correctly with respect to concurrency).

6.2.3 Memory Layout

The last major area where the simulated program and the natively executed program differ is the layout of the data stored in memory, with the most important difference concerning the organisation of the stack. In the simulator, the stack is organised as a linked data structure with each local variable allocated as separate object referenced from the main stack through a pointer (the layout is discussed in more detail in Section 3.2). In native execution, the stack is a flat array of bytes, with boundaries between activation records and local

[8] The typical cause will be an out-of-bound memory access on a stack-allocated variable. In real execution, this is not in itself an error, but will be detected and reported by DiVM, since each local variable is allocated in its own memory object (as discussed in Section 3).

variables implied by the code. Neither of those boundaries is made explicit in memory or enforced in any way, though a debugger can reconstruct them using debug metadata (at least as long as the stack is consistent).

From user perspective, this means that variable values can appear in unexpected locations in memory. However, since the representation used by the simulator is strictly more structured than the one used during native execution, we believe that it does not take significant effort on the part of the user to locate the desired values. Nonetheless, it is still a clear discrepancy that the user needs to take into account when working with DivSIM.

Additionally, like in the case of register spills discussed in Section 6.2.2, the different organisation can lead to different behaviour if assumptions made by the compiler are violated in the program. Again, assuming soundness, the simulator will truncate any affected traces by generating an error state, making the inconsistent behaviour inaccessible.

A second layout-related issue is the numeric values of pointers, which will differ between the simulation and real execution. However, these often differ even between individual native executions (even with fixed inputs) due to intentional randomization. For this reason, we do not consider this to be a major problem.

7 Implementation

DivSIM is distributed as part of the DIVINE 4 toolkit and is available through the `divine sim` subcommand. All relevant source code is available online,[9] 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[10].

7.1 User Interface

The data structures and most of the code are independent of a particular user interface. In fact, two user interfaces exist for the simulator. The primary interface is command-driven, similar to terminal-based symbolic debuggers like `gdb`. The command-line parser and other interface-specific code entails approximately 800 lines of C++. Additionally, a third-party graphical interface is also available.[11]

The command interface uses *meta variables* extensively: each such meta variable holds a reference to a single debug node (cf. Section 3.5). There are two basic types of meta variables, *static* and *dynamic*.

Static variables 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 current program state). These variables are prefixed with a `$` sign, e.g. `$frame`.

7.2 Programming Language Support

Our simulator design is, to a large degree, independent of the particular high-level language in which the simulated program was developed. The structure 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` [15] 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 obtain certain values without resorting to very low-level means (printing bytes at particular addresses). Consequently, the amount of implementation work required to support a programming language in a debugger can be substantial.[12]

On the other hand, the debug graph implemented in our simulator (see Section 3.5) is language-neutral, and hence the features derived from this graph are independent of the original programming language. For this reason, we consider the debug graph to be an important contribution: it can be built from LLVM debug metadata in a comparatively small amount of code, but nonetheless provides a very convenient interface.

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

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

[11] The source code of the graphical user interface is available from the supplementary materials page at `https://divine.fi.muni.cz/2021/sim/`.

[12] We speculate that this is the primary reason why interactive simulators (and debuggers in general) are so scarce.

7.3 Application Programming Interface

An important aspect of the presented simulator is that it is not just a standalone program, but also a re-usable library. The primary API is in C++ and has two levels:

1. a low-level interface which builds on DiVM and is available as a standalone library, `libdivine-dbg` and provides the following components:
   a. an interface to construct and explore the debug graph associated with a particular program state (as captured in a DiVM snapshot),
   b. a *stepper*, which implements the low-level instruction- and statement-level single stepping, along with the scaffolding required to implement breakpoints,
   c. utility functions – printing (formatting) instructions, export of the debug graph into the graphviz `dot` language, comparison (diffing) of debug nodes and so on,
2. a higher-level interface which makes the entire simulator available as a library, similar to how a user interacts with it (this interface is provided by `libdivine-sim`).

