Automatic Verification of LLVM Code

Axel Legay¹, Dirk Nowotka², and Danny Bgsted Poulsen³

¹UCLouvain, Belgium
²Kiel University, Germany
³Aalborg University, Denmark

June 5, 2020

Abstract

In this work we present our work in developing a software verification tool for LLVM-code - Lodin- that incorporates both explicit-state model checking, statistical model checking and symbolic state model checking algorithms.

1 Introduction

Formal Methods, in particular Model Checking [1], have for many years promised to revolutionise the way we assert software correctness. It has gained a large following in the hardware design industry, but has yet to become mainstream in the software development industry - and this despite software being used in a large array of safety-critical components in e.g. cars and air planes. Nowadays, any non-trivial component of any system is controlled by an embedded microprocessor with a control program making software quality assurance more important than ever. Many case studies have shown that formal methods is a valuable tool - even in industrial contexts - but most successful applications have been conducted by academic researchers exploring formal methods usefulness.

One of the reasons that formal methods have not penetrated the software industry is, that formal methods require a translation of the source code to a formal model (e.g. Petri Nets or Automata) and the analysis conducted on these formal models. This is problematic as it requires industry engineers to invest quite some effort into understanding the formal modelling language and its associated tool. The diagnostic output for formal tools are also hard to understand without being an expert in formal methods. As a result, industry quality assurance relies on extensive testing - which will have to be done even after applying formal methods - and code reviews. Another complicating factor in applying the above mentioned workflow is, that sometimes the engineers do not know the source code intimately - parts of it might have been auto-generated and some of it might be legacy code. Attempting to translate code one has not developed to a formal model is very difficult and error-prone.

In summary, the learning curve of formal methods is steep thus industry engineers rely on other methods, and translating code to formal models is very hard and close to impossible. Formal tools are needed that understand the source code that industry already uses to ease the usage of formal tools in industry.

Academics have developed tools accepting pure code as inputs [2, 3, 13, 14]. A major breakthrough was achieved by tools such as BLAST [3] and SLAM [2] based around a Counter-Example-Guided-Abstraction-Refinement (CEGAR) [3], where a program text is explored symbolic based on a predicate abstraction of the program. The
predicates are continuously refined to make the abstraction as detailed as needed. Another approach, pioneered by the tool CBMC [16], is bounded model checking [6]. Here the program transition system is unrolled a number of times (in practice by unrolling loops and inlining function call), and encoded into a constraint system. During encoding the assertions can be added that has to be true along any execution (e.g. that a divisor is never zero). If the resulting constraint system has a solution where an assertion is true, then the system is not safe. CEGAR and Bounded Model Checking are incomplete, but are nevertheless both very successful in locating errors.

Nowadays the more successful software verification tools are CBMC [16] (bounded model checker) and CPACHECKER [4] (CEGAR-based tool - and direct successor of BLAST). The tools are among the dominating tools in Software Verification competitions.

CBMC and CPACHECKER are both tied to one source language thus major parts of the tools have to be implemented for each language they want to support. A better idea may be to base the analyses on an intermediate format that can capture the semantics of many high level languages. One such intermediate format is LLVM [17] which at least 4 tools are using:

1. LLBMC [13] follows in the footsteps of CBMC and performs bounded model checking on LLVM,

2. SeaHorn [15] has the objective of making verification platform for LLVM code, it seems to employ mostly CEGAR-based approaches,

3. Klee [3] is a symbolic execution engine performing a symbolic exploration of the state space, in order to find good test cases for testing, and

4. Divine [3] is an explicit-state model checker for LLVM code.

Although previously mentioned tools have paved the way for formal methods entering industry, they are not without flaws. A lot of them primarily focus on single-threaded programs which is a problem, because industry moves to multi core-architecture and verification thus needs to take interleaving into account. This interleaving is the cause of the state space explosion problem - a problem that the symbolic representation of LLBMC, CBMC and CPACHECKER cannot avoid. Although there has been some work in adapting at least CBMC to concurrent code, it is still an open problem how to verify concurrent programs efficiently.

In this paper we present the tool LODIN a fairly new tool [18] offering a range of verification techniques for LLVM. For concurrent programs it implements explicit-state reachability. Realising an exhaustive state space search will not scale for large programs, it also implements under-approximate state space searches through simulation. For single-threaded programs LODIN implements symbolic exploration akin to CBMC and LLBMC. In this way, LODIN distinguishes itself from existing tools by implementing several techniques into a joint framework.

LODIN achieves its ability to implement different techniques through its flexible architecture. Another feature of LODIN that sets it apart from other formal tools is its extensibility through platform plugins: the core of LODIN implements only the bare minimum semantics of LLVM and has no knowledge of the runtime environment of the program. In real-life programs, the executing program may call into the runtime environment which LODIN must know about in order to provide correct verification results. The platform plugins serves as a way to provide these implementations.

2 LLVM

Although the focus of this paper is not to describe the LLVM [17] language itself, we spend some time on presenting a simplified version of the LLVM instruction set and its semantics. The full LLVM language description is available online [12]. The description we provide is closely linked to the implementation inside LODIN.
LLVM-Listing 1: An example LLVM module with a single entry point @main.

2.1 Structure of LLVM programs

An LLVM module consists of functions of which some of them may be entry point functions which are starting points for an LLVM process. Functions are divided into Basic Blocks where a Basic Block is a sequence of instructions executed in a linear fashion. Basic blocks are named by labels, so that instructions can direct control to the basic block. Individual instructions within a basic block can be pure arithmetic operations, memory allocations, memory accesses, function calls or instructions that passes control to other basic blocks. Basic blocks are always terminated by the latter class thus these are called terminator instructions. Operands to the instructions of an LLVM program are kept in so-called registers, and a syntactical requirement for an LLVM is that it must be in single-static-assignment i.e. each register is only assigned once.

In LLVM-Listing 1 is shown a very short LLVM program. The program consists of a single function @main (which is also the entry point) that consists of three basic blocks init, blk and succ. The blocks covers lines 4 – 5, 7 – 10 and 12 – 13 respectively. The terminating instruction links init to block blk and links blk to succ and blk. We refer to Figure 1 for a graphical depiction of how the basic blocks are linked together.

LLVM Types All operations in LLVM are typed, either with an arbitrary width bitvector, a compound datatype\(^2\) or a memory pointer. The bitvector is denoted \(\text{in}_n\) where \(n\) is the width. For our discussion, we restrict ourselves to bitvectors that are multiple of bytes thus we let

\[
T_{\text{int}} = \{ \text{in}_n | n \in \{8, 16, 24, 32, \ldots, \} \}
\]

be the set of all integer types in LLVM. If \(t_1, \ldots, t_n\) are LLVM types then \(\langle t_1, \ldots, t_{n-1} \rangle\) is a compound type. We denote by \(T_{\text{comp}}\) all compound LLVM types. For a type \(\langle t_1, \ldots, t_{n-1} \rangle\) and sequence of integers \(i_1, \ldots, i_k\) we let

\[
T_{i_1, \ldots, i_k} (\langle t_1, \ldots, t_{n-1} \rangle) = T_{i_2, \ldots, i_k} (t_{i_1})
\]

\[
T_i (t) = t_y.
\]

A memory pointer type to a type \(t_y\) is denoted \(t_{y*}\). LLVM leaves the bithwidth of pointer types unspecified - for the remainder of this paper we assume it is 64 bit. As is customary in C-style languages, LLVM includes the void type used to signify a function does not return a value.

It will often be convenient to talk about the byte-

---

\(^2\)Like C-Style structs
size of a type. We therefore define the function
\[
BSize(ty) = \begin{cases}
\frac{n}{m} & \text{if } ty = in \\
\sum_{i=1}^{n} BSize(ty_i) & \text{if } ty = \langle ty_1, \ldots, ty_n \rangle \\
8 & \text{if } ty = in^*
\end{cases}
\]

We let \( T \) denote the set of all types in LLVM.

**LLVM instructions** Let \( R \) be a set of registers, \( BL \) be a finite set of basic block labels and let \( Fs \) be a finite set of function names, then Table 1 displays the instruction set used in our discussion of LLVM. In the table \( BInst(R) = Arith(R) \cup Log(R) \cup Mem(R) \cup Cmp(R) \cup Intrin(R) \) are the basic instructions while \( Term(R, BL) \) are instructions terminating a basic blocks (e.g. jumps). A short description of the intended meaning of the instruction classes may be in order:

- **Arith(R)** Instructions in this class are arithmetic instructions that takes two registers \( \%inp1 \) and \( \%inp2 \), perform the mathematical operation and store the result in \( \%res \). It is worth noting that since LLVM has no signed and unsigned types it instead has signed and unsigned versions of some instructions. Prime examples of this is the remainder (\( \text{rem} \)) and the division (\( \text{div} \)) instructions. Signed and unsigned versions are distinguished by the prefixes ‘s’ and ‘u’.

- **Log(R)** This class consists of instructions performing bitwise operations. It might be worth mentioning the bit shift operations. Shifting to the left, \( \text{lshr} \), is performed by moving the bit pattern towards the most significant bit and pad with zeros. For Shifting to the right, LLVM has to operations \( \text{shl} \) and \( \text{ashr} \). The \( \text{lshr} \) is similar to left shifting with the difference that the pattern is shifted to the least significant bit and called a logical shift. The \( \text{ashr} \) is on the other hand a arithmetic right shift, which preserves the sign bit of the pattern.

- **Mem(R)** This instructions class has instructions for allocating memory, loading a value from a memory address and a value at a memory address. A special instruction in this class is the \( \text{getelementptr} \) instruction indexing into a compound type stored in memory. It can be thought of as the dereferencing operator in C.

- **Cmp(R)** This class of instructions are used for comparing the values of registers. As an example, \( \%res = \text{cmple} \%inp1, \%inp2 \) compares if \( \%inp1 \) is less than or equal to \( \%inp2 \) while interpreting \( \%inp1 \) and \( \%inp2 \) as unsigned integers.

- **Term(R, BL)** This class consists of instructions terminating a block. A terminating action can either be a jump to another block or a return from a function. For jumping there are two different version: The unconditional version \( \text{br \ label \ %block} \) that jumps to the specified block no matter what, and the conditional \( \text{br \ %cond, \ label \ %ttblock, \ label \ %ffblock} \) that jumps to \( \text{ttblock} \) if the pattern in \( \%cond \) corresponds to true and to \( \text{ffblock} \) otherwise. There are also two return instructions: an instruction \( \text{ret \ void} \) that does not return a value and one that does \( \text{ret \ %ty, \ %res} \).

- **CInst(R, Fs)** Instruction for calling other functions. The instruction for calling a function with name \( @\text{func} \) is \( \%res = \text{call \ ret \ @\text{func} \ %ty1, \%p1, \ldots, \%tyN, \%pN} \). As one would expect, this pass control to the function \( @\text{func} \), passes \( \%p1, \ldots, \%pN \) as parameters and stores the result of the function call into \( \%res \).

