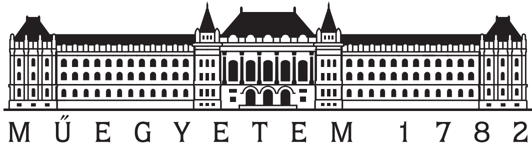


Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

LLVM IR-based Transformations for Software Model Checking

Master's Thesis

Author:

Gyula Sallai

Advisor:

Ákos Hajdu

2019

Contents

Kivonat i

Abstract ii

1 Introduction 1

2 Background 3
- 2.1 Program Representations 3
  - 2.1.1 Control Flow Graph Dominators 4
  - 2.1.2 Data Dependencies 5
  - 2.1.3 Program Dependence Graphs 8
  - 2.1.4 LLVM 9
- 2.2 Formal Verification 13
  - 2.2.1 Mathematical Logic in Model Checking 14
  - 2.2.2 The Model Checking Problem 14

3 LLVM for Model Checking 16
- 3.1 Instrumentation 16
  - 3.1.1 Protecting Undefined Values 17
  - 3.1.2 Checks 17
  - 3.1.3 Traceability Instrumentation 18
- 3.2 LLVM IR Transformations 18
  - 3.2.1 Lifting Error Calls 19
  - 3.2.2 Optimization Pipeline 21
- 3.3 Memory Models 22
  - 3.3.1 Memory SSA 23
  - 3.3.2 A Simple Memory Model 26
- 3.4 Transforming the LLVM IR into CFA 26
  - 3.4.1 Control Flow Automata within GAZER 27
  - 3.4.2 Transformation of Control Flow 27

3.4.3 Translation of Memory Instructions . . . 30
3.4.4 Optimizations during Transformation . . . 30

4 Verification **31**
4.1 Bounded Model Checking . . . 31
4.1.1 Inlining Techniques . . . 33
4.2 CEGAR-based Model Checking with THETA . . . 36
4.3 Traceability . . . 38
4.3.1 The Trace Format of GAZER . . . 38
4.3.2 From Counterexamples to Traces . . . 39
4.3.3 Executable Test Harnesses . . . 40

5 Implementation **42**
5.1 Architecture . . . 42
5.2 Technical Solutions . . . 42
5.2.1 Lifetime Management . . . 43
5.2.2 Expression Pattern Matching . . . 44
5.2.3 Expression Visitors . . . 45
5.3 Usage . . . 46

6 Evaluation **49**
6.1 Benchmark Environment . . . 49
6.2 Benchmark Results . . . 50
6.2.1 Evaluation of the BMC Backend . . . 51
6.2.2 Evaluation of the THETA Backend . . . 53
6.3 Summary . . . 53

7 Conclusion **55**

Acknowledgements **57**

List of Figures **58**

List of Tables **59**

Bibliography **64**

HALLGATÓI NYILATKOZAT

Alulírott *Sallai Gyula*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 22.

Sallai Gyula
hallgató

Kivonat

Ahogy a beágyazott rendszerek egyre inkább életünk szerves részévé válnak, biztonságos és hibamentes működésük egyre kritikusabb a felhasználók és a gyártók számára egyaránt. A modellellenőrzés egy hatékony technika, mely nem csak a hibák jelenlétét, hanem hiányát is képes bizonyítani, ezáltal egy kiváló eszköz biztonságkritikus rendszerek verifikációjához. Az elmúlt évtizedek hatalmas áttöréseket hoztak a modellellenőrzés területén, de ennek ellenére még mindig nem része az általános szoftverfejlesztési munkafolyamatnak. Ennek oka legtöbbször az elérhető modellellenőrző eszközök nehézkes használata.

Munkánkban bemutatunk egy felhasználóbarát és hatékony szoftver-modellellenőrző folyamatot, mely C programokat alakít át modellellenőrző-eszközök bemeneti nyelvére. A folyamat során a bemeneti programot különböző instrumentációs módszerekkel bővítjük, melynek segítségével a modellellenőrzés eredményét automatikusan vissza tudjuk vetíteni az eredeti forráskódra. Hogy elkerüljük a bemeneti nyelv teljes elemkészletének és lehetséges változatainak a feldolgozását, az LLVM fordítóprogram-infrastruktúra keretrendszert és annak a köztes nyelvét használjuk a leképezéshez. Az LLVM segítségével kiegészítjük a transzformációs folyamatot különböző beépített és saját optimalizációs algoritmusokkal. A memóriakezelést megvalósító utasítások leképezéséhez egy könnyen kiegészíthető memóriamodell keretrendszert definiálunk, melynek segítségével különböző előnyökkel és hátrányokkal rendelkező memóriamodellek implementálhatók.

A transzformációs folyamat végén kapott modelleket az általunk implementált korlátos modellellenőrző algoritmusunkkal vagy a THETA keretrendszer segítségével ellenőrizhetjük. Az implementált keretrendszerünk mindkét esetben felhasználóbarát és könnyen érthető visszavetítést biztosít az eredeti forráskódra, mellyel képesek vagyunk egy végrehajtható tesztkörnyezetet is generálni. A munkafolyamat működőképességét és hatékonyságát mérésekkel demonstráljuk.

Abstract

As embedded systems are becoming more and more common in our lives, the importance of their safe and fault-free operation is becoming even more critical. Model checking can prove both the presence and absence of certain errors in software systems, making it suitable for verifying safety-critical systems. The field of model checking has made a tremendous progress in the last decades, providing a great set of fast and effective algorithms. However, despite all this, model checking is still not widely adopted in a standard development workflow due to its poor accessibility.

In this work, we propose a user-friendly and efficient model checking workflow, that is able to translate C programs into the language of multiple model checking engines. The workflow uses a check instrumentation workflow to represent certain software properties as reachability problems and automatically map verification results back to the level of the original source code. In order to avoid translating from a possibly diverse set of input language dialects, we make use of the LLVM compiler infrastructure framework and its intermediate program representation. Furthermore, we shall use LLVM to extend the transformation workflow with a set of built-in and custom optimization transformations. To handle the ambiguity of memory-related language constructs, we propose a generic memory model interface that may be used to implement memory models of different strengths and trade-offs.

At the end of the workflow, the resulting models may be verified by our own implementation of the bounded model checking algorithm, or by using the CEGAR-based THETA model checking framework. We provide a traceability component, which is able to represent verification verdicts in a user-friendly format and also able to generate an executable mock environment that demonstrates found errors in an executable program. In addition, we provide benchmarks to demonstrate the usability of this workflow.

Chapter 1

Introduction

As our reliance upon safety-critical embedded software systems grows, so does our need for the ability to prove their fault-free behavior. Software model checking techniques can offer reliable proofs of the presence or absence of certain bugs in a computer system. These algorithms operate on mathematically precise formal models which describe the semantic behavior of the system under verification and are able to answer queries on its properties.

Despite the vast amount of research, efficient algorithms [11, 12, 36] and promising results [8], model checking techniques are still not widely present in a standard developer workflow. Developers expect efficient, user-friendly, and accessible tools, with easy-to-understand verification verdicts and as few false positives as possible [41, 16]. This signals the definite need for model checking tools that are able to reason efficiently about already implemented source code, yielding verification results in a human-readable format.

In this work, we present GAZER, a formal verification frontend, designed to leverage the power of model checking techniques in a user-friendly and traceable manner. We propose a source-to-model transformation workflow to translate imperative computer programs into formal models, which are then verified by model checking algorithms. We use traceability instrumentation both on the input program and the formal model to map the resulting verification verdict back to the original source code. This allows us to present a human-readable error message to developers or to automatically generate an executable test harness [31, 10] that can be used to understand and locate faults using widespread developer tools, such as debuggers.

Our source-to-model transformation uses the intermediate representation of the LLVM compiler infrastructure framework [44] to instrument, optimize, and translate the input program into our own program representation format, which will be translated to the language of a supported model checking engine. This offers us several benefits: by using an intermediate representation, we merely need to translate a small, well-defined instruction set instead of directly parsing diverse and often ambiguous input languages. This intermediate language is then instrumented with a set of so-called *checks* using our extensible check instrumentation framework with built-in traceability support. Furthermore, LLVM comes with an extensible optimization pipeline, which we can use to speed up verification by using LLVM's rich built-in optimization library and by implementing some additional transformations present in literature, such as program slicing [61] and assertion lifting [42].

As LLVM retains the ambiguity of memory operations due to (possible) pointer aliasing, the translation of memory-accessing instructions pose a challenge. Several *memory models*

have been proposed [15, 48, 35, 60] to disambiguate and formally represent program memory, each of them having their own advantages and shortcomings. Within GAZER, we define an extensible memory model interface and a toolchain that may be used to conveniently implement an arbitrary memory model. To demonstrate its applicability, we implement our own proof-of-concept *flat memory model*.

To ease the transformation step, we introduce an intermediate formal model, that is suitable to represent all instructions of the LLVM IR with formal semantics. This model is then translated further into the language of one of our supported verification backends (i.e. model checking engines), which in turn verify the formal model. One of these is our own implementation of the *bounded model checking* algorithm [12] combined with a state-of-the art inlining technique [43] and a small optimization of our own design. Another is the highly configurable CEGAR [18] framework, THETA [57, 36], developed at the FTSRG Research Group of Budapest University of Technology and Economics.

Related work. Popular software model checking tools such as CBMC [21] and CPACHECKER [11] have traditionally shipped with their own parsers and compilation processes. The BOOGIE verification language [45] provides a formalism which aims to be a program representation that could be a convenient input for a variety of model checking engines, such CORRAL [43] and VCC [22]. With the increasing popularity of LLVM, more and more model checking tools have chosen to adopt an LLVM frontend, such as LLBMC [48], SMACK [51] (which translates the LLVM intermediate representation to the language of BOOGIE and invokes CORRAL on it), and SEAHORN [34].

This thesis work is structured as follows. Chapter 2 offers some background information on program representations, dependency analyses, and model checking techniques. Chapter 3 describes the LLVM frontend of GAZER, whereas Chapter 4 presents its supported verification backends and possible error trace formats. Chapter 5 gives a short description about the architecture of GAZER with some implementation details and a user guide. Chapter 6 evaluates our workflow and measures the effects of certain transformation options. Finally, Chapter 7 concludes our work and lists possible directions for future development.

Chapter 2

Background

This chapter gives a brief introduction to the theoretical background of the algorithms and data structures presented later in this work. Section 2.1 gives an overview of some common program representations suitable for further analyses, whereas Section 2.2 lays the theoretical groundwork upon which over our verification solutions will be built.

2.1 Program Representations

Control flow graphs (CFG) [4] are language-agnostic intermediate representations of imperative computer programs. As such, they are a very important part of a compiler, and they are the main representation used for optimization and later code generation.

Definition 1 (Control flow graph). A *control flow graph* is a triple (S, E, s_0), where

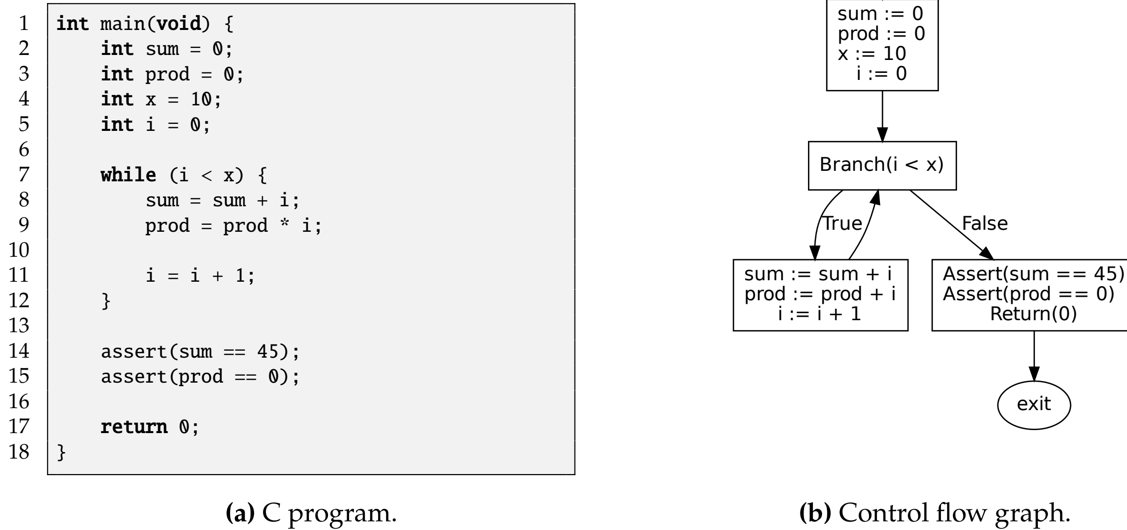
- $S = \{s_0, s_1, \dots, s_n\}$ is a set of program instructions,
- $E \subseteq S \times S$ is a set of directed edges, each representing a possible execution path in the program,
- $s_0 \in S$ is the entry point, from which all execution paths must begin. ▪

As there is a one-to-one correspondence between program instructions and CFG nodes, we shall use these terms interchangeably. For convenience, sometimes we shall refer to an "invisible" *exit node* s_q, at which all execution paths terminate.

In order to reduce the graph's size and to create a safe window for local optimizations, it is usual to merge linear instructions into a single entity, called the *basic block*. A basic block may only contain sequential instructions and has only one entry point (the first instruction) and one exit point (the last instruction). However, it may have multiple predecessors and successors, and may even be its own successor. Therefore a control flow graph can also be defined in the terms of basic blocks, where the CFG is a triple $(\mathcal{B}, E, B_0)$, $\mathcal{B}$ is the set of basic blocks, E is a set of directed edges between them, and $B_0 \in \mathcal{B}$ is the entry block. Throughout this work, for a basic block B in a control flow graph, the notation $\mathsf{pred}(B)$ marks the set of B's predecessors, $\mathsf{succ}(B)$ marks the set of B's successors.

Example 1. *An example is shown in Figure 2.1. Figure 2.1a shows a simple C program, Figure 2.1b shows its corresponding control flow graph. Notice the edges on the labels: in many cases, it is useful to augment the edges, and store information on the reason why a particular control path*

was chosen. In this case, the labels show how a path will be taken depending on the result of the branching comparison.



(a) C program.

(b) Control flow graph.

Figure 2.1: An example control flow graph.

2.1.1 Control Flow Graph Dominators

It may not be a trivial task to identify the control structures present within a control flow graph, as it requires the recognition of several patterns, which may span through multiple blocks. Luckily, the theory of flow graph dominators offers structures for easier recognition of loops and branches [2]. The dominator relation shows us which instructions (nodes) will always be executed before reaching a certain point of the program. This information will later be used to find useful dependency relations.

Definition 2 (Dominator). Let s and t be two nodes of a control flow graph. The node s *dominates* t (denoted as $s \text{ dom } t$) if all paths from the entry node to t contains s. If $s \neq t$ then s *strictly dominates* t. If every other dominator of t dominates s, then s *immediately dominates* t ($s \text{ idom } t$). ▪

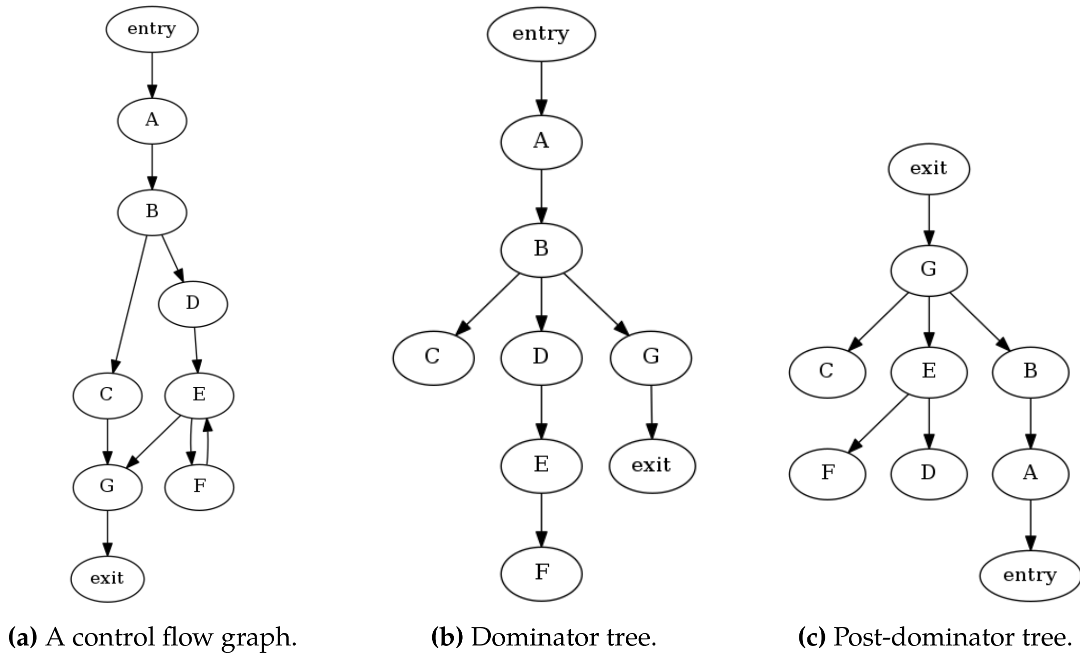
As it can be observed, according to this definition every node dominates (but not strictly dominates) itself, and the entry node dominates all nodes of the graph. If we wish to determine which instructions depend on the execution of a particular branch, we also need to introduce the following definition.

Definition 3 (Post-dominator). Let s and t be two nodes of a flow graph containing a single exit location. The node s *post-dominates* t (denoted as $s \text{ pdom } t$) if all paths from t to the exit node contains s. If $s \neq t$ then s *strictly post-dominates* t. If every other post-dominator of t dominates s, then s *immediately post-dominates* t ($s \text{ ipdom } t$). ▪

As every node can have at most only one immediate (post-) dominator, we can build a tree structure in which a node's parent will be its immediate (post-) dominator. This tree

is called the *dominator tree*. The dominator tree allows easy and efficient queries on the dominator information.

Example 2. *Consider the flow graph shown in Figure 2.2a. Figure 2.2b shows its dominator tree, Figure 2.2c shows its post-dominator tree. As it can be observed, if node Q is a descendant of node P, in the (post-) dominator tree, then P dom Q (P pdom Q) holds.*



(a) A control flow graph. **(b)** Dominator tree. **(c)** Post-dominator tree.

Figure 2.2: An example control flow graph and its dominator and post-dominator trees.

Dominance also allows us to see the range of influence of a particular instruction and the point at which this influence ends. In order to do this, we need the definition of the dominance frontier.

Definition 4 (Dominance frontier). Let s be a node in a flow graph. The *dominance frontier* of s (denoted as $\mathsf{DF}(s)$) is the set of all nodes w such that s dominates a predecessor p of w, but s does not strictly dominate w. ▪

The dominance frontiers $\mathsf{DF}(s)$ of s tells us the points at which the dominance of s stops. As an example, the dominance frontier of node D in Figure 2.2a is $\mathsf{DF}(D) = \{G\}$.

2.1.2 Data Dependencies

During program analysis, it is often useful to know whether an instruction writes variables that are later read by another instruction. This information is captured by the notion of *data dependency* [2].

Definition 5 (Definition, use). Let $\mathcal{P}$ be a program with I being its set of instructions. Let v be a variable within $\mathcal{P}$. Let $s, t \in I$ be instructions in $\mathcal{P}$.

1. The instruction s *unambiguously defines* the variable v if s explicitly assigns a value to v. The instruction s *ambiguously defines* the variable v if s is an operation which may or may not modify the value v.

