Masaryk University
Faculty of Informatics

Generic Platform for Explicit-Symbolic Verification

master's thesis

Vojtěch Havel

Brno, 2014

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vojtěch Havel

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Acknowledgement

Firstly, I would like to thank my thesis advisor Jiří Barnat for all the motivation he gave me during my work on this thesis. I would also like to thank Petr Bauch for collaboration on this interesting topic; it was an enlighting experience to work with somebody with such inspiring enthusiasm to research and wide knowledge of various formal verification research areas. I thank to both for numerous fruitful discussions.

Thanks also belong to all Paradise laboratory members, primarily to Jiří and Ivana, for creating an environment where I have always found motivation, inspiration and where are people with whom it is a joy to collaborate.

Abstract

In this thesis we present a new generic platform SymDivine that may serve as a basis for a program analysis tool that can handle both single-threaded and multi-threaded programs written in the LLVM intermediate language. SymDivine implements the so-called control explicit-data symbolic approach in order to cope with programs with both nontrivial thread interleaving and large input domains. We experimentally evaluate a symbolic execution-like algorithm implemented on the top of SymDivine on a large set of benchmarks and compare it with CPAchecker.

Keywords

model checking, llvm, divine, symbolic execution, satisfiability modulo theories

Contents

1 **Introduction** . . . 1
2 **Preliminaries** . . . 3
2.1 *LLVM* . . . 3
2.2 *Control Explicit–Data Symbolic Model Checking* . . . 5
2.3 *Related Work* . . . 9
2.3.1 Explicit-State Model Checking . . . 9
2.3.2 Symbolic Execution . . . 10
2.3.3 Bounded Model Checking . . . 11
2.3.4 Abstraction . . . 11
3 SymDivine . . . 12
3.1 *Introduction* . . . 12
3.2 *Program State* . . . 13
3.2.1 Explicable Variables . . . 17
3.2.2 Pointers . . . 17
3.3 *Successor Computation* . . . 18
3.3.1 Path Reduction . . . 21
3.4 *Final Remarks* . . . 24
3.4.1 Implementation . . . 24
3.4.2 Models . . . 25
3.4.3 Limitations . . . 25
4 **Data Storage** . . . 27
4.1 *Data Representation Module Interface* . . . 27
4.2 *SMT* . . . 28
4.2.1 Optimizations . . . 30
5 **Program Analysis with** SymDivine . . . 32
5.1 *Reachability* . . . 32
5.2 *Symbolic Execution* . . . 33
5.3 *Symbolic Execution with Limited Equality Check* . . . 34
6 **Experiments** . . . 36
6.1 *Motivation* . . . 36
6.2 *Setting* . . . 37
6.2.1 Testing Environment . . . 37
6.2.2 Benchmark Selection . . . 38
6.2.3 Results . . . 38
6.3 *Weaknesses of Our Approach* . . . 39
7 **Conclusion** . . . 42
A **Content of the attached archive** . . . 47

1 Introduction

Software bugs are a burden of software development. The verification and validation phase of the development process consumes a lot of time and increases the overall development costs. Moreover, errors in a released product degrade its quality and put the product in a bad light. Widely adopted method for bug finding is testing. Over the last few decades, tremendous effort was put to devise formal verification methods that would at least partially replace testing, with the vision of automatic bug finding leading to more reliable software.

The more complex the software is, the harder the formal verification tasks are. There are two main sources of complexity in programs. The first is *control flow* of the program. Each new line of code added to a program may generate new branching or data operations, and may substantially increase the number of interleavings of a multi-threaded program. The second source of program complexity is the amount of data gathered from the environment (e.g. reading from input). These two program components behave differently and require different methods for their analysis, as some approaches well suited to handle one usually fail on the other. [BB13] proposed hybrid approach for formal verification of parallel programs with shared-memory parallelism, where control flow is handled explicitly and data symbolically. This combination benefits from fast state space exploration of contemporary explicit-state model checking approaches (for example DIVINE[BBH+13] explores up to one hundred of thousands of states per second) and recent advances in *Satisfiability Modulo Theories (SMT)* solvers[DMB08, BB09], which enable more effective data manipulation.

For a formal verification tool to be successful, it is essential the tool operates on unmodified code of the program. Creating models for verification manually generates a new obstacle for software engineers that make the tool harder to use. Therefore a majority of contemporary software verification tools operate on some intermediate language which is suitable for automatic analysis. Two mostly used intermediate languages in software verification tools are *C Intermediate Language (CIL)[NMRW02]* and *LLVM bitcode[LA04]*.

In this thesis we present a new generic platform SymDivine enabling combined enumerative-symbolic approaches to program analysis of LLVM bitcode. The platform handles both single-threaded and multi-threaded programs that read data from the input. The chosen approach is similar in the way of composing enumerative and symbolic approaches to [BBH14b] and

[BBH14a] where the languages of interest were Simulink and DVE. We have chosen LLVM as the modeling language due to its widespread use, its robust infrastructure and its clean and simple syntax convenient for automatic analysis. SymDivine can serve as a base not only for a LTL model checking tool, but offers building blocks for other state space exploration-based and symbolic execution-based verification techniques.

This thesis is structured as follows. In Chapter 2, we briefly introduce LLVM and its intermediate representation on the top of which we build the SymDivine platform, then we give some theoretical background for the control explicit–data symbolic approach. In Chapter 3 we describe SymDivine in detail. Chapter 4 is focused on data representations in context of LLVM bitcode verification. Chapter 5 briefly outlines few different program analysis approaches that may be build on the top of SymDivine, and in Chapter 6 we experimentally evaluate program analysis with SymDivine. Finally, we conclude the thesis in the last Chapter 7.

2 Preliminaries

2.1 LLVM

LLVM[LA04][1] is a compiler infrastructure. The central part of LLVM is its well defined language *LLVM intermediate representation (IR)*. IR resembles a three address code and it is in *Static Single Assignment (SSA)* form. LLVM offers the middle player of a typical compiler scheme, it stands between compiler frontend (i.e. it is not input language dependent) and backend (i.e. it is not target dependent). The most popular use of LLVM is in chain with Clang, a C/C++ frontend. See Figure 2.1 for partial overview of utilization of LLVM infrastructure in contemporary compilers.

LLVM project offers a variety of support tools and libraries operating on the top of LLVM IR to help both programmers and users to work with LLVM more easily. The most important include:

- *LLVM Core libraries*

  These libraries are essential part of LLVM. They offer IR parsing and generation, optimization, support for interpreters and JIT compilers, etc.

- *libc++*

  A C++ library, targeting the modern C++11 standard.

- *LLDB*

  A LLVM debugger.

- *IR Just-In-Time interpreter*

LLVM IR

LLVM IR has three different forms: assembly language that is human readable, bitcode format for fast parsing and loading and in-memory IR representation. Core libraries provide translation facilities for every pair of these formats.

Basic Structure

The basic LLVM IR unit is a *module*. A module consists of functions, global variables and other symbols. Each function contains a graph of *basicblocks*.

1. `http://llvm.org/`

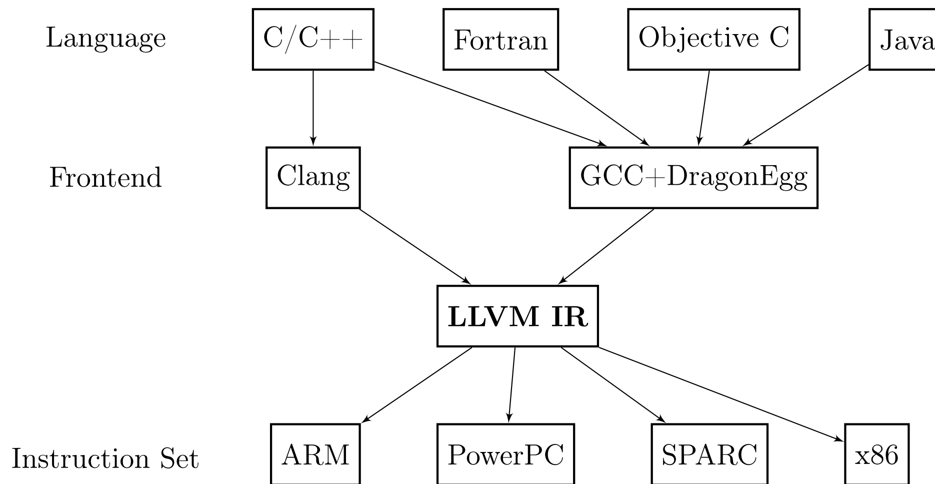

Figure 2.1: Possible uses of LLVM as an intermediate language in different combinations of input and target languages. The list of input languages and target instruction sets in the figure is not complete.

Each basicblock represents a linear sequence of instructions. Jump instructions in LLVM are restricted to jump only to first instruction of the target basicblock (in fact, not instruction labels, but basicblock labels are parameters for jump instructions). The last instruction of every basicblock must be a *terminator instruction* (e.g. jump or return).

Type System and Memory

LLVM IR has a rich type system. Basic types are integers with bitwith ranging from 1 to $2^{23} - 1$, few floating point types and the x86_mmx type, a special type used for MMX instructions. From a set of types we can construct derived types. Most important are pointers, arrays and structures.

Registers and Memory

The number of registers in a LLVM function is not bounded. A code in LLVM IR has a syntactic restriction that every register is assigned a value at most once – it is in so-called SSA form. A register may be seen as a name for the result of the underlying instruction, that can be later used to refer to that value. The type of a register is statically determined by the instruction to which the register corresponds. Values in the memory (i.e. with an address) can be accessed only trough explicit Load and Store instructions.

More detailed description of LLVM IR is out of scope of this thesis, therefore we point the reader to the official LLVM Language Reference[2] for more information.

2.2 Control Explicit–Data Symbolic Model Checking

As we mentioned in the introduction, our aim is to verify properties of software with shared-memory parallelism. This task is hard even if most of data is abstracted from the model due to the non-determinism introduced by thread interleaving. The exponential number of all possible interleavings (with respect to the number of threads and their size measured in lines of code) pose a great challenge in parallel software analysis, addressed for example in [MPC$^+$02] and many others.