- **Phi(R, BL)** The instruction class \( \text{Phi(R, BL)} \) consists of instructions selecting a value based on which basic block control flowed from. The instructions are needed, because LLVM-programs are in single-static-assignment form. The instructions are only allowed in the start of a basic block and must be executed simultaneously i.e. the evaluation of one phi-instruction cannot affect the result of another in the same block.

- **Intrin(R)** This class is a set of “extension instructions” used by LODIN. Currently it only consists of instructions that returns a non-deterministic value.
### LLVM-Listing 2: Example program for using phi i32

```c
1 define dso_local i32 @main() {
2 init:
3 %1 = call i32 (...) @__VERIFIER_nondet_int()
4 %2 = icmp ne i32 %1, 0
5 br i1 %2, label branch, label end
6 branch:
7 %4 = add nsw i32 %1, 1
8 br label end
9 end:
10 %, 0 = phi i32 [ %4, branch ], [ %1, init ]
11 ret i32 %, 0
12 }
```

**Remark 1.** All instructions in Table 1 can take constants as parameters in addition to real registers. For ease of exposition we will, however, treat constants as standard registers.

### Formal Definitions of LLVM Modules

In the introduction to this section, we mentioned that LLVM programs consists of functions (of which some may be program entry points) and functions consists of basic blocks. We are now turning towards giving proper formal definitions of these concepts.

**Definition 1 (Basic Block).** Let $BL$ be a set of labels, $Fs$ be a set of functions names and $R$ be a set of registers, then a basic block, $B$, is a finite sequence $I_0I_1...I_n$ of instruction where

- for all $i < n$, $I_i \in BInst(R) \cup CInst(R, Fs) \cup Phi(R, BL)$,

- $I_n \in Term(R, BL)$ and

- if $I_i \in Phi(R, BL)$ then $\forall j < i$, $I_j \in Phi(R, BL)$.

We denote the set of all possible basic blocks over $BL$, $R$ and $Fs$ by $BB(R, BL, Fs)$

As a convention, if $B = I_0I_1...I_n$ is a basic block then we write $|B| = n$ for its length and we let $B[i] = I_i$.

**Definition 2 (Function).** A function $F$ with $n$ parameters over the function names $Fs$ is a tuple $(\alpha_R, R, P, BL, BBs, Bm, ret)$ where

- $\alpha_R \in Fs$ is the functions name,

- $R$ is a set of registers,

- $P = p_1, ..., p_k$ where for all $i, p_i \in R$, is a sequence of registers used as parameters,

- $BL$ is a finite set of labels with the requirement that $init \in BL$,

- $BBs \subseteq BB(R, BL, Fs)$ is a finite set of blocks,

- $Bm : BL \rightarrow BBs$ assigns each block label a basic block and

- $ret \in T$ is the return type of the function.

**Definition 3 (Program Entry Point).** A program entry point is a function $(\alpha_R, R, \emptyset, BL, BBs, Bm, \text{void})$.

**Definition 4 (Module).** An LLVM module $M$ is a tuple $(F, E)$ where

- $F = \{ F_1, ..., F_n \}$ is a collection of functions where $\forall i, F_i = (\alpha_R, R, P_i, BL_i, BBs_i, Bm_i, ret_i)$, and for all $k \neq j, R_k \cap R_j = \emptyset$ and

- $E = k_1, ..., k_m$ is a list of indices defining the entry functions i.e. $\forall 1 \leq i \leq m, F_i$ is an entry point function.

For module $M = (F, E)$ we abuse notation slightly and allows writing $F \in M$ whenever $F \in F$.

### Well-typedness

For each register in $\%r \in R$ we assign a type from $t \in T$ and write $\%r : t$ to denote that $\%r$ has type $t$. If a list of registers $\%r_1, ..., \%r_n$ has the same type $ty$, we write $\%r_1, ..., \%r_n : ty$. Generalising this notation to an instruction $inst$, we write $inst : ty$ to denote $inst$ is well-typed with type $ty$.

**Figure 2** shows the type rules of LLVM instructions. For a function $F = (\alpha_R, R, P, BL, BBs, Bm, \text{retty})$ we write $\text{Rets}(F)$ to get all return instructions within that functions basic blocks. Given this we say that $F$ is well-typed $(F : \text{retty})$ if for all $inst \in \text{Rets}(F)$, $inst : \text{retty}$ and all other instructions are well-typed.
| Arith(R) | %res = add ty %inpl, %imp2 | %res = sub ty %inpl, %imp2 |
| Log(R)   | %res = shl ty %inpl, %imp2 | %res = lshr ty %inpl, %imp2 |
| Mem(R)   | %res = alloc ty           | %res = getelementptr ty, ty* %ptr, tylinpl, ..., tynindn |
| Cmp(R)   | %res = cmp eq ty %inpl, %imp2 | %res = cmp ne ty %inpl, %imp2 |
|          | %res = cmp ugt ty %inpl, %imp2 | %res = cmp ult ty %inpl, %imp2 |
|          | %res = cmp sgt ty %inpl, %imp2 | %res = cmp slt ty %inpl, %imp2 |

| Term(R, BL) | ret void |
|            | br label %block |

| Phi(R, BL) | %res = phi ty [%inpl, %lab1] ... [%inpl, %labm] |

| CInst(R, Fs) | %res = call ret %func (ty1 %p1 ... tyn %pn) |

Table 1: Basic instructions over a set of registers R and basic block names BL, where %cond, %res, %inpl, ... is in R, block, tbblock, fbblock, lab1, ..., labm in BL, @func in Fs and for all i, indi ∈ Z.

Modelling External Dependencies  
A common problem in software verification is that the system we want to verify depends on external library functions (e.g. libc), or functions interacting directly with the operating system (e.g. pthread). In principle we could extend the LLVM language with implementations for all these external function calls but it would unnecessarily inflate the semantics, and the semantics would have to be redefined for each external library and operating system.

LODIN combats this problem in two ways: 1. LODIN extends the LLVM language with the %1 = lodindty instruction that returns nondeterministic values, allowing a programmer to replace external function calls with %1 = lodindty and thereby explore all possible results of external function calls, and 2. LODIN allows programmers to extend the LODIN interpreter through platform plugins that provide implementations of external functions. Calls to external function calls are syntactically indistinguishable from function defined in the LLVM module itself.

2.2 Contextual Interface

LODIN has been developed with reusability in mind allowing to use core components for both explicit state analysis and symbolic state analysis. The semantics we present in the following reflect this reusability by defining the core semantics in terms of a context. The context is responsible for representing the register values, how memory is represented and for implementing operations on registers. The core semantics “just” translate the LLVM instruction set to operations on context states and keeps track of the control flow. In some sense one could consider the context being a “virtual machine”.

A context provides the LLVM program with an infinite set of register variables which the context maps to actual values. The intention is that a
Figure 2: Type rules for LLVM for which we have \((\%res = \text{inst ty} \%imp1, \%imp2) \in \text{Arith}(R) \cup \text{Log}(R)\) and \((\%res = \text{cmp cc ty} \%imp1, \%imp2) \in \text{Cmp}(R)\)

A collection of operations are needed for a LLVM program to manipulate the states of a context. Most of these operations are just semantical functions for LLVM instructions (see Table 2). Instead of writing \(\circ(S, t_1, t_2) = R\) when applying an operator, we use an infix notation \([t_1 \circ t_2]_S = R\). Besides the instructions in Table 2, we need instructions for creating new register variables (\(\text{mReg}\)), evaluate the value of a register variable (\(\text{Eval}_A^{\text{ty}}\)), loading (\(\text{load}_A^{\text{ty}}\)) and storing (\(\text{store}_A^{\text{ty}}\)) values from/to memory, allocating memory (\(\text{alloc}_A^{\text{ty}}\)) and freeing memory (\(\text{free}\)). We discuss them briefly in the following from a usage-perspetive:

\[\text{mReg}_A : S_A \times R \rightarrow S_A \times R\]  
This function takes a context state \(s_A\) and a register \%r, where \%r : ty. It returns a register variable \(r \in R\) that can be used to store values of ty and a new context state \(s\). Naturally, the context must ensure that the register variable \(r\) is not already used in \(s_A\).
| Instruction | Operator | Signature |
|-------------|----------|-----------|
| Addition    | add      | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Subtraction | sub      | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Multiplication | mul | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Unsigned Division | div | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Signed Division | sdiv | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Signed Remainder | rem | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Unsigned Modulo | srem | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Shift left | shl | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Logical Shift right | lshr | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Arithmetic shift right | ashr | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Bitwise and | and | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Bitwise or | or | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Bitwise xor | xor | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow \mathcal{Y}^{\text{dom}_A(ty)}$ |
| Equality | eq | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Non-equality | ne | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Signed Greater than | sgt | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Signed Greater than or equal | sge | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Signed Less than or equal | sle | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Signed Less than | slt | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Unsigned Greater than | ugt | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Unsigned Greater than or equal |uge | $S_A \times \text{dom}_A(18) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Unsigned Less than or equal | ule | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |
| Unsigned Less than | ult | $S_A \times \text{dom}_A(ty) \times \text{dom}_A(ty) \rightarrow S_A \times \text{dom}_A(18) \times \{T, \bot\}$ |

Table 2: Operations for a context $\mathcal{A} = (S_A, s^{\text{init}}, \text{dom}_A, \mathcal{R})$. They each take as input a context state and operands and returns a new context states and a return value. The compare instructions also return a value in $\{T, \bot\}$.

**Eval** $^\mathcal{A} : S_A \times \mathcal{R} \rightarrow \text{dom}_A(ty)$ This function takes a context state $s$ and register variable $r \in \mathcal{R}$, and returns a value in $\text{dom}_A(ty)$.

**Set** $^\mathcal{A} : S_A \times \mathcal{R} \times \text{dom}_A(ty) \rightarrow S_A$ This function takes a context state $s$, register variable $r \in \mathcal{R}$ with type $ty$ and a value $v \in \text{dom}_A(ty)$. It returns a new context state $s'$ with $r$ bound to the value $v$.

**load** $^\mathcal{A} : S_A \times \text{dom}_A(ty) \rightarrow \text{dom}_A(\mathcal{Y})$ This function takes a context state $s$ and a memory address in $\text{dom}_A(\mathcal{Y})$ and returns a subset of $\text{dom}_A(ty)$.

**store** $^\mathcal{A} : S_A \times \text{dom}_A(ty) \times \text{dom}_A(\mathcal{Y}) \rightarrow S_A$ This function takes a context state $s$ and values $v \in \text{dom}_A(ty)$ and $a \in \text{dom}_A(\mathcal{Y})$. It returns a new $s'$ where the value the memory address $a$ has been updated to the value $v$.

**alloc** $^\mathcal{A} : S_A \rightarrow S_A \times \text{dom}_A(\mathcal{Y})$ This function takes a context state $s$ and returns a tuple $(s', t_1)$ where $t_1 \in \mathcal{Y}$ is a newly allocated memory address with space for a type $ty$, and $s'$ is a new context state updated with information that $t$ is no longer free for allocation.