2. A *definition* is a pair $d = (s, v)$ where s is an instruction and v is a variable ambiguously or unambiguously defined by s. In that case s *generates* d.
3. An *use* is a pair $u = (t, v)$ such that t reads the value of v. ▪

In order to reason about the data dependencies of a particular instruction, we need to find out which other instructions may have observable effects on it. In general, an instruction s may affect another instruction t if it defines a variable v which t uses. As a variable may be defined (i.e. assigned to) multiple times in a program, it is important to know which is the definition we should take into account when analyzing the uses of v.

Definition 6 (Reaching definition). An instruction t *kills* the definition $d = (s, v)$, if $t \neq s \in I$ and t unambiguously defines v. Given a node $x \neq s$, d is a *reaching definition* for x, if there is a control flow path between s and x, where d is not killed along that path, in which case d is said to be *alive* at x. If s has a reaching definition for x, then x is said to be *data dependent* on s. ▪

Calculating the reaching definitions for a given instruction yields the variable assignments which have direct impact on it. This information can be organized into a structure, which contains every reaching definition of every use in our program. Such structure is called a use-define chain [2].

Definition 7 (Use-define chain). Let $\mathcal{P}$ be a program. Let $D = \{d_1, d_2, \dots, d_k\}$ be the set of $\mathcal{P}$'s definitions, and $U = \{u_1, u_2, \dots, u_n\}$ the set of its uses. The *use-define chain* of $\mathcal{P}$ is a set of pairs $\{(u_1, D_1), (u_2, D_2), \dots, (u_n, D_n)\}$, where $D_i \subseteq D$ is the set of definitions reaching u_i. ▪

Example 3. *Consider the CFG describing a code snippet shown in Figure 2.3a and its corresponding CFG in Figure 2.3b. The definition set of the snippet is* $D = \{(i_1, x), (i_2, y), (i_4, y), (i_5, u)\}$. *The use set is* $U = \{(i_2, x), (i_3, x), (i_5, x), (i_5, y)\}$. *The use-define information of Figure 2.3b is*

$$\begin{aligned}(i_2, x) &\rightarrow \{(i_1, x)\},\\ (i_3, x) &\rightarrow \{(i_1, x)\},\\ (i_5, x) &\rightarrow \{(i_1, x)\},\\ (i_5, y) &\rightarrow \{(i_2, y), (i_4, y)\}.\end{aligned}$$

Static Single Assignment

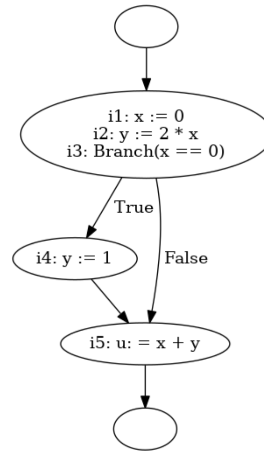
While use-define chains can be calculated using data-flow equations, most modern compilers encode programs in a way that shows this information explicitly. The *static single assignment* (SSA) form [53] does this by making sure that each program variable is assigned only once. If a variable (for example x) is assigned multiple times in the original program, then it is broken up to several variables ($x_0, x_1, \dots$), one for each assignment.

The main benefit of SSA-formed programs is that every use of a variable has exactly one reaching definition, thus they explicitly show use-define information. This removes the need to compute use-define chains separately (at the cost of a larger program representation and a less closer resemblance to the original program).

However, an issue arises at the merging point of divergent code paths. As each path may contain a different definition of a given variable, we need to know which definition to use at the merging point. For this purpose, the concept of ϕ*-nodes* was introduced.

```
1  int main(void) {
2      int x = 0;
3      int y = 2 * x;
4      if (x == 0) {
5          y = 1;
6      }
7
8      int u = x + y;
9  }
```

(a) C code snippet.



(b) Control flow graph.

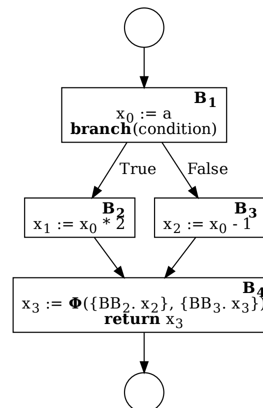
Figure 2.3: Use-define chains example.

Definition 8 (ϕ-node). Given the uses of a variable x in a basic block B with the set of possible reaching definitions $\{x_1, \ldots, x_n\}$, the ϕ-*node* for x is a $\phi_x : \mathsf{pred}(B) \rightarrow \{x_1, \ldots, x_n\}$ function. In other words, the ϕ-node selects a single definition x_i depending on the control flow paths the program took to reach B. ▪

Example 4. *Figure 2.4a shows a simple C program with a diverging code path and a merging point just before the return instruction. Figure 2.4b shows the SSA-formed CFG of this program. Note that the definitions of the variable x have received different integer subscripts to make each definition unique. The ϕ-node of B_4 selects the correct definition of the return value from the available definitions.*

```
1  int fn(int a, int cond)
2  {
3      int x = a;
4      if (cond) {
5          x = x * 2;
6      } else {
7          x = x - 2;
8      }
9
10     return x;
11 }
```

(a) A C program.



(b) SSA-formed CFG.

Figure 2.4: SSA transformation of a simple C program.

Standard SSA construction algorithms use dominance frontiers to determine the placement of ϕ-nodes. SSA requires each definition d to dominate all of its uses. This property is trivially satisfied for sequential paths. Definition d of some variable v dominates all uses of v until it is killed by another definition d', at which point d' will dominate all uses along said path. This property is satisfied until a merging point of divergent control flow

is reached, therefore this is the place where we should put the ϕ-nodes. As this is also the point at which the dominance of definition d stops, it is found in the dominance frontier of d, $\mathsf{DF}(d)$.

After the ϕ-nodes are placed, the algorithm performs a renaming pass, which renames all uses of a variable v to the proper form. The standard SSA renaming algorithm [56], presented in Algorithm 1, works by doing a preorder walk over the dominator tree. When processing a block, the algorithm keeps track of the current dominating name of each variable x using a stack ($S[x]$) and a counter ($C[x]$) per variable. When a new name is required for x, this new name is acquired by incrementing the counter $C[x]$ and adding its value as a subscript for x. This new name then gets pushed onto the stack of x, $S[x]$.

In each block, we first insert new names for the values determined by the ϕ-nodes. Then, we process each instruction of a block – all uses of x are renamed so they refer to the top element of $S[x]$, whereas definitions push a new name to the top of $S[x]$. In the next step, the algorithm fills in all the uses in the ϕ-nodes of successor blocks, so they will use the top of $S[x]$ in the incoming value list. Then, in another pass, we recursively call the rename procedure on each successor of B in the dominator tree. Lastly, we pop all newly introduced names from the name stack of each variable.

Algorithm 1: Standard SSA renaming algorithm.

```
1  Function NewName(x)
2  │  i := C[x]
3  │  push x_{i+1} onto S[x]
4  │  C[x] := i + 1
5  │  return "x_{i+1}"
1  Procedure Rename(B)
2  │  foreach φ-node in B, x ← φ(...) do
3  │  │  rename x as NewName(x)
4  │  end
5  │  foreach assignment in B, x ← y op z do
6  │  │  rewrite y as top(S[y])
7  │  │  rewrite z as top(S[z])
8  │  │  rewrite x as NewName(x)
9  │  end
10 │  foreach successor B′ in succ(B) do
11 │  │  fill in φ-node parameters of B′ w.r.t. S
12 │  end
13 │  foreach successor B_succ of B in the dominator tree do
14 │  │  Rename(B_succ)
15 │  end
16 │  foreach definitions in B, x ← φ(...) and x ← y op z do
17 │  │  pop(S[x])
18 │  end
```

2.1.3 Program Dependence Graphs

A *program dependence graph* [29] is a program representation which explicitly shows data and control dependency relations between two nodes in a control flow graph. The control dependencies show if a branch decision in a node affects whether another instruction gets

executed or not, and data dependencies tell us which computations must be done in order to have all required arguments of an instruction. Program dependence graphs are constructs which allow easy and efficient querying on these properties. In order to formalize the notion of control dependency, we shall define it using the theory of flow graph dominators, described in Section 2.1.1.

Definition 9 (Control dependency). Let s and t be nodes in a flow graph. t is *control dependent* on s if and only if:

- there exists a directed path P from s to t with any $u \in P$ $(u \neq s, u \neq t)$ post-dominated by t $(t \text{ pdom } u)$,
- s is not post-dominated by t. ▪

A program dependence graph is the result of the union of a *control dependence graph* and a *flow dependence graph*. Control dependence graphs contain the edge $s \rightarrow t$ if t is control dependent on s, whereas flow dependence graphs are merely the graph representations of the use-define information described in Section 2.1.2, therefore they contain the edge $x \rightarrow y$ if y is flow dependent on x.

Definition 10 (Program dependence graph). A *program dependence graph* is a 3-tuple (V, C, F), where

- $V = \{v_1, v_2, \dots v_n\}$ is a set of instructions,
- $C \subseteq V \times V$ is a set of control dependency edges and
- $F \subseteq V \times V$ is a set of flow dependency edges.

The edge $(v_i, v_j) \in C$ if v_j is control dependent on v_i. The edge $(v_m, v_u) \in F$ if v_u is flow dependent on v_m. ▪

Example 5. *The program dependence graph of the program shown in Figure 2.1a can be seen in Figure 2.5. Solid lines represent control dependency, dashed lines represent flow dependency.*

2.1.4 LLVM

LLVM [44] is a compiler infrastructure framework, which provides a language-agnostic intermediate program representation (the *LLVM IR*), multiple optimization algorithms, and has a frontend for several programming languages, such as C, C++, Fortran, Swift, or Rust. In addition, LLVM offers support for a wide variety of backend architectures (such as x86, ARM, SPARC).

The LLVM IR is a powerful intermediate representation in the form of a typed, SSA-based, high-level instruction set in a three-address code [2] scheme. The representation is dual: while it is defined as an assembly-like language, it may also be viewed and manipulated as a control flow graph.

The highest-level entity of an LLVM IR program is the *module*, which corresponds to a compilation unit in the input program. A module may contain a set of global variables, functions, and auxiliary information such as metadata.

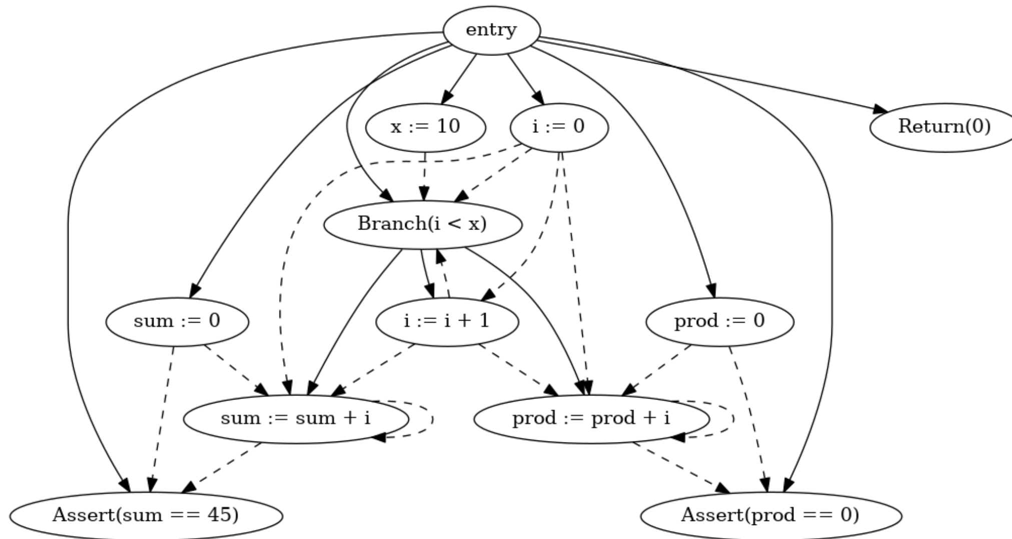


Figure 2.5: Program dependency graph of the program and CFG shown in Figure 2.1.

Functions are built from basic blocks, each block having a unique label, a set of sequential instructions and *terminator instruction* at the end. The terminator is either used to manipulate control flow (with instructions like **`br`** or **`switch`**), exit the function (**`ret`**) or mark unreachable code paths (**`unreachable`**). Instruction values and arguments are stored in so-called *registers*, which are required to be an SSA-form, thus they can be assigned to only once. The ϕ-nodes introduced by SSA are represented by the special **`phi`** instruction which must be at the beginning of the block.

Types and Operations

The LLVM IR is a typed language, providing *scalar types* such as integers and floating-point numbers. In addition, it provides *composite types* such as structures and arrays. Instructions are polymorphic, meaning that an instruction may work with several input types (e.g. an arithmetic instruction accepts different integer bit widths and SIMD vector types).

Integers are represented as bit-vectors of different bit widths (as an example, **`i32`** represents a 32-bit integer). LLVM has no knowledge about the sign of an integer value – it merely stores bit patterns and different instructions may be used to interpret values as signed or unsigned. As an example, **`sdiv`** divides two integers as signed values, while **`udiv`** interprets them as unsigned before the performing the division. Simpler arithmetic operations (**`add`**, **`sub`**, **`mul`**) do not have this distinction as two's complement arithmetic is the same for signed and unsigned values in their case.

The floating-point types supported by LLVM are the ones defined in the IEEE-754 [1] standard: **`half`**, **`float`**, **`double`**, **`fp128`**, representing 16-bit, 32-bit, 64-bit, and 128-bit floating-point types, respectively. Furthermore, there is support for the non-standard types **`x86_fp80`** and **`ppc_fp128`**, used by Intel x86 and PowerPC processor architectures.

The composite types most relevant to this work are arrays (for example, **`[10 x i32]`** is an array of ten 32-bit integers) and structures (such as **`{ i32, float }`**, which is a pair of an

integer and a single precision float). LLVM also offers vector types for SIMD[1] processing, but vector types are out of the scope of this work.

In LLVM, pointers are syntactically similar to the pointer types of C. They are denoted with an asterisk: **i32*** is a pointer to a 32-bit integer. Pointer arithmetic is calculated using the **getelementptr** instruction, which is also used to calculate the offset of array or struct members.

Casts between types are handled through explicit cast instructions. Integer casts, such as **sext** or **zext** may be used to cast integer types to a larger bit width, either by sign-extending or zero-extending the original number. The instruction **trunc** may be used to truncate an integer value. Semantic transformations such as **inttoptr**, **pointertoint**, **fptoui**, **fptosi**, **uitofp**, **sitofp** cast values to an entirely new type. In addition, the **bitcast** instruction is used to reinterpret a given bit-vector as a value in another type.

Memory Accesses in LLVM

While registers are required to be in a strict SSA-form, this does not apply to the memory state. A memory access can either be a write to an arbitrary memory location given by a pointer (**store**) or a read from a location (**load**).[2]

Local variables which have their address taken cannot be represented by registers, thus this done by using the **alloca** instruction. **alloca** allocates memory in the local scope (the program stack in most implementations) for a particular type and returns a pointer to the allocated memory. This pointer may then be used by all memory-manipulating instructions as usual.

Example 6. *Consider the C program in Figure 2.6a. Figure 2.6b shows the program encoded in the LLVM IR, with some additional comments to help readability. Notice how the input variable information is lost in the SSA-formed LLVM IR. Complex operations are broken down into primitive instructions each of them doing one simple task, such as performing an addition, doing a comparison, calculating and offset, etc.*

One of the more notable instructions in the example is the ***phi*** *node present at the beginning of* ***bb2****. As it can be seen,* ***bb2*** *has two predecessors: the entry block* ***bb*** *and the latch (that is, the block which has edge for the next loop cycle of a loop). The ϕ-node selects the initial value of* i *(zero) if control comes from the entry block (i.e. this is the first loop iteration). Otherwise, it selects the incremented value from the previous iteration.*

The ***getelementptr*** *(*GEP *for short) instruction in line 18 calculates the memory offset which we will access.[3] Finally, we write the value we got from the call (****%tmp5****) to the offset identified by the pointer we obtained from the GEP instruction.*

Figure 2.6c also shows the LLVM IR function in its control flow graph representation.

[1]Single instruction, multiple data: a parallel execution scheme where a single process operates on multiple data at once.

[2]LLVM also offers the atomic read-modify-write (**atomicrmw**) and atomic compare-and-swap (**cmpxchg**) instructions to manipulate memory in parallel programs, but these are out of the scope of this work.

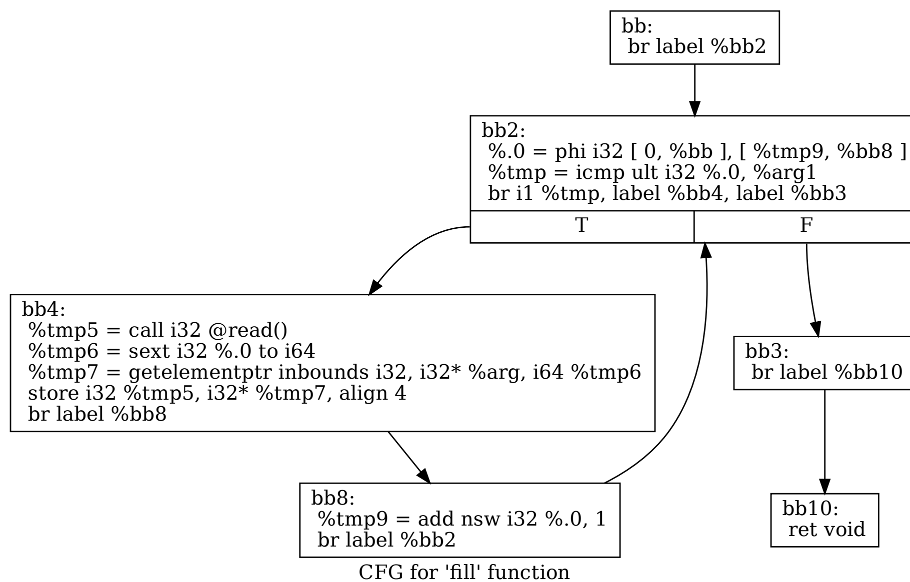
[3]Note that the **inbounds** flag does not indicate a check of the buffer size: it merely describes the instruction's behavior in the case of a pointer arithmetic overflow (i.e. what happens if we compute an address which is outside of the address space).

```c
1  extern int read();
2
3  void fill(int* t, int siz)
4  {
5      for (int i = 0; i < siz; ++i) {
6          t[i] = read();
7      }
8  }
```

(a) A C program with memory access.

```llvm
1   declare i32 @read()
2
3   define void @fill(i32* %arg, i32 %arg1) {
4   bb:
5       br label %bb2                                          ; unconditional jump to bb2
6
7   bb2:
8       %.0 = phi i32 [ 0, %bb ], [ %tmp9, %bb8 ]              ; i
9       %tmp = icmp slt i32 %.0, %arg1                         ; i < siz
10      br i1 %tmp, label %bb4, label %bb3                     ; conditional jump based on %tmp
11
12  bb3:
13      br label %bb10
14
15  bb4:
16      %tmp5 = call i32 @read()                               ; tmp5 = read()
17      %tmp6 = sext i32 %.0 to i64
18      %tmp7 = getelementptr inbounds i32, i32* %arg, i64 %tmp6    ; &(t[i])
19      store i32 %tmp5, i32* %tmp7, align 4                   ; t[i] = tmp5
20      br label %bb8
21
22  bb8:
23      %tmp9 = add nsw i32 %.0, 1                             ; i += 1
24      br label %bb2
25
26  bb10:
27      ret void
28  }
```

(b) The LLVM IR assembly of the program.



(c) The control flow graph representation.

Figure 2.6: An LLVM IR example.