We transfer ideas from [BB13, BBH14a, BBH14b] and adapt them to our setting. The chosen approach divides the verification task into two independent parts. The first is to handle the *control flow* of the verified program. Fast state space exploration is a necessity for verification of any programs larger than few lines of code, therefore we explore different control-flow locations by exhaustive enumerative exploration in a similar manner to an explicit model checker in order to cope with large numbers of different control locations in some reasonable time. On the other hand, enumerative exploration is not able to handle programs that read non-deterministic data from larger input domains. For example reading even only a few 32 bit integer values from the input causes tremendous state space explosion. Thus we need to utilize symbolic data representation in order to represent data sets in a feasible manner.

In the rest of this section we formally define and describe the control explicit–data symbolic approach to state space exploration. This theoretical part gives a ground for the next chapter.

Simple Program Model

In order to be able to give formal basis to our work, we start this section with a definition of a formal model of programs that abstracts from technical aspects of LLVM IR or real programs in general.

2. `http://llvm.org/docs/LangRef.html`

Definition 1. *A* linearized program $\mathcal{P}$ *is a tuple* $(C, c_0, E, X, Cmd, Cond)$, *where*

- C *is a (possibly infinite) set of program control locations*
- $c_0 \in C$ *is the initial location*
- $E \subseteq C \times Cmd \times Cond \times C$ *is a transition relation*
- X *is a finite set of program variables*
- Cmd *a finite set of commands*
- $Cond$ *is finite set of conditions*

Elements of the set of commands Cmd *are of two possible forms. The first are arithmetical assignments* $x_0 := x_1 \circ x_2$ *where* x_0, x_1 *and* x_2 *are variables and* $\circ \in \{+, -, *, /\}$, *the second form is* $x_0 = input_{lim}$ *(non-deterministic choice). Conditions from* $Cond$ *are simple arithmetic expressions defined in BNF as:*

$$
\begin{array}{lcl}
\langle \textit{Condition} \rangle & ::= & \langle \textit{SimpleCondition} \rangle \\
& | & \langle \textit{Condition} \rangle \wedge \langle \textit{Condition} \rangle \\
& | & \langle \textit{Condition} \rangle \vee \langle \textit{Condition} \rangle \\
& | & \neg \ \langle \textit{Condition} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{SimpleCondition} \rangle & ::= & x \leq y \\
& | & x < y \\
& | & x > y \\
& | & x \geq y \\
& | & x = y
\end{array}
$$

This definition deserves a clarification at one point. We consider programs with infinite program control locations as we do not limit the set C to be finite. This is indeed a reasonable assumption; we would like to distinguish two different program configurations when their call stacks are not same, and a program with recursive function may in theory have an infinite number of different call stacks. Of course, we might have different definition of a program where we would allow the use of functions and the infinite call stack would be described implicitly by the semantics of a program, but we would like to keep the definition as simple as possible, this definition matches our notion (in context of previous work) of *control* better and finally, infinity of the set C does not pose a problem in the rest of the thesis.

Definition 2. *A* state *of a program $\mathcal{P}$ is a tuple (c, v), where:*

- $c \in C$ *is a program location (program counter)*
- $v : X \to \mathbb{Z}$ *is a variable valuation*

Definition 3. *To each program $\mathcal{P} = (C, c_0, E, X, Cmd, Cond)$ we assign a semantics, called* state space *of $\mathcal{P}$, denoted as $[\![\mathcal{P}]\!]$, defined as a Kripke structure $(S, I, \to, L)$, where*

- $S = C \times V$*, where V is the set of all valuations of variables X*
- $I = \{(c_0, \lambda x.0)\}$
- $(c, v) \to (c', v')$ *iff* $\exists (c, e, g, c') \in E$ *where condition g holds in valuation v and command e evaluates v to v', formally:*

$$v'(x) = \begin{cases} v(x) & x \neq x_0 \\ v(x_1) \circ v(x_2) & \textit{if } x_0 = x \textit{ and } e = x_0 := x_1 \circ x_2 \\ k, \textit{ where } 0 \leq k < lim & \textit{if } x_0 = x \textit{ and } e = input_{lim} \end{cases}$$

We call the above defined semantics as *explicit state space* of $\mathcal{P}$. This state space represents fully "unrolled" state space of the program. Kripke structure $[\![\mathcal{P}]\!]$ is the structure that explicit model checking approaches explore for error location reachability in case of assert violation checking or for accepting cycle presence in case of automata-based LTL model checking.

We finally move from purely explicit world to control explicit–data symbolic world.

Definition 4. *A* set-based reduction *of a state space of a given program $\mathcal{P} = (C, c_0, E, X, Cmd, Cond)$ is a Kripke structure $(\hat{S}, \hat{I}, \hat{\to}, \hat{L})$ that satisfies the following conditions:*

1. $\hat{S} \subseteq C \times 2^V$*, where V is the set of all valuations of variables X*
2. $\hat{I} = \{(c_0, \{\lambda x.0\})\}$
3. *if $(c, v) \to (c', v')$ then there is D, D' such that $v \in D, v' \in D'$ and $(c, D) \hat{\to} (c', D')$*
4. *if $(c, D) \hat{\to} (c', D')$ then for all $v' \in D'$ exists $v \in D$ such that $(c, v) \to (c', v')$*

where $\to$ is transition relation of the explicit state space $[\![\mathcal{P}]\!]$

The motivation behind this definition is that we would like to group some states into a single state which represents the whole set of states in some compact way. If such a compacted state space comply this definition, it would preserve some important properties, as we show later. There is some degree of freedom in the definition. A simplest set-based reduction (defined in [BB13]) for a program $\mathcal{P}$ would be state space isomorphic to $[\![\mathcal{P}]\!]$, where we just "rename" all states (c, v) to $(c, \{v\})$. The other extreme might be a state space defined for a program $\mathcal{P} = (C, c_0, E, X, Cmd, Cond)$ as $K = (S, I, \rightarrow, L)$, where K is defined inductively as:

- **Base:** $I = \{(c_0, \{\lambda x.0\})\}$ and $(c_0, \{\lambda x.0\}) \in S$
- **Inductive step:** if $(c, D) \in S$ and there exists $(c, e, g, c') \in E$ then $(c, D) \rightarrow (c', D')$ where $D' = \{v' \mid \exists v \in D$ such that (c, v) and (c', v') are in transition relation of $[\![\mathcal{P}]\!]\}$

In this state space we follow the rule *do not branch if you do not have to* – we merge all successors with same control location into a single state. In this work, we will need to allow two successor with same control location, therefore we keep the definition less restrictive.

Finally, the definition of set-based abstracted state space would be of no use if the abstracted state space does not preserve important properties.

Theorem 1 (Path preservation)**.** *Let $\mathcal{P} = (C, c_0, E, X, Cmd, Cond)$ be a program and $\hat{K} = (\hat{S}, \hat{I}, \hat{\rightarrow}, \hat{L})$ set-based abstracted state space of $\mathcal{P}$. Then the following two conditions hold:*

1. *For each, possible infinite, run $(c_0, v_0), (c_1, v_1), ...$ in $[\![\mathcal{P}]\!]$, there exists a run $(c_0, D_0), (c_1, D_1), ...$ in $\hat{K}$ such that $\forall i \; v_i \in D_i$.*
2. *For each, possible infinite, run $(c_0, D_0), (c_1, D_1), ...$ in $\hat{K}$ there exists a run $(c_0, v_0), (c_1, v_1), ...$ in $[\![\mathcal{P}]\!]$ such that $\forall i \; v_i \in D_i$.*

Proof. 1. Suppose a run $(c_0, \lambda x.0), (c_1, v_1), ...$ in $[\![\mathcal{P}]\!]$ for which the condition 1 does not hold. Let $w = (c_0, \{\lambda x.0\}), (c_1, D_1), ..., (c_{n-1}, D_{n-1})$ be the longest run in $\hat{K}$ such that $\forall i < n . v_i \in D_i$. But $\hat{K}$ is set-based reduction, hence there is D_n such that $(c_{n-1}, D_{n-1}) \hat{\rightarrow} (c_n, D_n)$ and $c_n \in D_n$ (see the third condition in Definition4), which is contradiction with maximality of w.

2. Conversely suppose a run $(c_0, \{\lambda x.0\}), (c_1, D_1), ...$ in K for which the condition 2 does not hold and $w = (c_0, \lambda x.0), (c_1, v_1), ...(c_{n-1}, v_{n-1})$ the longest run in $[\![\mathcal{P}]\!]$ such that $\forall i < n . v_i \in D_i$. Similarly, we can extend this run by (c_0, v_n) due to second condition in Definition 4. □

The Theorem 1 implies the following corollaries:

- a program location c is reachable in the explicit state space if and only if it is reachable in a set-based reduced state space and
- a given LTL formula φ holds in explicit state space if and only if it holds in a set-based reduced state space.

Since a proof of these statemens would require lots of very technical definitions and proofs, we leave them on the intuitive level.

To this point, we have not given any formal labeling to states, but the LTL formula preservation property of set-based reduction needs one. For simplicity, let assume atomic predicates over control point locations only (i.e. atomic predicate φ_c holds in states of the form (c, v)), because such atomic predicates are valuation-independent and therefore we can assign a validity of atomic predicate in an abstract state uniquely (as all concrete states (c, v) in some abstract state (c, D) share the control location).

Note: The set-reduced state space does neither lose any information about the original state space nor introduce spurious behavior; set-based reduction represents the state space precisely, but it may represent it more *compactly* by abstracting from (in some sense) equivalent behavior (it is an *exact abstraction*).

2.3 Related Work

Over the last few years LLVM IR has gained popularity in the program analysis community. In this section we give a brief overview of program analysis methods and tools operating either on LLVM IR or directly on C/C++. Of course, the list is not complete, we tried to pick the most relevant approaches.

2.3.1 Explicit-State Model Checking

Explicit-state model checking is perhaps the closest approach to ours. The core idea of explicit-state model checking is to explore the whole state space of a program in a graph-traversal manner, where a state corresponds to the exact program state (one concrete state of memory, call stack etc.).