**free** $^\mathcal{A} : S_A \times \bigcup_{t \in \mathcal{Y}} \text{dom}_A(\mathcal{Y}) \rightarrow S_A$ This function takes a context state $s$ and a value in $k \in \mathcal{Y}$.
union \( \cup_{k \in \mathbb{N}} \mathsf{dom}_A(i_{\text{ret}}) \). It returns a new context state \( s' \) where the memory pointed to by \( k \) has been released.

\[
\mathsf{NonDet}^y_A : S_A \to S_A \times 2^{\mathsf{dom}_A(ty)} \quad \text{This function takes a context state } s \text{ and returns a subset of } \mathsf{dom}_A(ty) \text{ and a new context state.}
\]

\[
\mathsf{PtrAdd}^y_A : \mathsf{dom}_A(ty) \times \mathbb{Z} \to \mathsf{dom}_A(ty) \quad \text{This function takes a pointer } p \text{ and natural number } b \text{ and returns a pointer new pointer after adding } b \text{ bytes to } p.
\]

**Core Semantics**

We are now ready to define the core semantics for a single LLVM process relative to a given context. The state of a single process (e.g. instruction to be executed, what function it is executing, which block was previously executed, mapping the functions register to context register variables) is kept in an activation record. The activation record also has a list of memory addresses, that must be deallocated when control leaves the currently executing function. If a function calls another function, an activation record is pushed in front of the current one thus forming a stack of activation record.

**Remark 2.** An activation record roughly corresponds to the well-known concept of a stackframe. LLVM does however not assume the existence of a stack and rather in the activation keeps a set of memory addresses that must be released when removing the activation record (corresponding to popping the stackframe in stack-based systems).

**Definition 6** (Activation Record). An activation record, relative to a context \( (S_A, s_A, \mathsf{dom}_A, R, ff) \) is a tuple \( (F, \text{prev}, \text{cur}, \text{pc}, \pi, \mathsf{Free}) \) where

- \( F = (\mathbb{N}, R, P, B, B, B) \) is the LLVM function currently being executed,
- \( \text{prev} \in \mathbb{N} \) is the label of the block executed before the current one,
- \( \text{cur} \in \mathbb{N} \) is the label of the currently executed basic block,
- \( \text{pc} \in \mathbb{N} \) is a pointer into the current basic block to locate the next instruction to be executed,
- \( \pi : R \to R \) maps registers to register variables of the context and
- \( \mathsf{Free} \) is a set of memory addresses that must be deleted when removing this activation record.

**Remark 3.** Intuitively, an activation record is split into two parts: 1. A static part that indicates which instruction to be executed, given by \( F, \text{prev}, \text{cur} \) and \( \text{pc} \), and 2. a dynamic part that links the process to the memory model of the context, given by \( \pi \) and \( \mathsf{Free} \).

A stack of activation records is a structure \( s_1 : s_2 : \cdots : s_n \) where each \( s_i \) is an activation record. The empty stack is denoted by \( \epsilon \). In the transition rules in Figure 2 we usually use the notation \( s_1 : \mathbb{SL} \) meaning that \( s_1 \) is the head of the stack and \( \mathbb{SL} \) is the remaining part of the stack. We also write \( \mathsf{Inst} \text{ def} (\mathsf{EXPR}) \) to denote that \( \mathsf{Inst} \) is syntactically equivalent to \( \mathsf{EXPR} \). The transition rules are defined relative to a context state \( s \) and a module. Given a context state \( s' \) and module \( M \) the rules define how to execute an instruction \( \mathsf{Inst} \) from state \( (s, \mathbb{SL}) \), where \( s \) is an activation record and \( \mathbb{SL} \) is a stack, to produce the tuple \((s', \mathbb{SL}')\) where \( s' \) is a new state and \( \mathbb{SL}' \) is a new context state. We write this as

\[
s, M \vdash (s, \mathbb{SL}) \xrightarrow{\mathsf{Inst}} (s', \mathbb{SL}'), s'.
\]

The rules may look intimidating but most of them are fairly straightforward. As an example let us briefly consider the rule for binary operators (that are not comparisons) i.e.

\[
\begin{array}{ll}
\text{Binary} & \text{Inst} = \mathsf{Add}, \mathsf{Sub} \quad I \in \{\mathsf{Inst}_2^R(\mathsf{Assign}) + \mathsf{Inst}_2^S(\mathsf{Assign})\} \\
1, M \vdash (x, \mathbb{SL}) & \xrightarrow{\mathsf{Inst}_2^R(\mathsf{Assign})}(y, \mathbb{SL} + 1, \text{prev}, \text{cur}, \text{pc} + 1, \mathsf{Free}, R, \mathsf{Set}_2^R(x, y))
\end{array}
\]

This rule says, that in order to execute an instruction \( \%\text{res} = \mathsf{inst}_{ty} \%\text{in1}, \%\text{in2} \) we first figure out which register variables in \( s \) that contain the values of \( \%\text{in1}, \%\text{in2}, \%\text{res} \). This look up is done with
Figure 3: Transition Rules for memory instructions

calls to \( \pi \) and results kept in \( r_1, r_2, r_{\text{res}} \). Then we evaluate the value of \( r_1 \) and \( r_2 \) in \( s \) via calls to \( \text{Eval}^y \), and the operation corresponding to \( \text{inst} \) is looked up with \( \circ \) (see Table 2) for this mapping and applied (\( [\text{Eval}^y(s, r_1)] \circ (\text{inst}) \text{Eval}^y(s, r_2)]_s \) giving a new context state \((s')\), and the value of the operation \((v)\). \( \text{Set}^y (s', r_{\text{res}}, v) \) stores this new value in \( r_{\text{res}} \) and returns the new context state. Finally we update the program counter \((pc + 1)\).

In the rules special care has to be taken for the \text{phi} \( ty \) instructions. All of these must be evaluated simultaneously. We therefore evaluate the them in a big-step fashion where the evaluation of one instruction also result in evaluating the next instruction (if it is also a \text{phi} \( ty \) instruction). For the \text{getelementptr} rule, we use the auxiliary function
\[
T_i(\text{ty}_1, \ldots, \text{ty}_n) = \sum_{k=1}^{i-1} \text{BSize}(\text{ty}_k) + T_{i-k}(\text{ty}_i)
\]
\[
T_i(\text{ty}) = 0
\]
to calculate the offset needed to access the correct element of the designated type.

Remark 4. If LODIN has some functions defined in a platform plugin, the call rule in Figure 7 is replaced by the implementation described in that module instead. Platform functions are executed atomically in LODIN.

---

**Figure 4:** Transition rules for terminator instructions

**Figure 5:** Compare Rules for Phi instructions

**Figure 6:** Compare Rules for comparison instructions
Binary

\texttt{Inst} = \texttt{Inst(\text{cur}[\text{pc}])} \quad \forall \text{ pc} \in \{\text{Eval} \} (s, t_i) \circ (\text{Inst}) \text{Eval} (s, t_i) s, m \models (s, sl) \xrightarrow{\text{Inst}} ((\text{F, prev, cur, pc} + 1, r, \text{Free, SL}), \text{Set}^\text{\texttt{F}}(s, r, m, v))


\text{Call Function}

\text{Inst} = \text{Inst(\text{cur}[\text{pc}])} \quad \forall \text{ pc} \in \{\text{Eval} \} (s, t_i) \circ (\text{Inst}) \text{Eval} (s, t_i) s, m \models (s, sl) \xrightarrow{\text{Inst}} ((\text{F, prev, cur, pc} + 1, r, \text{Free, SL}), \text{Set}^\text{\texttt{F}}(s, r, m, v))

\text{NonDet}

\text{Inst} = \text{Inst(\text{cur}[\text{pc}])} \quad \text{NonDet} (s) = V, s' \quad \forall V \in \text{V} s, m \models (s, sl) \xrightarrow{\text{Inst}} ((\text{F, prev, cur, pc} + 1, r, \text{Free, SL}), \text{Set}^\text{\texttt{F}}(s', r, m, v))

\text{Get registers}

\text{Inst} = \text{Inst(\text{cur}[\text{pc}])} \quad \forall \text{ pc} \in \{\text{Eval} \} (s, t_i) \circ (\text{Inst}) \text{Eval} (s, t_i) s, m \models (s, sl) \xrightarrow{\text{Inst}} ((\text{F, prev, cur, pc} + 1, r, \text{Free, SL}), \text{Set}^\text{\texttt{F}}(s, r, m, v))

\text{Figure 7: Miscellaneous rules Rules

3 Representations in Lodin

In the preceding section we developed the semantics of LLVM programs abstractly i.e. we defined an “interface” to a context of the semantics, allowing instantiating different semantics by modifying the instantiation of this interface. In this section we develop two instantiations (\texttt{Eand S}) of the interface. The resulting transition semantics for module \texttt{M}, \texttt{L}^\texttt{E}(\texttt{L}^\texttt{S}), we call the explicit (symbolic) semantics.

3.1 Explicit Representation

\textbf{Bitvectors} Let \mathbb{B} = \{0, 1\} then a bitvector of width \text{n} is an element in \mathbb{B}^\text{n}. Two special bitvectors are \text{0}^\text{n} = (0, 0, \ldots, 0) \in \mathbb{B}^\text{n} and \text{1}^\text{n} = (1, 1, \ldots, 1) \in \mathbb{B}^\text{n}. If \text{b} = (b_0, b_2, \ldots, b_{\text{n}-1}) \in \mathbb{B}^\text{n} is a bitvector, then we can access individual bits by indexing into \text{b} i.e. \text{b}[i]. We also allow extracting the sub-vector \text{b}(i, \ldots, j) by \text{b}[i : j + 1]. If \text{b} = (b_0, b_2, \ldots, b_{\text{n}-1}) \in \mathbb{B}^\text{n}, \text{c} = (c_0, \ldots, c_{\text{n}-1}) \in \mathbb{B}^\text{k}, k \in \{0, \ldots, \text{n} - 1\} and \text{k} + i < \text{n} then we let \text{c}[k : k + i] = (b_0, b_1, b_{k+1}, c_0, \ldots, c_{i-1}, b_{k+i}, \ldots, b_{\text{n}-1}).

Let \text{b} = (b_0, b_2, \ldots, b_{\text{n}-1}) \in \mathbb{B}^\text{n} be a bitvector, then we can interpret it as either an unsigned integer or a signed integer. In the prior case we use the standard binary encoding and define \text{b} = \sum_{i=0}^{\text{n}} b_i \cdot 2^i. In the latter case we use 2’s-complement encoding.
and let \( \langle \bar{b} \rangle = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i2^i \). To encode a number \( n \in \mathbb{N} \) in either binary or 2s-complement we write \( \langle n \rangle^{-1} \) and \( \langle n \rangle^{-1} \) respectively.