Metadata and Debug Information

Within the LLVM IR, values (instructions, functions, modules, etc.) may have a set of *metadata* attached to them. They may be used to convey additional information about a particular value, which then may be used by the optimization passes or the code generator. The most prominent use of metadata (and the one we will concern ourselves with) is the tracking of source-level debug information. Debug metadata includes information about compilation units (`!DICompileUnit`), types (`!DIBasicType`, `!DIDerivedType`, `!DICompositeType`, etc.), functions (`!DISubprogram`), variables (`!DIGlobalVariable`, `!DILocalVariable`) and locations in the source program (`!DILocation`). Metadata nodes may have child nodes and refer to others (e.g. a variable metadata node may refer to the node about its type).

Run-time debug values are connected to the source-level variable metadata by so-called debug *intrinsic functions*, or intrinsics for short. Intrinsic functions are a set of built-in function declarations, that have some well-defined semantics within the compiler. These functions may not have bodies, and must only be referred to by function calls. Within LLVM, intrinsic functions are used to track debug information, insert assumptions for the code optimizer, represent some common idioms[4] and special instructions not supported by the LLVM IR.[5]

The most important intrinsic function for debugging is `llvm.dbg.value`, which takes three parameters: a register value (wrapped as a metadata node), a local variable metadata, and a complex expression which tells the debugger how to derive the actual value from the given register.

Example 7. *Figure 2.7 shows some of the metadata nodes defined for the program shown in Figure 2.6. Certain attributes have been removed in order to improve readability. As we can see, the metadata contains information about each variable (`!15`, `!16`, `!17`), and their types (`!12`, `!13`).*

Imagine line 8 in Figure 2.6b is followed by the debug intrinsic call:

```
call void @llvm.dbg.value(metadata i32 %.0, metadata !17,
    metadata !DIExpression()), !dbg !20
```

This call indicates that variable `i` (indicated by `!17`) has been assigned a new value (`%.0`) at line 5. The source location is attached as a location metadata to this call (`!20`), from which the exact line can be read out by following the scope information in `!18`).

2.2 Formal Verification

Model checking is a formal verification technique of mathematically proving correctness or faultiness of computer programs by systematically exploring their state space [20]. Model checking has two main requirements: a *formal model*, that is, a well-defined model of the system being checked, and a *formal property*, which must be proven about the model (e.g. an unsafe state cannot be reached). Model checking builds heavily upon mathematical logic, described in the following section.

[4] This includes intrinsics for the standard C library functions, such `llvm.memcpy`, `llvm.memset`, `llvm.sin`, etc. In certain cases, the optimizer may even recognize a custom memory setting loop and transform it into the corresponding intrinsic. Representing these basic library functions as intrinsics allows the optimizer to make more informed decisions about certain operations.

[5] As an example, LLVM has no built-in support for certain bit operations, such as byte swapping. However, if the target architecture provides an efficient instruction for byte swaps, the `llvm.bswap.*` intrinsic may be used to inform the code generator about this operation, so it can emit the proper CPU instructions.

```
!0 = distinct !DICompileUnit(language: DW_LANG_C99, file: !1)
!1 = !DIFile(filename: "example.c", directory: "/tmp")
!9 = distinct !DISubprogram(name: "fill", scope: !1, file: !1, line: 3, type: !10, scopeLine: 4, unit
    : !0, retainedNodes: !14)
!10 = !DISubroutineType(types: !11)
!11 = !{null, !12, !13}
!12 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !13, size: 64)
!13 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!14 = !{!15, !16, !17}
!15 = !DILocalVariable(name: "t", arg: 1, scope: !9, file: !1, line: 3, type: !12)
!16 = !DILocalVariable(name: "siz", arg: 2, scope: !9, file: !1, line: 3, type: !13)
!17 = !DILocalVariable(name: "i", scope: !18, file: !1, line: 5, type: !13)
!18 = distinct !DILexicalBlock(scope: !9, file: !1, line: 5, column: 2)
!20 = !DILocation(line: 0, scope: !18)
```

Figure 2.7: Some of the metadata defined for the program shown in Figure 2.6.

2.2.1 Mathematical Logic in Model Checking

Propositional logic is a branch of mathematical logic. The basic elements of the logic are *propositional variables* (e.g. P and Q). A *formula* φ is constructed from propositional variables and *logical connectives* such as $\top$ (*true*), $\bot$ (*false*), $\neg$ (*negation*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*) and $\rightarrow$ (*implication*). An *interpretation* assigns a truth value to every propositional variable.

The *boolean satisfiability problem* (SAT) is the problem of deciding whether a formula φ is satisfiable, i.e. whether there is an interpretation which yields that φ is true. Despite the problem's NP-completeness [23], modern SAT solvers in cases can handle models with millions of formulas [40]. However, describing a computer program using merely propositional variables and truth symbols is a rather complicated issue, resulting in impractically large models.

First order logic (FOL) extends propositional logic with *variables*, *function symbols*, *predicate symbols* and *quantifiers*. A first order interpretation is based on a *domain*, which is a set that may contain any abstract objects (such as numbers, animals or teapots). Function symbols are interpreted as functions over the domain, and predicate symbols are interpreted as relations over the domain. First order logic is undecidable in general [17, 58].

However, decidability can be achieved by semantically restricting first order logic to a class of interpretations. The *satisfiability modulo theories* (SMT) problem [13] is the problem of deciding whether a first order logic formula φ is satisfiable in a combination of certain (usually quantifier free) theories. In our case, the theory (and interpretation) used is the theory of integers, bit-vectors, arrays, and floating-point numbers. As an example, the theory of integers interprets $+$ as the well-known integer addition, $\leq$ as the usual total order over integers, etc., whereas the theory of bit-vectors defines these (and additional) operators in terms of bit-vectors. In addition to being decidable, these first order theories enable reasoning over data structures commonly used in computer programs (e.g. arrays), thus are convenient to describe program semantics.

2.2.2 The Model Checking Problem

There are several program representations suitable for model checking. While it is achievable to perform model checking using the well-known control flow graph formalism, most algorithms are better suited for automaton-like systems. Therefore we use an automaton-

like formalism which is able to represent program control flow in automata semantics, known as the control flow automaton. A *control flow automaton* (CFA) [9] is a 4-tuple (L, E, ℓ_0, ℓ_q) where

- $L = \{\ell_0, \ell_1, \ldots, \ell_n\}$ is a set of locations, representing values of the program counter,
- $E \subseteq L \times L$ is a set of *transitions* (edges), representing control flow,
- ℓ_0 is the distinguished entry location,
- ℓ_q is the designated exit location.

Performing a jump along the path described by a particular transition is called *firing*. The transition (ℓ_1, ℓ_2) is labeled with the *operations* which get executed when the control flow jumps from ℓ_1 to ℓ_2. Furthermore, transitions may have Boolean predicates called *guards*, which mean that a transition can fire only if the guard evaluates to true.

In this work, we shall use control flow automata as formal models to represent computer programs and a special *error location* ℓ_e to represent possible errors – if ℓ_e is a reachable location in a possible execution, then the program is considered faulty. Thus an instance of the *model checking problem* is a pair $(\mathcal{A}, \ell_e)$ where $\mathcal{A} = (L, E, \ell_0, \ell_q)$ is a control flow automaton and $\ell_e \in L$ is the designated *error* location. In our work, we use ℓ_e to represent erroneous behavior (failing assertions, division by zero, etc.) in a C program. The model cheking problem targets whether there exists an executable program path π from ℓ_0 to ℓ_e in $\mathcal{A}$. If such path exists, it is a counterexample to the property that can be reported to the programmer, otherwise the property holds.

Checking merely for the graph-theoretical reachability of the error location (i.e. not considering the semantics of the operations on the edges) yields false-positive results. It might be possible that a path $l_0 \implies l_e$ exists, however, it is not evident whether that erroneous control flow path can be executed for any inputs. Therefore model checking requires checking the semantic interpretation of the input CFA and only reporting a counterexample if it is actually a viable error path.

There are various solutions available for software model checking. Abstraction-based model checkers operate by "hiding" certain details of the program. If the abstraction fails to model the original system precisely (by hiding too much information), then the abstraction is refined, until it faithfully represents the original input model [18]. Several software verification tools use this approach, such as CPAChecker [11], Ultimate Automatizer [38], BLAST [39], SLAB [14], UFO [3], and theta [57]. Partial order reduction [32] is mostly used for verifying concurrent programs. If a program runs multiple threads whose paths to a certain state overlap in some order, then many of these orderings can be represented with only one of them. Abstract interpretation methods [24] prove the unreachability of a certain state (in our case, the error state), by iteratively extending an over-approximation of the set of reachable states. Bounded model checkers [12], such as CBMC [19], LLBMC [48], and Corral [43] use an approach which searches for error paths within a given maximal bound and iteratively extends this bound if no errors were found.

From the ones listed above, our work uses two types of model checking algorithms: bounded model checking and an abstraction-based one. Each of these are presented in greater detail in Chapter 4.

Chapter 3

LLVM for Model Checking

This chapter describes our use of LLVM as a frontend and optimization infrastructure for our software verification framework. In this section we propose a workflow that is able to transform C programs to control flow automata, using the LLVM IR as an intermediate step. In order to reduce the size of the resulting models, we enhance this process with compiler optimizations and program slicing. The resulting models can then be verified using an arbitrary verification algorithm.

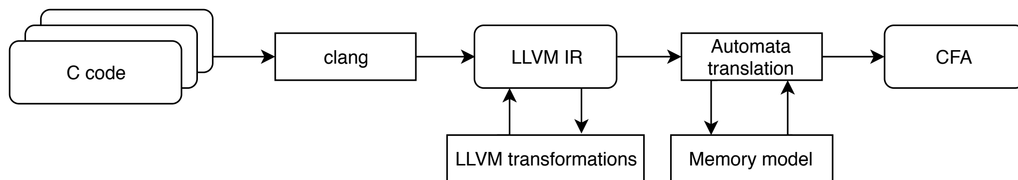


Figure 3.1: Transformation workflow.

An overview of the transformation workflow is shown in Figure 3.1. First, we take a set of C source code files as an input. These source files are then parsed into LLVM's control flow graph representation, the LLVM IR (Section 2.1.4) using the Clang[1] compiler. If multiple input files are supplied, we automatically link them together into a single LLVM IR module, using the `llvm-link` utility. The resulting CFG is then simplified by applying a set of LLVM IR transformations, including instrumentation (Section 3.1), built-in LLVM optimizations and custom transformations (Section 3.2). The instrumented and simplified program is then analyzed w.r.t. a memory model (Section 3.3) and finally translated into a control flow automaton (Section 3.4).

3.1 Instrumentation

Instrumentation is the process of injecting certain monitoring, measurement or traceability information into a program, without changing its functional behavior. In the context of GAZER, we shall use certain instrumentation passes to convey important information to later analyses (such as the presence of undefined behavior and unsafe states in the program) and help traceability.

[1] `http://clang.llvm.org/`

3.1.1 Protecting Undefined Values

Undefined values may occur in many places in a C program, most notably when reading uninitialized memory locations or when performing an operation that yields undefined behavior. LLVM encodes undefined values with a special **undef** constant, available for each LLVM type. Undefined values allow the optimizer to choose any bit-pattern for an undefined value, enabling optimizations which would not be legal otherwise. As we wish to optimize *then* verify the program, this may pose some problems.

As the compiler is free to pick any value for an **undef**, an aggressive optimization may choose to eliminate whole branches and basic blocks just by picking an arbitrary (but fixed) value for a nondeterministic branch. This also means that safety properties depending on undefined values may be removed as well. Note that these are valid and expected transformations, yet they may take away the soundness of our later analyses.

Observe the LLVM IR code shown in Figure 3.2a. An aggressive optimization may choose to pick the value of the **undef** in line 3 to be always zero, thus concluding that the subsequent branching instruction will always take the true path. This may result the complete removal of the assertion failure call, after which the verification algorithm would falsely return that the program is safe.

Our solution is to replace each **undef** operand in an instruction with the result of a nondeterministic function call to function **gazer.undef_value**. This protects the undefined value from being removed by further optimizations passes and will provide useful information when extracting traces from a counterexample (this will be discussed in Section 4.3 in more detail). Figure 3.2b shows the program after this transformation step.

```
1  define i32 @main() {
2  bb:
3    %tmp = icmp ne i32 undef, 0
4    br i1 %tmp, label %bb1, label %bb2
5
6  bb1:
7    br label %bb3
8
9  bb2:
10   call void @__assert_fail()
11   unreachable
12
13 bb3:
14   ret i32 0
15 }
```

(a) A simple program with undefined behavior.

```
1  define i32 @main() {
2  bb:
3    %undefv = call i32 gazer.undef_value.i32()
4    %tmp = icmp ne i32 %undefv, 0
5    br i1 %tmp, label %bb1, label %bb2
6
7  bb1:
8    br label %bb3
9
10 bb2:
11   call void @__assert_fail()
12   unreachable
13
14 bb3:
15   ret i32 0
16 }
```

(b) The instrumented code where the undefined behavior is protected from being removed by further optimization.

Figure 3.2: Undefined value protecting transformation example.

3.1.2 Checks

In order to derive a safety property from the input program, GAZER instruments the LLVM IR code with so-called *checks*. Checks usually insert pre- or postconditions for a given instruction (e.g. the second operand of a division must not be zero). This is done by inserting a branch condition before or after the given instruction to check whether the required property is satisfied or not. If the condition is true, we continue program

execution as usual, otherwise control flow is set to reach an *error call*. The error call is special intrinsic function that indicates a property violation and signals it using an *error code*, unique to each inserted check. By assigning a program location (acquired from the compiled debug information) to each error code, this allows us to trace possible counterexamples back to a location in the original source code. During verification, error calls will be automatically translated to represent property violations in the language of the verification backend.

Currently GAZER supports the following checks.

- Assertion failures with support for the common implementations of the assert macro in C (such as `__assert_fail`) and the `__VERIFIER_error` function found in the SV-Comp benchmark suite [8].
- Division by zero.
- Signed integer over- and underflow.

Under the hood, checks are just regular LLVM transformations with special additions for traceability support. As such, they are very convenient to write for anyone who is familiar with the syntax of the LLVM IR, and the common methods used to manipulate it. As a result, our check instrumentation system offers a powerful, traceable and easily extensible way to represent pre- or postconditions as reachability properties for a verification engine.

3.1.3 Traceability Instrumentation

As part of the instrumentation process, we also insert some traceability information into the input LLVM module. These functions are simple markers, which are used by a trace builder component (described in Section 4.3 in more detail) to track events back to the LLVM IR level. This will be required later when we wish to map the verification result back to the original source-code, as discussed in Section 4.3. We currently use the following the intrinsic functions for traceability.

- **`gazer.function_entry.T...(metadata fn, T args...)`**: Marks the entry point of a function `fn`, with the call arguments listed in `args`. The intrinsic is overloaded, meaning that we will have a new declaration for each possible argument type. As an example, for functions taking two integers, the function declaration will be **`gazer.function_entry.i32.i32(metadata fn, i32 arg1, i32 arg2)`**.
- **`gazer.return_void(metadata fn)`**: Indicates that function `fn` has exited without a return value.
- **`gazer.return_value.T(metadata fn, T ret_val)`**: A function return, with the return value of `ret_val`. Overloaded intrinsic.

These functions allow us to keep track of functions, their arguments and return values, even after running transformations such as function inlining.

3.2 LLVM IR Transformations

This section describes the reduction techniques we perform on the LLVM IR in order to reduce the size of the resulting formal model.

3.2.1 Lifting Error Calls

After check instrumentation, error calls in the program are spread over all throughout every procedure. In the verification phase, both supported model checking engines require an entry point and a single reachability property in order to perform verification. As such, these error calls must be combined into a single universal error location. Furthermore, as the entry procedure may not contain an error call, the verification engine may conclude that the program is safe by definition without looking into any function calls, yielding a false negative result. In order to avoid this, the program has to be structured in a way such that the verification engine does not falsely believe that function calls cannot fail.

A possible solution to this problem would be the inclusion of a global failure bit which is set if a deep assertion fails in a called procedure. In the main procedure, calls to possibly-failing functions then would be followed by a check of the failure bit, thus forcing the verification engine to look inside the function for possible failures. This technique is known as *error-bit instrumentation*. It has the benefit of being simple and not too intrusive, however, it may unnecessarily guide the verification algorithm into calls that cannot fail.

Another solution is *assertion lifting* [42], in which we lift assertions (or in our case, error calls) deeply nested in the call graph into the main procedure. Using this method, error-bit instrumentation and global variables are avoided by copying the body of each function that may fail into the main procedure. Assertion lifting operates on the observation that a function either fails, or succeeds and returns a value. Therefore the transformation creates a copy of a possibly failing function $\mathcal{F}$ and inlines it into the main procedure. All call sites referencing $\mathcal{F}$ are transformed into nondeterministic branches choosing between an invocation of $\mathcal{F}$ or a jump to its inlined copy. These copies are then modified to branch into the error call of the main procedure. It is sufficient to insert one copy per function as different call sites may jump to the same error copy.

To represent failures in a traceable manner, we insert one universal error call into the main procedure, which will be reachable only through the always failing copies or the possible error calls in main. Later on, this one error call will serve as the verification goal.

The copies may be optimized: the resulting copy can be stripped of the instructions which are irrelevant to the failure case. On the contrary, error calls are completely removed in the always-succeeding functions. Removed instructions are marked dead and are eliminated by a subsequent run of LLVM's dead code elimination (DCE) pass, reducing the size of the duplicated functions. Further reduction may be achieved by running a static program slicing [61] algorithm on the resulting main procedure with the slicing criterion being the universal call yielding the error code, as presented in Section 3.2.2.

Example 8. *Consider the already-instrumented program shown in Figure 3.3a. The program calls the function* **`calc`** *to calculate the quotient of two numbers. If the divisor is zero, the function terminates with an error call. Figure 3.3b shows the program after assertion lifting – the error checks and calls have been moved to into the main procedure. The error check copy of* **`calc`** *in main is present in the basic block* **`calc.fail`***, where we jump using nondeterministic a branch. As we always assume that the failing copy exhibits erroneous behavior, the error check has been transformed into an assume statement (***`verifier.assume`***). As we only use one copy per procedure, we use the* **`phi`** *instruction to differentiate between the possible arguments of the original call.*

```
1  declare void @gazer.error_call(i8)
2  declare i32 @read()
3
4  define i32 @calc(i32 %arg1, i32 %arg2) {
5  bb:
6    %error.cond = icmp eq i32 %arg2, 0
7    br i1 %error.cond, label %bb1, label %bb2
8
9  bb1:
10   call void @gazer.error_call(1)
11   unreachable
12
13 bb2:
14   %div = sdiv %arg1, %arg2
15   ret i32 %div
16 }
17
18 define i32 @main() {
19 bb:
20   %1 = call i32 @read()
21   %2 = call i32 @read()
22   %3 = call i32 @read()
23   %4 = call i32 @calc(i32 %1, i32 %2)
24   %5 = call i32 @calc(i32 %1, i32 %3)
25   br label %bb1
26
27 bb1:
28   ret i32 0
29 }
```

(a) The original program.

```
1  declare void @gazer.error_call(i8)
2  declare i32 @read()
3  declare void @verifier.assume(i1)
4
5  define i32 @calc(i32 %arg1, i32 %arg2) {
6  bb:
7    %div = sdiv %arg1, %arg2
8    ret i32 %div
9  }
10
11 define i32 @main() {
12 entry:
13   %1 = call i32 @read()
14   %2 = call i32 @read()
15   %3 = call i32 @read()
16   br i1 undef, label %calc.fail, label %bb1
17
18 bb1:
19   %4 = call i32 @calc(i32 %1, i32 %2)
20   br i1 undef, label %calc.fail, label %bb2
21
22 bb2:
23   %5 = call i32 @calc(i32 %1, i32 %3)
24   br label %bb3
25
26 calc.fail:
27   %calc.arg2 = phi i32 [ %2, %entry ], [ %3, %bb1 ]
28   %error.cond = icmp eq i32 %calc.arg2, 0
29   call void @verifier.assume(%error.cond)
30   br label %error
31
32 error:
33   %error_code = phi i8 [ 1, %calc.fail ]
34   call void gazer.error_call(i8 %error_code)
35   unreachable
36
37 bb3:
38   ret i32 0
39 }
```

(b) The transformed program after assertion lifting.

