Improving Precision of Loop Acceleration for C Programs

Charles Babu M

June 2017

Supervisor
Mandayam K. Srivas

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science

in

Chennai Mathematical Institute

Contents

Abstract

Veriabs is a software verifier for C programs that can verify and refute program assertions. While its general philosophy to over-approximate reachable states scales well for large programs, it becomes highly imprecise for loops with control-flow.

The tool abstracts loops using abstraction techniques based on widening and bounded model-checking. Due to the over-approximation, inexisting executions are considered that lead to false alarms.

In this thesis, we study the symbolic framework of acceleration and various acceleration techniques, both exact and abstract, for an efficient integration of them in the framework of Veriabs. We evaluate the efficiency of the acceleration techniques integrated into Veriabs on software-verification benchmarks and compare it with other state-of-the-art model checkers.

Chapter 1

Introduction

Several verification techniques have emerged and evolved to enable automatic software verification in practice. Different techniques have different strengths and performs better on those programs where technique can utilize its strength most. Modern day softwares are diverse in nature to address varied customer requirements or execution environment. Therefore, to maximize automatic software verification on wide variety of programs, diverse set of verification techniques are needed. VeriAbs emerged as a one such portfolio verifier since first participation in SV-COMP 2017. VeriAbs' primary aim is to verify safety properties in ANSI-C programs with loops, as loops form major bottleneck in effective formal verification. VeriAbs' journey in SV-COMP started with novel loop abstraction techniques for loops with numerical arithmetic operations and loops processing large sized arrays [11].

In contrast to traditional static analyzers, Veriabs does not calculate the abstract fix-point of a program by iterative application of an abstract transformer. Instead, it calculates symbolic abstract transformers for program fragments (e.g., loops) using a loop summarization presented in [11]. Veriabs computes abstract transformers starting from the inner-most loops, which results in linear run-time of the summarization procedure and which is often considerably smaller than the traditional fixpoint procedure of abstract interpetation.

Veriabs uses the following very coarse over-approximation: replace the loop by a piece of code that "havocs" the program state by setting all variables written by the loop to non-deterministic values and then strengthens summaries using loop invariants. While it does not aim to discover invariants by itself, it draws invariants from a library of abstract domains. Once

all loops have been summarized, the resulting program is a loop-free over-approximation of the input program. This program is then handed to a symbolic execution engine CBMC to check for violations of any assertions.

Veriabs' general philosophy to over-approximate reachable states, although it scales well for large programs, become highly imprecise for loops with control-flow. Control-flow in loops has always been a challenge to static analyzers due to the complex interleaving of paths in the concrete execution of the loop. The consequence of loop abstraction strategies based on summarization (a form of widening) in Veriabs is that, inexisting executions are considered, and often these lead to false alarms. One approach to handle control-flow in loops is to strengthen loop summaries using disjunctive invariants, but this is a challenging problem and static analyzers in general rely on partitioning heuristics.

One may ask the question whether there are certain cases where widening based summarization can be performed precisely. This leads us to so-called *acceleration* techniques, The idea of accelerations was first studied for communicating finite-state machines and counter automata [8, 12, 2], which then led to investigating the problem of finding subclasses of numerical programs for which the reachable sets can be computed exactly. These methods, roughly, can handle loops with a restricted form of linear arithmetic in programs, where reachable states are characterized in Presburger arithmetic.

Inspired by the earlier approaches on acceleration, this problem has been reformulated in the abstract interpretation framework by Gonnord and Halbwachs who proposed *abstract acceleration* [20] for Linear relation analysis as a complement to widening. These approaches aim at computing abstract summary of loops whenever possible, and otherwise use traditional widening. These methods discover complex invariants in the domain of convex polyhedra but restricted to simple loops.

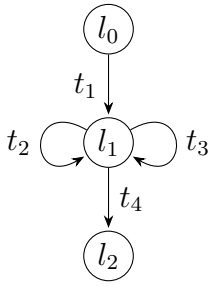
In this thesis we study the framework of acceleration, both exact and abstract, and identify practical acceleration techniques to be integrated into the framework of software model-checking by Veriabs. The two key contributions in Veriabs are: 1) an algorithmic solution to apply the specialized analyses for precise summarization for a class of loops called *flattable* loops with restricted finite monoid affine transformations [1, 28]; 2) an algorithmic solution for abstract loop summarization of general linear loops where summaries are strengthened using invariants on the domain of convex polyhedra [21].

1.1 Motivation: Common Veriabs' failures

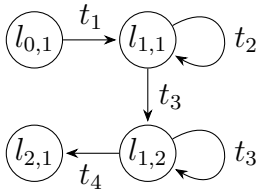
Common Veriabs failures are mainly due to the control flow in the programs which are encountered in practice.

```
1  unsigned int x = 0;
2  unsigned int y=5000000;
3  while(x<10000000){
4    if(x<5000000){
5      x=x+1;
6    }else{
7      x=x+1;
8      y=y+1
9    }
10 }
```

(a) 1a



(b) 1b



(c) 1c

Figure 1.1: 1a) Program P_1; 1b) CFG of P_1, where we omit intermediate (non-cutpoint) locations. The transition t_1 initializes the variables entering the loop. The transitions t_2 and t_3 traverse the loop by taking the `if` and `else` branch, respectively. Transition t_4 exit the loop; 1c) Flattening of the CFG of P_1 in 1b.

Consider the control-flow graph (CFG) of program P_1 shown in Fig.1.1b. Here, the transition formulas corresponding to t_2 and t_3 are defined as follows:

$$t_1 \stackrel{\text{def}}{=} (x = 0 \wedge y = 5000000)$$

$$t_2 \stackrel{\text{def}}{=} (x < 10000000 \wedge x < 5000000 \wedge x' = x + 1 \wedge y' = y)$$

$$t_3 \stackrel{\text{def}}{=} (x < 10000000 \wedge x \geq 5000000 \wedge x' = x + 1 \wedge y' = y + 1)$$

$$t_4 \stackrel{\text{def}}{=} (x \geq 10000000 \wedge x' = x \wedge y' = y)$$

Veriabs applies various abstraction techniques on this program. One such method is to over-approximate loops is by introducing non-deterministic variables $\mathtt{k}, \mathtt{k}_1, \mathtt{k}_2$ that corresponds to the total loop iterations of the loop, the `if`-block, and of the `else`-block respectively. Although it strengthens abstract summary using constraints on the variables like $\mathtt{k} \geq 0 \wedge \mathtt{k}_1 \geq 0 \wedge \mathtt{k}_2 \geq 0$ and $\mathtt{k} = \mathtt{k}_1 + \mathtt{k}_2$, these are too coarse of an over-approximation and fail to prove the property that `x` == `y`. Another one such method in Veriabs is bounded

model-checking, where Veriabs unwinds the program to a certain depth. The analysis fails on P_1 due to the large unwinding needed to verify the safety assertion.