SymDivine differs from explicit-state model checking approaches by its ability to handle nondeterministic data. On programs without reading any data, SymDivine and an explicit-state model checker can be hardly distinquished.

Related Tools

divine[BBH+13] is an explicit-state model checker, that accepts LLVM IR as one of many modeling input languages. divine tackles the state space explosion with parallel and distributed processing, tree-compaction and other methods. Current versions of divine do not offer any method for dealing with nondeterministic input data. Reading a nondeterministic value from the input causes the state space to branch for all possible values.

A very recent LLVM IR model checker, similar to divine in the capabilities from the language point-of-view, is MCP[TBS].

LLVM IR model checking was recently added to the LTSmin explicit-state model checker[vdB13, LvdPW11]. It aims at verification of parallel programs with a weaker memory model.

2.3.2 Symbolic Execution

Symbolic execution[Kin76] is a program analysis technique that, in contrary to the explicit-state model checking, aims at verification of programs that read nondeterministic values from the input. SymDivine and symbolic execution share the execution-based nature with symbolically stored path conditions (and in fact, symbolic execution-like program analysis may be build on the top SymDivine, see the Chapter 5).

Symbolic execution is rarely applied to (real) parallel programs due to increased level of path explosion introduced by interleaving, therefore we chose not to limit SymDivine to do pure symbolic execution, we keep the possibility to detect equivallent states and merge them.

Related Tools

klee[CDE08] is a symbolic virtual machine for LLVM IR. It may be used not only for symbolic execution, but also for other symbolic execution-based bug-finding techniques like high-coverage test generation. Another tool for LLVM IR symbolic execution is Rudla[SST13], a tool based on bugst. Rudla implements *compact symbolic execution*, which is a classical symbolic execution extended by an additional preprocessing step that may help to summarize some cycles in the given program into a single transition. The main difference between SymDivine and the two above mentioned tools is the ability of SymDivine to handle multi-threaded programs while neither klee nor Rudla support it.

2.3.3 Bounded Model Checking

Bounded model checking is another important approach to program analysis. Bounded model checking is fundamentally different from explicit-state model checking and symbolic execution – it is not *execution*-based, it does not explore all possible states or execution paths of a given program separately. Bounded model checking describes all possible paths violating some property as a formula in some logic and by satisfiability checking decides whether the program is correct. Despite being less related to our work we included it in this overview due to the raising popularity of bounded model checking-based tools in the program analysis community.

Related Tools

The most known bounded model checker for LLVM IR is LLBMC[MFS12]. It targets at complete C/C++ support with a precise memory model with dynamic memory, casts, etc. For user-specified limits on number of cycle iterations and recursive calls LLBMC unrolls all cycles and inline all functions up to these bounds and then it code the program into a bitvector formula which is satisfiable if and only if the program violates given conditions. LLBMC does not support multi-threaded programs.

Another tools based on bounded model checking are CBMC[KT14] and ESBMC[MRC+14]. Both operate directly on the C programming language and support threads.

2.3.4 Abstraction

All the program analysis approaches we have mentioned so far are exact in a sense that if the analysis ends with some result, we are guaranteed that the result is correct. Abstraction-based methods compute an approximation (often over-approximation) of a program that is easier to analyse. Since an analysis on such approximated program may give a wrong result it is necessary to wrap the procedure into some refinement loop.

Related Tools

Prominent program analysis tool implementing various kinds of abstraction is CPAchecker[BK11]. Dominant feature of CPAchecker is its configurable design that should allow to implement new program analyses (mostly based on some kind of abstraction) easily. CPAchecker operates directly on C programs.

3 SymDivine

3.1 Introduction

SymDivine is a generic platform enabling program analysis of parallel programs written in LLVM IR. SymDivine stands between the input formalism (LLVM IR) and a program analysis algorithm, forming a layer that provides a clean view to a LLVM program state space without any complex details of LLVM IR. More precisely, SymDivine offers a set-based reduction *generator* for programs written in LLVM IR.

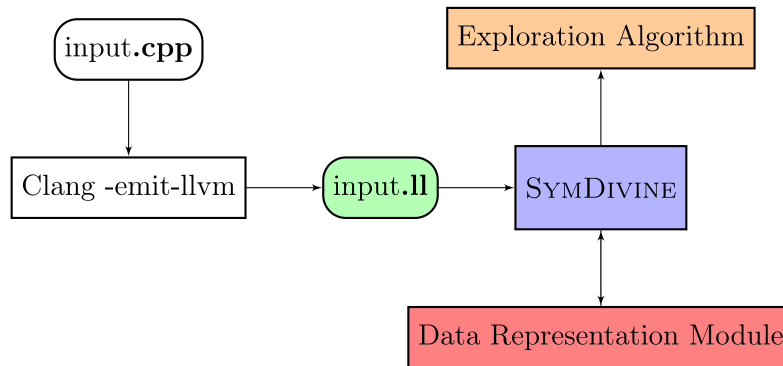

Figure 3.1: A high level schema of a program analysis tool build on top of SymDivine. The input file `input.ll` may be generated differently, not only from a C/C++ source code. SymDivine reads an input LLVM IR program and offers successor relation in reduced state space for exploration algorithm.

For set-based reduction construction, SymDivine needs to internally represent sets of valuations. Current state-of-the-art program analysis methods rely on few different data set representations, for example *Binary Decision Diagrams* (BDD), or formulae in some logic. SymDivine is not limited to one particular representation, rather it assumes a data representation module with a certain interface to be "plugged" into it. In this chapter, we describe SymDivine in detail, without any connection to any particular data representation used or program analysis algorithm running on the top of SymDivine. For a detailed view on data representations and data representation implementation, see Chapter 4; Chapter 5 is dedicated to program analysis with SymDivine.

From a high level perspective SymDivine operates in same manner as a

state space generator in the divine model checker (and in fact, it emerged from an experimental implementation of control explicit–data symbolic generator); for a given LLVM IR code on the input SymDivine computes the initial state and for a given state it computes all its successors. Basically, these two functions of SymDivine may be used for state space exploration. In the rest of this chapter, we look in detail how SymDivine represents a state of a LLVM program and how, given a state and a LLVM instruction, the smallest "step" in LLVM IR, SymDivine computes the set of successors. When more threads are allowed to execute (e.g. not waiting for mutex), we choose nondeterministically one; SymDivine therefore capture all possible interleavings in generated state space.

3.2 Program State

At each point, SymDivine holds a representation of the state on which it currently operates (computes its successors). The main parts of state that is required to capture in the representation are:

1. control flow
   (a) call stack for each thread
2. data
   (a) memory shape (length of the memory used etc.)
   (b) stack content for each thread
   (c) heap content
3. state flags (e.g. error flag)

SymDivine strictly distinguishes between control flow and data in the state representation, because a control flow–data symbolic program analysis algorithm is then able to treat control flow and data separately; most importantly SymDivine stores control flow information canonically, therefore the algorithm can compute a hash value of the control part and use it for effective state look-up via hashtables. (We cannot assume the same property for data; we require a data representation to provide an `equal` operator that compares two states, and it may or may not be syntactical comparison, typically a data representation by formulae in some logic is rarely canonical, therefore equality of states is not the same operation as the syntactical comparison in this case).

Control Flow Representation

Representation of the control flow is straightforward – we need to keep information about call stacks of all active threads, therefore control flow is in SymDivine represented as tuple of program locations sequences (i.e. call stack for each thread).

Effect of Instruction Execution on Control Flow

The control flow representation is updated with each instruction executed by SymDivine. Each instruction increments the program counter of the active thread by one, branching instructions change the program counter to the program location given as argument and call instructions push current program counter on the call stack and update the program counter to the program location of the called function. A few special functions have some special effect on control flow (`pthread_create` creates new thread, `pthread_exit` destroys it, `pthread_join` and `pthread_mutex_lock` forbid the program counter to advance until a certain event happens etc.).

Data Model

As we mentioned earlier, SymDivine does not handle data representation itself, instead, it offloads it to a data representation module. We leave details on particular data representations on next chapter, in this chapter we rather describe the interplay between SymDivine and a data representation module.

There are two major data model types in symbolic program analysis, *flat byte-array* (used for example in LLBMC[MFS12]) and *typed memory model* (used in older versions of CBMC[CKL04]). In the former, all memory in a state is view as a byte-array and all register or memory values are mapped into this array. The drawback of this approach is in difficulty of arithmetic operations on multi-byte values, because we need to obtain the value of a multi-byte value from several bytes before actually computing the arithmetic operation. Typed memory model on the other hand considers the memory as a sequence of variables with a specified bit-width. In this model, arithmetic operations does not come with the overhead as in the flat byte-array model, which possibly makes it more effective. The problem with the typed memory model is the weak type system of C. For example suppose a program that allocates memory with a `malloc` call; the obtained memory is hidden behind a `void*` pointer and later possibly typed to different type, `int` for example. In flat byte-array model this does not necessarily pose a problem as the type

of a variable influences only the way of composing bytes into a wider variable in arithmetic operations.

Since SymDivine is designed to be data representation independend and we dont want to rely on data representation module ability to (effectively) compose single bytes to larger values, the typed memory model is used. We currently do not support LLVM IR programs that access the same piece of memory through two pointers of different types. In real programs, this is very restrictive policy.

Memory Structure

SymDivine has a layer between the data representation module and its instruction execution core, that keeps information about *memory layout*. As memory layout we understand mapping from variables in LLVM program to variables in data representation. When new block of memory is allocated in a LLVM program, it is assigned a unique identifier. Then, each variable in each state is uniquely addressed by a pair of positive numbers `(segment_id, offset)`, where `segment_id` is the unique identifier of the block memory to which the variable belongs to and `offset` is the offset of this variable inside this block (not *byte offset*, but *variable offset*, i.e. first variable is on the offset 0, second on the offset 1, etc., no matter how many bytes wide are the variables).