Figure 3.3: Assertion lifting example.

3.2.2 Optimization Pipeline

LLVM comes with a plethora of built-in optimization passes, some of which we will use to reduce the size of the resulting model. LLVM defines multiple types of optimizations, depending on the target scope. *Module passes* act on the level of the whole module: they may insert or remove functions, global variables and modify the program in any way. A *function pass*, on the other hand is constrained to operate on a single function at a time, whereas *loop passes* can only work on loops. Local peephole optimizations are implemented as *basic block passes*, which can only modify the instructions in a single block. Furthermore, LLVM's pass manager is able to provide important analyses (such as a dominator tree) and schedule passes in a way that it can avoid recomputing analysis information if possible.

When executing the verification pipeline, we first begin with a set of early optimizations – basic simplification of control flow (CFG simplification pass), common subexpression elimination (CSE), removal of unused global values and unused function arguments.

An interesting optimization that we also use in this phase is what the LLVM terminology calls SROA – scalar replacement of aggregates. This pass examines structures and small arrays (remember that arrays in LLVM always have a known size) in a function, and attempts to replace them with a set of scalar registers. As an example (shown in C for readability), a struct definition `struct S { int x; float y; }; s` is replaced by two variables, `sx` and `sy`. All operations on individual fields (e.g. `s.x = a + b`) are modified to update the newly introduced scalars instead (`sx = a + b`).

A nice benefit of SROA is that it eliminates some trivial memory instructions, thus the analyses and transformations that follow have an easier time reasoning about them. As most optimizations are defined for LLVM registers, lifting aggregate operations also enables transformations for individual aggregate members that would not have possible without running SROA.

For easing interprocedural analysis, we also use function inlining (the procedure of replacing a function call with the callee's body). In our work, we use it to obtain more information on the behavior of an interprocedural program, as without more thorough interprocedural analysis, function calls would act as black boxes. A model checker algorithm may extract more information from the entire inlined function body than from merely just a function call. Furthermore, one of our model checking backends (THETA) requires a single, call-free model as an input, thus inlining is a must in its case. And while our bounded model checker backend is able to handle function calls, inlining opens up better optimization opportunities, thus we are generally able to generate a simpler model, reducing verification time. If there is no recursion in the system, we are able to promote global variables into locals.

Program Slicing

Program slicing is a technique first described by Mark Weiser [61]. He suggested that while debugging a complex program, a programmer only pays attention to a smaller subset of the entire source code. This subset contains only the instructions and variables relevant to the problem being debugged. Attempting to formalize this practice, Weiser gave the following definition for program slices:

Definition 11 (Program slice). *A program slice $\mathcal{P}'$ of a program $\mathcal{P}$ with respect to the criterion of (s, V) is an executable subset of $\mathcal{P}$, producing the same output and assigning the same values to the variables $V = v_1, \ldots, v_n$ as the original program $\mathcal{P}$ in its statement s.* ▪

In our work, we shall use program slicing to remove instructions which are irrelevant to the error call inserted by check instrumentation and assertion lifting. The most commonly used approach for slicing is a technique known as *backward slicing*. Backward slicing produces accurate slices, while retaining all instructions which are crucial to the slicing criterion. Given a criterion instruction s, backward slicing finds all nodes on which s transitively data depends. As some branching decisions may affect whether a particular instruction is reachable or not, these also need to be included in the slice. These branching decisions are the same as the control dependencies of a given instruction. Of course, these dependencies can have dependencies which need to be taken into account. Therefore backward slicing is done by retaining all instructions on which the criterion control or flow depends transitively. This information can be queried from a program dependence graph.

A backward slicer algorithm marks all nodes which are backwards reachable (walking backwards on both data and control dependency edges) from the criterion node in the program dependence graph [29, 50]. As the PDG explicitly shows the control and flow dependency relations of every instruction, this method will include all required nodes in the slice. Figure 3.4 shows an example of this procedure with the non-trivial assertion node being the criterion. The red dashed edges represent flow dependency, blue edges represent control dependency. The filled nodes are those backwards reachable from the criterion node.

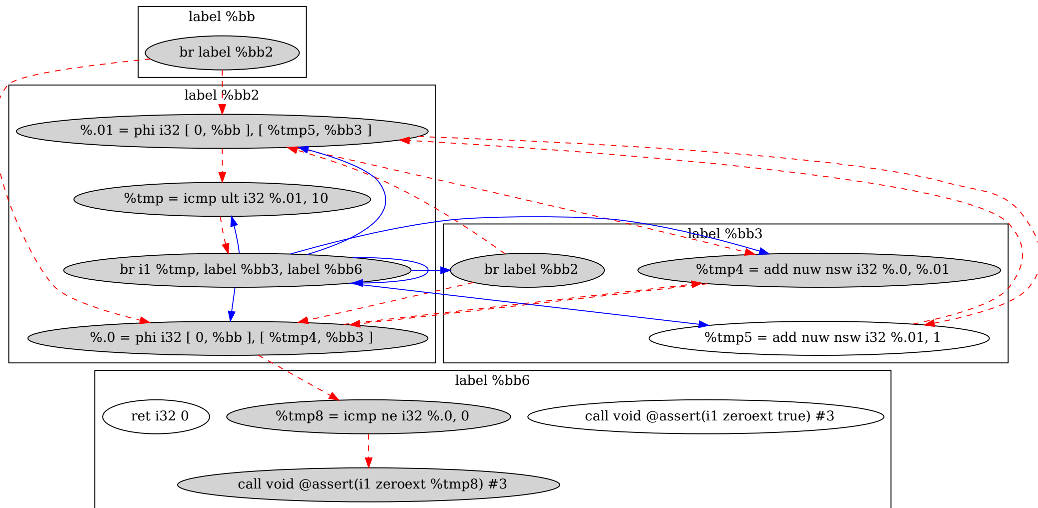


Figure 3.4: PDG and a slice of a program.

3.3 Memory Models

While the translation of basic arithmetic instructions is straightforward, the same cannot be said about memory operations. Memory operations and pointers introduce an ambiguity into the system which cannot be modeled in a simple manner. A *memory model* is an

abstraction which can help us model system memory in a way that can be represented for an SMT solver. Memory models may come in different flavors and may make trade-offs between certain aspects of verification: a model may choose precision over speed, while another may choose to drop soundness for fast verification.

The most straightforward memory model, called the *flat memory model* models the entire memory as one large array of bytes. This model is simple, precise and can safely model type-unsafe operations, such as bitcasts and unions. However, it suffers from scalability issues as each operation must reconstruct its value from a sequence of bytes. Furthermore, the flat model has no knowledge about certain aliasing rules (such as distinct globals never alias), therefore it cannot benefit from the inherent disjointedness of memory regions.

The *Burstall model* [15] splits memory into several arrays, one for each distinct type in the system. The model is based on the assumption that pointers of distinct types never alias – a statement that is true for several type-safe languages, such as Java. This offers better performance, but is usually unsound for a type-unsafe language such as C.

A third, and emerging family of memory models is the one we can collectively refer to as *partitioned memory models* [52, 60, 35]. A partitioned memory model relies on static pointer analyses to determine aliasing rules and ensure that distinct alias groups are put into separate arrays. This usually yields better performance, and by applying proper care, can be made sound in the presence of type-unsafe memory operations.

The rest of this section is organized as follows. Section 3.3.1 describes our generic memory SSA framework, designed to provide a convenient translation interface for various memory models. Section 3.3.2 describes our simple, proof-of-concept flat memory model we have implemented within GAZER.

3.3.1 Memory SSA

As discussed in Section 2.1.4, LLVM does not translate memory operations into an SSA form. However, this makes translating and reasoning about memory operations more difficult: it is not evident how an instruction modifies a particular memory location, and querying a pointer alias analysis for this information can be costly. Memory SSA form [49] attempts to solve this problem by constructing an explicit static single assignment form for the variables stored in memory. In usual compiler implementations, memory SSA usually does not provide an SSA form for each distinct variable in memory – rather, the memory is represented as a single variable, and the memory SSA form is built for just one memory object.[2] Opposed to that, our memory SSA analysis is generic, and relies on the underlying memory model to partition the memory and will generate a proper memory SSA form for each partition.

In the memory SSA of GAZER, each allocation site corresponds to a *memory object*. A memory object is an abstract entity used to describe the type and allocation semantics of a piece of memory, thus it is never referenced directly in the final automaton. Memory objects may have *memory object definitions*, instructions that possibly change the value at the location the memory object describes. *Memory object uses* are instructions which possibly read the value referenced memory location of a memory object. In order to ensure the SSA-property of memory SSA, we also need to introduce the concept of the

[2]This is the main point in which our implementation differs from the built-in memory SSA infrastructure found in LLVM. Our tool offers a generic framework with the possibility to define multiple memory objects – possibly introducing an increase in analysis time. In contrast, memory SSA in LLVM was designed with analysis speed in mind, therefore it trades precision for performance.

memory ϕ-nodes as a subset of memory definitions. As with a regular ϕ-node, a memory ϕ-node also selects one single definition form a set of possible incoming definitions, based on the control flow paths the program took to reach said node.

The memory SSA is partly built by the memory model. During memory analysis, the memory model inspects the input program and inserts the memory objects based on the allocation sites of the program. In the next step, the memory model makes another pass over the program and annotates memory-accessing instructions with their possible memory object definitions and uses. Each instruction may have multiple definitions and uses due to the often occurring ambiguity of pointer usage. As certain instructions may define or use a memory object in different ways, we distinguish between several types of uses and definitions. Table 3.1 offers an overview of the memory object definition and use types used by GAZER.

Table 3.1: Memory objects definitions and uses.

Name	Scope	Description
Definitions		
AllocaDef	`alloca` instructions	Represents an initial allocation of the object on the stack, i.e. the declaration of a local variable.
LiveOnEntry	Functions	Indicates that the memory object is alive when entering function.
GlobalInit	The entry function	One-time initialization of a global variable.
StoreDef	`store` instructions	A definition through a `store` instruction.
CallDef	`call` instructions	Indicates that a call possibly modifies the memory object.
PhiDef	Basic blocks	A memory object ϕ-node placed by our memory SSA construction algorithm.
Uses		
LoadUse	`load` instructions	Use through a `load` instruction.
CallUse	`call` instructions	The call may read the memory object's value, either by passing a memory object as a call parameter or through a global variable.
ReturnUse	`ret` instructions	Indicates that the memory object is alive when the function returns.

After the memory model has placed all definitions and uses, we calculate an SSA-form over the definitions. This is done by using the dominance frontier information, presented in Definition 4. For each memory object we use the dominance frontier to calculate the locations of the memory ϕ-nodes and perform the standard SSA construction and renaming algorithm to give a unique name to each memory object definition.

Example 9. *Figure 3.5 shows a program with global variables annotated by our memory SSA construction algorithm w.r.t. a flat memory model. Definition and use annotations are placed according to the memory model, from which the final SSA-numbering and memory ϕ-nodes were placed by the memory SSA algorithm. Each definition annotation has a unique number given by the SSA renaming algorithm and the name of the defined memory object. Use annotations store the name of the used memory object and the number of its reaching definition.*

```c
int a = 1, b = 1, c = 3;

int main(void) {
    int input = 1;
    while (input != 0) {
        input = __VERIFIER_nondet_int();
        if (input == 1) {
            a = a + 1;
        }
    }
    assert(!(a < b));
    return 0;
}
```

(a) A C program with global variables.

```llvm
; Declared memory objects:
; (0, "Memory", objectType=[Bv64 -> Bv8], size=Unknown)
; (1, "StackPointer", objectType=Bv64, size=64)
; (2, "FramePointer", objectType=Bv64, size=64)
define i32 @main() {
bb:
  ; 0 := liveOnEntry(Memory)
  ; 1 := liveOnEntry(StackPointer)
  ; 2 := liveOnEntry(FramePointer)
  ; 3 := globalInit(Memory, i32 1)
  ; 4 := globalInit(Memory, i32 1)
  ; 5 := globalInit(Memory, i32 3)
  br label %bb1

bb1:
  ; 8 := phi(Memory, { [7, bb8], [5, bb] })
  %.0 = phi i32 [ 1, %bb ], [ %tmp3, %bb8 ]
  %tmp = icmp ne i32 %.0, 0
  br i1 %tmp, label %bb2, label %bb9

bb2:
  %tmp3 = call i32 @__VERIFIER_nondet_int()
  %tmp4 = icmp eq i32 %tmp3, 1
  br i1 %tmp4, label %bb5, label %bb8

bb5:
  ; load(Memory, 8, i32 %tmp6)
  %tmp6 = load i32, i32* @a, align 4
  %tmp7 = add nsw i32 %tmp6, 1
  ; 6 := store(Memory)
  store i32 %tmp7, i32* @a, align 4
  br label %bb8

bb8:
  ; 7 := phi(Memory, { [8, bb2], [6, bb5] })
  br label %bb1

bb9:
  ; load(Memory, 8, i32 %tmp10)
  %tmp10 = load i32, i32* @a, align 4
  ; load(Memory, 8, i32 %tmp11)
  %tmp11 = load i32, i32* @b, align 4
  %tmp12 = icmp slt i32 %tmp10, %tmp11
  br i1 %tmp12, label %error, label %bb13

bb13:
  ; ret(Memory, 8)
  ret i32 0

error:
  %error_phi = phi i16 [ 2, %bb9 ]
  call void @gazer.error_code(i16 %error_phi)
  unreachable
}
```

(b) The LLVM IR annotated with memory SSA information.

Figure 3.5: MemorySSA example.

3.3.2 A Simple Memory Model

In this work, we use a simple flat memory model to demonstrate the usability and capabilities of our workflow. As discussed previously, the flat memory model encodes memory as a single array of bytes. Therefore the only memory object we declare is $M : Ptr \rightarrow Bv8$, where *Ptr* is the pointer type according to the target architecture (e.g. *Bv32* or *Bv64*). Note that the theory of arrays [47] within SMT (therefore within GAZER) requires array stores to be side-effect free. As such, writing the value v into an array A on location p returns a new array A', which has the same values at every index as A, except for p. In the following, we denote such array stores as $A' = A[p \leftarrow v]$.

Each memory writing instruction will serve as a new definition of our single memory object. Storing and loading values from this memory model requires us to break the value down into its bytes and write them into the array, according to the endianness of the platform. As an example, storing the 32-bit value x into M at location p may be encoded in the following fashion:

$$
\begin{aligned}
M_1 &= M[p \leftarrow x_{0..7}] \\
M_2 &= M_1[p + 1 \leftarrow x_{8..15}] \\
M_3 &= M_2[p + 2 \leftarrow x_{16..23}] \\
M_4 &= M_3[p + 3 \leftarrow x_{24..31}]
\end{aligned}
$$

where $x_{i..j}$ represents the bits between i and j of the bit-vector x. Loads are encoded similarly, using bit-vector concatenation. Calculating pointer addresses using the **`getelementptr`** instruction is done in a straightforward manner. Recall that a pointer **`%p`** of type τ not only represents a particular address in memory, but the pointer arithmetic operations over blocks of the type of τ. If τ is n bytes long, then each pointer addition over pointers of this type will advance the stored memory address of **`%p`** by n bytes. As such, we can encode a **`getelementptr`** instruction with the base pointer q of type τ and offset i as $p = q + \mathsf{sizeof}(\tau) \cdot i$.

Allocations are handled by defining special memory objects for the stack and frame pointers. An **`alloca`** instruction (thus, an alloca definition) allocates memory into the main memory array starting from the offset shown by the stack pointer. When entering a function, the frame pointer is moved into the current position of the stack pointer, and upon exit, all memory locations on the stack past the frame pointer are invalidated (i.e. reads from these locations return undefined values).

Calls are handled by passing in the single memory array into the called function as input–output parameter, indicated by the call use and call definition annotations. Undefined functions are assumed not to modify memory, unless they have a pointer operand, return a pointer, or if their return value is void. While this can make the analysis unsound in certain corner cases (e.g. a possible write of global variables in a undefined function), it offers the huge benefit of avoiding a large number of false-positives.

3.4 Transforming the LLVM IR into CFA

This section describes our approach on transforming the LLVM intermediate representation into the control flow automaton formalism of GAZER framework.

3.4.1 Control Flow Automata within GAZER

Our tool, GAZER comes with its own control flow automata formalism, designed to be a convenient intermediate representation between the LLVM IR and the input formalism of the eventual formal verification algorithm. In order to provide a safe and easily translatable format (in both directions), the control flow automata found in GAZER have the following restrictions.

1. Procedures must be in a single assignment form, where there is at most one assignment for each variable in every path between two locations.
2. The control flow of the procedure must form a directed acyclic graph (DAG), thus having loops is forbidden.
3. All procedures must be stateless – global variables are disallowed.

The first condition is a slight relaxation from the original definition of SSA (where it is forbidden for a variable to be the target of an assignment more than once).[3] This relaxation is needed if we want to avoid having ϕ-nodes in our CFA formalism without explicitly encoding the path condition of each ϕ-node entry.

The second and third restrictions have two major consequences. First, in order to represent loops, they must be encoded as tail-recursive functions. For this encoding to be possible, we must impose a limitation on the input program: all loops in the input CFG must be *natural loops*, that is, the loop must have a single entry point (the loop *header*) which must dominate all nodes within the loop. Flow graphs which only contain natural loops are called *reducible flow graphs* [37] and most programs in practice are reducible. It can be easily seen, that all structured programs (i.e. programs without **`goto`**) fall into this category. Even when using **`goto`** statements, the only way to build an irreducible control flow is by jumping into the middle of a loop from the outside, which is extremely rare. Furthermore, irreducible control flow poses challenges way beyond the verifiability of the input program, as most loop recognition and optimization algorithms in compilers also assume that all loops are natural.

Second, the CFA formalism does not include any global variables, but manages global state by passing global values as procedure inputs, and reading them as procedure outputs. This allows GAZER's built-in analysis algorithms to reason more efficiently about the properties of the formal model as they are not required to track any global information.

3.4.2 Transformation of Control Flow

At the end of the workflow, the (optimized) LLVM IR is transformed into our control flow automaton formalism. The transformation process introduces two new locations in the CFA for each basic block in the program, which will represent the entry and exit points of the block, respectively. We also add an edge between these locations, which is labeled with the instructions of the block. For branching blocks, we add edges between the predecessor block's exit location and the entry locations of the successors blocks. For conditional branches, a guard condition is also placed on the edges: the boolean predicate

[3]Strictly speaking, the literature [28, 7] would refer to programs in this form as *dynamic single assignment*-formed (DSA). In DSA form, a variable may be present on the left side of an assignment more than once, but it is guaranteed that at most one assignment of a variable would be executed during any program execution. However, as SSA is a much more common term, we will continue to refer to this program representation as SSA-formed.

of the branch condition (say p) for the "then" path and its negated form ($\neg p$) for the "else" path.

Another issue comes with the handling of ϕ-nodes, as our CFA formalism does not offer support for the ϕ-node construct. However, as the semantics of the ϕ-node can be interpreted as a statement which picks a value for a variable, given all incoming edges to that variable's block. An equivalent representation can be achieved by eliminating the ϕ-node and labeling the incoming edges with the corresponding assignments. Note that this transformation satisfies the single assignment requirement of our CFA formalism.