The classic bitwise operators, and, or, xor and negation, between vector \( \bar{b}_1, \bar{b}_2 \in \mathbb{B}^n \) are defined as usual and denoted (\( \bar{b}_1 \text{ and } \bar{b}_2 \)), (\( \bar{b}_1 \text{ or } \bar{b}_2 \)), (\( \bar{b}_1 \text{ xor } \bar{b}_2 \)) and (\( \text{neg } \bar{b}_1 \)) respectively. If \( \bar{b}_1 \in \mathbb{B}^n \) is a bitvector, \( d \in \mathbb{N} \) is a number and \( d < n \) then we define bit shifting operations as

\[
\begin{align*}
\bar{b}_1 \text{ lshl } d &= \bar{o}^n[0 : d/b_1[0 : n - d]], \\
\bar{b}_1 \text{ lshl } d &= \bar{b}_1[n - d : n] \\
\bar{b}_1 \text{ ashr } d &= \begin{cases}
\bar{o}^n[0 : d/b_1[0 : n - d : n]] & \text{if } b_1[n - 1] = 0 \\
\bar{b}^n[0 : d/b_1[0 : n - d : n]] & \text{if } b_1[n - 1] = 1
\end{cases}
\end{align*}
\]

The \texttt{lshl} (\texttt{lsr}) operator is a logic left (right) bitshift i.e. shift all bits to left (right) and pad with zero. The \texttt{ashr} is arithmetic right shift where instead of padding with zero, the bit vector is padded with the original value of the most significant bit.

**Memory Modelling** In the explicit semantics we model the memory state of a computer as a (possibly) infinite length array of memory blocks. Memory blocks are tagged with their size and the actual content of the block. Formally, the memory state of program is a function \( M : \mathbb{N} \rightarrow (\mathbb{N} \times (\bigcup_{i \in \mathbb{N}} \mathbb{B}^i)) \cup \{ \mathcal{S} \} \). An entry \( MB(i) \) means that block \( i \) of the memory has not been used. If \( MB(i) = (k, \bar{b}) \) and \( \bar{b} \in \mathbb{B}^k \), then we say that block \( i \) is consistent, has \( k \) and \( \bar{b} \) is the content of that block.

To modify and read from memory, we define the functions:

- \texttt{new}(M, i) = (M[i \mapsto \langle i, \bar{0} \rangle], n \text{ where } n = \min\{g \mid M(g) = \mathcal{S}\}),
- \texttt{Memfree}(M, \texttt{Used}, i) = (M[i \mapsto \mathcal{S}]),
- \texttt{read}(M, b, f, len) = \bar{b}[f : f + \text{len}] \text{ where } M(b) = (i, \bar{b}) \text{ and } f + \text{len} < i \text{ and }

\[
\begin{array}{c|c}
\text{offset} & \text{size} \\
\hline
0 & \bot \\
1 & \bot \\
\vdots & \\
\text{block} & (\text{size,}) \\
\end{array}
\]

**Figure 8:** Memory representation in LODIN. Pointers are 64bit integers split into a 32bit base and a 32bit offset. LODIN uses a redirection table \( M \) that stores memory blocks, and block indexes into this table, while offset indexes into the memory blocks. The symbol \( \bot \) indicates an entry in \( M \) is unused.

- \texttt{write}(M, b, f, \bar{c}, \text{len}) = (M[b \mapsto \bar{b}][f : f + \text{len}/\bar{c}]\text{Used}) \text{ where } M(b) = (i, \bar{b}) \text{ and } f + \text{len} < i \text{ and }

The initial state of the memory is the function \( M_{\text{init}} \) where for all \( i, M_{\text{init}}(i) = \mathcal{S} \).

Given both a representation of the register values and the memory, we can now define the explicit context. In the explicit context, we assign to a type in the domain \( \mathbb{B}^n \) and any pointer type is assigned the domain \( \mathbb{B}^{64} \). Using a 64-bit bitvector for representing pointers allows us to use the 32 most significant for indexing into \( M \) of the memory and use 32 least significant bits to index into the actual block. For a pointer \( p \in \mathbb{B}^{64} \) we let \( \text{block}(p) = p[32 : 64] \) and \( \text{offset}(p) = p[0 : 32] \). See Figure 8 for a graphical depiction of how this work.

**Definition 7** (Explicit Context). The explicit context is the tuple \( E = (S_E, S_E^{\text{init}}, \text{dom}_E, N, ff_E) \) where

- \( S_E = \{(M, N, F) \mid M \text{ is a memory state } \land N \subset \mathbb{N} \land F : N \rightarrow (\bigcup_{i \in \mathbb{N}} \mathbb{B}^i) \cup \{ \bot \} \}
- \( S_E^{\text{init}} = (m_{\text{init}}, \emptyset, F) \text{ where for all } i, F(i) = \bot, \)
- \( \text{dom}_E(e) = \mathbb{B}^{N_{\text{Size}(e)}} \)
- \( ff_E = (0)^{-1} \).
\[
\text{\texttt{mReg}_E((M, N, F), \%r)} = (M, N \cup \{i\}, F), i \text{ where } i = \min(N \setminus N)
\]

\[
\text{\texttt{Eval}_E((M, N, F), i)} = \begin{cases} 
F(i) & \text{if } F(i) \in \text{dom}_E(\text{ty}) \\
\text{Error} & \text{Otherwise}
\end{cases}
\]

\[
\text{\texttt{alloc}_E((M, N, F)) = (M', N, F), i} \text{ if } \text{ty} = \text{in} \land \text{new}(M, \text{BSize}(\text{ty})) = M', i
\]

\[
\text{\texttt{free}_E(((M, \text{Used}), N, F), i)} = \begin{cases} 
\text{Memfree}((M, \text{Used}), k), N, F)) & \text{if } k = (i[32:64]) \in \text{Used} \\
\langle i[0:32]\rangle = 0 & \text{otherwise}
\end{cases}
\]

\[
\text{\texttt{load}_E(((M, \text{Used}), N, F), i)} = \begin{cases} 
\{((M, \text{Used}), N, F), \text{read}((M, \text{Used}), k, o, m)\} & \text{if } k = (i[32:64]) \in \text{Used} \\
\langle i[0:32]\rangle \in \text{Used} & \text{otherwise}
\end{cases}
\]

\[
\text{\texttt{store}_E(((M, \text{Used}), N, F), v, p)} = \begin{cases} 
\{\langle \text{write}((M, \text{Used}), k, o, v, m), N, F)\} & \text{if } k = (p[32:64]) \in \text{Used} \\
\langle p[0:32]\rangle \in \text{Used} & \text{otherwise}
\end{cases}
\]

Figure 9: Operations for the explicit semantics.
The operations for modifying the explicit context is provided in Figure 9 and Figure 10. The rules are derived from the informal description provided at [12]. For the comparison operators, we give the definition of \( >_E \) and \( >_s \) below, and note that the remaining comparison operators are easily generalised from these. In the rules we let \( tt \in \text{dom}(E) \) and require \( tt \neq ff \).

\[
>_E(s, r_1, r_2) = \begin{cases} 
(s, tt, \top) & \text{if } \langle r_1 \rangle > \langle r_2 \rangle, \\
(s, tt, \bot) & \text{otherwise}
\end{cases}
\]

\[
>_s(s, r_1, r_2) = \begin{cases} 
(s, tt, \top) & \text{if } \langle r_1 \rangle > \langle r_2 \rangle, \\
(s, ff, \bot) & \text{otherwise}
\end{cases}
\]

**Remark 5.** Instantiating a model with the explicit context as described so far result in a possibly infinite state space. As a result, an exhaustive enumeration of all possible states may not terminate.

### 3.2 Symbolic Representation

We have already mentioned that an explicit representation of values in a program will explode (even without concurrency) in the presence of non-deterministic values. As an example of this, consider **LLVM-Listing 4** which can call the function `@@error` if and only if \( \%2 \) is set to 5. It is easy for easy for humans to realise that `@@error` can be called, but a computer with an explicit representation has to enumerate all \( 32^2 - 1 \) possible values of \( \%2 \).

For combatting this, LODIN provides a symbolic context representation. Instead of representing values explicitly, the symbolic context gathers all operations performed during exploration into one large logical formula - known as the path formula - that can since be passed to an SMT-solver. The SMT-solver can then determine if the formula is satisfiable and thus if the explored path is feasible.

#### 3.2.1 Satisfiability Modulo Theories

An SMT-instance is principally a first order logic formula where some predicates and functions have special interpretations. These special interpretations are encapsulated into what is called theories. An SMT-instance of the theory \( T \) can be determined to be satisfiable or not satisfiable by SMT-solver supporting the \( T \). We will not invest too much time here in talking about how SMT-solvers work, but will rather informally discuss the theories we need.

**Theory of Bitvectors** In the theory of bitvectors, variables are given a bitvector type \( i^n \). The operations that can be performed between bitvectors are

- the classic bitwise operations, i.e. `and`, `or`, `neg`, `xor`, `lshl`, `lshr` and `ashr`
- arithmetic operations (modulo \( 2^n \)), i.e. `add`, `sub`, `divu`, `divs`, `mul`, `remu`, `rems` - as in the LLVM discussion we need both signed and unsigned versions of some operations (indexed by \( u \) and \( s \))
- comparisons e.g. `=`, `\leq`,
- boolean operations e.g. `(\land, \lor, \neg)`
- concatenation of bitvectors `\circ`,

\[\text{Note we reuse the type name from LLVM}\]
\begin{align*}
\text{PtrAdd}_E(b_1, k) &= b_2 \quad \text{where } \text{block}(b_2) = \text{block}(b_1) \quad \text{and } \text{offset}(b_2) = \text{offset}(b_1) + k \\
\text{trunc}(b_1 / b_2) &= \begin{cases} 
\left\lfloor \frac{b_1}{b_2} \right\rfloor, & \text{if } b_2 \neq 0 \\
\left\lfloor \frac{b_1}{b_2} \right\rfloor, & \text{otherwise} 
\end{cases} \\
\text{lsr}(b_2) &= b_2 \quad \text{otherwise} \\
\text{asr}(b_2) &= b_2 \quad \text{otherwise} \\
\text{xor}(b_1, b_2) &= \{ (b_1 \text{ xor } b_2) \}
\end{align*}

Figure 10: Operation for the explicit semantics. Throughout these rules we assume that \( \text{dom}_E(\tau_E) = \mathbb{B}^m \), for some \( m \). In the rules we use \text{trunc} to denote a rounding operation towards zero.
• extraction of subvectors i.e. if $v$ is a bitvector then be$[0 \ldots n]$ extract a bitvector with bits 0 to $n-1$.

**Remark 6.** We reuse the operatorions from our discussin of bitvectors in subsection 3.1 and require that the SMT-solver implements the semantics of the operations as described there. Likewise we write constant bitvectors using the notation from subsection 3.1.