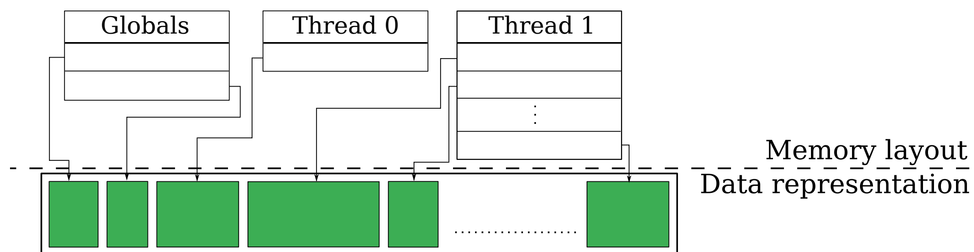

Figure 3.2: Data model in SymDivine. The bottom part belongs to a data representation module; the green blocks represent data segments stored inside the data representation module, on the other side, SymDivine keeps *memory layout* – information about which segments are currently available, to which threads they belong and which program variables each segment represents. Each segment may contain a different number of variables.

The Memory Layout Layer (MLL) offers the following run-time functionality:

- When a thread executes the `call` instruction, MLL creates new mem-

ory segment for automatic variables in the function and assigns new unique identifier to it.

- When a thread executes the `alloca` instruction, MLL creates new memory segment for newly allocated memory and assigns new unique identifier to it.
- When SymDivine executes an instruction that references some registers (as result or as operands), MLL provides `(segment_id, offset)` for the register.

The purpose of MLL is to allow communication of SymDivine and a data representation module. Now, equipped with such mechanism, we can finally describe the interplay between SymDivine and data representation module. SymDivine assumes that the data representation module provides the following interface (the list is not complete, more details can be found in the source code):

- `addSegment`, `eraseSegment` – functions, that create or delete a segment. `addSegment` takes a list of bit widths of variables in the segment
- $\texttt{implement}_{op}$`(r, a, b)` for any arithmetic operation *op*. Each argument is a variable name from MLL, i. e. a pair `(segmant_id, offset)`. After call to $\texttt{implement}_{op}$, data representation module is supposed to compute `a` *op* `b` and store it in `r`
- $\texttt{implement}_{\texttt{input}}$`(r)` stores nondeterministic value to `r`
- $\texttt{prune}_{op}$`(a, b)` for any relational operator – each restricts the state to values that satisfy `a` *op* `b`
- `empty` returns true if and only if the set of currently stored values is empty
- `equal` compares two sets of valuations for set equality

SymDivine uses any data representation module as long as it correctly implements this set of functions. When SymDivine computes a successor, for example labeled with `add` instruction, it takes care of advancing the program counter of underlying thread, uses MLL to compute variable identifiers `segment_id, offset` of instruction operands and instruction result, and calls $\texttt{implement}_{+}$ with the relevant arguments. At the end of successor generation, SymDivine checks whether the set of variable valuations is empty (with the `empty` call) and if so, the successor is not generated.

3.2.1 Explicable Variables

It is possible that in some state a variable can have only one possible value. According to our observation, lots of values (especially stored in registers) are of this kind. While such variables may be stored symbolically inside the data representation module, it is more effective to store such values explicitly (i.e. as concrete values); arithmetic operations on concrete numbers can be computed few magnitudes faster then computing them symbolically.

Authors of [BBH14a] stored all values symbolically first, but in the state generation process they checked dynamically for variables that may be evaluated to only one possible value in that state. Similar optimization may be found in the KLEE symbolic virtual machine[CDE08]. The whole model checking process was sped-up, but the dynamic check costs a considerable amount of time. In LLVM, every model usually contains a huge number of variables (in contrast to models in [BBH14a]), therefore checking for such "explicable" variables dynamically would come with a huge time penalty. Therefore in SymDivine we chose a compromise; we analyze input LLVM program beforehand and by static analysis we compute the set of variables whose value may depend on the input; other variables may be stored explicitly (as concrete values). It is obvious that it is possible that a value is stored symbolically (i.e. in data representation module) even if it has only one possible valuation at some point – suppose a program, that on some execution path reads a non deterministic integer value to register `x`, then it checks whether `x == 5`. At this point, SymDivine creates two successors, one which is pruned by $\texttt{prune}_{=}\texttt{(x, 5)}$ and second by $\texttt{prune}_{\neq}\texttt{(x,5)}$. The variable `x` can be evaluated to only one value (5) in the first branch, but as the variable held a nondeterministic value, it is still stored symbolically.

Our experiments show that the difference in performance between symbolic manipulation with data and manipulations with concrete numbers is significant, moreover, an algorithm using SymDivine can benefit from this optimization too; explicitly enumerated values may be used for hash computation, which enables more effective state matching and state look-up.

3.2.2 Pointers

We have defined a memory model for our symbolic interpreter; hence we need to choose how to represent *pointers* and give a semantics to pointer arithmetics. Firstly, we try to formulate some properties we believe to characterize the pointer semantics "correctness".

1. *dereference consistency*: Suppose we point a pointer to a particular memory location. Unless the pointer is altered or the memory cell vanishes, loads and stores via this pointer have same effect as writing to or reading from that memory cell directly.

2. *pointer arithmetics correctness*: Suppose a set of program memory locations that is guaranteed to be contiguous (i.e. allocated through `alloca, malloc` or fields of a vector type). Then adding some offset x to pointer value creates new pointer pointing at the memory location which is x elements farther.

3. *bounds checking possibility*: We must be able to determine whether a pointer points to legal memory location or not (detect overflows or underruns) to enable memory safety analysis.

We have omitted pointer casting in this list, although it needs to be addressed too; but as we have mentioned before, we currently use simpler *typed memory model* and casting will be part of future work.

SymDivine represents pointers as a 64 bit number, where the upper 32 bits is segment number and the lower 32 bits is offset inside the segment. Such pointer mapping corresponds to our memory layout scheme, therefore dereferencing does not need any complex mapping. Currently we forbid pointers with a symbolic value, more precisely, we require all pointers to be explicable (see previous section for details). Although symbolic pointers have sufficient theoretical basis and can be used[DMB08, BB09, BMS06, FMS14], allowing it would require the data representation module to handle it. At this time we want to keep SymDivine as generic as possible, therefore we have decided to leave symbolic pointers for a future consideration.

This pointer arithmetics then naturally maps to regular integer arithmetics. SymDivine allows segments to contain at most $2^{32} - 1$ fields, therefore adding a valid offset (i.e. new offset is at most $2^{32} - 1$) to a pointer alters only the lower 32 bits. Such pointer arithmetics comply with both pointer arithmetic-related conditions (second and third), as each segment is allocated with a particular number of elements and therefore we are able to check whether a pointer is legal or not.

3.3 Successor Computation

SymDivine offers two main functions for an exploration algorithm: `initial` and `advance`. `initial` creates the initial state of a given LLVM program,

the “entry point” for most program analysis algorithms. Creating the initial state consists of the following steps:

- create new segments for global variables
- initialize each global variable, possibly by calling data representation module functions when the variable is not explicable
- create main thread
- create new segment for `main` function stack

Computation of `advance` consists basically of three phases. First, we list all active threads and fetch their current instructions. For each such a instruction, in second phase, we execute it in context of the underlying thread. Finally, we check whether the state is nonempty and if so, we pass the state to the exploration algorithm. LLVM IR consists of large number of different instructions and so a detailed description of the instruction execution phase is out of scope of this thesis. Instead, we illustrate SymDivine successor generation process on a small example.

Example

In Figure 3.3 a state space possibly generated by SymDivine is depicted. For a reader not familiar with LLVM IR, a brief comment on the program: the program consists of two functions `main` and `f`. Function `f` reads two 32bit integers from the input, compute their sum and returns single bit that is set to true if and only if the sum computed is a positive number. The entry point of the program, the function `main`, in a loop calls `f` until it returns true; then it jumps to the `exit` label and returns from `main`.

```
1  declare i32 @input();
2
3  define i32 @main() {
4  start:
5    %0 = call i1 @f()
6    br i1 %0, label %exit, label %start
7  exit:
8    ret i32 0
9  }
10
11 define i1 @f() {
12   %1 = call i32 @input()
13   %2 = call i32 @input()
14   %3 = add i32 %1, %2
15   %4 = icmp sgt i32 %3, 0
16   ret i1 %4
17 }
```

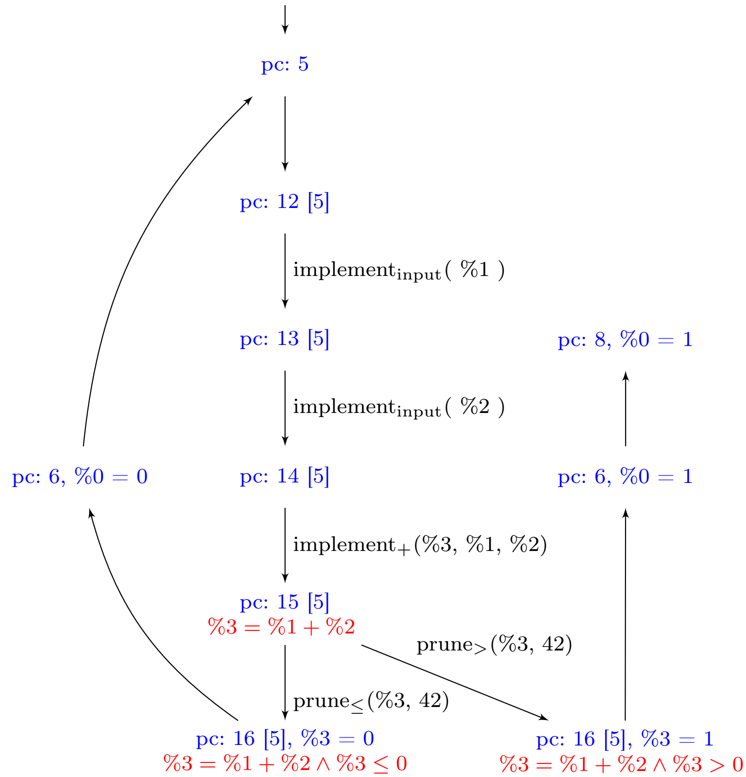

Figure 3.3: simple program written in LLVM IR and a possible state space generated by SymDivine

Now we go through the example step-by-step:

- **pc: 5**: The initial state of this program consists of one thread with program counter at the first instruction in the `main` function (line 5).
- **pc: 12 [5]**: It has one successor, a state, where the main thread calls the function `f` and therefore the old program counter is saved on call stack and the program counter is set to first instruction in `f`.
- **pc: 13,14 [5]**: Next two steps are reading from the input – SymDivine calls the data representation module to instruct it that variables corresponding to registers %1 and %2 hold a nondeterministic value.
- **pc: 15 [5]**: SymDivine executes the `Add` instruction. Because its result (the register %3) is not explicable (depends on input), SymDivine calls data representation module to store the result of %1 + %2 in %3.
- **pc: 16 [5]**: `icmp` instructions cause branching – SymDivine generates two states, "positive" and "negative" branch, each pruned respectively. Both branches of a `icmp` instruction are candidates for being empty – therefore SymDivine checks by calling the `empty` function on both successors. In this case, the register %3 may contain both positive or nonpositive integer, hence neither of new states is empty.
- **pc: 6** (both branches): The single bit value is returned to the caller and the function is left. SymDivine calls `eraseSegment` on the data representation module. The right branch simply jumps to line 8 and in next state it returns from `main`.
- **pc: 5**: The left branch simply jumps to line 5.