We see how acceleration techniques can be applied to this program to prove the safety property of interest. Here, the natural representation of the loop in P_1 represented using control-flow graph in Fig.1.1b is not *flat*. A CFG G is *flat* if for each location l, there exists at most one elementary cycle (the cycle do not touch the same location twice) in G containing l. The CFG in Fig.1.1b is not flat as l_1 is part of two cycles corresponding to the paths t_2 and t_3. For CFGs known to be *flattable* [1] with accelerable relations, flat acceleration technqiues [28, 1] can be used to compute precise disjunctive loop summaries. However, to check if a loop is flattable or not is undecidable. As a consequence, VeriAbs implements incomplete procedures that can flatten when a flattening of a loop exists [28, 1] and computes precise disjunctive loop summaries.

We now see that the example CFG shown in Fig.1.1b is indeed flattable. The analysis starts with $x = 0$ and $y = 50$. Initially, the execution of the loop stays in path t_2 and increments x until it violates the guard $x < 50$. After violating the guard in t_2, the execution continues to stay in path t_3 and increments both x and y until the loop condition ($x < 100$) in t_3 violates. The flattening of the given CFG is shown in Fig.1.1c. VeriAbs implements the technique in [28] for constructing flattenings of loops, that use *path-dependency automaton* (PDA) and Z3 SMT solver. Since P_1 is flattable and precisely accelerable, the precise conditional summary of its flattening in Fig.1.1c is $(x < 50 \wedge x < 100 \wedge k_{23} = 50 - x \wedge k_{34} = 100 - x \wedge x' = 100 \wedge y' = 100)$, where k_{ij} encode the number of iterations of t_i before t_j is taken, in Fig.1.1c. This precise summary proves the safety of the assertion $(x == y)$.

1.2 Contributions and outline

The goal of this thesis is to study and implement acceleration technqiues for C programs. We cover the following aspects:

- Origins: We recall the principles of the symbolic framework for model-checking and motivate the results of Presburger-based acceleration
- Theoretical Background: We revisit the symbolic framework for acceleration for various classes of systems and revisit results known from Presburger-based acceleration

– The methods: We then discuss developing practical algorithms for implementing acceleration techniques, both exact and abstract.

– Tool: We briefly describe the tool Veriabs that implement these methods and give some experimental results

Chapter 2

Premilinaries

2.1 Sets and Relations

Given two sets A and B, we denote $A \cup B$, $A \cap B$, $A \setminus B$, and $A \times B$, the union, intersection, difference, and cartesian product of A and B. The empty set is denoted by $\emptyset$. The cardinality of the finite set is denoted by $|A|$. A relation R between A and B is a subset $R \subseteq A \times B$. Given two relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, the composition of R_1 and R_2, written as $R_1 \bullet R_2 \subseteq A \times C$ is defined by: $(x, x'') \in R_1 \bullet R_2$ if $(x, x') \in R_1$ and $(x', x'') \in R_2$ for some $x' \in B$.

Let A be a set. A *binary relation* R on A is a relation between A and itself. The *identity relation* Id_A on A is a set $\{(x, x) | x \in A\}$. For R on A, we denote R^i the relation defined inductively by: R^0 is the identity relation on A and $R^{n+1} = R \bullet R^n$. We denote the *reflexive transitive closure* of R by R^*, and it is defined as $R^* = \bigcup_{i \geq 0} R^i$.

The set of integers is denoted by $\mathbb{Z}$. The complete linearly ordered set $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, \infty\}$. $\mathbb{B} = \{0, 1\}$ denotes the set of Boolean values. Similarly, the set of rationals and reals are denoted by $\mathbb{Q}$ and $\mathbb{R}$ respectively.

A partially ordered set $\mathbb{D}$ is called a *lattice* if and only if any two elements $x, y \in \mathbb{D}$ have a greatest lower bound and a least upper bound, denoted respectively by $x \wedge y$ and $x \vee y$. A lattice is a *complete lattice* if and only if any subset $Y \subseteq \mathbb{D}$ has a greatest lower bound and a least upper bound, denoted by $\bigwedge Y$ and $\bigvee Y$. The least element $\bigvee \emptyset$ of a complete lattice is denoted by $\bot$. The greatest element $\bigwedge \emptyset$ is denoted by $\top$.

Consider two partially ordered sets $\langle \mathbb{D}_1, \leq_1 \rangle$ and $\langle \mathbb{D}_2, \leq_2 \rangle$. A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ if and only if $f(x) \leq_2 f(y)$ for all $x, y \in \mathbb{D}_1$ with $x \leq_1 y$

Theorem 1 (Knaster/Tarski[27]). *Let $\mathbb{D}$ be a complete lattice and $f : \mathbb{D} \to \mathbb{D}$ a monotone function. The operator f has a least fixpoint and a greatest fixpoint, respectively denoted by μf and νf, where $\mu f = \bigwedge\{x \in \mathbb{D} | f(x) \leq x\}$ and $\nu f = \bigvee\{x \in \mathbb{D} | x \leq f(x)\}$*

Let $f : X \to Y^k$, i.e., for all $i \in \{1, ..., k\}$, the mapping $f_i : X \to Y$ is given by $f_i(x) = (f(x))_i$ for all $x \in X$. The set $\overline{\mathbb{R}}^m$ is partially ordered by the component-wise extension of $\leq$, which we again denote it as $\leq$.