**Theory of Arrays** In this theory an array is a mapping between elements. Elements from an array can be read using a `select` function, and an element stored in an array using a `store` function. We introduce the array type $\{ i n \} \rightarrow \{ i m \}$ mapping elements from $i n$ to $i m$. If $v : \{ i n \} \rightarrow \{ i m \}$, $v_1 : i n$ and $v_2 : i m$ then we write `store$(v, v_1, v_2)$` to create a new array that is equal to $v$ with the only difference that the value of $v_1$ now maps to the value of $v_2$. We also write $v_2 = \text{select}(v, v_1)$ to set $v_2$ equal to the value kept at position $v_1$.

In the following we use $\mathcal{V}$ to denote an infinite set of SMT variables. We also use the restricted sets $\mathcal{V}^\text{ty} = \{ v \in \mathcal{V} \mid v : \text{ty} \}$. Similarly we refer by $\mathcal{W}$ to all SMT expressions over $\mathcal{V}$ and $\mathcal{W}^\text{ty}$ to all SMT expressions with type $\text{ty}$.

**The Symbolic Context**

The symbolic context in LODIN maps its register variables to SMT variables and uses a so called path formula to capture all constraints (assignments and comparisons) encountered during a program execution. Memory is represented using a SMT array and a SMT variable points to first place in memory that is free for allocation.

**Definition 8** (Symbolic Context). The symbolic context for the symbolic semantics is the tuple $\mathcal{S} = ( \mathcal{S}_I, s_{\text{init}}, \text{dom}_\mathcal{S}, \mathcal{N}, \mathcal{f}_\mathcal{S} )$ where

- $\mathcal{S}_I$ are tuples $(v_M, v_f, N, F, \psi, \text{used})$ where
  - $v_M : \{ i 64 \} \rightarrow \{ i 8 \}$ is an array representing the memory state of the program,
  - $v_f : i 64$ is a pointer into memory
- $N \subseteq \mathbb{N}$ is a set of used register variables,
- $F : \mathbb{N} \rightarrow \mathcal{V} \cup \{ \bot \}$
- $\psi$ is an SMT formula - the path formula - encoding the constraints that an explored path has to satisfy, and
- $\text{used} \subseteq \mathcal{V}$ is a set of used SMT variables.

- $s_{\text{init}} = (\mathcal{M}, 0, F, \mathcal{f}_\mathcal{S} = \mathcal{f}_\mathcal{S}, \emptyset)$ where for all $n \in \mathbb{N}$, $F(n) = \bot$.
- $\text{dom}_\mathcal{S}(i_1) = \mathcal{W}^{\text{ty}1}$, $\text{dom}_\mathcal{S}(\text{ty}1) = \mathcal{W}^{164}$, and $\text{dom}_\mathcal{S}((\text{ty}1, \ldots \text{ty}1n)) = \mathcal{W}^{8 \cdot \text{BSize}((\text{ty}1, \ldots \text{ty}1n))}$
- $\mathcal{f}_\mathcal{S} = \emptyset^8$.

The arithmetic instructions (e.g. $+^S(v, v_1, v_2)$) that we need to implement for the context is straightforward to represent. All we need to do is to create an SMT expressions corresponding to the operation. Below we give a generalised definition of the rule:

$$\sim^S_\mathcal{S}((v_M, v_f, F, \psi, \text{used}), v_1, v_2) = v_1 \text{SMTOp} v_2$$

For the mapping between $\sim^S_\mathcal{S}$ and SMTOp we refer to Table 3.

The comparison operators are very similar to the binary operator, and below we provide an example for the $>^S_\mathcal{S}(s, v_1, v_2)$ function where $s = (v_M, v_f, F, \psi, \text{used})$

$$>^S_\mathcal{S}(s, v_1, v_2) = (v_M, v_f, N, F, \psi \land (v_1 >_u v_2), \text{used}, v_1 >_u v_2, T)$$

For the remainder of the operations we refer the reader to Figure 11 and Figure 12.

**Example 1.** We briefly return to the module $(\mathcal{M})$ in LLVM Listing 3 and consider how we can use the symbolic representation of LODIN to determine if the function `@error` can be called. We simply instantiate the symbolic transition system $\mathcal{L}_\mathcal{M} = (\mathcal{N}, n_0, \rightarrow^S)$ and generate symbolic states from $n_0$ until we reach a state $n_f = (s_1 : s_2 \cdots : \epsilon, s_S, \mathcal{M})$.
feasibility, we invoke a SMT-solver and checks if it is feasible. To ensure the register multiple times thus it is only applicable for programs without any loops in their control-flow graph. It is usual convenient to merge symbolic context states into one state. Merging Symbolic States It is usual convenient to merge symbolic context states into one state. This allows exploring several computational paths simultaneously and helps combat path-explosion.

Remark 7. The symbolic context assigns each register of an LLVM program a single SMT-variable, and gathers constraints over these SMT-variables in a path formula. Assignments to LLVM registers is captured by equality between the SMT-variable and SMT-expressions. A result of this is that the symbolic context does not support assigning to the same register multiple times thus it is only applicable for programs without any loops in their control-flow graph.

\[ mReg_S((v_M, v_f, N, F, \psi, used), i) = (v_M, v_f, N \cup \{i\}, F, \psi, used), i \] where \( i = \min(N \setminus N) \)

\[ \text{Eval}^v_S((v_M, v_f, N, F, \psi, used), i) = \begin{cases} F(i) & \text{if } F(i) \in \text{dom}(v) \\ \text{Error} & \text{Otherwise} \end{cases} \]

\[ \text{Set}^v_S((v_M, v_f, N, F, \psi, used), l, v) = \begin{cases} (v_M, v_f, N, F, \psi \land (F(l) = v), used) & \text{if } l \in N \land v \in \text{dom}(v) \\ \text{Error} & \text{Otherwise} \end{cases} \]

\[ \text{alloc}^v_S((v_M, v_f, N, F, \psi, used)) = \begin{cases} (v_M, v_f, N, F, \psi \land (v_f = v_f \text{ add } n), used \setminus \{v_f\}), v_f & \text{if } v = \text{ in} \\ (v_M, v_f, N, F, \psi \land (v_f = v_f \text{ add } 64), used \setminus \{v_f\}), v_f & \text{if } v = \text{ in*} \end{cases} \]

\[ \text{PtrAdd}_S(v_b, k) = v_b \text{ add } k \]

\[ \text{load}^n_S((v_M, v_f, N, F, \psi, used), i) = \text{SymbLoad}^n(v_M, F(i)) \]

\[ \text{SymbLoad}^n(v_M, v_a) = \begin{cases} \text{select}(v_M, v_a) & \text{if } i = \text{ in} \\ \text{select}(v_M, v_a) \circ \text{SymbLoad}^n\text{ } S(v_M, v_a \text{ add } 1) & \text{otherwise} \end{cases} \]

\[ \text{store}^v_S((v_M, v_f, N, F, \psi, used), v_0, v_p) = (v_M', v_f, N, F, \psi \land (v_M' = \text{SymbStore}^v(v_M, v_0, v_p)), used, v_0, v_p) \]

\[ \text{SymbStore}^n((v_M, v_0, v_p)) = \begin{cases} \text{store}(v_M, v_p, v_0) & \text{if } i = \text{ in} \\ \text{SymbStore}^n\text{ } S(\text{store}(v_M, v_p, v_0[0 \ldots 8]), v_p \text{ add } 1, v_0[8 \ldots n]) & \text{otherwise} \end{cases} \]

Figure 11: Evaluation and setting registers in symbolic context.

Figure 12: Store and Load operations in the symbolic context.
Table 3: Mapping between semantic operators and SMT operators

| SMT operator | C operator | SMT operator | C operator |
|--------------|------------|--------------|------------|
| +<sup>ty</sup> | add | ÷<sup>ty</sup> | lshr |
| -<sup>ty</sup> | sub | ×<sup>ty</sup> | ashr |
| ×<sup>ty</sup> | mul | ▸<sup>ty</sup> | diva |
| ÷<sup>ty</sup> | div | >><sup>ty</sup> | rema |
| ≪<sup>ty</sup> | diva | /<sup>ty</sup> | rema |
| ≫<sup>ty</sup> | lshl | ⊕<sup>ty</sup> | xor |

Problem - which is a big problem for symbolic execution engines such as KLEE.

For merging context-states

\[ s_S = (v_M, v_f, N, F, \psi, used) \]

and

\[ s'_S = (v'_M, v'_f, N', F', \psi', used') \]

where for all \( n \in N \cap N' \) it is the case that \( F(n) = F'(n) \) we introduce the function \( \text{merge} : S_S \times S_S \rightarrow S_S \) defined as

\[ \text{merge}(s_S, s'_S) = (v''_M, v''_f, N \cup N', F'', (\psi \lor \psi') \land \psi'' \land \psi''', used \cup used' \cup \{v''_M, v''_f, v_P\}) \]

where

- \( F''(n) = \begin{cases} F(n) & \text{if } n \in N \\ F'(n) & \text{if } n \in N' \end{cases} \)
- \( v''_M, v''_f, v_P \not\in used \cup used' \)
- \( \psi'' \overset{\text{def}}{=} (v''_M = \text{ite}(v_P, v_M, v_M')) \)
- \( \psi''' \overset{\text{def}}{=} (v''_f = \text{ite}(v_P, v_f, v_f')) \)

Here \( v_P \) with type \( \text{is} \) is a fresh SMT variable and \( \text{ite}(v_P, v, v') \) evaluates to \( v' \) if \( v_P = \text{ff} \) and to \( v \) otherwise.

4 Explicit Reachability Checking

Model Checking \cite{Henzinger:00,Clarke:99} is a technique widely used in academia for validating that a formal model of a program behaves correctly - according to a specification given by a logical formula. A basic specification is a reachability specification, where we are interested in finding a state where a given proposition is true. This is the main focus in LODIN, and thus we will limit our discussion to this setting.