The last step is the most interesting one. It is the first moment when SymDivine generated a state with the control part equal to some other already visited state (the initial state in this case). It depends on exploration algorithm's decision whether it maintains a database of already visited states and checks for state equality. If it does, after the last step of the example (where the state with program counter 5 is generated), comparing the new and the initial state yields a call to the `equal` function of the data representation module.

3.3.1 Path Reduction

LLVM IR is a low level language. Although its low level of abstraction matches the degree of details needed to fully describe behavior of real pro-

grams, it is not always necessary to build program state space with such precision. Sequences of arithmetic operations typically can be computed in only one step instead of many; reducing the path in such way not only saves some space by not storing all "intermediate" states, but it also reduces the blow-up caused by interleaving, because the original path may interleave with other threads.

Similar problem is addressed in CPAchecker[BCG+09, BKW10] where some sequences of actions are collapsed into a single action in order to reduce the overhead that comes with each newly computed abstract state (merging with other states and checking termination). In contrast to CPAchecker we allow programs with more threads, hence we have to take thread interleaving into account too. Other work addressing the same issue is [BBGK11], that is still not concerned with threads, but is more close to our approach. We adopt τ-reduction introduced in [BBR12] to the divine model checker with few modifications towards the more effective τ+reduction[RBB13].

The basic idea behind τ-reduction is that effects of instructions are *observable* from other threads only when the thread accesses main memory via `Load` or `Store` instructions and few others; after such instructions we have to generate a successor to enable interleaving of these observable actions with instructions executed by other threads. We can benefit from this fact by executing more instructions from one thread and accumulate them into a single transition.

divine's LLVM interpreter has one property that SymDivine does not. When divine picks one thread to execute in a particular state, it has only one possible successor (except some exceptions like calls to special functions), because states in divine are concrete and instructions in LLVM IR do not introduce nondeterminism by itself. Therefore τ-reduction collapses *paths* a single transition.

In SymDivine one state may have multiple successors that result from executing a single instruction from one fixed thread (currently this can happen when executing compare instruction `icmp` (see example 3.3) or branching instructions `br` and `switch`). One solution to this problem could be to consider each possibly branching instruction to be observable; then we would collapse paths only, but that would be too restrictive.

Our approach to τ-reduction in state space generation is as follows. Suppose we want to generate the set of successors of a given state. The idea is to run a graph traversal from that state on the original state space restricted to instructions belonging to one fixed thread and collect all nearest observable states. A more precise description of the successor set computation function is in Algorithm 1.

Algorithm 1: Computes the set of all successors in the τ-reduced state space.

```
begin compute_successors(state S)
    succs = {};
    foreach thread t do
        to_do := {S};
        while to_do not empty do
            S' := any state from to_do;
            to_do := to_do \ S';
            Inst := current instruction of thread t in state S';
            foreach successor R of S' via instruction Inst do
                if Inst is observable then
                    add R to succs;
                end
                else
                    add R to to_do;
                end
            end
        end
    end
end
```

The algorithm assumes that on each cycle there is at least one observable instruction, in other case it may not terminate (because the graph traversing lacks the "visited" check when discovering new states). However, this is not a restriction at all; we cannot collapse cycles into single transitions, therefore we need at least one observable state on each cycle anyway.

SymDivine considers as observable instructions, that:

- change global state: `Load`, `Store` or calls to functions that changes state of locks (`pthread_mutex_lock, pthread_mutex_unlock`)
- possibly create/destroy threads: `Ret` (return-from-function instruction), calls to `pthread_create`, `pthread_destroy`
- change the validity of atomic proposition or create error state: calls to `assert`
- change program control flow to a *lower* address

The first two classes of instructions must be observable because they change global state of a program, to enable interleaving with other such actions. The third bullet are instructions that change validity of atomic propositions. Hiding such changes in collapsed paths may change the verification outcome. The last class are branching instructions; the sole purpose of making these instructions observable is to ensure that on each cycle is at least one observable instruction. An example how SymDivine generates the τ-reduced set of successors is given in Figure 3.4.

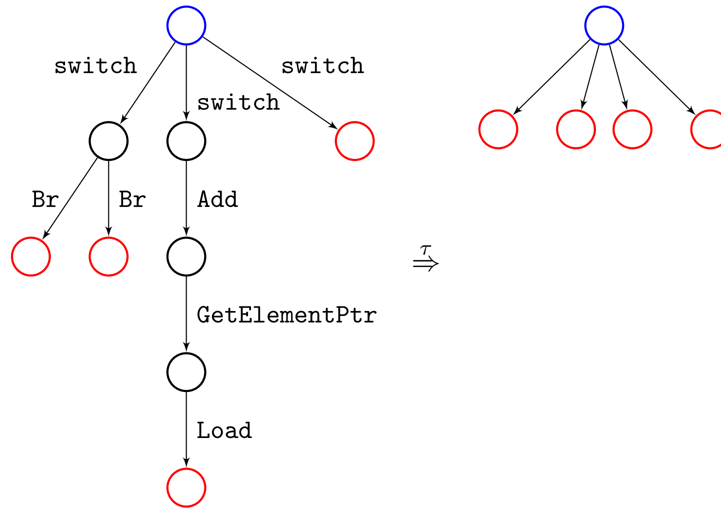

Figure 3.4: τ-reduction in SymDivine; the state space on the left is unreduced and restricted to actions of one fixed thread, blue is the start node, black state are not observable whereas red ones are observable. On the right side the reduced state space is depicted, computed by the Algorithm 1. (Note: the set of all successors may not always form a tree, but it is always an acyclic graph.)

3.4 Final Remarks

3.4.1 Implementation

We have implemented SymDivine in the C++ programming language. The choice was made for a number of reasons. Exploring large state spaces calls for a language that compiles into an effective code. Next reason is that we may be able to find some intersection in SymDivine's and divine's functionality and share it in the future. The most important reason why we have chosen C++

is that most related libraries and tools offer native C++ interface (LLVM core libraries, z3 and other). We parse the input file with the help of LLVM core libraries.

3.4.2 Models

SymDivine expects the model to be in a LLVM IR assembly file (i.e. in the human-readable form of LLVM IR). It is assumed the module contains a `main` functions which is the entry point of the program. We have tested only LLVM IRs that originated from C or simple C++ programs. Other LLVM IR programs should be supported too, unless they use some advanced LLVM IR features (the limits are subject of the next section).

SymDivine partially supports the pthread standard for threads, namely dynamic thread creation via the `pthread_create` function, mutex locking via `pthread_mutex_(un)lock`, the thread joining function `pthread_join` and thread termination via `pthread_exit`. Functions `input` and `__VERIFIER_-nondet_*` read nondeterministic value from the input. The bitwith of the input value is not tied to the name of the called function but it is always determined by the return type of the called function.

3.4.3 Limitations

Not all LLVM programs are handled by SymDivine. In this section we summarize the limitations of SymDivine from the modeling language point of view. The first limitation we have already came across before – we use the typed memory model in SymDivine and it does not handle pointer casts. This is a serious limitation, because it makes memory allocation via `malloc` nearly unusable (`malloc` allocates untyped block of memory that is usually casted to desired type later), and therefore it should be a high priority future task.

As we mentioned earlier, SymDivine does not handle symbolic pointers. To be precise, we consider a pointer to be symbolic if there is a point at some execution path when a pointer may have at least two different values, in other words, a symbolic pointer is a pointer variable that holds a value that is not determined exactly by the execution path (for example reading a character from the input and indexing a table by its numeric value results in a symbolic pointer). Most programs we could find that contain symbolic pointers did also allocate memory via `malloc`, as programs with fixed-size datatypes rarely use symbolic pointers, therefore implementing a support for symbolic pointers can wait until we overcome the previous limitation.

LLVM IR has lots of features. Some of them were not needed in our experimental evaluation and are less frequently used; therefore we have not spent time on implementing a support for them. Currently we do not support vector operations, floating and intrinsic functions. Except a few exceptions SymDivine supports the rest of LLVM IR instructions. SymDivine requires the program on the input to be *self-contained* or *closed* – it does not call any external functions whose definition is not available in the program (for example call to standard library function as `memcpy`). These features are very diverse in their level of usefulness and should be prioritized on demand.

We deal with parallel shared-memory programs. The semantics of a parallel program is influenced by a *memory model*. Currently SymDivine assumes *Sequential Consistency*[Lam79] which may not be precise for some programs. Instead, we should consider other memory models and take LLVM atomic memory ordering constraints into account (as did authors of [vdB13]), possibly in some form of store buffer simulation. Verification with store buffers is a challenging task even without nondeterministic data[ABP11], hence this feature has rather low priority.