Translation of Loops

Loops are a very important special case of our translation algorithm. As our CFA formalism does not support loops, we must encode loops as stateless tail-recursive functions. To present our transformation algorithm, we make use some of the terminology described in the following definition.

Definition 12. Let L be a natural loop, and the basic block B be an arbitrary block in L.

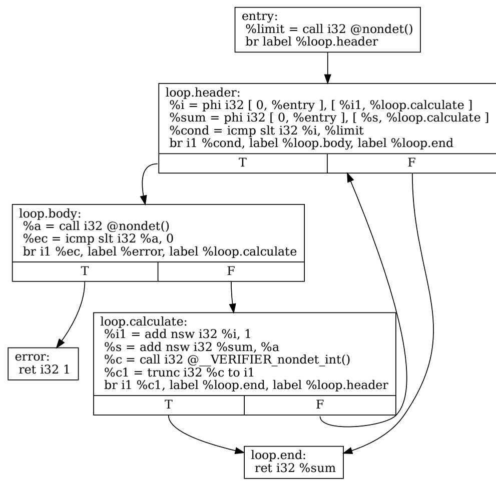
- B is a *latch block* if B has an outgoing edge which targets the loop header. Edges between a latch block and the header are called *back-edges*.
- B is a *exiting block* if B has a successor B' outside of the block. In such case, B' is called an *exit block*. ∎

Let L be a loop, $\mathcal{A}_L$ its corresponding automaton. In the first step, we create a new variable for each value defined within L. In order to do this, let **`%x`** be an arbitrary LLVM register, v the variable corresponding to **`%x`** in $\mathcal{A}_L$.

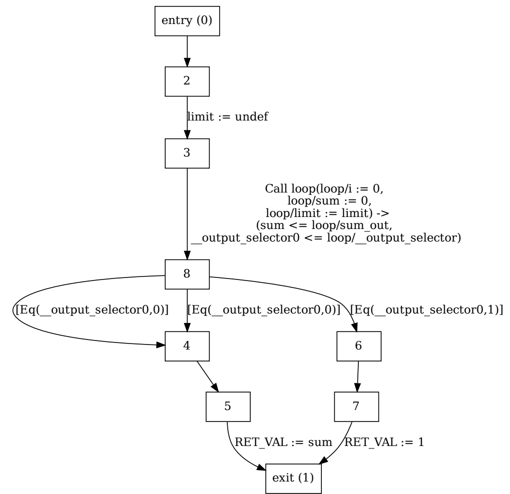
1. If **`%x`** is has a use within the loop, but is defined outside of the loop, v will become an input argument.
2. If **`%x`** is defined within the loop, then v will be a local variable of the new automaton.
3. If **`%x`** is a ϕ-node, defined in the loop header, v will be an input argument.
4. If **`%x`** is defined within the loop (possibly as a ϕ-node in the header) and has uses outside of the loop, v shall be an output argument.

During translation, the *back-edges* of the loop (the edges from within the loop targeting the loop header) are represented by recursive call transitions to $\mathcal{A}_L$. As $\mathcal{A}_L$ is required to be a DAG, said call transition's source is the location corresponding to the end of the source block of the back-edge. There are two kinds of calls to $\mathcal{A}_L$ in the system: first, the recursive call within the loop, second the "original" entry point into the loop from the outside. If the call is in the entry point, each regular input variable will be passed into the call as they are. For inputs obtained from the header ϕ-nodes, we pass the value corresponding to the loop header predecessor in the ϕ-node entry list. In the case of the tail-recursive call, ϕ-node input arguments will have the value of the ϕ-node for its predecessor latch block.

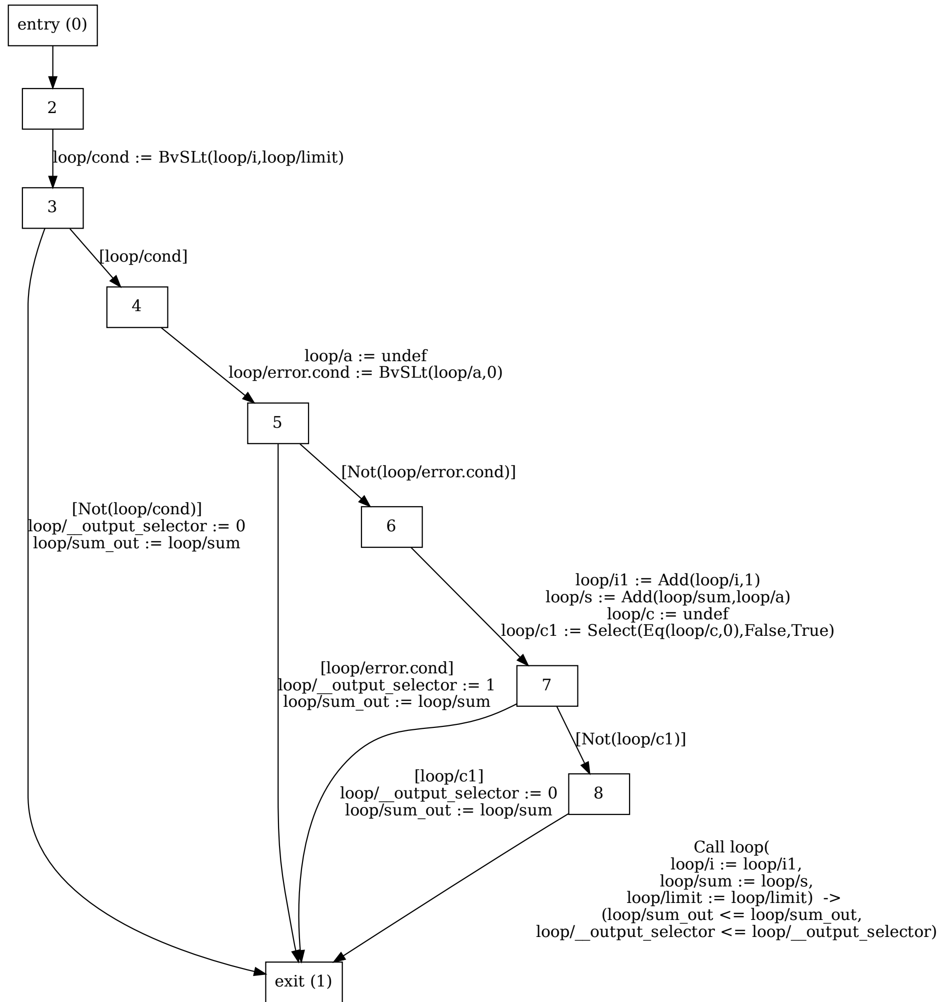
Example 10. *Figure 3.6 offers and example of this transformation, with the presence of ϕ-nodes and multiple loop exits. The program in Figure 3.6a is transformed into two automata: one for the main control flow (shown in Figure 3.6b) and a recursive automaton for the loop (Figure 3.6c). The variables corresponding to* **`%i`** *and* **`%sum`** *became both inputs and outputs in the loop automaton, while the variable corresponding to* **`%limit`** *is a simple input.*



(a) The source CFG with a loop.



(b) The translated main procedure.



(c) The translated loop automaton.

Figure 3.6: Transformation of a SSA-formed CFG into a CFA.

3.4.3 Translation of Memory Instructions

Memory instructions are handled by querying the constructed memory SSA and the memory model that produced it. Each memory object *definition* d_m will have a corresponding variable v_m in the CFA. If the definition indicates that its memory object is alive upon entry, the variable v_m will become an input variable. If the definition is alive upon exit (indicated by a return use), v_m shall become an output variable. Memory object definitions in loops are handled similarly as discussed for regular LLVM registers.

Access to memory are translated according to the rules defined by the memory model. When encountering a memory use (such as a load), the memory model is asked to translate the load instruction into a valid assignment. If the pointer referenced by the load is ambiguous, the memory model may choose to generate a sequence of if-then-else expressions to resolve the pointer value during verification time.

3.4.4 Optimizations during Transformation

In order to reduce the size and complexity of the generated automata, the translation process applies certain optimizations. The most notable is *variable elimination*, where certain variables are eliminated by inlining their assigned values into their users. In order to do so, we inspect assignments in the form of $v := E$, where v is a variable in the CFA, and E is an arbitrary expression. As the automaton is in a single assignment form, we can assume that v is not defined anywhere in the procedure[4]. As such, all uses of v are dominated by this assignment, and can be replaced with E. Note that this is not possible on the LLVM IR level: LLVM's three-address code scheme requires a new variable for every operation, thus it does not allow us to replace instructions with complex nested expressions.

In order to avoid building overly large expressions and to avoid losing useful traceability information, variable elimination has a few restrictions.

- Variables obtained from call instructions are never eliminated, as they might be needed to produce a traceable counterexample.
- If the CFA was translated from a loop, variables that correspond to loop outputs are not eliminated.
- Variables translated from ϕ-nodes are never eliminated, as their definitions are ambiguous due to the semantics of our ϕ translation algorithm.

By default, we only eliminate a variable if it has only one use and has no eliminated operands. This keeps the size of expressions at a manageable level. However, there exists an "aggressive" variable elimination setting, in which we eliminate all variables that satisfy the restrictions listed above.

[4]The only exception to this assumption are the variables obtained from ϕ-nodes, as discussed in Section 3.4.2. Due to this reason, we never perform this transformation on these kinds of variables.

Chapter 4

Verification

This chapter describes the verification backends offered by GAZER. Section 4.1 presents our inlining-based bounded model checker algorithm, while Section 4.2 describes the backend built upon the THETA model checking framework. Section 4.3 describes the process in which we trace possible counterexamples from the formal verification backend back to the LLVM IR level, and furthermore, to the original C code.

4.1 Bounded Model Checking

A *bounded model checker* (BMC) algorithm [12] traverses the state space of the program, searching for an erroneous path from the initial state of length at most k (the bound – hence the name *bounded* model checker). If such an error path is present, it is refutation of correctness. If no error paths were found for a given k, the algorithm increases the bound to $k + 1$.

As the algorithm can only prove safety for a bound of k, but has no information whether it is safe for $k + 1$, the bound may grow indefinitely. Due to this reason bounded model checking is not complete: it may not check all possible error paths, thus it cannot prove correctness. However, with a proper (i.e. bit-precise) encoding, BMC can be a very effective method to discover errors in a program without any (or a minimal) number of false-positives [8].

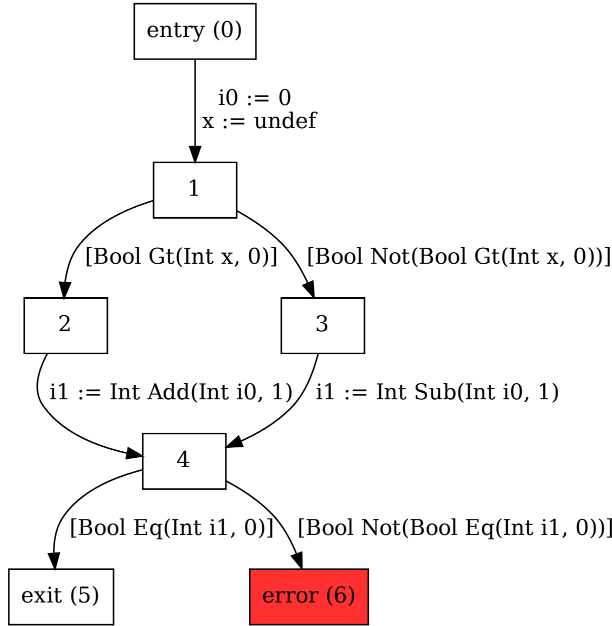
The algorithm works as follows: given a control flow automaton $\mathcal{A} = (L, E, \ell_0, \ell_q)$, an error location $\ell_e \in L$, and a bound $k \geq 0$, we encode all possible $\ell_0 \Longrightarrow \ell_e$ paths as a single SMT formula. The encoding needs to replace assignments with equivalence predicates, thus it requires having a unique variable for each occurrence of a program variable on the left-hand side. This property is satisfied out of the box by using GAZER's CFA formalism, as it is SSA-formed by definition. Furthermore, our CFA formalism guarantees that an automaton is always a DAG, thus we can construct a topological sort of its locations. This allows us to do the encoding with a simple dynamic programming approach.

Let $\ell_i \in L$ be an arbitrary location in $\mathcal{A}$, where i is the index of said location in the topological sort. Let $\mathsf{SMT}(\ell_x, \ell_y)$ be the function which represents the SMT encoding of the edge between ℓ_x and ℓ_y. Let $\mathcal{T}$ be an array where $\mathcal{T}[i]$ is contains the SMT encoding of all paths between ℓ_0 and ℓ_i. Initially $\mathcal{T}[i] = \bot$ for every $\ell_i \in L$. The algorithm fills the values of $\mathcal{T}$ according to the following rules:

$$\mathcal{T}[0] := \top$$
$$\mathcal{T}[i] := \bigvee_{\ell_p \in \mathsf{pred}(\ell_i)} (\mathcal{T}[p] \wedge \mathsf{SMT}(\ell_p, \ell_i)).$$

When this process finishes, we can get the encoding of all $\ell_0 \implies \ell_e$ paths by querying $\mathcal{T}$ with index of ℓ_e in the topological sort. As the CFA formalism guarantees that a variable may be defined at most once in each path starting from ℓ_0, this encoding introduces no inconsistencies. Furthermore, with i and p being the indices of a topological sort, it is guaranteed that $\ell_p \in \mathsf{pred}(\ell_i)$ contains a valid encoding by the time we process ℓ_i.

Example 11. *Consider the CFA shown in Figure 4.1a. Node labels show the index of each location in a topological sort. The SMT encoding process of the bounded model checker algorithm is shown in Figure 4.1b. Note how the nondeterministic definition of variable x is not present in formula – there is no need to encode it as all of its uses will be treated as undefined values. The path condition for the error location (and thus the verification condition) is the formula calculated in $\mathcal{T}[6]$.*



(a) A simple control flow automaton.

$$\mathcal{T}[0] := \top$$
$$\mathcal{T}[1] := (i_0 = 0)$$
$$\mathcal{T}[2] := \mathcal{T}[1] \wedge (x > 0)$$
$$\mathcal{T}[3] := \mathcal{T}[1] \wedge \neg(x > 0)$$
$$\mathcal{T}[4] := (\mathcal{T}[2] \wedge (i_1 = i_0 + 1))$$
$$\vee\, (\mathcal{T}[3] \wedge (i_1 = i_0 - 1))$$
$$\mathcal{T}[5] := \mathcal{T}[4] \wedge (i_1 = 0)$$
$$\mathcal{T}[6] := \mathcal{T}[4] \wedge \neg(i_1 = 0)$$

(b) The SMT encoding of all path conditions.

Figure 4.1: SMT encoding example.

In order to reduce formula size and lift some of the burden on the SMT solver, we have implemented a set of term simplification rules. These rewriting rules include simple constant propagation, arithmetic simplification, and some logical optimization for if-then-else expressions.

4.1.1 Inlining Techniques

If no errors were found for a given bound k, the BMC algorithm needs to increase the bound to $k + 1$. To do this, the algorithm must unroll loops and inline procedure calls to account for these control structures. As our CFA formalism requires loops to be encoded as recursive procedures, increasing the bound always means inlining a procedure into another one. As such, we shall discuss bound increases as terms of inlining (possibly recursive) procedures.

The easiest solution is to perform this statically, where the algorithm just inlines all procedures up to a maximum bound in one pass, encodes the resulting automaton into an SMT formula, and passes said formula to an SMT solver. While this approach is simple, it scales poorly for large programs due to a potentially exponential increase in program size.

A slightly more sophisticated method is to inline iteratively, increasing the bound every time the program was proved to be correct for the bound of k. This delays the exponential explosion of the state space and may be noticeably more effective than eager static inlining if an error is present at a lower bound than the defined maximum.

Stratified Inlining

Stratified inlining [43] further improves this concept by using under- and over-approximations to decide which procedures should be inlined. First, an under-approximated program is checked whether a property violation could be found without going through any calls. If not, then we over-approximate each call in the program with the *summary* of their called procedures. If a (possibly spurious) counterexample π was found, we inline all calls which were present in π, and start the process again.

The process (slightly adapted to our CFA formalism) is described in Algorithm 2. The algorithm operates on an entry automaton $\mathcal{A}$ (i.e. the program) and tracks the set C of all call transitions found in $\mathcal{A}$. The *cost* of a call transition is its depth in the call chain – initially all calls have the cost of 1. If inlining a procedure with a cost of w lifts a call transition c into $\mathcal{A}$, then the cost of c shall be $w + 1$.

Each iteration of the loop in line 2 searches for property violations up to the current value of k. The loop in line 3 performs the under- and over-approximation loop described above. First, we under-approximate all calls by blocking execution from going through them. This can be achieved by annotating call transitions with "assume false" guards. This yields the modified program $\mathcal{A}'$, which then encoded into a formula and checked using an SMT solver. If the formula was satisfiable, we have found a valid counterexample π, thus we can return with $\textsc{Cex}(\pi)$.

If no errors were found in the program stripped of its call transitions, we create an over-approximation. To do this, we split the call transition set C into sets C_1, C_2. Calls which have a cost lower than the current bound are put into C_1 and will later be over-approximated. Calls in C_2 will stay blocked as their cost is higher than the current bound.

Given a call c in C_1, the simplest form of over-approximation summaries are havocs over all variables possibly modified by executing c. Once again, due to our SSA-formed CFA format, no explicit havocs are required: a simple over-approximation with "assume true" is enough to leave variables unconstrained. In more advanced cases, static analysis methods (such as abstract interpretation) could be used to calculate procedure invariants that may be used as summaries [30, 5].

If the over-approximated program was proven to be correct and C_2 is empty, then all call transitions have been over-approximated, thus the original program is also correct. If there are still some call transitions present (i.e. C_2 is not empty), the current verification bound must be increased (line 18).

If the over-approximated program yielded a satisfiable formula and thus a (spurious) counterexample π, we check which over-approximated call transitions were present in π. All such calls are inlined into $\mathcal{A}$ and the algorithm jumps back to line 3.

Algorithm 2: Stratified inlining algorithm.

Input: A control flow automaton $\mathcal{A} = (L, E, l_0, l_q)$
Input: An error location $\ell_e \in L$
Input: A maximum bound $MAX > 0$
Output: SUCCESS, CEX(π) or BOUNDREACHED

1. C := call transitions in $\mathcal{A}$
2. **for** $k := 1$ **to** MAX **do**
3.   **while** *true* **do**
4. $\mathcal{A}' := \mathcal{A}[\forall_{c \in C}\ c \leftarrow \textbf{false}]$
5. φ := SMT encoding of all $\ell'_0 \Longrightarrow \ell'_e$ paths in $\mathcal{A}'$
6.   **if** φ *is satisfiable* **then**
7. π := the error path from φ
8.   **return** CEX(π)
9.   **end**
10. $C_1 := \{c \in C \mid cost(c) \leq k\}$
11. $C_2 := \{c \in C \mid cost(c) > k\}$
12. $\mathcal{A}' := \mathcal{A}[\forall_{c \in C_1}\ c \leftarrow summary(c),\ \forall_{c \in C_2}\ c \leftarrow \textbf{false}]$
13. φ := SMT encoding of all $\ell'_0 \Longrightarrow \ell'_e$ paths in $\mathcal{A}'$
14.   **if** φ *is unsatisfiable and* $C_2 = \emptyset$ **then**
15.   **return** SUCCESS
16.   **end**
17.   **if** φ *is unsatisfiable and* $C_2 \neq \emptyset$ **then**
18.   **break**
19.   **end**
20. π := the error path from φ
21. $\mathcal{A}$:= all $\{c \in C_1 \mid c \in \pi\}$ calls inlined into $\mathcal{A}$
22. C := call transitions in $\mathcal{A}$
23.   **end**
24. **end**
25. **return** BOUNDREACHED

Optimization

Many modern SMT solvers offer support for incremental solving, a technique where a formula may be pushed onto or popped from the solver in a stack-like (LIFO) manner. Push inserts a new solver state, which will preserve its list of asserted formulae between invocations and the lemmas deducted from said formulae. Popping the solver state removes each formula that was added after the latest push, but preserves all learned lemmas stored in the "lower" scopes. This opens up an optimization opportunity where

we can avoid encoding the entirety of the program in each under- or over-approximation step, and do the encoding incrementally.