4.1 General Reachability Checking

At the core of any reachability checking algorithm is a transition system to search and a set of atomic propositions. In the case of LODIN, the state space we search is \( L^*_M = (N, n^0, \rightarrow^*) \). Atomic propositions of a program are elements that may be true or false in a state (for instance whether \( x = 5 \) or if a state has a DataRace \cite{Lodin:08}). An interpretation (over states \( N \)) of an atomic proposition, \( p \), is a function \( P_p : N \rightarrow \{\text{tt, ff}\} \), where \( \text{tt} \) indicates \( p \) is true and \( \text{ff} \) indicates it is false. Atomic propositions may be combined with the classical boolean operators \( \land, \lor \) and \( \neg \). The interpretation of these combined propositions are defined recursively below as,

- \( P_{\psi_1 \land \psi_2}(n) = P_{\psi_1}(n) \land P_{\psi_2}(n) \)
- \( P_{\psi_1 \lor \psi_2}(n) = P_{\psi_1}(n) \lor P_{\psi_2}(n) \)
- \( P_{\neg \psi}(n) = \neg P_{\psi}(n) \)

where \( \psi_1, \psi_2 \) are combined proposition themselves. Checking reachability for the proposition \( \psi \) is now to check whether we, from the initial state, can reach a state \( n \) where \( P_{\psi}(n) = \text{tt} \). The classical approach for such a search is the fix-point algorithm in \( \text{Algorithm 4} \)

For a finite state system \( \text{Algorithm 4} \) obviously terminate, as \( \text{Passed} \) eventually contains the entire reachable state space - and thus no further states can be put into \( \text{Waiting} \) and therefore \( \text{Waiting} \) will eventually become \( \emptyset \). Equally straightforward is it
Data: Property : φ
Data: Initial state: n
Result: ⊤ or ⊥
Passed := 0;
Waiting := {n};
while Waiting ≠ ∅ do
    Let nc ∈ Waiting;
    Waiting := Waiting \ {nc};
    if Pφ(nc) then
        | return ⊤
    end
    Waiting := Waiting ∪ {n | ∃i,Inst.t. nc \ Inst $\xrightarrow{i}$ n};
end
return ⊥

Algorithm 1: The classic reachability algorithm. States that have not been explored (but found) are kept in the set Waiting, and states that has already been processed are kept in Passed.

to realise that [Algorithm 4] produces correct results. [Algorithm 4] is non-deterministic in selecting an element from Waiting and in generating successors of the currently considered state. The latter can easily be determinised by generating states in a fixed order, while the prior can be determinised in different ways: the two usual ways is to keep the elements of Waiting in a stack or on a queue and let the order induced by these define the search order.

Remark 8. As mentioned earlier, the explicit state space may in fact be infinite thus [Algorithm 4] may not terminate. In LODin we have added options for terminating any verification after a user defined time or after using a user defined size of memory.

LLVM Propositions LODin has support for propositions specifying classic programming errors (division by zero, data race, out of bounds errors, etc). Furthermore, it is possible to do comparisons between registers and check if a specific function is called by a process. The use case for the latter is, that the user can modify the verified program to call an error function and check if that function is called. In LODins propositional language, registers and numbers are typed to signed bitvectors or unsigned bitvectors with the suffixes uin and sin where n ∈ {8, 16, 32, 64}. For any production rule R in Figure 13 we write Ψ(R) for the language generated by that rule. An expression like %0,F,%tmp3; uin32 == 3; uin32, means take register %tmp3 in the function %0 of the 0th process. Interpret it as an unsigned 32bit integer, and compare it for equality with 3 also interpreted as a unsigned 32bit integer. For comparisons to make sense, the two expressions being compared must, naturally, have the same type.

For evaluating the value of a register in a state (n = (s_0, s_1, ..., s_n, s,M)), we define

\[ A_{\text{0k,F,%tmp:uin}(n)} = \]
\[
\left\{ \begin{array}{ll}
\langle \text{Eval}^{\text{ty}}_{\text{cm}}(s, r) \rangle & \text{if } s_k = ((\%\text{prev,cur,pc,π,Free}), \text{SL}) \\
\tilde{0}^{16} & \text{otherwise}
\end{array} \right.
\]

Figure 13: Grammar generating verification queries of LODin.
Notice that we assign the default value of zero to registers that are not present in the current activation record. If the register is present in the activation record, we just extract the bitvector and apply the interpretation function for signed/unsigned numbers.

For evaluating numbers (e.g., 3;ui32) we write \( A_{3;ui32}(n) \) and it has the obvious implementation. Given these notations, we can define how propositions are evaluated within LODIN in Figure 14. A short discussion may be in order about the evaluations in Figure 14.

- Division by zero (\( \text{DivZero} \)) are determined in the obvious manner, where we simply check if any process executes any instruction involving a division \( \div \) and check if the second operand is zero.

- Buffer overflows (\( \text{OverFlows} \)) are likewise easily checked by checking if any process accesses memory, and for each of those that do access memory we check if their read/write to memory exceeds the length of the buffer they are writing/reading into/from.

- The instruction for checking whether a specific process number \( i \) can call a function \( \text{func} ([i, \text{func}]) \), we first check if process \( i \) performs a \text{call} instruction and if so, if the functions being called matches \( \text{func} \).

- The most difficult proposition to check is without a doubt \( \text{DataRace} \). For evaluating this instruction, we iterate over all processes and finds pairs of read/write and write/write to the same pointer base. Afterwards we check if their \( \text{offset} + \text{length} \) overlaps.

**Example 2.** As a short example of using LODIN for reachability checking let us consider LLVM-Listing \( I \) and consider we are interested in whether \( \%x \) and \( \%z \) can ever be equal. Notice that since all \( \text{phi} \) instructions should be executed atomically in the beginning of a block, this should never be possible - thus checking this with LODIN actually

\[
\mathcal{P}_{\text{DivZero}}(n) =
\begin{cases}
\top & \exists s_i = ((F, \text{prev}, \text{pc}, \pi, \text{Free}), \text{SL}) \\
& F = ([F, R, P, BL, BBS, Bm, ret]) \\
& \text{Bm}(\text{cur})(\text{pc}) = \text{store ty} \%\text{inp1}, ty + \%\text{inp2} \\
& r = \pi(\%\text{inp2}) \\
& (\text{Eval}(\pi, s_i, r)) = 0 \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\mathcal{P}_{\text{OverFlows}}(n) =
\begin{cases}
\top & \exists s_i = ((F, \text{prev}, \text{pc}, \pi, \text{Free}), \text{SL}) \\
& F = ([F, R, P, BL, BBS, Bm, ret]) \\
& \text{Bm}(\text{cur})(\text{pc}) = \text{store ty} \%\text{inp1}, ty + \%\text{inp2} \\
& r = \pi(\%\text{inp2}) \\
& \pi(\%\text{inp1}) \in \mathbb{B}^i \\
& (\text{offset}(r) + \text{len} = 0) \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\mathcal{P}_{\text{DataRace}}(n) =
\begin{cases}
\top & \exists s_i = ((F, \text{prev}, \text{cur}, \text{pc}, \pi, \text{Free}), \text{SL}) \\
& F_1 = ([F, R, P, BL, BBS, Bm, ret]) \\
& \text{Bm}(\text{cur})(\text{pc}) = \text{load ty}, ty + \text{ptr}_1 \\
& p_i = \text{Eval}(\pi, s_i, (\text{ptr})) \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\mathcal{P}_{\text{OverFlows}}(n) =
\begin{cases}
\top & \exists s_i = ((F, \text{prev}, \text{cur}, \text{pc}, \pi, \text{Free}), \text{SL}) \\
& F_1 = ([F, R, P, BL, BBS, Bm, ret]) \\
& \text{Bm}(\text{cur})(\text{pc}) = \text{store ty} \%\text{val}, ty + \text{ptr}_1 \\
& p_i = \text{Eval}(\pi, s_i, (\text{ptr})) \\
& \text{block}(p_i) = \text{block}(p_i) \\
& \text{offset}(p_i) \ldots \text{offset}(p_i) + \text{BSize}(\text{ty}), \ldots \text{offset}(p_i) + \text{BSize}(\text{ty}) \neq 0 \\
\bot & \text{otherwise}
\end{cases}
\]

Figure 14: Evaluation of propositions in LODIN where \( A_1, A_2 \in \Psi(\text{Register}) \cup \Psi(\text{Number}), \in \Psi(\text{OP}) \) and \( s = ([M, \text{Used}], N, F) \). For \( \text{OverFlows} \) we have only shown the rule for overflows at writes, but naturally there is an equivalent rule for reads.
Lodin example.ll example2.q
Lodin 0.3 (Jul 8 2019)
Revision: 0.2-802-ga42644cf
Importance Ratio: double
LLVM: 8.0.0

LLVM module modifications:
Remove Unused instructions
Warning: No entry-point specified. Assuming main.
Random seed: 1562587068
System: NaiveGraph-explicit
Platform: PThread
Storage: SharedMem Storage
Successor: Standard
Prob-Successor: Standard
Passed-Waiting: Standard
SMT-Backend: Boolector 3.0.0
Verifying: E<>((0).main.b ==) ;
Warning: Casting register main.b to integer type UI8 - can’t guarantee LLVM uses this register as such

Lodin Output 1: Output from Lodin.

checks if Lodin implements the phi instructions behaviour correctly.

In Lodin we can check the property by asking the query E <= (0).main.b++;)

Unfortunately Lodin reports that this is indeed possible even though it should not be. There is a logical explanation for this: both registers are initialised by Lodin to 0 thus in the initial state they are equal. For this reason, it is more reasonable to use the %b register for our check thus we check the query E <= (0).main.%b; ui32 == (0).main.%b; ui32).

4.2 State Space Reductions

A well-known problem for explicit-state reachability checking of parallel systems is the notorious state space explosion problem i.e. that the combined state space increases exponentially when each process of the system increases linearly. This is a huge problem when considering high-level programs and exacerbated when using LLVM as input, because LLVM programs has more instructions per process. For making explicit-state reachability checking possible we thus need ways of limiting the size of the state space. A first realisation to reduce the state space is, that processes can only influence each others behaviour at predefined points, namely when accessing memory. Due to our specification language allowing to query whether functions can be called, we also consider call instructions to affect the external behaviour of a process. We say that an instruction Inst is internal if Inst if it is a load/store or call instruction. We denote the set of all internal instructions by Internal(R). In the following we describe the two state space reductions that are implemented inside Lodin. They both define a new transition relation, that can directly replace \( \rightarrow \).

4.2.1 State Space Reduction 1

Our first state space reduction is based on the idea, that when a process performs a transition step it will perform all following transitions that executes internal instructions. More formally, we replace the transitions relation \( \rightarrow \) with \( \rightarrow^{\mathcal{E}} \) where \( \rightarrow^{\mathcal{E}} \) is defined according to the rule

\[
\begin{align*}
\ell & \rightarrow^{\mathcal{E}} n_k \\
\ell & = \ell_{k-1} \rightarrow_{i_k} n_k \\
\ell & = \ell_{0} \rightarrow_{i_0}^{\mathcal{E}} n_0, \quad \forall k > 2, \forall \ell_k \in \text{Internal}(R),
\end{align*}
\]

Notice, that there is no lower length in then size of the sequence \( \ell_1, \ldots, \ell_n \). To achieve the largest reduction, Lodin always uses the longest possible sequence.

In this state space reduction, all processes that perform internal instructions execute simultaneously while all other processes execute independently. The transition relation \( \rightarrow^{\mathcal{E}} \) is defined by two rules

\[
\begin{align*}
\ell & \rightarrow^{\mathcal{E}} n_k \\
\ell & = \ell_{k-1} \rightarrow_{i_k} n_k \\
\ell & = \ell_{0} \rightarrow_{i_0}^{\mathcal{E}} n_0, \quad \forall k > 2, \forall \ell_k \in \text{Internal}(R),
\end{align*}
\]

21
Figure 15: LODIN state space reductions. Transitions going left originate from one process while transitions going to the right correspond to another. Dashed arrows indicate visible actions.