4 Data Storage

In this chapter we look closely at data representation in SymDivine. Our needs motivate us to strictly separate the LLVM-specific instruction execution core and the data representation in order to allow us to experiment with different data representations easily, without any need to dip into technical details of LLVM IR. We have chosen often used data representation by contemporary program analysis tools – logic formulae (from bitvector theory in our case). To decide state emptiness and state equality we use a SMT solver. With the "SMT" data representation (which is described in detail in Section 4.2) we did few experiments of SymDivine that is available in Chapter 6

4.1 Data Representation Module Interface

We briefly touched data the representation interface in the previous chapter in the SymDivine and data representation module interplay description. This section is dedicated to important details of data representation module interface that delimits the requirements put on data representation module. Data representation is *stateful*; it is supposed to internally hold a set of valuations representation that changes with calls to its interface.

Semantics of Operations on Data Representation Module

Formally, the state of data representation is a set of valuations. The domain of valuations must be the same for all valuations in one such set, but is not fixed among different states – variables are created and deleted through the computation, because LLVM can dynamically allocate and deallocate memory. We give precise semantics for the functions that data representation module must offer as operations over these sets of valuations. In other words, we define how executing an operation (for example `implement_add`) on data representation changes the state the data representation currently holds. Arithmetic operations should consider the valuations as bitvectors (i.e. all operations should be modular) to capture the LLVM IR semantics precisely.

The first class of operations on data representation are arithmetic operations. Suppose the state currently represented by data representation module is a valuation $\mathcal{C}$ and r, a, b are variables in the domain of functions in $\mathcal{C}$. Then `implement_add(r,b,c)` executed on $\mathcal{C}$ changes the state

to $\mathcal{C}' = \{v^{+}_{r,a,b} \mid v \in \mathcal{C}\}$, where

$$v^{+}_{r,a,b}(x) = \begin{cases} v(x) & x \neq r \\ v(a) + v(b) = v(x) & x = r \end{cases},$$

other arithmetic operations are defined in similar manner.

Other operation is $\texttt{implement}_{\texttt{input}}$. All operations from the previous class have one property in common – the operation preserves cardinality of a set of valuations, on contrary, $\texttt{implement}_{\texttt{input}}$ enlarges the input set of valuations. For a given set of valuations $\mathcal{C}$ and a variable a, $\texttt{implement}_{\texttt{input}}\texttt{(a)}$ creates set of valuations $\mathcal{C}' = \{v^{input}_{a,k} \mid v \in \mathcal{C}, k \text{ is a bitvector value}\}$, where

$$v^{input}_{a,k}(x) = \begin{cases} v(x) & x \neq a \\ k & x = a. \end{cases}$$

The last class of operations are pruning operations $\texttt{prune}_{op}$. $\texttt{prune}_{op}$ does not modify any valuation, it only allows valuations that satisfy given condition. Formally, for a set of valuations $\mathcal{C}$ and variables a, b, $\texttt{implement}_{<}$ executed on $\mathcal{C}$ changes the state to $\mathcal{C}' = \{v \in \mathcal{C} \mid v(a) < v(b)\}$. Similarly we define pruning for other relational operators.

`empty` and `equal` do not change the set of valuation at all, they only query the set whether it is empty or equal to another set respectively.

4.2 SMT

Motivation

One possible set representation was described in [BBH14b]. The approach given was to represent a set of valuations as bitvector formulae. We chose this representation as our experimental SymDivine data representation due to several reasons. SMT representation was already a subject of research and its limits are known. Next reason is that we find it very flexible when experimenting with new techniques.

Data Representation

The state of SMT data representation is a quantifier-free first-order bitvector formula Ψ, called *path condition.* The set of (free) variables in Ψ is subset of $V \times \mathbb{N}$, where V is the set of program variables; we need multiple copies (the second component) of variables to code "generations" of variables – each new

assignment to a program variable uses fresh new formula variable (with new generation number).

Notation: Suppose we have a formula variable $(a,k) \in V \times \mathbb{N}$. Then we call it *k-th generation of the variable a*. For a formula Ψ we denote $y(a)$ the variable (a,k) such that (a,k) is subformula of Ψ and k is maximal ($y(a)$ is the most fresh copy of variable a in Ψ).

Formally, executing

- `implement`$_{op}$`(a, b, c)` on a state Ψ results in the formula
$$\Psi \wedge ((a, k+1) = y(b)\ op\ y(c)),$$
where $y(a) = (a,k)$, and
- `prune`$_{op}$(a,b) on a state Ψ results in the formula $\Psi \wedge (y(a)\ op\ y(b))$.

SMT data representation handles these operations quite effectively, they only slightly modify the formula. (Compare for example to a BDD-based solution, creating a BDD that represents e.g. multiplication of two words is a very expensive operation.) The computationally expensive part of SMT data representation is hidden in the following operations.

- `empty`(Ψ) is true if and only if Ψ is not satisfiable
- `equal`(Ψ, Φ) is true for two states Φ and Ψ if and only if both
$$notsubseteq(\Phi, \Psi) \text{ and } notsubseteq(\Psi, \Phi)$$
are not satisfiable. $notsubseteq(\Phi, \Psi)$ is shortcut for
$$\Phi \wedge \forall (a_0, k_0), ..., (a_n, k_n)(\Psi \implies \bigvee((a_i, y(i)) \neq (b_i, y(i)))),$$
where $(a_0, k_0), ..., (a_n, k_n)$ are all variables used in Ψ and a_i and b_i iterate through all program variables of Φ or Ψ respectively. Intuitively, $notsubseteq(\Phi, \Psi)$ is true for such variable valuations in Φ for which there is no valuation in Ψ that agrees on the values of the most fresh variables.

Satisfiability of a bitvector formula can be decided, we use the Z3 SMT solver[DMB08]. Calls to SMT solver are in most cases computationally expensive, deciding satisfiability of a quantifier-free formula (in the `empty` check) is usually cheaper than satisfiability of a quantified formula (in the `equal` check). In contrary, these operations are both constant with a suitable BDD-based representation, but building a BDD that represents e.g. multiplication of two words is expensive.

Discussion

The SMT data representation allows fast arithmetic and relational operations, because formula construction is relatively cheap to compute. This representation may be seen as *lazy* in a sense that the representation internally keeps information about relations among inputs and variables in logical formulae and the hard part (empty and equal checking) is offloaded to a SMT solver in the time when needed (i.e. when `equal` or `empty` are called).

SMT data representation is transparent in a sense that the structure of the logical formula representing a state corresponds to the "history" of the state (i.e. which sequence of operations has created it). Moreover, formulae allow simple analysis and manipulation, therefore we have proposed and implemented few optimizations to the SMT data representation to improve its performance.

4.2.1 Optimizations

- **Simplification**

Motivation: Suppose that during a longer SymDivine's execution the SMT representation create a state represented by a complex formula Ψ. SymDivine may compare it to multiple other formulas for equivalence. Ψ may be equivallent to some other formula Φ that would allow simple equality matching.
Optimization: Each time new formula is created we call the SMT solver to simplify it first (Z3 offers the `ctx-solver-simplify` tactics or its lighter version `ctx-simplify` that applies only few simple rewrite rules to the formula).
Expected result: We want to improve the time SMT solver consumes for the queries by precomputation.

- **Unused Definitions**

Motivation: Lots of variables are created and destroyed during the lifetime of a LLVM program, mostly variables corresponding to registers with each call to or return from a function. The content of these variables will not be used in the future unless their value is "propagated" outside the function's scope (i.e. returned from function or stored through pointer).
Optimization: Most variables have a *definition* in the formula. As definition of a variable (a, k) we consider the subformula that is conjuncted to the formula while executing $\texttt{implement}_{op}$ call; e.g. after $\texttt{implement}_{+}\texttt{(a,b,c)}$

SMT data representation adds conjunct of form $(a, k) = (b, l) + (c, m)$ to the formula that represents the state. This subformula is then considered as the definition of (a, k).

When a variable (a, k) is destroyed (by `eraseSegment`) and $(a, k) = \varphi$ is its definition, we search for each occurence of (a, k) in the formula and substitute it for φ.
Expected result: The formula contains less variables and therefore the satisfying assignment or quantification is smaller in the `equal` and `empty` calls which should shorten the times the SMT solver needs to decide satisfiability.

- **Syntactical Equivalence and Canonization**

Motivation: Queries to the SMT solver are a performance bottleneck in the overall successor generation process, even if the queries are relatively simple. By a quick experiment with the Z3 SMT solver we have found out that the upper limit Z3 is able to handle is less than one thousand queries per second[1]. It is therefore desirable to identify simple queries and solve them without quering the SMT solver. When analyzing parallel programs, it often happens that the SMT data representation compares two states represented by the same formula due to "diamonds" in the state space.
Optimization: When comparing two states in the `equal` call, we try a syntactic equality of the formulae first. We also use few rewrite rules to enlarge the number of cases of syntactically equal formulae (e.g. variable renaming).
Expected result: Syntactic check is *very* effective, hence the `equal` check should run few orders of magnitude faster in case of states with syntactically equal states.

1. We implemented a program in C++ that repeatedly queried Z3 whether formula $true$ or $false$ is satisfiable; Z3 was linked to the program as a library and called via its C++ interface. The experiment was performed on a personal laptop with 1.9 GHz AMD processor and compiled with GCC-4.7.3 with maximal optimization level.

5 Program Analysis with SymDivine

In this chapter we give some examples how can be SymDivine utilized in some program analysis algorithms. To summarize the previous chapter, SymDivine provides the following functions, on top of which we build the algorithms.

- `initial()` – initial state computation
- `successors(S)` – set of all successors for a given state S
- equality of two states

5.1 Reachability

The first algorithm is the most straightforward one. Equipped with the `initial` and `successors` functions we have an implicit state space description that we use for breadth-first search-like exploration. The algoritm search for a state that is erroneous – SymDivine marks as erroneous states that executed the `assert` function with zero as argument.