Definition 2 (**Uninterpreted System**). *An uninterpreted system S is a tuple $S = (L, T, \Sigma)$, where*

- *L is a finite set of locations*
- *Σ is a (possibly infinite) set of formulae called actions*
- *$T \subseteq L \times \Sigma \times L$ define a finite set of control-flow edges*

Definition 3 (**Interpreted System**). *An interpreted system G is a pair (S, I) of an uninterpreted system $S = (L, T, \Sigma)$ and an interpretation $I = (\Sigma, C, [\![\cdot]\!])$ of Σ, shortly written as $S = (L, \Sigma, T, C, [\![\cdot]\!])$*

The *set of configurations* $\mathcal{C}_S$ is $L \times C$. Here, each transition $t \in T$ is interpreted as a relation $\subseteq \mathcal{C}_S \times \mathcal{C}_S$, and this extends to a sequences of transitions $\pi \in T^*$. Here, $\xrightarrow{\epsilon}= Id_{C_S}$ and $\xrightarrow{t \cdot \pi}= \xrightarrow{t} \bullet \xrightarrow{\pi}$. Similarly, for any language $\mathcal{L} \subseteq T^*$, we define $\xrightarrow{\mathcal{L}}= \bigcup_{\pi \in \mathcal{L}} (\xrightarrow{\pi})$

2.2 Reachability Problem

We are interested in the reachability problems as model-checking *safety properties* can be reduced to reachability. For any given configurations $X \subseteq \mathcal{C}_S$ and any $\mathcal{L} \subseteq T^*$, we define $\texttt{post}_S(\mathcal{L}, X) = \{x' \in \mathcal{C}_S | \exists x \in X; (x, x') \in \xrightarrow{\mathcal{L}}\}$. We define $\texttt{post}(T, X)$ as the set of configurations reachable in one step from X and $\texttt{post}(T^*, X)$ as the set of all configurations reachable from X. For simplicity, we denote this reachability set of X by $\texttt{post}^*(X)$.

Model-checking a safety property P amounts to asking whether $\texttt{post}^*(X_0) \subseteq P$ for a set of initial configurations X_0. Since reachability sets are not in general recursive, we rely on partial correctness procedures with no guarantees of termination or over-approximating reachability sets.

2.3 Symbolic Framework

In practice model checking procedures use symbolic representations (called here regions) to manipulate sets of configurations.

Definition 4 (Symbolic framework)**.** *A symbolic framework is a tuple* $SF = (\Sigma, C, [\![\cdot]\!], D^\sharp, \gamma)$*, where* $I = (\Sigma, C, [\![\cdot]\!])$ *is an interpretation,* $D^\sharp$ *is a set of formulae called regions,* $\gamma : D^\sharp \to 2^C$ *is a region concretization, and such that there exists a decidable relation* $\sqsubseteq$ *and recursive functions* $\sqcup$*, POST satisfying:*

1. $\bot \in D^\sharp$ *such that* $\gamma(\bot) = \emptyset$
2. $\sqsubseteq \subseteq D^\sharp \times D^\sharp$ *is such that for all* $d_1, d_2 \in D^\sharp$*,* $d_1 \sqsubseteq d_2$ *iff* $\gamma(d_1) \subseteq \gamma(d_2)$
3. $\sqcup : D^\sharp \times D^\sharp \to D^\sharp$ *is such that for all* $d_1, d_2 \in D^\sharp$*,* $\gamma(d_1 \sqcup d_2) = \gamma(d_1) \cup \gamma(d_2)$
4. $POST : \Sigma \times D^\sharp \to D^\sharp$ *is such that* $\forall a \in \Sigma$*,* $\forall d \in D^\sharp$*,* $\gamma(POST(a, d)) = [\![a]\!](\gamma(d))$

Here, we also consider weaker versions of symbolic reprensentations. A *weak inclusion* ensures only that $d_1 \sqsubseteq d_2$ implies $\gamma(d_1) \subseteq \gamma(d_2)$ and a *weak union* satisfies $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$. The operations like join and widening in abstract interpretation typically have these weaker notions.

2.3.1 Limits of the symbolic approach

We say that a set of configurations $X \subseteq \mathcal{C}_S$ is $D^\sharp - definable$ if there exists $d \in D^{\sharp^{|L|}}$ such that $\gamma(d) = X$. Using the symbolic framework $D^\sharp$, computing $\texttt{post}^*(X)$ is feasible only if $\texttt{post}^*(X)$ is $D^\sharp - definable$. In general it is undecidable to check if $\texttt{post}^*(X)$ is $D^\sharp - definable$. $D^\sharp - definability$ does not give sufficient condition for computing the $\texttt{post}^*(X)$. We say that $\texttt{post}^*(X)$ is *effectively* $D^\sharp - definable$ if there exists a recursive function $g : D^{\sharp^{|L|}} \to D^{\sharp^{|L|}}$ such that $\forall d \in D^{\sharp^{|L|}}$, $\texttt{post}^*(X) = \gamma(g(d))$.

2.3.2 Standard symbolic procedure

We now give standard procedure for computing reachability sets, and this is shown in Fig. 1. Here, termination is not guaranteed in general.

For weaker symbolic representations as typically seen in abstract interpretation, widening operators are used to enforce termination

```
1: function REACH(d_0, G)
2:     input: d_0 ∈ D^{♯|L|}
3:     d ← d_0;
4:     while POST(d) ⋢ d do
5:         d ← POST(d) ⊔ d
6:     end while
7:     return d
8: end function
```

Figure 2.1: Procedure 1: Standard symbolic procedure

2.3.3 Widening operator

An operator $\triangledown : D^\sharp \times D^\sharp \to D^\sharp$ is said to be a widening operator if it satisfies the following properties:

- $d_1, d_2 \in D^\sharp,\ d_1 \sqsubseteq^\sharp d_1 \triangledown d_2$
- $d_1, d_2 \in D^\sharp,\ d_2 \sqsubseteq^\sharp d_1 \triangledown d_2$
- For every chain $(X_i^\sharp)_{i\in\mathbb{N}}$, the increasing chain $(Y_i^\sharp)_{i\in\mathbb{N}}$ defined by:

$$Y_0^\sharp = X_0^\sharp$$

$$Y_i^\sharp = Y_{i-1}^\sharp \triangledown X_i^\sharp,\ \forall i > 0$$

If the abstract domain does not satisfy the ascending chain condition, we thus can compute fixpoint of $F^\sharp$ in finite time using a widening operator. Widening operators are not necessarily neither commutative neither monotone, nor associative, and these are crucial for chaotic iteration fixpoint algorithms