Of course, the high-level interface hooks into the low-level one where appropriate: for instance it provides low-level access to the debug node of the currently active stack frame, which can then be accessed using the low-level interface and used as a starting point for exploring the debug graph. Same for global variables and so on. The high-level API also gives the programmer access to each user-level command for simple automation tasks, with the user-directed output sent to a text stream of programmer's choosing. This makes it easy to hide, post-process or redirect the output, as needed.

Since the API is quite simple, only exposes a few classes (perhaps most importantly the debug node class), and can be used without using advanced C++ features (templates, smart pointers and so on), it is easily exported to object-oriented scripting languages, like Python.[13]

8 Evaluation

To demonstrate robustness and quantify performance of the simulator, we have executed it across a collection of C and C++ test programs, in two modes:

1. For the 608 test cases which contain a bug (and hence a counterexample), we have loaded the trace into the simulator, stepped to the end and printed a backtrace. We have then checked that the simulator reached the same error that was reported by the model checker.
2. The remaining 2090 test programs do not contain a bug, and hence also no counterexample to load and follow. On each of these programs, we have performed two random walks, one which consists of first 10000 atomic steps (atomic from the point of concurrency, as outlined in Section 5.1) and another which spanned 100000 LLVM instructions. In deterministic, sequential programs, the random walk obviously follows the only available execution. In programs with non-determinism (either due to concurrency, or due to automatic data abstraction), the result of each choice is simply picked at random (using a pseudo-random generator with a fixed seed, so that the same walk is easily reproduced).

The programs are sorted into these categories:

1. `abstract`: abstract data domains (Section 4.4),
2. `bricks`: unit tests of a C++ utility library,
3. `demo`: simple programs demonstrating various bugs,
4. `dios`: test cases for DiOS APIs,
5. `pv264`: assignments in an advanced C++ course,
6. `lang`: language feature tests for C and C++,
7. `libc`: tests for features of the standard C library,
8. `libcxx`: upstream unit tests from `libc++`,
9. `posix`: POSIX APIs (files, processes, signals, etc.),
10. `pthread`: POSIX threads,
11. `svcomp`: a selection of SV-COMP test cases (including symbolic data, Section 4.5),
12. `sym`: test cases for handling of symbolic values,
13. `undef`: detection of undefined values in C programs,
14. `vm`: virtual machine interface (hypercalls),
15. `weakmem`: threading with relaxed memory models.

8.1 Trace Statistics

We have collected some statistics about these runs, summarized in Table 1. The tabulated times are wall clock times when running on a contemporary CPU with a 2GHz base clock. The times include startup and loading of the bitcode, but not compilation of the source code (i.e. the input was a pre-compiled bitcode file). The main takeaways are:

1. The average times are favourable in almost all categories, for both the random walks and for counterexamples. Considering the length of the walks

[13] At the time of this writing, work is in progress to provide simple Python bindings for the C++ API, via Boost.Python. We believe the Python API will make DivSIM more accessible to 3rd-party developers.

and counterexamples, there would be hardly any delays when responding to stepping commands. One notable exception is the `pv264` category, where the average runtime is significantly longer. Overall, we note that C++ programs take a longer time to process (though part of this effect is also due to longer load times caused by larger bitcode files).

2. The maximal times for random walks are generally well-behaved as well. A notable case is `libcxx` where the slowest random walk along 100000 instructions took over 20 seconds, averaging to about 4500 instructions per second (in interactive use, this could translate to a noticeable delay for longer steps). The other notable figure appears in the category `pv264`, though in this case, the maximum is in line with the category average.
   The times for random walks that counted atomic steps are more varied, reflecting significant variation in the length of such steps. The trends are similar, though a few additional categories show fairly long maximal times (namely `bricks`, `dios` and `sym`).
3. Counterexample lengths span 3 magnitudes, though even very long counterexamples (millions of instructions) are processed in what we believe is a reasonable time (about 2 minutes, compared to about 20 minutes it takes the model checker to find and generate the counterexample). The overall average counterexample length is not very far from the 100000 instructions which we have used for the random walks of correct programs. This makes the average time in the correct and incorrect case roughly comparable, indicating that processing counterexamples is in fact marginally cheaper than walking the state space at random.