The exploration keeps the set of visited states. Each time a state is encountered, it is checked againts the visited set whether the state is new or not. Since the data part of state may neither have canonical representation nor be orderable, we are left with the only option – to check semantical equivalence of the data part of state against each state that has the control part (call stacks and explicable variables, i.e. part of the state that is stored explicitely) of state the same. (In the implementation we do not use the exact state equality in the "visited" check, but we use *subsumption* instead, i.e. we check whether the set of visited states contains some state with equal control part and larger set of valuations in the data part).

In case of the SMT data representation, satisfiability of all quantified bitvector formulae can be decided, at least in theory, however, for practical verification of real programs some queries are very hard to compute, which sometimes causes the reachability not to terminate in a reasonable time.

Our initial motivation included LTL model checking of LLVM IR parallel programs. We implemented a facility to query SymDivine whether a particular global variable in a state contains nonzero value, which should enable to build a LTL model checking algorithm on the top of SymDivine. Since the range of tools that enable LTL model checking of LLVM IR is very limited, we have not spent any effort to examining the capabilites of our approach in this direction as we would not have an opportunity to a comparison.

Algorithm 2: A reachability algorithm. Assumes SymDivine to be properly initialized with the given LLVM IR module. Explores the state space and searches for errors.

```
begin Reachability()
    to_do := stack();
    to_do.push(initial());
    visited := {initial()};
    while to_do not empty do
        S := to_do.pop();
        foreach S' ∈ successors(S) do
            if S' ∉ visited then
                visited := visited ∪ {S'};
                if S' is an erroneous state then
                    return (Unsafe, S');
                end
            end
        end
    end
    return Safe;
end
```

5.2 Symbolic Execution

Another approach that can be build on the top of SymDivine is symbolic execution. In this algorithm, we explore all paths separately, we proceed in a breadth-first manner in order not to “fall” into some infinite or at least very long path.

As the readed might notice, this algorithm is very similar to the previous one (reachability), the only difference is that symbolic execution does not maintain the set of visited states, which means that symbolic execution does not check equality of two states. The benefit of this fact is that equality check are usually very hard and sometimes do not terminate in a reasonable time. In other hand, as some paths may contain same state, symbolic execution continues in exploring both paths independently, whereas reachability detects that the state was already visited. This particulary helps in case of parallel programs that usually generate state space with high average branching degree due to interleaving.

Algorithm 3: Symbolic execution. Explores all possible paths in the program and searches for errors on each path.

```
begin SymbolicExecution()
    to_do := stack();
    to_do.push(initial());
    while ¬to_do.empty() do
        S = to_do.pop();
        foreach S′ ∈ successors(S) do
            if S′ is an erroneous state then
                return (Unsafe, S′);
            end
            else
                to_do.push(S′);
            end
        end
    end
    return Safe;
end
```

5.3 Symbolic Execution with Limited Equality Check

The drawbacks of the previous two program analyses (hard equality checking for reachability, path explosion for symbolic execution) served as an impulse for us to find some hybrid solution that takes the better from both approaches.

While experimenting with SymDivine we have observed that lots of equal checks are very easy to compute. This is due to state space shapes in parallel programs, where lots of interleavings may result in the same valuation of variables. For SMT data representation with few optimizations described in the Section 4.2.1 our experiments showed that lots of equality queries on some models can be solved even without calling a SMT solver, only by comparing them syntactically.

This observation led us to a hybrid algorithm combining reachability and symbolic execution. We keep maintaining the set of visited states, but instead of checking exact equality against all visited states, we limit the check to simple instances only – in the SMT case, for example, we can limit the equal check to check syntactical equality only. (In the implementation we went one step further: if the formulae are not syntactically equal, we let the SMT

Algorithm 4:

```
begin SymbolicExecutionWithLimitedEquality()
    to_do := stack();
    to_do.push(initial());
    visited := {initial()};
    while to_do not empty do
        S := to_do.pop();
        foreach S' ∈ successors(S) do
            if S' is not synctically equal to any state from visited then
                visited := visited ∪ {S'};
                if S' is an erroneous state then
                    return (Unsafe, S');
                end
            end
        end
    end
    return Safe;
end
```

solver to try to prove their equivalence for some small amount of time. The time limit is increasing exponentially with each unsolved instance).

This approach lies between reachability and symbolic execution – if a newly discovered state is same as some already visited state, we may recognize it and save some time that would symbolic execution spend exploring it. On the other hand, if equality of the new state and the visited is unrecognized, we can proceed and possibly find an error in further exploration, although wasting some effort on exploring equivallent states. The drawback of this approach is its memory consumption; it has to maintain the *visited* set, but symbolic execution maintains only the *to_do* stack. Moreover, some states may have some equivallent "copies" in this set. This algorithm is similar to symbolic execution presented in [XMSN05] and [APV06], where a conservative approximation of subsumption is used to prune the state space.

6 Experiments

6.1 Motivation

The main contribution of this thesis is SymDivine, a generic platform for program analysis of LLVM IR. To experimentally validate our work we implemented the Symbolic execution with limited equality check algorithm with the SMT data representation and used it for a limited set of benchmarks from *Competition on Software Verification (SV-COMP)*[1][Bey14]. The reason why we have chosen SV-COMP is that the competition offers a large set of benchmarks and the community of program analysis tools participating in SV-COMP is very active.

We also considered different algorithms and data representations for SymDivine, but none was good enough to handle a benchmark set like SV-COMP, or there is virtually no reason for using it: regarding the choice of algorithm, we have also experimented with pure symbolic execution and pure reachability, but in vast majority of verification tasks it is at most as good as the setting we chose. Pure reachability sometimes simply fails when proving equality of states, but the algorithm 4 continues and can possibly find an error. Pure symbolic execution has an advantage of not storing the set of visited states, however, the overhead induced by doing so is much lower then the time spent on successor generation (mainly in the emptiness check), therefore a state lookup in the visited set when the equality check is effective can save some computation time by not exploring some already visited paths. We also tried to use BDDs instead of formulae to represent data, but it failed even on small examples; SymDivine needs to store lots of variables during its computation and hence the number of nodes in BDDs increased too fast.

Unfortunately, SymDivine does not support full C standard yet; mainly it lacks support for heap memory allocations, we cannot evaluate it on the full benchmark yet. A large part of the overall score measures how complete the tool is in supporting C programs, but since we have described the limitations of input program, it would be superfluous to measure it again.

We have selected benchmarks that are accepted by SymDivine and that are diverse enough to pinpoint strong and weak parts of our program analysis approach. We measured the number of instances we can check for reachability of an error label in a limited time and memory. For a reference we run the same set of benchmarks in the same setting with CPAchecker-1.3.4[BK11].

---

1. Benchmarks, results, rules and other informations about SV-COMP can be found at `http://sv-comp.sosy-lab.org/2014/`

Unfortunately, CPAchecker currently does not support threads and recursion (whereas SymDivine does). Threads are supported by CBMC, ESBMC, CSeq-Lazy and CSeq-Mu, but a comparison with a bounded model checking approach may be misleading since bounded model checkers usually "guess" the answer when the problem is too hard (i.e. when the bounds are insufficient), whereas when our tool gives an answer, it is always correct, and CPAchecker has the same property. In the last SV-COMP[Bey14], CPAchecker won silver medal in the overall category, and it was the best non-bounded model checking-based tool in the competition.

6.2 Setting

6.2.1 Testing Environment

We ran all benchmarks on the Intel Xeon 2.266 GHz processor, each verification task was allocated one processor core and 15 minutes of processor time and 15 gigabytes of memory. CPAchecker takes as input C source files directly, our tool needs the C code to be compiled into LLVM IR, which we generated using clang with the -O3 flag to turn on the highest level of optimization. Since we compiled all benchmarks into LLVM IR before verification, we did not include the compilation time in the overall time limit.

We encountered a few problems with the benchmark set and the competition rules, that forces us to use preprocessing steps before compiling C to LLVM IR:

- The error state is marked by the `ERROR` label. The label identifier may be lost in the translation to LLVM IR, therefore we have to add `assert(0);` right after the label to be able to detect an error on the LLVM IR level.
- Some benchmarks contain undefined behavior and it should be interpreted as nondeterminism according to the competition rules. But in case of undefined behavior clang and its optimizer is free to optimize such behaviors, for example:
  - *undefined variables:* may have arbitrary value, but clang initializes them to zero in compiled IR in some cases
  - *non-volatile shared variables:* changes to such variables may or may not be visible to other threads, optimizers usually makes them local

We initialized all undefined variables to an input value and added a `volatile` keyword to all global variables in multithreaded tests.

6.2.2 Benchmark Selection

We selected 386 benchmarks from the total of 2952 benchmarks available at the SV-COMP SVN repository[2]. We excluded following benchmarks:

- directories `ddv-machzwd`, `heap-manipulation`, `list-*`, `memsafety*`, `ntdrivers*`, `product-lines`, `ssh` (767 benchmarks) that use memory allocated on heap
- directories `ldv-*` (1529 benchmarks) for calls to undefined functions and heap memory allocations via `kmalloc` in over half of instances
- directory `pthread-ext` (45 benchmarks) for unbounded number of threads (the main function creates theads in infinite loop)
- directories `seq-mthreaded, seq-pthreaded` (199 benchmarks) that contain sequentialized versions of parallel programs
- few other benchmarks from various categories that allocate heap memory (26 benchmarks)

6.2.3 Results

The comparison of SymDivine and CPAchecker on the benchmarks is captured in the table 6.1. Since we used newer version of CPAchecker than the version that participated in the competition, CPAchecker solved more instances than in the competition, although we ran all tests on slower processor than the `Intel i7-2600` processor used in the competition. We used the newer version of CPAchecker in order to provide fair comparison, since the benchmarks were available when we had been working on the implementation that gave us an opportunity to find weaknesses of our implementation. The version of CPAchecker that participated in the competition did not have the same advantage.