2.4 Families of systems

Definition 5. *Given an interpretation $I = (\Sigma, C, [\![\cdot]\!])$, the familiy of systems built on I denoted by $\mathcal{F}(I)$ is the class of all systems $S = (Q, \Sigma, T, C, [\![\cdot]\!])$ where actions are interpreted using I*

Many well-known systems can be instantiated in the above definition. Minsky machines have $C = \mathbb{N}^{Var}$ where actions have increments and guarded decrements. For Counter systems $C = \mathbb{Z}^{Var}$, and all actions are defined in Presburger arithmetic. For Pushdown systems $C = \Gamma^*$ where Γ^* is the set of all words on the stack alphabet and actions add or remove letters from the top of the stack. Channel systems has the domain $C = (\Gamma^*)^C$ where C is a set of fifo channels, and Γ is some alphabet of messages. Actions add messages at one end of the channels and remove messages at the other end. Some other systems include Timed systems, Hybrid systems, etc.
Various symbolic representations for the above family of systems are:

Regular Languages: Used for representing sets of configurations of push-down systems and channel systems.

Number Decision Diagrams are automata recognizing subsets of $\mathbb{Z}^{Var}$ and are used for counter systems.

Convex polyhedra for hybrid systems, etc.

Presburger formulas for counter systems

2.5 Presburger-based acceleration

In practice, an iterative symbolic reachability set computation similar to the one of procedure 1 will surely fail. Acceleration methods aim to compute the exact set of reachable states in numerical transition systems. Acceleration identifies classes of systems for which computing reachability set $\texttt{post}^*(X)$ is decidable.

2.5.1 Presburger Arithmetic

Presburger Arithmetic [26] is the additive first order theory over the integers $\langle \mathbb{Z}, \leq, + \rangle$. It is well known that satisfiability and validity of Presburger formulas are both decidable. We denote a Presburger formula as $\phi(\mathbf{x})$ where $\mathbf{x}$ is a n-dimensional vector of free variables. The set of vectors satisfying ϕ is denoted by $[\![\phi]\!] \subseteq \mathbb{Z}^n$, and a set $X \subseteq \mathbb{Z}^n$ is said to be *Presburger-definable* if the there exists a Presburger formula $\phi(\mathbf{x})$ such that $X = [\![\phi(\mathbf{x})]\!]$. The efficiency of the algorithms based on Presburger definable sets depends strongly on the symbolic representation used for manipulating these sets. *Number decision diagrams* (NDD) are one such symbolic representation. Automata-based representations have a crucial advantage over Presburger formulas where they

have canonicity. A minimization procedure provides a canonical representation for NDD-definable sets. Presburger formulas on the other hand lack canonicity, where a simple representation could be represented in a complicated way.

2.5.2 Counter Systems

Counter systems are a subclass of program model defined in with $C = \mathbb{Z}^n$, where transition relations $\mathcal{P}(C^2)$ are defined by Presburger formulas. Counter systems generalize Minsky machines. From a practical point of view, these systems allow the modelling of,for example, communication protocols, multi-thread programs or programs with pointers. From a theoretical view, many well-known classes appear to be encom-passed by counter systems, like Minsky machines, Petri nets extended with reset/ inhibitor/ transfer arcs, reversal-bounded counter machines and broad-cast protocols. The counterpart of the expressiveness of counter systems is that only two counters with increment, decrement, and test-to-zero can simulate a Turing machine. Then checking even basic safety properties of counter systems is undecidable. As the reachability sets are not always Presburger definable, the idea is to identify the class of accelerable relations for which the transitive closure $\texttt{post}^*(X)$ is Presburger definable.

2.5.3 Presburger Linear relations with finite monoid

Definition 6 (**Monoid $\mathcal{M}$**). *The multiplicative monoid generated by the matrix M is denoted by $\mathcal{M}$, where the monoid $\mathcal{M}$ is written as $\langle M \rangle = \{I_n, M, M^1, M^2, ..., M^m, ...\}$*

Definition 7 (Presburger-linear relations with finite monoid). *The transition relation $R(\mathbf{x}, \mathbf{x}') = (\phi(\mathbf{x}) \wedge \mathbf{x}' = M\mathbf{x} + \mathbf{d})$ is Presburger with finite monoid iff ϕ is a Presburger formula and $\langle M \rangle$ is a finite multiplicative monoid.*

Theorem 8 (Presburger-definable transitive closure [2]). *If R is a Presburger-linear relation with finite monoid, then R^* is Presburger-definable.*

The finiteness of the monoid is polynomially decidable [7].

2.5.4 Convex translation

A convex translation of a Presburger linear relation $R(\mathbf{x}, \mathbf{x}') = (\phi(\mathbf{x}) \wedge \mathbf{x}' = M\mathbf{x} + \mathbf{d})$ where M is an identity matrix and $\mathbf{d}$ is a constant vector.

The geometrical properties of convex sets help alleviate the transitive closure construction. Here, the Presburger formula encoding the binary relation R^* can be replaced by:

$$\exists k \geq 0, \mathbf{x}' = \mathbf{x} + k\mathbf{d} \wedge \phi(\mathbf{x}) \wedge (k = 0 \vee \phi(\mathbf{x}' - \mathbf{d})) \tag{2.1}$$

It is shown that R^* is proved in [2] to be computable in quadratic in the size of the NDD and exponential in the number of counters. This improvement became possible because of the less number of quantifiers in the Presburger formula encoding R^*.

2.5.5 Flat systems

A system is called flat if it has no nested loops, or more precisely if any location of the system is contained in at most one elementary cycle of the system [1]. This notion allows us to identify a class of systems for which the set of reachable states can be computed exactly

Theorem 9 (Presburger-definable flat systems [1])**.** *The reachability set* $\texttt{post}^*(X)$ *of a counter system is Presburger-definable if the system is flat and all its transitions are Presburger-linear relations with finite monoid.*

Chapter 3

Acceleration in Symbolic Model Checking

We now dicuss a theoretical framework for symbolic model checking with acceleration. We identify three natural levels of accelerations: 1) loop; 2) flat; 3) global.