We will operate on the observation that after an unsuccessful under-approximation (i.e. the under-approximated formula was unsatisfiable), all possibly feasible error paths must go through a call. As such, a location from which all error paths involving at least one call transition offers us a convenient place to save the solver state. In our implementation, we pick the lowest common ancestor (LCA) of all call transitions (more precisely, their edge source locations) in the dominator tree of the automaton, denoted by ℓ_{lca}. The fact that ℓ_{lca} dominates all call transitions means that *every* path starting from the entry node and going through a call must contain ℓ_{lca}. Furthermore, as ℓ_{lca} is the closest of all common dominators, it allows us push as much information onto the solver as possible. Also note that such node always exists in a flow graph with an entry point – if nothing else, it is the entry location.

Due to our automata being DAGs, we can provide a fast algorithm to calculate ℓ_{lca}. We exploit the fact that if $p_1, p_2, \dots, p_k$ are the predecessors of a node n, then node $d \neq n$ dominates n if and only if $d \operatorname{dom} p_i$ for every $1 \leq i \leq k$ [2]. This means that finding the set of n's dominators $D(n)$ requires finding the dominator set $D(p_i)$ of every $p_i \in \operatorname{pred}(n)$. Furthermore, due to the definition of dominance, every node dominates itself. This can be summarized by the following equations:

$$D(n_0) = \{n_0\}$$
$$D(n) = \{n\} \cup \bigcap_{p \in \operatorname{pred}(n)} D(p)$$

which mean that every node's dominator set contains the node itself and the intersection of its predecessors' dominators. Using this, we can calculate ℓ_{lca} in a single pass (without building the dominator tree) by going through the topological sort and building the dominator sets[1] and maintaining a special D' set of possible ℓ_{lca} candidates. After finding the dominator information, ℓ_{lca} will be the element from D' having the largest index in the topological sort.

The final algorithm. The optimized algorithm works by calculating path conditions from a *current starting location*, denoted by ℓ_{start}. Initially, the starting location ℓ_{start} is the entry location ℓ_0. The algorithm extends the stratified inlining presented in Algorithm 2 with the following steps.

1. During under-approximation, we push the path condition φ_1 of all $\ell_{start} \implies \ell_e$ paths to the solver state. If the under-approximated formula was unsatisfiable, φ_1 is popped from the solver.
2. In the next step, we calculate the lowest common dominator of all call transitions, denoted by ℓ_{lca}. We push the path condition of $\ell_{start} \implies \ell_{lca}$ onto the solver, thus encoding all call-free paths between the current starting point and the LCA node. Note that this formula will never popped from the solver – this information will always be present in subsequent solver queries.
3. The over-approximation step will then push the path condition φ_2 of all $\ell_{start} \implies \ell_{lca}$ paths onto the solver. If the over-approximation step is unsuccessful, we pop φ_2,

[1]Our implementation uses bit-vectors to represent such sets as they have a smaller memory footprint and provide $O(1)$ insertion time.

set ℓ_{start} to be equal to ℓ_{lca}, and either start the under-approximation step again or increase the bound as in the original algorithm.

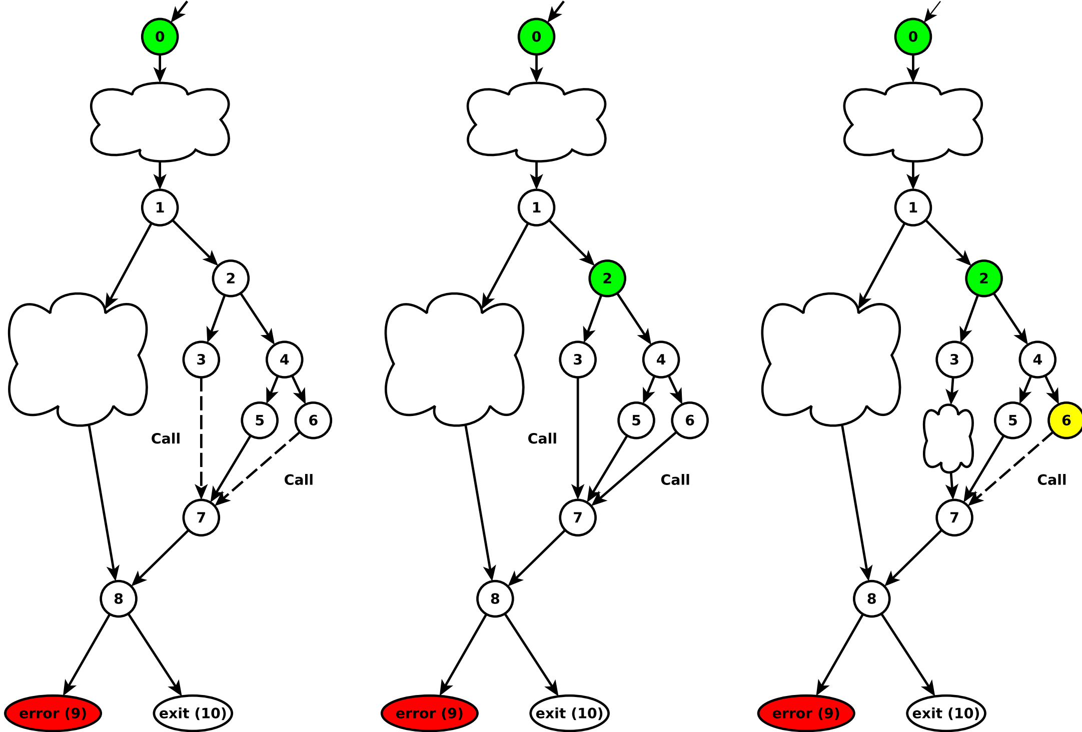


Figure 4.2: Our stratified inlining process.

Example 12. *Figure 4.2 offers an example of our inlining process. Assume that clouds in the graph represent (possibly) large call-free node clusters. Assignments and guards are stripped from the transitions to improve readability. Initially, the automaton contains two calls: the transitions between* $\ell_3 \longrightarrow \ell_7$ *and* $\ell_4 \longrightarrow \ell_6$. *Locations colored in green show the locations at which a new path condition was pushed onto the solver.*

When executing the BMC algorithm, we begin by encoding all $\ell_0 \Longrightarrow \ell_{error}$ *paths while under-approximating calls by adding* **assume false** *guards to them. If the under-approximation was unsuccessful, we pop the last condition from the solver and we calculate the path condition between the entry location and the lowest common dominator of the call transitions* (ℓ_2). *This path condition is then pushed onto the solver and we begin over-approximation: we calculate the condition between* ℓ_2 *and* ℓ_{error}. *If the over-approximated formula was satisfiable, we pop the last formula from the solver, inline calls present in the counterexample (in this instance,* $\ell_3 \longrightarrow \ell_7$*) and start with the under-approximation step once again. If this step is yet again unsuccessful, we set* ℓ_6 *as the new starting point (marked with yellow color), pushing the encoding between* ℓ_2 *and* ℓ_6 *onto the solver.*

4.2 CEGAR-based Model Checking with THETA

Counterexample-Guided Abstraction Refinement (CEGAR) is an automatic algorithm that starts with a coarse initial abstraction and refines it based on the counterexamples until the proper precision is reached [18, 36]. CEGAR was first described using transition systems and predicate abstraction, but since then many variants have been developed.

The *abstraction* algorithm maintains an *abstract reachability graph* (ARG) which is an over-approximation of the state space over an *abstract domain* with a given initial precision. As it is an over-approximation, if no erroneous state is reachable in the ARG, then the whole program can be considered safe. On the other hand, reachability of an error state (shown by an *abstract counterexample*) may or may not indicate the presence of an actual error. If the abstract counterexample is feasible in the original model, the program is considered unsafe. Otherwise, a *refinement* algorithm adjusts the precision of the abstraction.

In our work, we shall use THETA [57], a highly configurable CEGAR framework, which incorporates many variants of the algorithm. As discussed above, the framework explores the abstract state space in a given abstract domain with a given *exploration strategy*. Currently, Boolean predicate abstraction [33, 6], Cartesian predicate abstraction [6] and explicit value [19] domains are supported. Supported exploration strategies are breadth-first (BFS), depth-first (DFS), and error-oriented [36]. The framework supports refinement based on binary [25], backward binary [36] or sequence interpolation [59], as well as unsat cores [46]. The model is then checked again with the refined abstraction and this procedure is repeated until the model is proven to be safe or a feasible counterexample is found.

Translating Programs into the Format of THETA

One of distinguishing features of THETA is that it is able to handle multiple input formalisms, such as symbolic transition systems, timed automata, and control flow automata. Each input formalism has its own domain-specific language (DSL), in which said formalism can be represented. As we use control flow automata within GAZER, our main task is the translation of GAZER automata into the CFA format of THETA. Currently, THETA imposes strict restrictions upon its input format. As an example, automaton calls are not supported – all procedures must be inlined into the main procedure. Furthemore, currently there is no support for bit-vectors and floating-point operations, thus these must over-approximated by using unbounded mathematical integers and undefined values, respectively. While it is possible to approximate floating-point operations using rational arithmetic, our current implementation only replaces them with undefined values at the moment.

As there is no support for recursion in THETA, we must transform our recursive automata into cyclic ones, i.e. replace all tail-recursive automaton calls with loops. We do this by traversing the call graph of the input system, starting from the entry procedure, and looking for tail-recursive procedures. If we find a tail-recursive automaton $\mathcal{A}$ with the entry location ℓ_0, we inline into the entry automaton $\mathcal{A}_{\text{main}}$. All locations and edges of $\mathcal{A}$ are cloned into $\mathcal{A}_{\text{main}}$, such that each ℓ_i location in $\mathcal{A}$ will have a corresponding ℓ_i' in $\mathcal{A}_{\text{main}}$. Naturally, the same applies to the edges and variables of $\mathcal{A}$.

The recursive call is transformed into a back-edge, pointing to the inlined entry location clone ℓ_i'. Input and output arguments placed on the original call transition are replaced with the corresponding assignments. The original procedure call in $\mathcal{A}_{\text{main}}$ is replaced with a simple transition into ℓ_i', where all inlined input variables are initialized to the values of the call parameters.

During the recursive-to-cyclic transformation, we maintain a map of locations and variables back into the original CFA, which will be later used for traceability information (Section 4.3). Next, we build an internal THETA CFA syntax tree, which maps THETA entities into our automaton representation. Finally, this syntax tree is serialized into the textual format of the THETA CFA DSL, which is then passed to the model checker.

4.3 Traceability

One important characteristic of software model checking algorithms is that they can produce a counterexample. This counterexample is a trace which describes how an erroneous state may be reached from the entry point of the program. However, counterexamples may come in many abstraction levels and formats, such as:

- the violation model extracted from (and in the language of) the underlying SMT solver,
- a custom trace which maps assignments to the formal model used by the model checking engine,
- a standardized witness format [8] which then may be used by other tools to validate the counterexample.

The further away the counterexample is from the source language, the harder it is for a programmer to understand and interpret the results. While many tools offer visualization and interpretation aids to help the user trace the counterexample back to the original program, these are often not sufficient enough to be adopted into a proper software development workflow. Studies [41, 16] show that false positives (i.e. unreproducible counterexamples) and hard-to-understand tool outputs are two major reasons of why analysis tools are still not widely adopted by developers.

Due to the reasons listed above, providing informative and reproducible error traces was one of the most important goals during development. To achieve this, we have leveraged the power of some instrumentation methods (as described in Section 3.1) and LLVM's source-level metadata and debug information. We have developed a convenient, language-agnostic, source-level error trace representation format, which will be generated from the counterexamples coming from the verification backends.

4.3.1 The Trace Format of GAZER

Traces within GAZER are composed of *trace steps*. A trace step is a single event of interest, such as a variable assignment or entering a function. Currently we support the following kinds of trace steps:

- variable assignments,
- entering a function,
- returning from a function (with or without a return value),
- calls to external functions,
- undefined behavior.

Certain trace steps, such as assignments refer to *trace variables*, which store the name, location, size, and encoding (signed, unsigned, float, etc.) of said variable. All values related to a particular trace variable are represented in the appropriate encoding of the variable. This is important because LLVM and the underlying verification engine has no distinction between signed or unsigned types – all they see are bit-vectors, with different operations, thus the resulting counterexample will make no distinction either. Storing the proper encoding allows us to avoid presenting a negative number as the value of an unsigned variable to the developer.

4.3.2 From Counterexamples to Traces

Figure 4.3 gives a brief overview of our trace transformation process. Verification backends yield counterexamples in their own custom counterexample format. Our process translates these counterexamples into the trace format of GAZER, using the traceability information available in the LLVM module. The translation process begins with the counterexample acquired from the used verification backend. This is translated into a counterexample format designed for the CFA formalism used within GAZER, which we will refer to as the *raw trace*. The raw trace is an alternating sequence of $(\ell_0, \sigma_0, \ell_1, \sigma_1, \ldots, \sigma_{n-1}, \ell_e)$. Each ℓ_i is a location along an error path between ℓ_0 and ℓ_e, while σ_i is a set of *actions* (i.e. assignments) that get executed when control jumps from one location to another. Note that these actions are different from the edges used within a CFA, as contrary to CFA edges, these assignments may also represent interprocedural information and global state.

Counterexamples returned by the verification backends are first translated into raw traces. For the BMC backend, this requires tracking each inlined location and variable back to its original automaton. In the case of the THETA backend, its native counterexample format[2] must be parsed and translated into a raw trace.

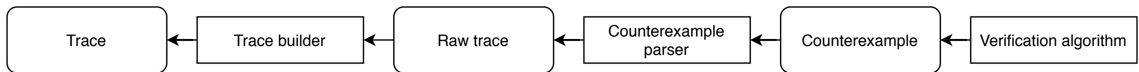


Figure 4.3: Our trace transformation workflow.

The raw trace is then transformed into the final trace format by "replaying" it over the original LLVM module. During trace construction, we maintain the mapping $\Sigma : \mathit{Vars} \rightarrow \mathit{Exprs}$, which will contain the actual value of each variable at the time we process ℓ_i. Starting from ℓ_0, for each location ℓ_i, we update Σ according to the assignments in σ_i. We use the encoded traceability information to determine which LLVM block the location ℓ_i corresponds to. Then we iterate over each instruction in the block, searching for traceability intrinsics and other instructions of interest. These are handled according to the following rules.

- **`llvm.dbg.value(v, var, e)`**: A new assignment event for the variable **`var`** with the current value of **`v`**. Complex expressions (the expression described by **`e`**) are not supported at the moment.
- **`gazer.function.entry.*(fn, args...)`**: A new function entry event into the function **`fn`** with the argument list **`args`**.
- **`gazer.function.return_value.*(fn, v)`**: A new function exit event from the function **`fn`** and return value of **`v`**.
- **`%1 = gazer.undef_value.*()`**: We store the information that reading the value of **`%1`** results in undefined behavior. Note that reaching this function (inserted during the process described in Section 3.1.1) does not necessarily indicate undefined behavior – it only occurs of the value of this register is actually read.

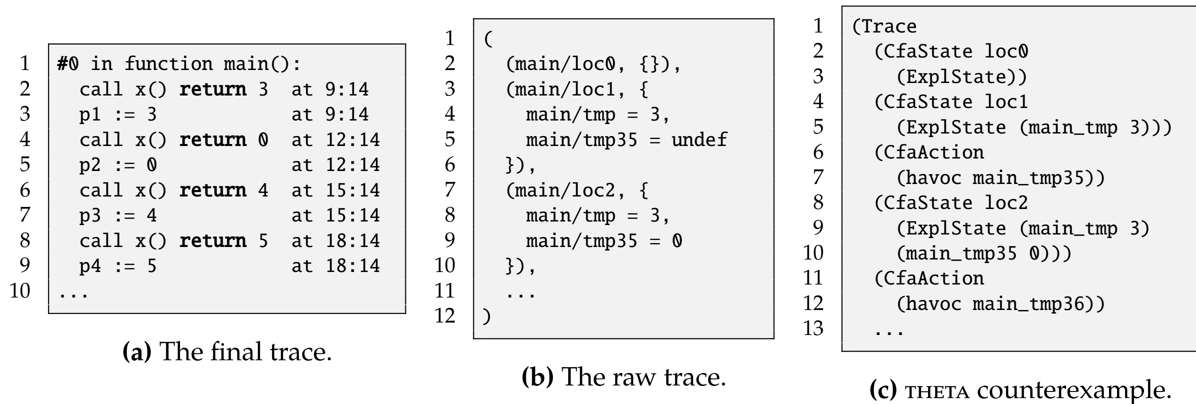
Calls to other nondeterministic functions insert a nondeterministic call event, which will be used later on in our test harness generation process. Operations using register operands marked as "possibly-undefined" trigger the insertion of an undefined behavior event.

[2]A structured LISP-like S-expression tree.

Variables, program locations and scopes are found and resolved using the LLVM source-level debugging and metadata tools described in Section 2.1.4. The variable encoding (signed, unsigned, floating-point, etc.) is determined by reading the type metadata available for each variable.

After the process finished, the resulting trace will contain all important steps of a possibly erroneous execution. This trace may then be transformed and represented in several ways, one of which is the textual error message printed out for the developer. As the text message lists values and variables as they are in the orignal C program, it offers an easy-to-understand error message.

Example 13. *Consider the possible counterexample snippet from the* THETA *verifier, shown in Figure 4.4c. This counterexample is interpreted for the* THETA *CFA, therefore we translate it into the raw trace format which is defined for the CFA formalism of* GAZER*, shown in Figure 4.4b. Finally, this raw trace is transformed into the final trace format using the inserted LLVM instrumentation and metadata, resulting in a trace similar to 4.4a.*



(a) The final trace.

(b) The raw trace.

(c) THETA counterexample.

Figure 4.4: Trace transformation example.

4.3.3 Executable Test Harnesses

Another way to report error traces in a more developer-friendly manner is by representing them as so-called *executable mock environments* [31, 10]. Such an environment is a module which may be linked together with the original program, thus producing an executable test harness that triggers the erroneous execution. This offers several benefits. First of all, developers may check and validate counterexamples easily, just by executing the test harness. Furthermore, as the test harness is a simple executable, the full repertoire of debugging utilities (such as `gdb`) is available to the developer to understand the bug found by the verification algorithm.