Overall, SymDivine solved 48 % of all benchmarks with error location reachable and 46 % of error-free benchmarks, CPAchecker solved 66 % of erroneous benchmarks and 82 % of error-free benchmarks. The reason why SymDivine was unable to solve an instance was in 70 % of benchmarks caused by the time limit, in the rest by the memory limit. Although the

2. `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14/`

	with error			safe		
	total	SymDivine	CPA	total	SymDivine	CPA
bitvector/*	9	8	9	36	15	32
eca/*	89	30	33	44	14	32
locks/*	2	2	2	11	11	11
loops/*	32	21	27	35	24	27
ssh-simp/*	12	8	12	14	7	14
systemc/*	37	17	37	25	5	19
sum	181	86	120	165	76	135

Table 6.1: Number of instances solved by SymDivine and CPAchecker.

	with error		safe	
	total	SymDivine	total	SymDivine
pthread/*	6	4	11	8
recursive/*	7	3	16	3
sum	27	11	13	7

Table 6.2: Number of instances solved by SymDivine in categories that are not supported by CPAchecker.

number of instances that SymDivine could not solve in the given time and memory is twice the number of instances CPAchecker could not solve, it shows that a straightforward algorithm build on the top of SymDivine is actually able to compete with a tool which is dedicated to a particular task: assertion-checking of sequential programs, and that has been developed over five years and tuned to perform well on SV-COMP tasks.

6.3 Weaknesses of Our Approach

Since the results provides us a very useful feedback about weak points of our approach we conclude the experimental evaluation by a brief overview of future directions that our approach may consider in order to increase the number of instances it can handle.

Language Features

The successor computation part of SymDivine does not handle full LLVM IR. Regarding the competition, the most restricting limitation is that SymDivine does not handle heap memory allocations. Supporting heap requires a change of memory model, as we discussed earlier 3.2.

Verification Performance Bottlenecks

Unrelated Code

Some benchmarks perform some complex computation and then the path to the error label is guarded by either completely unrelated condition (e.g. `loops/heavy_false.c`) or a condition that strongly restricts the values that were read from the input (e.g. `recursive/Ackermann03_true.c`).

Slicing[Wei84] can help with the simpler cases, when the program contains behavior completely unrelated to reachability of the error label. The big advantage of employing slicing into our verification scheme (as a preprocessing step) is that is may not require any large changes to SymDivine. Another method that could help in this direction is *abstraction*, which would, however, be harder to employ in our scheme. On the other side, CPAchecker's results show that it may be worth of consideration.

Wide State Spaces

Since we represent the control flow explicitely, a fastly branching control flow is a fundamental problem of our approach. For example, `pthread/fib_bench_*` are benchmarks where two threads add some value to a global variable in a cycle and the error label is reachable in one particular thread interleaving. Each interleaving generates different valuations of global variables, therefore every interleaving generates unique state. Since the number of all interleavings grows exponentially, we were not able to solve the larger `fib_bench_longer*` and `fib_bench_longest*` instances. (There was only one other test in the `pthread` directory that we were not able to solve with our approach. The cause for it was, again, a huge number of states generated by unrestricted interleaving.)

It seems that bounded model checking-based approaches handle such instances well (for example CBMC did solve all these instances in relatively short time), possibly because they represent control flow symbolically.

Deep State Spaces

Consider a program that increments a variable in some loop with high number of iterations, for example one million. A clever approach could possibly collapse the cycle into a single transition that would add million to the variable. SymDivine, instead, would go throught one million states to compute the state at the branch that exists the cycle. Although SymDivine's successor computation is relatively fast and can handle relatively long paths

when compared for example to bounded model checkers, it would improve the performance to somehow summarize the cycle.

To improve performance in this direction, we would have to employ some form of cycle summarization (for example [SST13]). Cycles with simple body are often optimized by compiler when compiling C to LLVM IR (this is one reason why SymDivine performs well in the `loops` benchmarks).

Complex Arithmetics

Analysis of programs that perform some complex arithmetic operations and then decide some properties of the computed values (e.g. `bitvector/jain*` benchmarks) is also a problem. This probably cannot be solved inside of our tool; in this direction we depend on advances in SMT solving.

Target Architecture

Most categories in SV-COMP assume 32bit architecture. SymDivine assumes 64bit architecture – pointers are 8 bytes long and in our experiments we generated LLVM IR files with clang targeting a 64bit architecture. We could save both space and time by allowing verification assuming 32bit architecture.

7 Conclusion

We designed and implemented a generic platform SymDivine that enables control explicit–data symbolic analysis of parallel programs given in LLVM intermediate representation. SymDivine employs few optimizations in its successor generation process to alleviate the impact of expensive data handling, such as τ-reduction and variable explicating. SymDivine is not designed as a single-purpose tool, it offers a clearly defined interface that allows use of different data representation and to implement a certain class a program analysis techniques.

To evaluate control explicit–data symbolic approach on some real benchmarks, we implemented hybrid symbolic execution-explicit exploration algorithm on the top of SymDivine and we tested it on the subset of SV-COMP benchmarks that we support. The results show that although CPAchecker, a mature and very sophisticated tool, solves at this moment substantially more instances than SymDivine, our approach is valid and promising.

A lot of work remain to be done ragarding SymDivine. Currently SymDivine does not support complete LLVM IR, and it cannot handle programs that allocate heap memory and symbolic pointers, which disables SymDivine to solve a large part of SV-COMP benchmarks and lots of C programs in general. SymDivine also opens a variety of program analysis methods that it allows to implement, possibly with use of our combination of SMT data representation and symbolic execution-like analysis.

Bibliography

[ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting Rid of Store-Buffers in TSO Analysis. In *CAV*, pages 99–115, 2011.

[APV06] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *SPIN*, pages 163–181. Springer, 2006.

[BB09] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 174–177. Springer-Verlag, 2009.

[BB13] Jiří Barnat and Petr Bauch. Control Explicit—Data Symbolic Model Checking: An Introduction. *CoRR*, abs/1303.7379, 2013.

[BBGK11] Sebastian Biallas, Jörg Brauer, Dominique Gückel, and Stefan Kowalewski. On-The-Fly Path Reduction. *ENTCS*, pages 3 – 16, 2011.

[BBH+13] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multi-threaded C & C++ Programs. In *Proceedings of Computer Aided Verification*, LNCS, pages 863–868, 2013.

[BBH14a] Jiří Barnat, Petr Bauch, and Vojtěch Havel. Model Checking Parallel Programs with Inputs. In *Proceedings of PDP 2014*, pages 756–759, Turin, 2014. IEEE Computer Society.

[BBH14b] Jiří Barnat, Petr Bauch, and Vojtěch Havel. Temporal Verification of Simulink Diagrams. In *Proceedings of High Assurance Systems Engineering*, pages 81–88, Miami, 2014. IEEE Computer Society.

[BBR12] Jiři Barnat, Luboš Brim, and Petr Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods*, Lecture Notes in Computer Science, pages 252–266. Springer Berlin Heidelberg, 2012.

[BCG+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.

[Bey14] Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In *Proceedings of TACAS 2014*, LNCS 8413. Springer-Verlag, Heidelberg, 2014.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, Proceedings of Computer Aided Verification, pages 184–190. Springer-Verlag, 2011.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 189–197. FMCAD Inc, 2010.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 168–176. Springer Berlin Heidelberg, 2004.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software*, TACAS/ETAPS, pages 337–340, 2008.

[FMS14] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the Theory of Arrays: memset, memcpy, and Beyond. In Ernie Cohen

and Andrey Rybalchenko, editors, *VSTTE '13*, Lecture Notes in Computer Science, pages 108–128. Springer-Verlag, 2014.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC–C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004.*, pages 75–86. IEEE, 2004.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[LvdPW11] Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-core LTSmin: Marrying modularity and scalability. In *NASA Formal Methods*, pages 506–511. Springer, 2011.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded Model Checking of C and C++; Programs Using a Compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 146–161. Springer-Verlag, 2012.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, pages 75–88, 2002.

[MRC+14] Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22 - (Competition Contribution). In *TACAS*, pages 405–407, 2014.

[NMRW02] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[RBB13] Petr Ročkai, Jiří Barnat, and Luboš Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In

NASA Formal Methods (NFM 2013), volume 7871 of *LNCS*, pages 1–15. Springer, 2013.

[SST13] Jiri Slaby, Jan Strejcek, and Marek Trtík. Compact symbolic execution. In *ATVA*, pages 193–207, 2013.

[TBS] Sarah Thompson, Guillaume Brat, and Karl Schimpf. The MCP Model Checker. *submitted to PEPM*, 8.

[vdB13] Freark van der Berg. Model checking LLVM using LTSmin. Master's thesis, University of Twente, 2013.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, pages 352–357, 1984.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *In TACAS*, pages 365–381, 2005.

A Content of the attached archive

- **symdivine/** directory contains the source code of SymDivine. We provided a makefile that builds the exactly same tool we used in experimental evaluation (`./reachability`). To compile SymDivine, it is necessary to have LLVM installed on the system. A source code of the Z3 SMT solver is packed with SymDivine and our makefile script should build it as well. If you want to build our tool with your system version of Z3, type `make reachability_without_z3` to link the executable to system Z3 library.

- **benchmarks/** directory contains the benchmarks in the form used for our experimental evaluation. We also provide the following bash scripts:

  - **final_tests/prepare.sh** substitutes the `ERROR:` label in all available C files with `ERROR: assert(0);`. The benchmarks in the archive are already processed; you do not need to run this script for them.
  - **final_tests/make_lls.sh** compiles all C files available in the current working directory into LLVM IR. The variable `$CFLAGS` is used to pass directives to the compiler (i.e. to build optimized LLVM IR files, just execute `CFLAGS=-O3 ./make_lls.sh`).
  - **final_tests/run_test.sh** executes our tool on a given LLVM IR benchmark. After running `./run_test.sh <benchmark>.ll` the script generates the following files:
    * `<benchmark>.ll.result` with the verification result (Safe/Unsafe/Unknown)
    * `<benchmark>.ll.output` with output from the tool
    * `<benchmark>.ll.time` with time (in seconds) consumed
  - **final_tests/run_all.sh** runs all tests available in current directory. Can be used to reproduce the experiments in this thesis (although on different machine, with different clang version the results may vary). It may take more than ten hours to finish for all benchmarks.