3.1 Acceleration

Definition 10 (**Acceleration).** *Let Act be set of actions and $D^\sharp$ be the abstract domain. The symbolic representation $D^\sharp$ supports*

1. loop acceleration *if there exists a recursive function $POST\_STAR$: $Act \times D^\sharp \to D^\sharp$ s.t. $\forall a \in Act, \forall x \in D^\sharp$, $[\![POST\_STAR(a,x)]\!] = [\![a^*]\!]([\![x]\!])$*

2. flat acceleration *if there exists a recursive function $POST\_STAR$: $Act^* \times D^\sharp \to D^\sharp$ s.t. $\forall \pi \in Act^*, \forall x \in D^\sharp$, $[\![POST\_STAR(\pi,x)]\!] = [\![\pi^*]\!]([\![x]\!])$*

3. global acceleration *if there exists a recursive function $POST\_STAR$: $RegExp(Act) \times D^\sharp \to D^\sharp$ s.t. for any regular expression e over Act, for any $x \in D^\sharp$, $[\![POST\_STAR(e,x)]\!] = [\![e]\!]([\![x]\!])$*

Counter systems equipped with finite monoid supports both loop and flat acceleration. Global acceleration is a very strong property, and global $\Longrightarrow$ flat $\Longrightarrow$ loop. It is obvious that most interesting families of systems do

not support global acceleration as they are Turing powerful in general. Flat acceleration is a good compromise which requires methods for computing $POST\_STAR$.

For counter systems which are not flat, Finkel [2] proposes to partially unfold the outerloops (circuits) of nested loops in order to obtain a flat system that is simulated by the original system. Such a system is called a *flattening*. The algorithm is based on heuristically selecting circuits of increasing length and enumerating flattenings of these circuits. Hence, the algorithm terminates in case the system is flattable,i.e., at least one of its (finite) flattenings has the same reachabil-ity set as the original system. This method is complete for *flattable* systems. However, flattability is undecidable

Symbolic representation such as abstract domain of template polyhedra over integers, reals, or rationals, which has weak union i.e. $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$, supports global acceleration using strategy iteration methods [15, 16, 14, 19]. We discuss these methods in section 5.1

3.2 Acceleration for Flattable Counter Systems

We have seen earlier in Theorem 9 that reachability set $\texttt{post}^*(X)$ of a flat counter system with finite monoid is Presburger-definable. But most of the systems we encounter are not flat. Here, we address such systems. A way to approach these systems is to consider flattenings [1]. A flattening S' of a system S is a flat system simulated by S. Flatening generalizes unfolding, where every location participates in at most one elementary cycle. A system S is flattable when there exists a flattening S' such that $\texttt{post}^*_{S'}(X') = \texttt{post}^*_S(X)$ for some X' derived from X, where S' is equivalent to S w.r.t reachability. Then with circuit selection and enumeration of flattenings, the set $\texttt{post}^*(X)$ can be computed if the system is flattable. Flattable systems appear to be a maximal class for reachability set computation by circuit accelerations.

We can enumerate all possible flattenings for a system S and check for every flattening S' of if $\texttt{post}^*_{S'}(X'_0)$ is a fixpoint of S and iterate. However, such a procedure would consume too many resources in practice. A systematic complete procedure for flattable systems is proposed in [1] based on restricted linear regular expressions, and this is explained below.

3.2.1 Complete Procedure for Flattable Systems

We introduce a notion called restricted linear regular expressions which will be used below for a systematic way of constructing reachability sets.

Definition 11. *Given a finite alphabet A, a restricted linear regular expression (rlre) over A is a regular expression ρ of the form $u_1^*...u_m^*$, where $u_i \in A^*$.*

Fixpoint computation for flattable systems is shown to be efficiently compuable using restricted linear regular expressions, where the problem reduces to exploring the set of rlre over a finite alphabet of transitions T. This can be computed by building iteratively an increasing sequence of rlre such that each $w \in T^*$ is present infinitely often in the sequence. The key issue in the procedure is to select $w \in T^*$ to be added at each step in the sequence such that the fixpoint computation is reached quickly. Instead of considering all possible sequences from T^*, we consider all sequences in T^* up to some length k. We denote these set of sequences by $T^{\leq k}$. A system S is k-flattable if length of circuits is limited to k. If the search fails, it is eventually stopped and we increment k.

Here, we use two procedures called `Watchdog` and `Choose`, where the former decides when the procedure should be aborted and the latter chooses a sequence $w \in T^{\leq k}$ at each step. For *fairness* we require that `Watchdog` fires infinitely often, but only after `Choose` picked each $w \in T^{\leq k}$ at least once. Assuming that the methods `Choose` and `Watchdog` respects fairness condition, the procedure obtained is correct and complete for flattable linear counter systems with finite monoid [1].

Theorem 12 ([1])**.** *For a finite counter system with finite monoid S,*

1. *when Procedure 2 terminates, $[\![REACH2(x_0, S)]\!] = \texttt{post}^*([\![x_0]\!])$ (partial correctness).*
2. *$REACH2$ terminates for any input if and only if S is $D^\sharp$-flattable (termination)*

3.3 Acceleration in Data-Flow Analysis

While acceleration techniques for concrete reachability set computations may be equivalently formulated in terms of control-flow path languages or control-flow graph unfoldings, this equivalence does not hold anymore for weaker

```
1: function REACH2(x_0, S)
2:     x ← x_0; k ← 0
3:     k ← k + 1
4:     Start
5:     while POST(x) ⋢ x do
6:         Choose fairly w ∈ T^{≤k}
7:         x ← POST_STAR(w, x)
8:     end while
9:     with
10:        when Watchdog stops goto Step 2
11:    return x
12: end function
```

Figure 3.1: Procedure 2: Flat acceleration

versions of symbolic representations as typically seen in abstract interpretation.

A data-flow analysis of a program basically consists in the choice of abstract domain which is a complete lattice of data properties over program variables and transfer functions for program instructions. The merge-over-all-path (MOP) solution provides the most precise abstract invariant. In data-flow analysis, usually MOP solution is approximated by mimimum fixpoint (MFP) solution, which can be computed using Kleene iteration. Using widening/narrowing operators to enforce termination, may not compute MFP solution. In §5.1, we discuss two acceleration techniques to data-flow analysis: 1). to compute MFP solution in the domain of template polyhedra using *Strategy iteration* methods and; 2) to compute a tight over-approximation of MOP solution for a restricted class of counter systems in the domain of convex polyhedra. Although the former support global acceleration, the latter approach supports only loop acceleration but computes a tight over-approximation of MOP solution.