8.2 Debug Graph Statistics

In addition to trace lengths and timing information, we have collected information about the size of debug graphs at the point where the programs stopped (that is either at the end of the counterexample, or after 100000 instructions). Since the sizes only span a fairly small range of values, we have plotted them in a histogram in Figure 8. The debug graph size indicates the number of distinct memory nodes (components were not counted separately) for which it was possible to derive type information (that is, untyped vertices were not counted).

By far the most common size was 9 nodes, which corresponds to a standard C or C++ program that has terminated: in this case, only kernel data remains reachable. This was the outcome for 1059 programs, i.e. approximately half of the 'correct' programs. The largest recorded graph had 55 nodes and was generated from a `pthread` test case (in fact, all debug graphs with 40 or more nodes were generated by threaded programs).

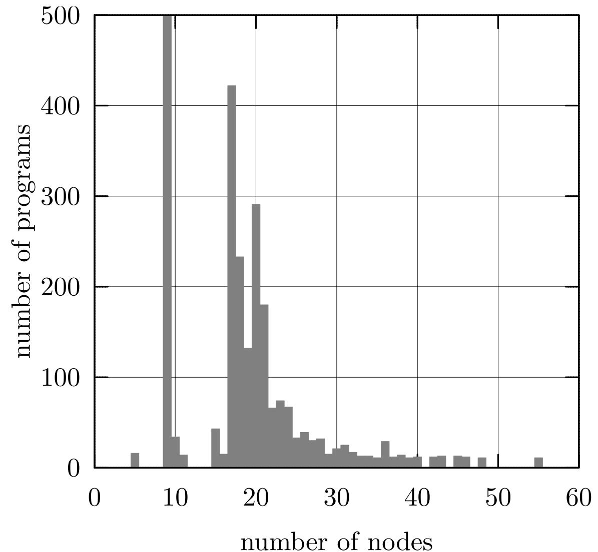

Fig. 8: Histogram of debug graph sizes. Small values have been exaggerated for readability (there is only 1 program with debug graph with 55 nodes) and the bar for 9 nodes has been cut off (the actual value is 1059 programs).

9 Conclusion

We have described and implemented a new approach to interactive analysis of real, multi-threaded C and C++ programs. In the wider context of automated verification and, in particular, model checking of software, we need every idea and every technique that can help us in the uphill battle against the complexity of the real world. We believe that the ideas presented in this paper are a piece of that puzzle, and will play a non-negligible role in the future of interactive tools in this space.

On the practical side, the inclusion of an interactive simulator has made DIVINE 4 substantially more useful (compared to earlier versions).[14] Additionally, following in the footsteps of DiOS, the first truly standalone, reusable component coming from the DIVINE ecosystem, DivSIM can be combined with existing LLVM-based tools. We hope this will foster inter-tool compatibility and cooperation in the broader LLVM space.

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

category	correct	avg(t_1)	max(t_1)	avg(t_2)	max(t_2)	c.e.	avg(len)	max(len)	avg(t)	max(t)
`abstract`	39	4.13	7.80	4.28	7.00	10	14129	57782	2.83	6.39
`bricks`	368	9.12	31.40	6.19	16.44	0	-	-	-	-
`demo`	0	-	-	-	-	6	17908	58104	6.97	29.52
`dios`	25	2.76	35.07	1.70	5.46	15	2598	8298	2.34	4.27
`pv264`	13	52.75	70.61	33.91	47.81	0	-	-	-	-
`lang`	55	2.61	16.38	2.54	4.39	32	4130	21138	2.35	4.61
`libc`	23	1.97	3.18	2.22	3.86	8	2670	3825	2.12	2.45
`libcxx`	902	10.16	68.69	4.75	22.35	8	14900	29559	7.27	13.55
`posix`	107	2.52	13.53	2.28	5.32	0	-	-	-	-
`pthread`	121	2.01	2.75	2.07	3.04	22	3559	8082	2.02	2.51
`svcomp`	228	4.10	28.46	2.89	6.53	276	121756	4101281	3.48	121.39
`sym`	168	4.41	63.56	3.01	4.79	102	5548	28159	2.80	3.79
`undef`	10	1.89	2.11	1.97	2.38	29	107	204	1.88	2.17
`vm`	21	2.67	11.32	2.22	3.65	33	1453	7293	2.38	3.99
`weakmem`	10	2.61	3.20	1.84	2.41	67	360394	522652	2.65	4.49
total	2090	7.61		4.37		608	97206		3.05	