Example 3. As an example of the state space reductions that $\mathcal{E}$ and $\mathcal{G}$ respectively do, consider the C-program in Figure 16 that executes petersons mutual exclusion algorithm. To use this program with LODIN, it must first be compiled to an .ll-file using clang\(^7\). After this step we can inspect the state space reductions achieved by asking LODIN the query EnumStates on the resulting .ll-file with the

\[ n \xrightarrow{\text{Inst}} n', \quad \text{Inst} \notin \text{Internal}(\mathcal{I}) \]

Table 4: State Space Reductions.

| State Generator | States | DataRace States |
|-----------------|--------|-----------------|
| $\mathcal{E}$   | 6573   | 16              |
| $\mathcal{G}$   | 4111   | 16              |
| $\mathcal{H}$   | 3057   | 8               |

Although the above state space reductions can reduce the state space due to interleavings dramatically, they cannot reduce the number of states caused by non-deterministic input. A program with just one non-deterministic 32bit value will end up having over $2^{32}$ states.

5 Simulation-Based Model Checking

In the preceding section we saw how LODIN can be used to perform an exhaustive state space search under an explicit context. We also realised, that the state space explosion problem poses a problem for any exhaustive search and showed how LODIN can reduce this explosion through state space reductions. The state space reductions also have their limits thus we need other strategies for handling this explosion. LODIN proposes to use a simulation-based technique, where random (step-bounded) traces are drawn from the program and inspected for satisfaction of the property at hand. At the heart of any simulation-based technique is an underlying simulation distribution. The simulation distribution may stem from actual knowledge of how the system behaves, in which case simulations can be used to calculate actual probabilities of the system satisfying the property using statistical methods - hence the name statistical model checking\(^{21}\). In case the simulation distribution is “arbi-

\(^7\)clang -S -c -emit-llvm file.c


```c
#include <stdio.h>

int flags[2] = {0,0};
int turn = 0;

void crit () {}

typedef struct {
  int *mflag;
  int *oflag;
  int* turn;
} Options;

void* petersons1 () {
  Options opt;
  opt.mflag = &flags[0];
  opt.oflag = &flags[1];
  opt.turn = &turn;
  *(opt.mflag) = 1;
  *(opt.turn) = 1;
  while (*(opt.oflag) && *(opt.turn) == 1)
    { // busy wait
        crit () ;
    } // critical section
  crit ();
  return 0;
}

void* petersons2 () {
  Options opt;
  opt.mflag = &flags[1];
  opt.oflag = &flags[0];
  opt.turn = &turn;
  *(opt.mflag) = 1;
  *(opt.turn) = 0;
  while (*(opt.oflag) && *(opt.turn) == 0)
    { // busy wait
        crit () ;
    } // end of critical section
  *(opt.mflag) = 0;
  return 0;
}

Figure 16: Petersons Mutual Exclusion Protocol
```

...trary”, then estimated probabilities are meaningless for the system itself, but serves as a way to predict how likely it is that a continued search will find the property searched for. In this case the technique is called Monte Carlo Model Checking.

### 5.1 Simulation Distribution

In LODIN each state $n$ of the state space $L^E_M = (N, n^0, E)$ is assigned a probability distribution $\gamma_n : N \to [0,1]$. The probability distribution assigns a probability to which process should perform an action. The function $\gamma_n$ should obviously only assign a probability mass to a process if that process can perform a transition thus we require that $\gamma_n(i) \neq 0 \implies n \xrightarrow{i} n'$, for some $n'$. Having selected who should perform an action, we also need a probability function for the result of that choice $i$. We do this by assuming a $\delta_{n,i} : N \to [0,1]$, where $N$ is the set of all states. The requirement to this function is, that it should only assign probabilities to states that can be reached by the $i$th process performing a transition from $n$, i.e. $\delta_{n,i}(n') \neq 0 \implies n \xrightarrow{i} n'$ for some instruction $i$.

Given these two probability mass functions, the probability that a system generates the finite transition sequence $\omega = n_0 \xrightarrow{i_1} n_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} n_n$, where $n_0$ is the initial state, is given by $P(\omega) = \prod_{k=1}^n \gamma_n(i_k) \cdot \delta_{n_{k-1},i_k}(n_k)$. For a transitions sequence $\omega = n_0 \xrightarrow{i_1} n_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} n_n$, we let $|\omega| = n$ be its length and $\omega[i] = n_i$. We also let $\Omega_{m,M}$ be the set of all transition sequences $\omega$ with $|\omega| = m$ of LLVM module $M$. Let $p$ be a proposition, and $\omega \in \Omega_{m,M}$ then we define the indicator function

$$\mathbb{I}_p(\omega) = \begin{cases} 1 & \text{if } \exists i \text{ s.t. } p(\omega[i]) = \text{true} \\
0 & \text{otherwise} \end{cases}$$

that returns 1 if $\omega$ at some point satisfies $p$ and 0 otherwise. With this at our hand, we define the probability that an execution trace of a program $M$ satisfies a proposition $p$ within $m$ steps as
Example 4. Before dwelling upon how to using simulation to do verification, let us briefly consider what kind of coverage of the state space we can expect with by doing simulations. To this end, we have implemented the query $\text{EnumStatesSMC} \leq 5000$ $n$. This query simply generates $n$ traces each of length $l$ and keeps tracks of how many different states it has visited in total. We show the results of running this query on Figure 16 in Table 5. Recall from previously, that the total number of states is 6573.

\begin{algorithm}
\begin{center}
\textbf{Algorithm 2:} Generating random traces in LODIN
\begin{algorithmic}
  \State \textbf{Data:} Initial state: $n_0$
  \State \textbf{Data:} Length: $n$
  \State $\omega = n_0$;
  \For {$i \in \{1, \ldots , n\}$} \Do
    \State $k \sim \delta_{n_{i-1}}$;
    \State $n_i \sim \gamma_{n_{i-1}, k(n_{i-1})}$;
    \State $\omega = \omega n_i$;
  \EndFor
  \State \Return $\omega$
\end{algorithmic}
\end{center}
\end{algorithm}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
$n$ & States & DataRace States \\
\hline
1 & 77 & 1 \\
100 & 1840 & 4 \\
1000 & 3579 & 11 \\
10000 & 4714 & 14 \\
\hline
\end{tabular}
\caption{State encountered with SMC. The used query is $\text{EnumStatesSMC} \leq 5000$ $n$.}
\end{table}

$\Pr_{M,m}(p) = \sum_{\omega \in \Omega_{m,M}} I_p(\omega) \cdot P(\omega)$

As the probability only depends on the state, we usually project out transitions and only generate the states. An algorithms for generating a sequence of states from $n_0$ according to the probability distribution can be seen in [Algorithm 2]. In the algorithm we use $k \sim P$ to mean that $k$ is distributed according to the probability mass function $P$.

Example 5. Let us consider the program in Figure 16 again and let us assess the probability that a data race is encountered. We can assess this with the query: $\Pr[\leq 5000]$ (<> DataRace). The 5000 in this query is the length of the runs. See LODIN-Output 2 for the output. From the output we can see that LODIN estimates the probability to

5.2 Statistical Model Checking

Statistical model checking tries answering two questions: 1. a quantitative “What is the probability $\theta$ of reaching $p$?” and 2. a qualitative “Is the probability of $\theta$ greater than $\theta_\epsilon$?” Both questions are answered by generating a number samples and using statistical techniques to infer the answer with a user specified confidence.

Quantitative Here we repeatedly generate runs and construct an interval $[\theta_l, \theta_u]$ for which we are confident that the probability $\theta$ is contained within. For the following we assume we are provided with $\epsilon$ being the wanted width of the interval and an $\alpha \in [0, 1]$ indicating the confidence $(1 - \alpha)$ we want in the interval.

Consider that we have generated a sequence of samples $\omega_1, \omega_2, \ldots$, and let $x_1, \ldots, x_m$ be random variables such that $x_i = I_p(\omega_i)$. Then each variable $x_i$ has a Bernoulli distribution with success probability $\theta_i$ and the sum $X_m = \sum_{i=1}^m x_i$ is binomially distributed. We construct a confidence interval using the exact confidence interval by Clopper and Pearson [10]: if we have $m$ samples then a Clopper-Pearson-interval with confidence $\alpha$ is given as the intersection $S_{\leq} \cap S_{\geq}$ where

$$S_{\leq} = \{ \psi \mid B_{m,\psi}(X_m) > \alpha / 2 \}$$
$$S_{\geq} = \{ \psi \mid 1 - B_{m,\psi}(X_m) > \alpha / 2 \}$$

and $B_{m,\psi}$ is the cumulative distribution function for a binomial distribution with $m$ samples and success parameter $\psi$. Notice that we are not in control of the resulting width of this interval - more samples will however shrink the width $\epsilon$ and thus we simply iteratively produce samples until we get the desired width.

Example 6. The query $\text{EnumStatesSMC} \leq 5000$ $n$. This query simply generates $n$ traces each of length $l$ and keeps tracks of how many different states it has visited in total. We show the results of running this query on Figure 16 in Table 5. Recall from previously, that the total number of states is 6573.
lie in the interval $[0.29, 0.30]$. The last part provides a histogram over the length of the satisfying runs. LODIN runs by default with $\alpha = 0.05$ and $\delta = 0.01$. These parameters can be tweaked by suffixing the query with \{$\text{Alpha} = \text{Float}$, $\text{Epsilon} = \text{Float}$\} where Float are numbers in $[0, 1]$. Running the query $\Pr[\leq 5000] \ (< > \text{DataRace})$ \{$\text{Alpha} = 0.01$, $\text{Epsilon} = 0.05$\} for instance gives the result $[0.27, 0.32]$.

**Qualitative.** Checking whether the probability $\Pr_{M,m}(p)$ exceeds a threshold $\theta$ can be answered by doing hypothesis testing. We test the hypothesis $H_0 : \Pr_{M,m}(p) \geq \theta$ against $H_1 : \Pr_{M,m}(p) < \theta$. In advance, we want to define two parameters, $\alpha$ (significance level) and $\beta$ (power level), that signifies how willing we are to reject a true hypothesis and how willing we are to accept a false hypothesis. In practice we want a test for which the probability of rejecting $H_0$ while $H_0$ is true is less than $\alpha$; while the probability of accepting $H_0$ while $H_1$ is true is less than $\beta$. Realising that acheiving both of these requirements is close to impossible in general \cite{22} we introduce an indifference region of width $2 \cdot \delta$ around $\theta$ and test instead the hypothesis $H'_0 : \Pr_{M}(p) > \theta + \delta$ against $H'_1 : \Pr_{M}(p) < \theta - \delta$. Wald \cite{20} developed a sequential hypothesis testing algorithm, see \cite{Algorithm 3} for exactly this case; the idea is to iteratively generate runs and based on these calculate a value $r$ - eventually this value will cross $\log(\beta/(1-\alpha))$ or $\log((1-\beta)/\alpha)$ and $H'_0$ is either rejected or accepted.