Chapter 4

Disjunctive Loop Summarization

4.1 Program Model

To improve the readability, in this section we consider the interpreted system program graphs over state space $C = \mathbb{Z}^n$ as the program model.

A CFG G is a directed graph $(L, \mathcal{R}, l_0)$, where

- L is a finite set of locations, $l_0 \in L$ is the initial location, and
- $(l, R, l') \in \mathcal{R}$ is a finite set of control-flow edges from $l \in L$ to $l' \in L$ with transition relations $\mathcal{R} \subseteq C^2$.

An *execution* of a CFG G is a sequence of location-state pairs $(l_0, s_0) \rightarrow^{R_0} (l_1, s_1) \rightarrow ... \rightarrow^{R_{k-1}} (l_k, s_k)$, $\forall k \geq 0$,

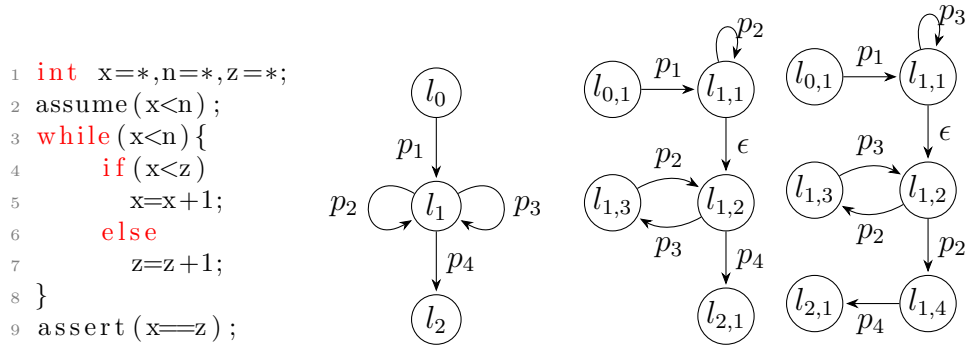
- $l_k \in L, s_k \in \mathbb{Z}^n$,
- $(l_k, s_k) \rightarrow^{R_k} (l_{k+1}, s_{k+1})$ if $R_k(s_k, s_{k+1})$.

In this section we look at acceleration that targets affine transition relations of the form $R : \mathbf{Ax} \leq \mathbf{b} \wedge \mathbf{x}' = \mathbf{Mx} + \mathbf{d}$, where $\mathbf{M}$ is a $n \times n$ diagonal matrix with elements from $\mathbb{Z}_{\geq 0}$.

4.2 Example

Consider the program P_2 shown in Fig. 4.1a and its corresponding CFG in Fig.4.1b with two paths in the loop. Here, we can see that the CFG is not

flat. Interestingly, we will see that this loop is flattable due to the path-interleaving it exhibits for various inputs. While we could use Presburger acceleration discussed in the previous section, we borrow similar ideas for synthesizing exact loop disjunctive summaries using SAT/SMT solvers. For this, we define a *path-dependency automaton* (PDA) that capture feasibility relation among all paths in the loop. Veriabs implements this technique in [28] for constructing flattenings of a loop that use PDA and Z3 SMT solver. This approach is in contrast with the previous approach of constructing flattenings, where it avoids enumerating all possible flattenings with cycles of length $\leq k$ [1].



(a) 1a (b) 1b (c) 1c (d) 1d

Figure 4.1: 1a) Program P_2; 1b) CFG of P_2, where we also omit intermediate (non-cutpoint) locations. The transition p_1 initializes the variables entering the loop. The transitions p_2 and p_3 traverse the loop by taking the `if` and `else` branches, respectively. Transition p_4 exit the loop; 1c) Flattening of the CFG of P_2 corresponding to the initial states $x < z$ of 1b; 1d) Flattening of the CFG of P_2 corresponding to the initial states $\neg(x < z)$ of 1b.

Let p_2, p_3 be the paths of the loop corresponding to `if`, `else` branches of the loop body, and p_4 be the exit path of the loop as shown in Fig.4.1. The transition formulas corresponding to p_2, p_3, and p_4 are defined as follows:

$$p_2 \stackrel{\text{def}}{=} (x < n \wedge x < z \wedge x' = x + 1 \wedge z' = z)$$

$$p_3 \stackrel{\text{def}}{=} (x < n \wedge x \geq z \wedge x' = x \wedge z' = z + 1)$$

$$p_4 \stackrel{\text{def}}{=} (x \geq n \wedge x' = x \wedge z' = z)$$

Interestingly, the path-interleaving executions of the loop can be represented using restricted regular expressions $r_1 : p_1p_2^*(p_3p_2)^*p_4$ and $r_2 : p_1p_3^*(p_2p_3)^*p_2p_4$ corresponding to inputs $x < z$ and $\neg(x < z)$ respectively. An explicit enumeration of all flattenings for this example using the procedure 2 in Fig.1 terminates with a fixpoint at $k = 2$. The CFG flattenings corresponding to these two different cases are shown in Fig.4.1c and Fig.4.1d. Although, procedure 2 in Fig.1 succeeds on this example, it has worst-case exponential search as the number of words to consider increase exponentially with increasing k.

In this section, we aim to solve this problem for loops with restricted transition relations with finite monoid, and then devise static analysis techniques to identify "good accelerable" words. This in turn results in the faster convergence of the fixpoint computation of procedure 2 at the expense of its termination for flattable systems. The resulting new procedure (procedure 3) that uses static analysis techniques to identify the next word in each iteration may not be complete for flattable systems as all words are not explored. If procedure 3 terminates, we compute disjunctive loop summary for the flattening using SMT solvers. The summaries are exact when the loop is flattable with accelerable relations.

4.3 Accelerated Disjunctive Loop Summaries for Flattable loops

Using SMT solvers, we discuss static analysis techniques for synthesizing good words such that fixpoint computation can be accelerated. Without loss of generality, we consider multi-path loops which are not nested loops. Without loss of generality, we also assume that each path has the transition relation of the form $R : \mathbf{Ax} \leq \mathbf{b} \wedge \mathbf{x'} = \mathbf{Mx} + \mathbf{d}$, where $\mathbf{M}$ is a $n \times n$ diagonal matrix with elements from $\mathbb{Z}_{\geq 0}$.