Table 1. Summary of benchmark data. The columns 'correct' and 'c.e.' give the number of test cases of the respective type in the given category. For 'correct' test cases, t_1 gives the time required to perform 10000 atomic steps and t_2 the time to run 100000 instructions. The 'len' columns gives the length of the counterexample in instructions (average and maximum).

Finally, the simple, compact and universal counterexample format produced by DiVM and consumed by DivSIM has the potential to further improve interoperability of existing tools. Having adapted KLEE to generate DivSIM-compatible traces is the first step in that direction.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. *Acta Informatica*, 54 (8):789–818, 2017. doi: 10.1007/s00236-016-0275-0.
2. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003.
3. 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.
4. 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.
5. Samik Basu, Diptikalyan Saha, and Scott A. Smolka. Getting to the root of the problem: Focus statements for the analysis of counter-examples. 2012.
6. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *SFM*, 2004.
7. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
8. Marek Chalupa, Tomáš Jašek, Lukáš Tomovič, Martin Hruška, Veronika Šoková, Paulína Ayaziová, Jan Strejček, and Tomáš Vojnar. Symbiotic 7: Integration of predator and more. In *TACAS*, pages 413–417, Cham, 2020. Springer. ISBN 978-3-030-45237-7.
9. Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *Computer Aided Verification*, LNCS, pages 453–456. Springer, 2004.
10. Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna Verification Tool: IC3 for parallel software (competition contribution). In *TACAS*, pages 954–957, 2016. doi: 10.1007/978-3-662-49674-9_69.
11. Ruben Kleiman, Mike Brayshaw, Marc Eisenstadt, and Marc Eisenstadt. Tales of debugging from the front lines, 1993.
12. Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *PLDI*, PLDI 2019, page 96–110, New York, 2019. ACM. doi: 10.1145/3314221.3314609.
13. Henrich Lauko, Petr Ročkai, and Jiří Barnat. Symbolic computation via program transformation. In

Theoretical Aspects of Computing – ICTAC, pages 313–332, Cham, 2018. Springer.

14. Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. Extending DIVINE with symbolic verification using SMT. In *TACAS*, pages 204–208, Cham, 2019. Springer.
15. Keith Lee. *Using LLDB*, pages 415–434. Apress, Berkeley, CA, 2013. ISBN 978-1-4302-5051-7.
16. Axel Legay, Dirk Nowotka, Danny Bøgsted Poulsen, and Louis-Marie Tranouez. Statistical model checking of llvm code. In *Formal Methods*, pages 542–549, Cham, 2018. Springer.
17. Jeff Magee. Behavioral analysis of software architectures using LTSA. In *ICSE*, 1999.
18. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.
19. Petr Ročkai and Jiří Barnat. A simulator for llvm bitcode. In *Formal Methods for Industrial Critical Systems*, pages 127–142, Cham, 2019. Springer.
20. 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. doi: 10.1016/j.jss.2018.04.026.
21. Petr Ročkai, Zuzana Baranová, Jan Mrázek, Katarína Kejstová, and Jiří Barnat. Reproducible execution of POSIX programs with DiOS. *Software and Systems Modeling*, pages 1–20, 10 2020. doi: 10.1007/s10270-020-00837-y.
22. Richard Stallman, Roland Pesch, and Stan Shebs. Debugging with gdb. 2010.
23. The LLVM Project. LLVM language reference manual, 2016. URL `http://llvm.org/docs/LangRef.html`.
24. Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. URDB: a universal reversible debugger based on decomposing debugging histories. In *PLOS '11*, 2011.
25. Willem Visser and Alex Groce. What went wrong: Explaining counterexamples. In *SPIN*, LNCS, pages 121–135. Springer, 2002.