Executable mock environments are built by generating a definition for each external function declaration and providing an implementation which returns values that trigger the faulty behavior. After linking the mock environment together with the original program, developers may simply execute and debug the resulting program and find the possible assertion failures and arithmetic exceptions. For elusive errors found due to undefined behavior (such as signed arithmetic under- and overflow), the erroneous behavior can be triggered during execution by using code instrumentation methods such as sanitizers.[3]

[3] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

Example 14. *Consider the program shown in Figure 4.5a, which demonstrates an error if the value of variable* **y** *is such that the addition causes the value of* **x** *to overflow. The generated mock environment*[4] *shown in Figure 4.5b and Figure 4.5c provides an implementation which tracks how many times the function was called and returns an appropriate value. The first call is the loop iteration limit, and the second is the value of* **y** *within the loop – a value high enough that causes the addition to overflow, thus yielding zero as the result.*

```c
#include <assert.h>

unsigned __VERIFIER_nondet_uint(void);

int main(void)
{
  unsigned i = 0;
  unsigned c = __VERIFIER_nondet_uint();

  unsigned x = 1;
  while (i < c) {
    unsigned y = __VERIFIER_nondet_uint();
    x = x + y;
    ++i;
  }

  assert(x != 0);

  return 0;
}
```

(a) An example program.

```c
unsigned __VERIFIER_nondet_uint(void)
{
  static unsigned test_vector[] = { 1, 4294967295 };
  static unsigned test_cnt = 0;

  return test_vector[test_cnt++];
}
```

(b) C implementation of the mock environment.

```llvm
@gazer.test_vector0 = private constant [2 x i32] [i32 1, i32 -1]
@gazer.test_cnt0 = private global i32 0

define i32 @__VERIFIER_nondet_uint() {
entry:
  %0 = load i32, i32* @gazer.test_cnt0
  %1 = add i32 %0, 1
  store i32 %1, i32* @gazer.test_cnt0
  %2 = getelementptr inbounds [2 x i32], [2 x i32]* @gazer.test_vector0, i32 0, i32 %0
  %3 = load i32, i32* %2
  ret i32 %3
}
```

(c) LLVM IR implementation of the mock environment.

Figure 4.5: Mock environment generation example.

[4]While the example shows a C implementation of the mock environment to help readability, our tool currently emits mock environments only in the LLVM IR. However, as the concept is universal, it is very easy to write a mock generator that emits C code.

Chapter 5

Implementation

This chapter describes some of the implementation details of GAZER. Section 5.1 gives an overview of the overall architecture, Section 5.2 describes some of our most notable technical solutions that could potentially be applied in other contexts as well. Section 5.3 concludes this chapter with a short user documentation and usage example.

5.1 Architecture

Figure 5.1 presents an overview of the GAZER architecture. The system is divided into several libraries. The core library, `GazerCore` contains all components related to types, expressions and control flow automata. This core library acts as the link between the verification frontend (`GazerLLVM`) and backends (`GazerBMC`, `GazerTheta`). As the bounded model checker requires an SMT-solver to operate, we also provide an interface for the Z3 [27] SMT solver (`GazerZ3Solver`).

The verifier component (`GazerVerifier`) offers useful interfaces for the verification backends, so that they can tie into the verification process seamlessly. The traceability library (`GazerTrace`) is used to produce counterexample traces and is present in every step of the verification workflow. Verification libraries are bundled together with the LLVM frontend and the traceability component into separate tools such `gazer-bmc` and `gazer-theta`.

All components were implemented in C++17, with dependencies on LLVM [44], the Boost software libraries[1] and the Z3 theorem prover [27]. The implemented tool is available online[2] under an open-source license.

5.2 Technical Solutions

This section lists some of the notable technical challenges and solutions we have encountered during the development of GAZER. All the solutions described in the following has required the author to further expand his knowledge about software architectures and C++ alike. We believe that the solutions presented below would be of interest to anyone wishing to implement a tool similar to GAZER.

[1] `https://www.boost.org/`

[2] `https://github.com/ftsrg/gazer`

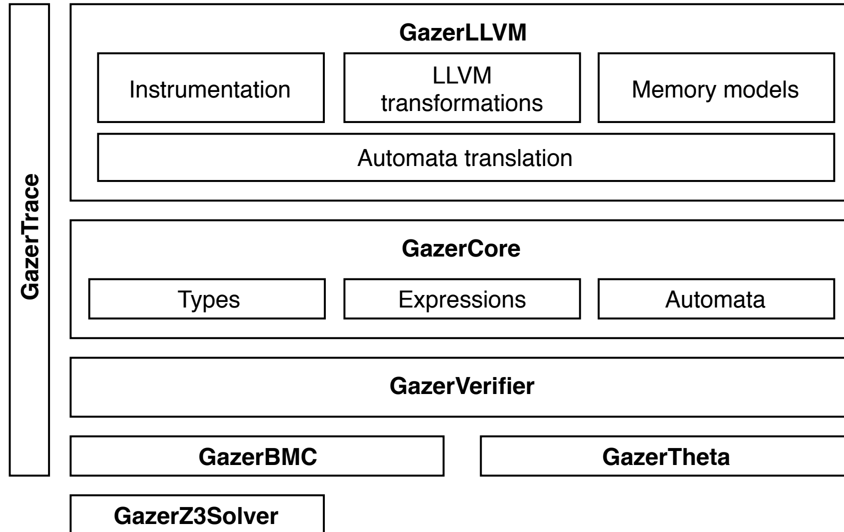


Figure 5.1: The architecture of GAZER.

5.2.1 Lifetime Management

The central entity of the GAZER infrastructure is the `GazerContext` class. It uniquely owns each type, variable, and expression, managing their lifetimes. Types and variables live "forever" and they are only deleted when their enclosing context is destroyed. Expressions, on the other hand, are reference-counted and allocated using the so-called *flyweight* pattern.

Flyweight is a structural design pattern that caches redundant immutable objects in a shared repository and returns lightweight handles to them. Objects are stored in and created by the so-called *flyweight factory*. Each time there is a request for a new object, the flyweight factory checks whether it already exists in the cache. If no such item is present in the cache, the flyweight factory will construct a the new object, place it into the cache, and return a handle to it. Otherwise, it just returns a handle to the already existing cached object.

In the case of GAZER, the flyweight factory is a class called `ExprStorage`, which is implemented as an intrusive hash table. As expression classes are immutable, all information about them is known at the time of construction. When a new expression object is requested, we calculate its hash code from the constructor arguments and check whether the expression storage already contains it. If so, we merely increment the reference count and return a handle to the stored expression, otherwise a new expression is constructed. This offers us two main benefits.

- Expressions belonging to the same context are semantically equal if and only if their addresses are equal. This means that instead of traversing a possibly huge expression tree to check semantic equivalence, we can do so by a mere pointer comparison.
- Memory usage is reduced, as an expression cannot be allocated simultaneously at different places. This is considerable in the case of constants as they are frequently repeated.

Reference counting is done via intrusive smart pointers (built upon the `intrusive_ptr` utility from the Boost C++ library), which we call `ExprRef`. If an expression's reference count drops to zero, the smart pointer automatically deletes it.[3]

Figure 5.2 shows the structure of the expression storage with an example of four expressions. As both reference counting and the hash table bucket list is intrusive, their bookkeeping data are embedded within the allocated expression object.

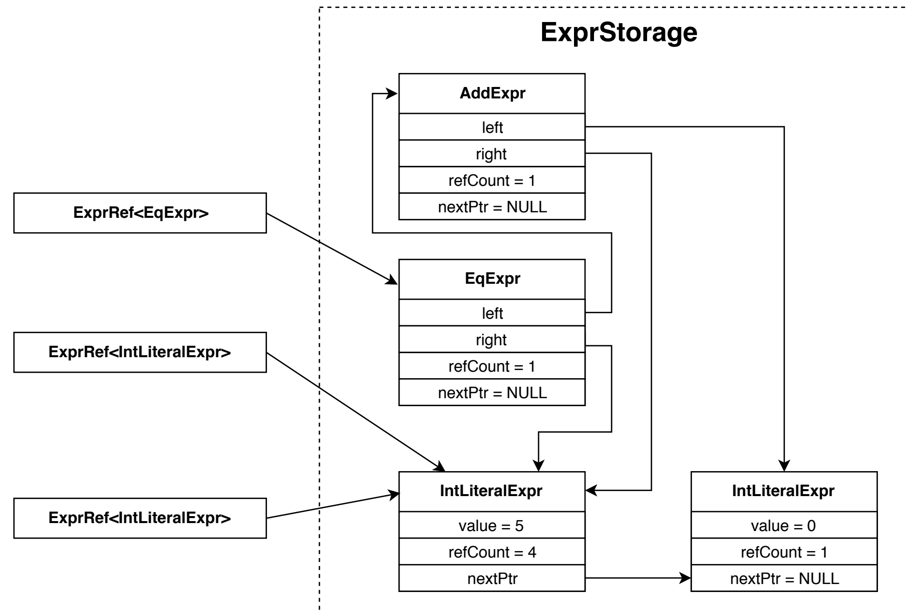


Figure 5.2: A possible state of the expression storage.

5.2.2 Expression Pattern Matching

In order to reduce expression size and help the underlying solver or model checking engine, we perform some automatic reductions on the generated expressions. In GAZER, we do this by using a set of tree-based pattern matching utilities. Each expression class has its own matcher, and each matcher may accept additional matchers to perform further matching on operands or a terminal to bind an operand to variable.

Matchers are implemented using the advanced template metaprogramming features offered by newer versions of C++, such as variadic templates[4] and `if constexpr`.[5]

The code in Figure 5.3 shows an usage example: given the expression in `input`, we use a set of *and* (`m_And`), and *or* (`m_Or`) matchers.[6] After finding the operands we are interested in, we bind them using the `m_Expr` terminal matcher. Having recognized the pattern, we can return a smaller, optimized expression using the bound operands.

[3]This is not entirely trivial as deleting considerably deep expression trees may overflow the call stack. On deletion, each expression will call the destructors of its operand pointers, which may destroy their expressions, thus calling operand destructors, and so on. This can cause an overly large call sequence chain of destructors and deletion functions. We overcome this issue collecting each expression that will be deleted and destroying them in one go, reusing the intrusive linked list pointers to avoid heap allocation during expression destruction.

[4]`https://en.cppreference.com/w/cpp/language/parameter_pack`

[5]`https://en.cppreference.com/w/cpp/language/if#Constexpr_If`

[6]The naming convention is based on the similar instruction pattern matching utility found in LLVM.

```
ExprRef<> input = ...
ExprRef<> f1, f2, f3;
if (match(input, m_And(m_Or(m_Expr(f1), m_Expr(f2)), m_Or(m_Specific(f1), m_Expr(f3))))) {
    // (F1 | F2) & (F1 | F3) -> F1 & (F2 | F3)
    return AndExpr::Create(f1, OrExpr::Create(f2, f3));
}
```

Figure 5.3: Expression rewriting using patterns.

Using this, we can perform expression simplification by defining a set of pattern-matching rules. As expressions are immutable, we can perform this step right at the point of construction, therefore we use a special `FoldingExprBuilder` class to construct and return already optimized expressions. Expressions built through the folding expression builder are guaranteed to be constant-folded, and we also define about 30 complex patterns to simplify expressions further.

5.2.3 Expression Visitors

Visitor is a well-known design pattern used to traverse heterogeneous data structures and perform some type-dependent actions. In the context of GAZER, visitors are used to translate our expression language onto the representation of the theorem prover. However, translating the possibly large formulas built during bounded model checking (using the method discussed in Section 4.1) may overflow the call stack.

In order to mitigate this problem, we introduced our own expression walker interface (called `ExprWalker`), which avoids recursion by using an explicit stack on the heap instead of the normal call stack. This allows us to avoid stack overflow errors in the case of large input expressions at the cost of imposing slightly more restrictions on the implementing classes. The order of the traversal is fixed to be post-order and a particular visitation step may query the class for the result of an already translated operand. This traversal order is convenient for expression rewriting and transformations, however it disallows custom traversals. To speed up transformations, visitors often wish to use caches to store the result of an already executed transformation. Our expression walker interface allows this by providing the methods `shouldSkip` and `handleResult`. The former method can query the cache and return true if it was hit, thus skipping the transformation of the input expression. The latter function may be used to insert already translated entries into the cache.

While the easiest way to provide a proper stack representation on the heap is by allocating each stack frame as a separate node, this comes with some obvious performance penalties. To avoid this, we have implemented a custom stack-like allocator, which allocates slabs of a given size (the default setting is 4 kilobytes) in a LIFO-like fashion, implemented as a linked list. If a slab is full, the allocator allocates another slab and appends it to the beginning of the slab list. When a slab gets empty, the allocator removes it from the list and deallocates the memory assigned to it. Our custom stack frames therefore allocated on the heap through this stack-like allocator and initialized using C++'s placement new[7] syntax.

[7] `https://en.cppreference.com/w/cpp/language/new#Placement_new`

5.3 Usage

As discussed previously, GAZER comes with two verification backends: the built-in bounded model checking engine and the one using the THETA model checker. These are deployed into two separate executables: `gazer-bmc` and `gazer-theta`. Each tool accepts a list of `.c`, `.ll` and .bc files, which are compiled and linked into a single LLVM IR module using Clang and `llvm-link`. The Clang compilation phase accepts some of the most common C compiler flags, such as preprocessor macro definitions (`-D`), include search paths (`-I`), and warning message preferences (`-W`). Later stages define their own set of settings. These settings and their arguments are summarized in Table 5.1.

Note that some of settings are forced in certain situations. For example, `--inline` and `--math-int` is a forced setting (meaning that they are implicitly turned on) in case of `gazer-theta`, as the underlying THETA model checker does not support function calls and bit-vectors.

Apart from the environment settings, `gazer-theta` accepts all algorithm settings specified by THETA, such as abstract domain (`--domain`), predicate splitting (`--predsplit`), refinement strategies (`--refinement`), and so on.[8] These settings are then passed to the THETA verification engine.

Example 15. *Consider the C program shown in Figure 5.4a (`example.c`). This program may attempt to divide by zero for certain values of `k`. We can verify this program with either verification backend:*

```
gazer-bmc   --trace --test-harness=harness.ll --bound 10 example.c
gazer-theta --trace --test-harness=harness.ll --refinement=SEQ_ITP
            --search=BFS example.c
```

The former command executes the bounded model checking workflow (with the maximum bound of 10) on `example.c`, while the latter uses the THETA framework with the sequence interpolation refinement and BFS search strategies.

In both cases, the verifier will return the verification verdict and error trace shown in Figure 5.4b. As discussed in the previous chapters, GAZER may strip away some of the variables it proves to be irrelevant to the verification task. However, such removed variables may still show up in the trace as undefined values (indicated by the text "???"). This behavior can be turned off using the `--no-optimize` and `--elim-vars=off` flags.

As the error trace shows, division by zero may occur the `ioread32` function returns -11. The requested test harness (`harness.ll`) contains the definition of each function which was declared but not defined within the input module. Linking this test harness against the original module yields an executable which may be used to replay the error scenario, as shown in Figure 5.4c.

[8] The interested reader is invited to consult the THETA paper [57] and website (`https://github.com/FTSRG/theta`) for further information.

Table 5.1: Command-line options.

Clang compilation settings	
-D=<macro[=<value>]>	Define <macro> to <value> (or 1 if <value> is ommited).
-I=<dir>	Add <dir> to the include search paths.
-W=<warning>	Enable the specified warning.
LLVM frontend settings	
--inline	Inline all non-recursive function calls into the entry procedure.
--inline-globals	Lower global variables into locals. May only be used together with --inline.
--no-assert-lift	Do not perform assertion lifting.
--no-optimize	Do not run LLVM optimizations passes.
LLVM IR translation settings	
--elim-vars	Variable elimination setting. Accepted values are: off, normal (default), aggressive.
--math-int	Represent integer values as unbounded integers instead of bit-vectors.
--no-simplify-expr	Do not simplify expressions.
Traceability settings	
--trace	Print counterexample trace.
--test-harness=<file>	Write test harness into <file>.
Bounded model checking settings (gazer-bmc only)	
--bound=<bound>	Set the maximum verification bound to <bound>.
--eager-unroll=<bound>	Unroll <bound> iterations eagerly.
Theta environment settings (gazer-theta only)	
--theta-path=<dir>	Full path to theta-cfa JAR file, defaults to the gazer-theta directory.
--lib-path=<dir>	Full path to the directory containing the Z3 libraries required by theta. Defaults to the gazer-theta directory.
--model-only	Do not run the verification engine, just dump the CFA in THETA's format.

```
1   #include <stdio.h>
2
3   extern int ioread32(void);
4
5   int main(void) {
6       int k = ioread32();
7       int i = 0;
8       int j = k + 5;
9
10      while (i < 3) {
11          i = i + 1;
12          j = j + 3;
13      }
14
15      k = k / (i - j);
16
17      printf("%d\n", k);
18
19      return 0;
20  }
```

(a) A simple C program.

```
1   Verification FAILED.
2   Divison by zero in example.c at line 15 column 11.
3   Error trace:
4   ------------
5   #0 in function main():
6     call ioread32() returned -11       at 6:13
7     k := -11  (0b11111111111111111111111111110101)     at 6:13
8     i := 0    (0b00000000000000000000000000000000)
9     j := ???
10    i := 0    (0b00000000000000000000000000000000)
11    i := 1    (0b00000000000000000000000000000001)     at 11:15
12    j := ???
13    j := ???
14    i := 1    (0b00000000000000000000000000000001)
15    i := 2    (0b00000000000000000000000000000010)     at 11:15
16    j := ???
17    j := ???
18    i := 2    (0b00000000000000000000000000000010)
19    i := 3    (0b00000000000000000000000000000011)     at 11:15
20    j := ???
21    j := ???
22    i := 3    (0b00000000000000000000000000000011)
23    i := 3    (0b00000000000000000000000000000011)
24    j := 3    (0b00000000000000000000000000000011)     at 10:5
```

(b) The error trace.

```
1   clang example.c harness.ll -o example_test
2   ./example_test
3   [1]    19948 floating point exception (core dumped)  ./example_test
```

(c) Checking the executable test harness.

Figure 5.4: Usage example on a small C program.

Chapter 6

Evaluation

This chapter presents an evaluation of our implemented tool. Our goal is twofold: first, we want to demonstrate the applicability of our transformation workflow. Second, GAZER offers a variety of transformation options, which we would like to compare to each other.

Section 6.1 of this chapter describes our benchmark goals and environment, Section 6.2 discusses our benchmark results. Finally, Section 6.3 summarizes our findings.

6.1 Benchmark Environment

The verification task set is acquired from the annual Competition on Software Verification (SV-Comp) [8] benchmark suite. They may be split into four distinct categories.

locks This task set consists of small (100-150 LOC) locking mechanisms described using nondeterministic integers and if-then-else constructs.

eca The ECA (event-condition-action) category describes event-driven reactive systems from moderate-sized programs to large ones (600-70000 LOC). The events are represented by nondeterministic integer variables, the conditions are simple if-then-else statements. While syntactically simple, verifying these programs requires verifying some of the largest models in the competition repertoire.

ssh-simplified These tasks describe moderate-sized (500-600 LOC) server-client systems. While these systems are rather complex, verifying the server-client communication is not part of this task, such factors are abstracted away with nondeterministic variables.

bitvectors The task set consists of simple bitvectors manipulations, such as addition, casting, etc. Most programs in this task set are small in size (100-300 LOC), but require bit-precise modeling of the system.

Our experiment was done using the BMC and THETA backends, with the combination of several flags presented in Table 5.1. The LLVM IR translation process was tweaked using the `--elim-vars`, `no-optimize` and `--no-simplify-expr` flags. For the locks and ECA categories, we also experimented with using mathematical integers instead of bit-vectors using the `--math-int` flag.

Contrary to previous experiences with our old frontend [54], preliminary measurements have established that program slicing was not doing any meaningful reductions in the

case of GAZER. Upon further investigation, we have found that picking the single error call as the criterion instruction introduced most of the program as a dependence of the error call into the PDG, not allowing the slicer to remove any instructions apart from some traceability instrumentation (which we wanted to retain). As such, we have chosen to omit program slicing from this experiment. Furthermore, while our proof-of-concept flat memory model worked for small programs with simple array operations, it did not scale well for any practical benchmarks. Therefore we have decided to drop the memory model and enable inlining for all benchmark programs. Another issue came with the representation of larger programs for the THETA backend: programs in the ECA task set were taking too long to verify, despite working well with our previous frontend. Combined with the fact that THETA is unable to handle bit-vector operations, this meant that we had to restrict the evaluation of THETA to the locks task set.