Definition 13 (Path Dependency Automaton (PDA)[28]). *Given a loop in a program P and a set of integer variables $\mathcal{X}$ with $C = \mathbb{Z}^n$. The path dependency automaton (PDA) over a finite alphabet is a 4-tuple $M = (Q, \delta, Q_0, F)$, where*

- *Q is the set of states, where each state q_i is associated with a 2-tuple $\langle i, R_i \rangle$ corresponding to the transition relation along the path i.*
- *$Q_0 \subseteq Q$, a set of initial states corresponding to the paths where execution may begin*

– *$F \subseteq Q$, a set of states corresponding to exit paths of the loop.*

– *For every transition $(q_i, \mathtt{a}, q_j) \in \delta$, where $\mathtt{a}$ denotes a 2-tuple $\langle k_{ij}, \psi_{ij} \rangle$.*

  – *k_{ij} is a constraint about no. of iterations of path p_i before p_j is taken in any execution of the loop.*

  – *ψ_{ij} is a guard constraint (a first-order linear arithmetic formula) on the concrete states C, obtained in any execution after path p_i, and from which path p_j is taken.*

We use SAT/SMT solvers to obtain transitions for the PDA of a given loop by checking the feasibility using path transition relations. For two states $q_i, q_j \in Q$ in the automaton, we construct a formula $\tau : R_i[x/\mathbf{x}, x'/\mathbf{x}'] \wedge R_j[x'/\mathbf{x}, x''/\mathbf{x}']$ and check its satisfiability. If the formula τ is SAT, we add a transition between q_i and q_j in the automaton and also annotate the transition with constraints $\langle k_{ij}, \psi_{ij} \rangle$ after synthesizing them as discussed in [28]. Note that existence of the transition in the automaton does not imply the existence of an execution with $(l_0, s_0) \to \ldots (l, s) \to^{p_i} (l', s') \to^{p_j} (l'', s'') \ldots$

4.4 Summarization

Definition 14 (Flat Fragment Program (FFP))**.** *Consider program graph $G = (L, \mathcal{R}, l_0)$. A program graph $G_F = (L^f, \mathcal{R}^f, l_0^f)$ is a flat fragment program of G if (1) G_F is flat and (2) there exists a map $\tau : L^f \mapsto L$ such that*

1. *$\tau(l_0^f) = l_0$*
2. *Let $l \in L$ and $l_e \in L^f$ such that $\tau(l_e) = l$. If $(l_e, R, l'_e) \in \mathcal{R}^f$, then $(\tau(l_e), R, \tau(l'_e)) \in \mathcal{R}$.*

Here, τ can be extended to subsets of configurations $\mathcal{C}_{G_F}$. Given $X' \subseteq \mathcal{C}_{G_F}$, the definition of FFP ensures that $\tau(\mathtt{post}^*_{G_F}(X')) \subseteq \mathtt{post}^*_G(\tau(X'))$. We say a program G is *flattable* iff there exists an FFP G_F such that for any $X \subset \mathcal{C}_{G_F}$ $\tau(\mathtt{post}^*_{G_F}(X)) = \mathtt{post}^*_G(\tau(X))$.

The key idea is to synthesize flat fragment programs $G_{F_1}, G_{F_2}, \ldots, G_{F_m}$ such that for any $X \subset \mathcal{C}_{G_F}$ we have $\tau_1(\mathtt{post}^*_{G_{F_1}}(X)) \cup \ldots \cup \tau_m(\mathtt{post}^*_{G_{F_m}}(X)) = \mathtt{post}^*_G(\tau(X))$. The flattening of the program G_F can be obtained from G_{F_1}, G_{F_2}, …, G_{F_m} as follows: the locations are the union of all locations of G_{F_i}, transitions are the union of transitions of $\mathcal{R}_{F_i}$, and the function τ is extended

as $\tau(l) = \tau_i(l)$ for $l \in L_{F_i}$. If such flattening exists with accelerable relations, the loop is flattable and is summarizable. Provided that the loop is summarizable, the summary of the loop is a finite disjunction of the summaries of each FFP G_{F_i}. For loops which are not flattable or contain transition relations which are not accelerable, we use over-approximated summaries to capture the effect of the loop.

Here, we perform a Depth-First Search (DFS) algorithm on PDA $\mathbf{M}$ to construct FFPs such that these FFPs cover all executions of the loop. We assume a procedure $\texttt{SUMMARY}(G_{F_i})$ that computes summary of any FFP which is flat with accelerable relations using the techniques discussed in [28, 2]. The summary of a FFP G_{F_i} of $\mathbf{M}$ is denoted by tuple $\langle c_{G_{F_i}}, \texttt{Summ}_{G_{F_i}} = \texttt{SUMMARY}(G_{F_i})\rangle$, where $c_{G_{F_i}}$ is constraint on the initial states and $\texttt{Summ}_{G_{F_i}}$ encodes the relation between those initial states $c_{G_{F_i}}$ and the output states that exit the loop head.

We denote the disjunctive loop summary $S_{\mathbf{M}}$ of $\mathbf{M}$ as:

$$\bigcup_i \{\langle c_{G_{F_i}}, \texttt{Summ}_{G_{F_i}}\rangle\} \tag{4.1}$$

Identifying "good accelerable words" using SMT solving

4.5 Example P_2.

Consider the example CFG G shown in Fig.4.1b which is not flat. We earlier saw that G is indeed for flattable for all inputs as shown in Fig.4.1c,d. How do we synthesize the flattening? The trivial approach is to try all possible flattenings. This approach although terminates for length $k = 2$ for this program, we aim for a method that avoids enumerating all possible flattenings.