6 Bounded Model Checking

In previous sections we described the symbolic representation of states used within LODIN, and we saw in an example how this representation could be used to explore many values registers simultaneously. We however did not give a structured way of using this symbolic representation in a verification framework. We make up for that in this section.
Data: Initial State: \( s \)
Data: Property: \( \Pr_{M,m}(p) \geq \theta \)
Data: Indifference Region: \( 2 \cdot \delta \)
Data: Significance Level: \( \alpha \)
Data: Power Level: \( \beta \)

Result: \( \top \) or \( \bot \)

\[ p_0 = \theta + \delta; \]
\[ p_1 = \theta - \delta; \]
\[ r = 0; \]

while \( d > \delta \) do
    \[ \omega = \text{generateRun}(s, m); \]
    \[ x = \|P(\omega); \]
    \[ r = r + x \cdot \log(P_1/p_0) + (1 - x) \cdot \log((1 - p_1)/(1 - p_0)); \]
    if \( r \leq \log(\beta/(1 - \alpha)) \) then
        \[ \text{return } \top \]
    end
    if \( r \geq \log((1 - \beta)/\alpha) \) then
        \[ \text{return } \bot \]
    end
end

Algorithm 3: Testing whether probability is larger than \( \theta \)

6.1 Symbolic Analysis of Loop-free program

In this section we show how LODIN uses its symbolic representation to analyse single-threaded programs without loops. For now, we will also restrict our attention to verify if a given function can be called at any time e.g. propositions as \([0, @\text{error}]\). Before going into details about the algorithm, we will setup up some convenient notations, to make the algorithm more readable.

A key concept we will need in the algorithm for analysing loop-free programs is converging basic blocks and diverging basic blocks: for a LLVM function \((@x, R, P, BL, BBs, Em, ret)\), we say that a block \( B \in BBs \) diverges control flow if \( B[B] \equiv (\text{true}, \bar{c}, \text{label}@true, \text{label}@false)\). For a block \( B \in BBs \) where \( Em(con) = B \) for some \( con \), we define the set of all blocks jumps to \( B \) as

Data: Property: \( \phi \)
Data: Initial state: \( n \)
Result: \( \top \) or \( \bot \)

Mergees := Mergees;
Waiting := \{s\};
while Waiting \( \neq \emptyset \) do

Let \( n_c \in \text{Waiting}; \)
Waiting := Waiting \( \setminus \{n_c\} \);
if \( P[i.@\text{func}](n_c) \) then
    \[ \text{return } \top \]
end

foreach \( n_n \in \{n \mid \exists i, \text{Inst}s.t. n_c \xrightarrow{\text{Inst}} S_n\} \) do

if \( \neg \text{Mergeable}(n_n) \) then
    Waiting := Waiting \( \cup \{n_n\} \);
else
    Let \( n_n = ((F, prev, cur, pc, \pi, Free) : S, s_S) \);
    if \( \exists (cur, n_o, n) \in \text{Mergees} \) then
        if \( n - 1 = 0 \) then
            Waiting := Waiting \( \cup \{\text{merge}(n_o, n_n)\} \);
        else
            Mergees :=
            Mergees \( \setminus \{(cur, n_o, n)\} \cup \{(cur, \text{merge}(n_o, n_n), n - 1)\} \);
        end
    else
        Mergees :=
        Mergees \( \cup \{(cur, n_n, \text{In}(n_n) - 1)\} \);
    end
end

return \( \bot \)

Algorithm 4: The symbolic reachability algorithm.
Handling Loops  Any nontrivial program will have loops, and as such verificaion techniques must cope with loops. LODIN can verify programs with loops, but relies on syntactically unrolling the loops before verification. In case the loop unroll is complete, then the verification is complete - otherwise the verification is only sound.

7  Implementation Details

LODIN- available at [www.fillthis.later] - is build around the LLVM-bitcode and uses the LLVM-libraries for parsing the input-files, and performing some LLVM modifications during. LODIN does, however, not use the infrastructure of LLVM for performing analyses. Instead it builds its own internal representation of the loaded LLVMmodule and implements its own state space successor generator.

7.1  LLVM Modifications

At load time LODIN can perform a number of modifications of the LLVM program - some of the modifications are enabled by default, some forced enabled by other. In the following we briefly discuss the modifications.

Naming Instructions  LLVM-bitcode files do not necessarily contain names for the registers. At load time LODIN therefore give names to all non-named registers in the program. This simplifies internally when providing error messages.

Constant Removal  LLVM-bitcode instructions can have constant expressions which the interpreter of LODIN would have to evaluate at run time. We replace these constant expressions with LLVM instructions thus simplifying the subset of LLVM that our interpreter needs to understand.

Simplify CFG  This is a standard LLVM modification that attempts to simplify the control flow
graph. LODIN provides an option for running this simplification, but does not run it by default as it modifies the program drastically and thus specifications of the user is perhaps no longer “valid”. The modification can be enabled by the user or forced by other modifications.

Eliminate Dead Code As the names suggests, this modification removes code that statically can be determined to be unreachable. This is standard LLVM modification that has to be enabled by the user.

Constant Propagation This is a standard LLVM modification that forwards constants in the LLVM-code and thereby reduce the number of instructions in the LLVM-code.

Mem2Reg This modification tries to promote memory operations to register operations. This is useful as it makes operations easier for some of the modifications. The modification can be enabled by the user or forced by other modifications.

Loop Unrolling This is the only modification that requires a user specified input n. The modification unrolls all detected loops in the program at most n times. If it can be determined a loop will only execute m < n times, it is of course only unrolled m times. The unrolling is implemented inside LODIN but borrows the unrolling strategy from the LLVM library. The reason the loop unrolling does not use the default LLVM unrolling method is that LODIN needs more control of the unrolling than the interface offered. Enabling loop unrolling force-enables Mem2Reg and Simplify CFG. The main usage of Loop unrolling is to support the unrolling needed by bounded model checking.

7.2 Architecture

LODIN employs a layered architecture (see Figure 17) where high-level algorithms - as detailed in previous sections - can be implemented without knowledge of low-level considerations such as how the states are represented. The algorithms depends on state generators implementing the the state space reductions or the probabilistic semantics. The generators in turns depends on a joint interpreter-platform unit, that will interact with an interface to a state representation (how activation records are stored etc.). The state representation then depends on a context-memory unit which performs the operations requested by the interpreter. At the lowest level of the architecture is the storage unit which is responsible for storing and saving states (used by the implementation of Passed/Waiting sets in Algorithm 4).

SMT Solvers LODIN uses external SMT-solvers for solving the contraints gathered by the symbolis context implementation. The constraints are represented in a solver-independent format and only at the last minute converted to SMT-solver specifics. This allows easily interchanging the used solver: currently LODIN is linked against Z3 [11] and Boolector [19] and uses Boolector by default.
8 Conclusion

We presented the fairly new tool LODIN. LODIN implements explicit-state model checking of LLVM with concurrent processes. To combat the state-space explosion problem LODIN supplements explicit-state model checking techniques with simulation-based techniques. For single-threaded programs LODIN implements a symbolic state space representation allowing it to verify programs with non-deterministic input precisely. The symbolic engine of LODIN uses off-the-shelf SMT-solvers - presently Boolector and Z3.

References

[1] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. MIT Press, 2008. ISBN 978-0-262-02649-9.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Conon, and Alain Finkel, editors, Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings, volume 2102 of Lecture Notes in Computer Science, pages 260–264. Springer, 2001. ISBN 3-540-42345-1. doi: 10.1007/3-540-4585-4_25. URL https://doi.org/10.1007/3-540-4585-4_25

[3] Zuzana Baranov, Ji Barnat, Katarna Kejstov, Tade Kuera, Henrich Lauko, Jan Mrzek, Petr Rokai, and Vladimir till. Model checking of C and C++ with DIVINE 4. In Automated Technology for Verification and Analysis (ATVA 2017), volume 10482 of LNCS, pages 201–207. Springer, 2017.

[4] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of Lecture Notes in Computer Science, pages 184–190. Springer, 2011. ISBN 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_16.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. STTT, 9(5-6):505–525, 2007. doi: 10.1007/s10009-007-0044-z. URL https://doi.org/10.1007/s10009-007-0044-z

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. Advances in Computers, 58:117–148, 2003. doi: 10.1016/S0065-2458(03)58003-2. URL https://doi.org/10.1016/S0065-2458(03)58003-2

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings, pages 209–224. USENIX Association, 2008. ISBN 978-1-931971-65-2. URL http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[8] Edmund Clarke, Orna Grumberg, and Doron Peled. Model Checking. MIT Press, 1999.

[9] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752–794, 2003. doi: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643

[10] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika, 26(4): 404–413, 1934.

[11] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof,
editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of Lecture Notes in Computer Science, pages 337–340. Springer, 2008. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[12] LLVM Developers. LLVM language reference manual. https://llvm.org/docs/LangRef.html#sub-instruction, 2018.

[13] Stephan Falke, Florian Merz, and Carsten Sinz. LLBMC: improved bounded model checking of C programs using LLVM - (competition contribution). In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of Lecture Notes in Computer Science, pages 623–626. Springer, 2013. ISBN 978-3-642-36741-0. doi: 10.1007/978-3-642-36742-7_48.

[14] Patrice Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of Lecture Notes in Computer Science, pages 476–479. Springer, 1997. ISBN 3-540-63166-6. doi: 10.1007/3-540-63166-6_52. URL https://doi.org/10.1007/3-540-63166-6_52.

[15] Arie Gurfinkel, Temesghen Kalsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of Lecture Notes in Computer Science, pages 343–361. Springer, 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_20. URL https://doi.org/10.1007/978-3-319-21690-4_20.

[16] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Abrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of Lecture Notes in Computer Science, pages 389–391. Springer, 2014. ISBN 978-3-642-54861-1. doi: 10.1007/978-3-642-54862-8_26. URL https://doi.org/10.1007/978-3-642-54862-8_26.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

[18] Axel Legay, Dirk Nowotka, Danny Bøgsted Poulsen, and Louis-Marie Traonouez. Statistical model checking of LLVM code. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of Lecture Notes in Computer Science, pages 542–549. Springer, 2018. ISBN 978-3-319-95581-0. doi: 10.1007/978-3-319-95582-7_32. URL https://doi.org/10.1007/978-3-319-95582-7_32.
[19] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I, volume 10981 of Lecture Notes in Computer Science, pages 587–595. Springer, 2018. ISBN 978-3-319-96144-6. doi: 10.1007/978-3-319-96145-3_32. URL https://doi.org/10.1007/978-3-319-96145-3_32.

[20] Abraham Wald. Sequential analysis. Courier Corporation, 1973.

[21] Håkan L. S. Younes, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. STTT, 8(3):216–228, 2006.

[22] Håkan L. S. Younes. Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon University, 2005.