As there are 3 variable elimination settings, 2 optimization settings, 2 expression simplification settings and 2 integer representation strategies, the total number of possible configurations is $3 \cdot 2 \cdot 2 \cdot 2 = 24$ for the BMC backend. In the case of THETA, we also experimented with different abstract domains (`PRED_CART`, `PRED_BOOL`, `EXPL`), interpolation strategies (`SEQ_ITP`, `BW_BIN_ITP`) and precision granularity settings (`GLOBAL`, `LOCAL`). As the mathematical integers setting is forced for the THETA backend, the final number of configurations is $3 \cdot 2 \cdot 2 \cdot 3 \cdot 2 \cdot 2 = 144$. With a such large number of possible configurations, we have chosen to use a pairwise combination [26] of settings (meaning that final configuration set covers all combinations of any two settings), generated by the PICT[1] tool.

All measurements were carried out on the following configuration:

- Intel Xeon Platinum 8167M CPU @ 2.00 GHz,
- 320 GB RAM (memory limit of 16 GB),
- Ubuntu Linux with Linux Kernel 4.15.0, 2019 x86_64 GNU/Linux.

6.2 Benchmark Results

This section presents the benchmark results for each benchmark category. All measurements were ran 3 times with a time limit of 5 minutes. We consider a verification run *verified* (i.e. successful) if it successfully verifies the input model at least once within the given time limit.

Unless indicated otherwise, results will be shown in a table with four columns. The first column lists the used configuration flags (except those which are present in every configuration, such as `--inline`), while the second column contains the number and ratio of the verified models. The third column shows the sum of the total verification time (without timeouts – all runs resulting in a timeout have a verification time of zero), including the compilation, instrumentation, optimization, and verification phases. In the case of the BMC backend, the additional fourth column shows the total time spent within the SMT solver.

Each table is first ordered by the ratio of verified programs (the further ahead a configuration is in the table, the more programs it verified), while the secondary sorting is based on total verification time (i.e. faster configurations show up earlier).

[1] `https://github.com/microsoft/pict`

6.2.1 Evaluation of the BMC Backend

This section presents our measurement results using the bounded model checker backend of GAZER. All measurements were ran with inlining enabled and the maximum bound of 12. Therefore all commands were in the form of

```
gazer-bmc --inline --inline-globals --bound=12 <config> <input>,
```

where <config> stands for one of the configuration combinations presented below.

The locks task set

Measurement results of the **locks** benchmark suite are shown in Table 6.1. As we can see, the bounded model checking algorithm performed extremely well with all configurations. While certain configurations were slightly slower, the magnitude of the verification times (most combinations verified 13 programs just under a second) does not allow us to declare any meaningful difference between configurations.

Table 6.1: Benchmarks results for the locks program set, using our BMC backend.

Configuration	Verified	Total time	Solver time
--elim-vars=aggressive --math-int	13/13	< 1s	< 1s
--elim-vars=normal	13/13	< 1s	< 1s
--elim-vars=off --math-int --no-simplify-expr	13/13	< 1s	< 1s
--elim-vars=normal --no-simplify-expr	13/13	< 1s	< 1s
--elim-vars=normal --math-int --no-simplify-expr --no-optimize	13/13	< 1s	< 1s
--elim-vars=aggressive --no-simplify-expr --no-optimize	13/13	1.5s	< 1s
--elim-vars=off --no-optimize	13/13	2.3s	< 1s

The ECA task set

The measurement results for the ECA task set is shown in Table 6.2. As we can see, turning off variable elimination greatly degrades the performance of the verification algorithm, resulting in a timeout more than half of the cases. Unsurprisingly, changing the integer representation from bit-vectors to unbounded mathematical integers results in the best performance.

As this benchmark set has resulted in the most meaningful results, we present the median verification time of each program with the best configuration combination in Table 6.3. As we can see from the results, the bounded model checking algorithm scales fairly well, being able to find the problems in programs with more than 10000 lines of code. Note that there is a mismatch between the expected and actual verification verdict in the case of two input programs, eca4_label09_false.c and eca4_label11_false.c. This is due to the incomplete nature of the bounded model checker algorithm: as the maximum bound was set to be 12, it could not find the problem in these programs within the given bound, therefore the verifier returned SAFE.

The column **Iterations** tells how many iterations (i.e. bound increases) has the algorithm performed. One advantage of the stratified inlining algorithm as opposed to static inlining is that the verification algorithm can terminate as soon as it encounters an error, clearly shown by the iteration count.

The bitvectors task set

Table 6.4 shows our results for the bitvectors program set. All configurations were able to verify all programs, but similarly to the ECA task set, turning off variable elimination yields the worst performance. Interestingly, the best configurations are the ones without LLVM optimizations (`--no-optimize`) with expression simplification turned on, followed by the ones without expression simplification. While our pairwise combination method does not allow us to derive definitive conclusions, the results suggest that LLVM optimizations slightly slow the verifier down.

Table 6.2: Benchmarks results for the ECA program set.

Configuration	Verified	Total time	Solver time
`--elim-vars=aggressive --math-int`	15/18	607s	496s
`--elim-vars=normal --math-int --no-simplify-expr --no-optimize`	15/18	680s	559s
`--elim-vars=normal`	15/18	839s	727s
`--elim-vars=normal --no-simplify-expr`	14/18	516s	427s
`--elim-vars=aggressive --no-simplify-expr --no-optimize`	14/18	686s	600s
`--elim-vars=off --math-int --no-simplify-expr`	7/18	114s	102s
`--elim-vars=off --no-optimize`	7/18	193s	180s

Table 6.3: Individual results of the best-performing ECA benchmark.

Program name	Size (LOC)	Expected result	Actual result	Time	Solver time	Iterations
eca1_label00_true.c	594	SAFE	SAFE	<1s	<1s	12
eca1_label15_false.c	594	UNSAFE	UNSAFE	<1s	<1s	5
eca1_label20_false.c	594	UNSAFE	UNSAFE	<1s	<1s	7
eca1_label31_true.c	594	SAFE	SAFE	<1s	<1s	12
eca2_label00_true.c	617	SAFE	SAFE	<1s	<1s	12
eca2_label13_false.c	617	UNSAFE	UNSAFE	<1s	<1s	3
eca3_label00_true.c	1669	SAFE	SAFE	13s	11s	12
eca3_label09_false.c	1669	UNSAFE	UNSAFE	2s	1s	6
eca4_label00_true.c	4827	SAFE	SAFE	51s	39s	12
eca4_label09_false.c	4827	UNSAFE	**SAFE**	76s	64s	12
eca4_label11_false.c	4827	UNSAFE	**SAFE**	74s	61s	12
eca4_label29_true.c	4827	SAFE	SAFE	52s	40s	12
eca5_label00_false.c	11141	UNSAFE	UNSAFE	70s	57s	8
eca5_label02_true.c	11141	SAFE	SAFE	234s	201s	12
eca6_label00_false.c	9484	UNSAFE	UNSAFE	38s	28s	6
eca6_label03_true.c	9484	SAFE	TIMEOUT	300s	-	-
eca7_label00_true.c	73698	SAFE	TIMEOUT	300s	-	-
eca7_label03_false.c	73698	UNSAFE	TIMEOUT	300s	-	-

The ssh-simplified task set

The ssh-simplified benchmark set (measurements shown in Table 6.5) further verifies our suspicion that turning off LLVM optimizations actually improves performance. Same as

Table 6.4: Benchmarks results for the bitvectors program set.

Configuration	Verified	Total time	Solver time
`--elim-vars=aggressive --no-optimize`	4/4	11s	10s
`--elim-vars=normal --no-optimize`	4/4	16s	14s
`--elim-vars=aggressive --no-simplify-expr`	4/4	24s	22s
`--elim-vars=normal --no-simplify-expr`	4/4	34s	32s
`--elim-vars=off --no-simplify-expr --no-optimize`	4/4	163s	160s
`--elim-vars=off`	4/4	170s	166s

Table 6.5: Benchmarks results for the ssh-simplified program set.

Configuration	Verified	Total time	Solver time
`--elim-vars=normal --no-optimize`	9/12	146s	139s
`--elim-vars=aggressive --no-optimize`	9/12	156s	147s
`--elim-vars=off --no-simplify-expr --no-optimize`	9/12	165s	156s
`--elim-vars=normal --no-simplify-expr`	9/12	266s	260s
`--elim-vars=aggressive --no-simplify-expr`	9/12	307s	300s
`--elim-vars=off`	8/12	16s	11s

in the case of the benchmarks shown previously, turning variable elimination off greatly reduces performance.

6.2.2 Evaluation of the THETA Backend

This section presents our measurement results using the THETA backend. As discussed in Section 6.1, we had to restrict the evaluation of this backend to the **locks** task set. All measurements were run with a command in the form of

```
gazer-theta --inline --inline-globals --math-int <config> <input>,
```

where `<config>` is one of the possible configuration combinations.

The measurement results are shown in Table 6.6. While the verification seems to be somewhat (10-15 seconds) slower compared to the results of the BMC backend, note that each invocation of THETA requires starting a Java virtual machine, introducing some fixed overhead into the verification workflow.

As with our experiences using the BMC backend, turning off variable elimination still seems to cause a noticeable drop in performance. Interestingly, the last configuration was only able to verify 7 programs out of 13 within the given time limit. While backward binary interpolation (`BW_BIN_ITP`) has offered good performance in conjunction with multiple abstract domains when variable elimination was turned on, turning variable elimination off completely broke the performance of this combination.

6.3 Summary

The evaluation and measurement results show the applicability of GAZER. Given the right configurations, the bounded model checker backend performs fairly well, being able to verify larger programs. On the other hand, the THETA CFA generation algorithm seems to generate models which are difficult to handle for THETA.

Table 6.6: Benchmarks results for the locks program set, using the THETA backend.

Configuration	Verified	Total time
`--elim-vars=aggressive --no-optimize --domain=PRED_CART --refinement=BW_BIN_ITP --precgranularity=GLOBAL`	13/13	12s
`--elim-vars=normal --no-simplify-expr --domain=PRED_CART --refinement=SEQ_ITP --precgranularity=GLOBAL`	13/13	12s
`--elim-vars=normal --domain=PRED_BOOL --refinement=SEQ_ITP --precgranularity=GLOBAL`	13/13	12s
`--elim-vars=aggressive --domain=EXPL --refinement=BW_BIN_ITP --precgranularity=LOCAL`	13/13	12s
`--elim-vars=normal --no-simplify-expr --no-optimize --domain=EXPL --refinement=BW_BIN_ITP --precgranularity=LOCAL`	13/13	12s
`--elim-vars=aggressive --no-simplify-expr --no-optimize --domain=PRED_BOOL --refinement=SEQ_ITP --precgranularity=LOCAL`	13/13	12s
`--elim-vars=off --domain=EXPL --refinement=SEQ_ITP --precgranularity=GLOBAL`	13/13	13s
`--elim-vars=off --no-simplify-expr --no-optimize --domain=PRED_CART --refinement=SEQ_ITP --precgranularity=LOCAL`	13/13	17s
`--elim-vars=off --no-optimize --domain=PRED_BOOL --refinement=BW_BIN_ITP --precgranularity=GLOBAL`	7/13	78s

Interestingly, LLVM optimizations have regressed the performance of the verifier, raising the argument that while an optimization may be useful for speeding up executable programs, it may not prove as helpful when trying to improve program verification.

Sadly, while program slicing has proven to be extremely effective with our previous frontend [54], it has failed to perform any meaningful reductions on our check-instrumented programs. Furthermore, the scalability issues with our flat memory model has prevented us from performing proper measurements on programs with memory accesses. Investigating these performance issues and their causes should be a direction of further development.

Chapter 7

Conclusion

The goal of this work was to provide an efficient and user-friendly model checking workflow for C programs, a goal which we believe we have met. During the preparation of this thesis, we have examined the theoretical background of compiler design, program dependency analyses, and software model checking. We have presented several program analysis (e.g. dominance, program dependence, memory analyses, etc.) and transformation (e.g. source-to-model transformation, optimization, program slicing, etc.) algorithms. We also described two model checking algorithms: bounded model checking with stratified inlining and CEGAR. We have also proposed a small and simple optimization technique for the bounded model checking algorithm.

From the practical point of view, we have implemented a moderate-sized (17000 lines of code) formal verification framework in C++, with a full source-to-model transformation workflow, a memory model and a built-in bounded model checker. We have used traceability instrumentation to translate low-level verification verdicts into a meaningful format on the source-code level. We have also added support for the generation of executable test harnesses, i.e. mock environments that allow developers to investigate faults in a familiar environment with tools such as debuggers.

We evaluated GAZER on parts of the Competition on Software Verification benchmark set. The implemented workflow has demonstrated its usability, albeit certain parts of it show signs of poor scalability. We believe that by applying some tweaks and improvements on these components will allow GAZER to be an efficient, accessible and user-friendly model checking library.

Contributions. The following list gives a brief overview of the main theoretical and practical contributions of this work.

- The design and implementation of GAZER, its expression library with some convenience utilities (Section 5.2), and a control flow automaton formalism (Section 3.4.1).
- The LLVM frontend of GAZER (Chapter 3), in which we have implemented a check instrumentation workflow to represent software safety properties as reachability problems with built-in traceability support (Section 3.1). In order to reduce the size of the program, we have used a set of built-in LLVM optimizations from LLVM's transformation library. In addition, we implemented some other transformations present in the literature, such as program slicing and assertion lifting (Section 3.2).

- A generic, extensible memory SSA implementation for LLVM (Section 3.3.1), designed to allow the convenient implementation of a wide variety of memory models, out of which we have shipped a proof-of-concept flat memory model (Section 3.3.2).
- The translation of LLVM instructions into our control flow automaton formalism, w.r.t. a memory model (Section 3.4).
- The implementation of a bounded model checking algorithm using the stratified inlining technique, extended with a optimization step of our own (Section 4.1).
- Seamless integration of the THETA model checking framework into our verification workflow (Section 4.2).
- The traceability component of GAZER, which presents verification results in a human-readable trace format and generates executable test harnesses to further aid the investigation of the erroneous behavior (Section 4.3).

Future work. While the implemented workflow may be considered complete, there are still several directions of future development.

- The implemented flat memory model is very simple and scales poorly. The implementation of a more refined memory model (either by the optimization of the current one or by the implementation of new ones) would allow us to verify real-world programs with memory operations.
- According to measurements, the THETA backend currently suffers from performance problems, likely introduced by the way the CFA generation algorithm has built its input model. This requires further investigation and an eventual fix.
- While our bounded model checking backend offers good performance, it is currently not complete: certain techniques, such as *k-induction* [55] could be used to prove correctness of fault-free programs.
- Currently, GAZER only considers sequential programs: support may be added for parallel programs and operations.

Acknowledgements

First of all, I would like to thank my supervisor, Ákos Hajdu for his support and guidance. His vast knowledge, advice and ideas were indispensable during both research and development. Furthermore, his positive attitude has kept me motivated all throughout the hardships I have encountered during the preparation of this work. I also would like to thank Zoltán Micskei, who has presented me several opportunities of professional development and has constantly followed my progress.

Before the preparation of this thesis, I have spent a year as a technical student intern at CERN. I would like to thank my former supervisor there, Dániel Darvas, who has made my internship professionally fruitful and from whom I learned a lot. I also wish to thank my former office-mate, the C++ sensei Bernhard Gruber, who has shown me tricks of the C++ language that I never would have thought to be possible beforehand.

Last but not least, I wish to thank my family and friends, who have supported me throughout my studies and made my university years a very enjoyable part of my life.

List of Figures

2.1 An example control flow graph. 4
2.2 An example control flow graph and its dominator and post-dominator trees. 5
2.3 Use-define chains example. 7
2.4 SSA transformation of a simple C program. 7
2.5 Program dependency graph of the program and CFG shown in Figure 2.1. 10
2.6 An LLVM IR example. 12
2.7 Some of the metadata defined for the program shown in Figure 2.6. 14

3.1 Transformation workflow. 16
3.2 Undefined value protecting transformation example. 17
3.3 Assertion lifting example. 20
3.4 PDG and a slice of a program. 22
3.5 MemorySSA example. 25
3.6 Transformation of a SSA-formed CFG into a CFA. 29

4.1 SMT encoding example. 32
4.2 Our stratified inlining process. 36
4.3 Our trace transformation workflow. 39
4.4 Trace transformation example. 40
4.5 Mock environment generation example. 41

5.1 The architecture of GAZER. 43
5.2 A possible state of the expression storage. 44
5.3 Expression rewriting using patterns. 45
5.4 Usage example on a small C program. 48

List of Tables

3.1 Memory objects definitions and uses. 24

5.1 Command-line options. 47

6.1 Benchmarks results for the locks program set, using our BMC backend. . . 51
6.2 Benchmarks results for the ECA program set. 52
6.3 Individual results of the best-performing ECA benchmark. 52
6.4 Benchmarks results for the bitvectors program set. 53
6.5 Benchmarks results for the ssh-simplified program set. 53
6.6 Benchmarks results for the locks program set, using the THETA backend. . 54

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic, 2008. Note: Standard 754–1985.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 672–678. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[5] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. HiFrog: SMT-based function summarization for software verification. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 207–213. Springer Berlin Heidelberg, 2017.

[6] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer*, 5(1):49–58, Nov 2003.

[7] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[8] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[9] Dirk Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design*, pages 25–32. IEEE, 2009.

[10] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. Tests from witnesses. In Catherine Dubois and Burkhart Wolff, editors, *Tests and Proofs*, pages 3–23, Cham, 2018. Springer International Publishing.

[11] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[12] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[13] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification*. Springer, 2007.

[14] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. In *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2007.

[15] Rod M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[16] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM.

[17] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936.

[18] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[19] Edmund Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[20] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[21] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[22] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[23] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[25] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[26] Jacek Czerwonka. Pairwise testing in real world practical extensions to test case generators, 2008.

[27] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[28] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[29] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[30] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[31] Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A. Navas, and Edward J. Schwartz. Executable counterexamples in software model checking. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, pages 17–37. Springer International Publishing, 2018.

[32] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, 1996.

[33] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[34] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361. Springer International Publishing, 2015.

[35] Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In Francesco Ranzato, editor, *Static Analysis*, pages 148–168. Springer International Publishing, 2017.

[36] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, Nov 2019.

[37] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

[38] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate Automizer and the search for perfect interpolants. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451, Cham, 2018. Springer International Publishing.

[39] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[40] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[41] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[42] Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 147–154, 2014.

[43] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *Computer-Aided Verification (CAV)*, July 2012.

[44] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[45] K. Rustan M. Leino. This is Boogie 2. June 2008.

[46] Martin Leucker, Grigory Markin, and Martin R. Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Hardware and Software: Verification and Testing*, volume 9434 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2015.

[47] J. McCarthy. Towards a mathematical science of computation. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification: Fundamental Issues in Computer Science*, pages 35–56. Springer Netherlands, 1993.

[48] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, pages 146–161. Springer Berlin Heidelberg, 2012.

[49] Diego Novillo. Memory SSA – a unified approach for sparsely representing memory operations.

[50] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[51] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 106–113. Springer International Publishing, 2014.

[52] Zvonimir Rakamarić and Alan J. Hu. A scalable memory model for low-level code. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 290–304. Springer Berlin Heidelberg, 2009.

[53] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27. ACM, 1988.

[54] Gyula Sallai, Ákos Hajdu, Tamás Tóth, and Zoltán Micskei. Towards evaluating size reduction techniques for software model checking. *Electronic Proceedings in Theoretical Computer Science*, 253:75–91, Aug 2017.

[55] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144. Springer Berlin Heidelberg, 2000.

[56] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[57] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017.

[58] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.

[59] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2009.

[60] Wei Wang, Clark Barrett, and Thomas Wies. Partitioned memory models for program analysis. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 539–558. Springer International Publishing, 2017.

[61] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.