Consider the PDA automaton $\mathbf{M}$ for the graph shown in in Fig.4.1b. Let q_1, q_2, q_3 be the three states in the PDA of the loop corresponding to the paths p_2, p_3, and the exit path p_4 respectively. For any transition in the PDA, we synthesize transition constraints $\langle k_{ij}, \psi_{ij}\rangle$ in the following way. Let k_{12} denote the constraint on the variables $\mathcal{X}$ that symbolically represents no. of iterations of path p_1 before p_2 is taken in any execution of the loop, and similarly k_{21} and k_{13}. Using SMT solving and extended Z3 tactics, we synthesize k_{12} using Z3 SMT solver. Precisely, this can be viewed as a quantifier elimination for the equation 2.1. Once we synthesize k_{ij}, we synthesize the constraint ψ_{ij}, which encodes for any execution of the program G with $(l_0, s_0) \rightarrow ... \rightarrow^{R_i} (l_k, s_k)$, if s_k satisfies ψ_{ij} then $(l_k, s_k) \rightarrow^{R_j} (l_{k+1}, s_{k+1})$. We

then synthesize ψ_{12} using k_{12} with Z3 solving and by simplification of linear arithmetic formulas. We then annotate a_{12} with $\langle k_{12}, \psi_{12}\rangle$ and add transition (q_1, a_{12}, q_2) to the automaton. Similarly, we add transitions (q_2, a_{21}, q_1) and (q_1, a_{13}, q_3) to complete the automaton

Note that, we avoid self-loop transitions in the PDA automaton, as we implicitly take care of iterations or multiple executions of a path using constraints like k_{ij}. We then complete the automaton by obtaining all other transitions using the aforementioned approach. Once, we complete the PDA $\mathbf{M}$ automaton for the given program graph, we summarize all *traces* of the PDA using the DFS traversal on PDA $\mathbf{M}$.

A trace of a PDA (also the run of the automaton) is a sequence of transitions $(q_0, a_{01}, q_1), (q_1, a_{12}, q_2), \ldots, (q_m, a_{mm+1}, q_{m+1})$ such that $q_0 \in Q_0$ and $q_{m+1} \in F$. A trace is *simple* if for any $j \in \{0, \ldots, m+1\}$, we have $q_j \neq q_i$, $\forall 0 \leq i < j$. A trace is *feasible* if there is an execution of the loop such that $(l_0, s_0) \rightarrow^{R_0} \ldots \rightarrow^{R_0} (l_h, s_i) \rightarrow^{R_1} \ldots\ldots \rightarrow^{R_1} (l_h, s_i') \rightarrow^{R_2} \ldots \rightarrow^{R_m} (l_h, s_i''') \rightarrow^{R_{m+1}} \ldots \rightarrow^{R_{m+1}} (l_e, i'''')$, where l_h and l_e are loop-head and loop-exit locations respectively. Note that for any simple trace, we can trivially construct a corresponding FFP of the CFG G with all loops being self-loops at every location.

Summarizing all feasible traces of the PDA is trivial if the PDA $\mathbf{M}$ has no cycles. If $\mathbf{M}$ has no cycles, then the FFP associated with each trace is trivially flattable and precisely accelerable. A trace $(q_0, a_{01}, q_1), (q_1, a_{12}, q_2), \ldots, (q_m, a_{mf}, q_f)$ is *simple-and-cyclic* if for any $i \leq u < v < j$ such that $q_i = q_j$ we have $q_u \neq q_v$. If $\mathbf{M}$ has cycles, then we check if each all its feasible traces are simple-and-cycle during the DFS traversal. If we encounter a trace which is not simple-and-cyclic, then program may not summarizable and checking if there exists a flattening is hard, and we fall back to explicitly enumerating all words as shown in the procedure 2. If a trace is simple-and-cyclic, then by trivial construction there exists a corresponding FFP of the trace.

Consider the DFS traversal of the PDA $\mathbf{M}$. Here, the trace $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_1 \rightarrow q_f$ is simple-and-cyclic where we encounter a cycle with a period $\langle q_2, q_1\rangle$. Here, using SMT solving we see that the constraints k_{12} and k_{21} get simplified to constants, i.e., $k_{12} = k_{21} = 1$. So, the cycle gives a periodic behaviour to the execution of the loop, and we obtained a good cycle that accelerates the fixpoint computation process. Once we know that a cycle is periodic, we add a new state to the automaton corresponding to the new composed path $p_2 \circ p_1$ and obtain transition constraints. We earlier restricted the transition relations in the program model such that accelerability is not

affected by the composition of transition relations. We then accelerate this composed path which is precisely accelerable and continue summarizing the trace using DFS traversal from this newly added state in the automaton. If the cycles are not periodic, we fall back to the original method of checking all possible cycles up to certain length $\leq k$.

We implement the above mentioned loop summarization techniques using the PRISM program analysis framework. We use Z3 SMT solver for constructing PDA and also use extended Z3 tactics to identify accelerable cycles [28] for constructing flattenings of loops.

4.6 Completeness for Flattable Systems?

This method is not complete for flattable systems as we do not explore all possible cycles. The original solution by Finkel et al [2] provides a semi-algorithm that is complete for flattable systems, where the fixpoint computation reduces to exploring the set of $\mathtt{r}$lre over a finite set of transitions. The key issue for both approaaches is to select the $w \in A_G^*$ to be added to the sequence at each step such that the fixpoint is reached quickly. The blottleneck issue with the two approaches is the cardinal of $T^{\leq k}$ being exponential in k. [12] proposes *reduction techniques* to decrease dramatically the number of useful words, thus avoiding enumeration of all sequences in $T^{\leq k}$. The key idea is that not all words are needed to compute reachability sets.

4.7 Optimization using Reduction Techniques

In the future, we plan to investigate the performance using these two reduction techniques: reduction by union [12] and reduction by commutation [2]

- *Reduction by union* consists in merging two transitions with different guards but same matrix M: $R_1 := \phi(\mathbf{x}) \wedge M\mathbf{x} + \mathbf{d}$ and $R_2 := \phi'(\mathbf{x}) \wedge M\mathbf{x} + \mathbf{d}$ are merged into a unique $R := (\phi(\mathbf{x}) \vee \phi(\mathbf{x'})) \wedge M\mathbf{x} + \mathbf{d}$
- *Reduction by commutation* consists in removing transitions $g \cdot f$ and $f \cdot g$ where $f, g \in T^{\leq k}$ for some k, whenever f and g satisfy $(\xrightarrow{f \cdot g})^*$ and $(\xrightarrow{g \cdot f})^*$ are then equal to $(\xrightarrow{f})^* \bullet (\xrightarrow{g})^*$

Chapter 5

Abstract Acceleration

5.1 Abstract Acceleration for Linear Counter Systems

We now study acceleration techniques in the context of data-flow analysis. Abstract interpretation of a program basically consists in the choice of abstract domain which is a complete lattice of data properties over program variables and transfer functions for program instructions. As discussed earlier, the-merge-over-all-path (MOP) solution provides the most precise abstract invariant. In data-flow analysis, usually MOP solution is approximated by mimimum-fixpoint solution (MFP), which can be computed using Kleene iteration. Using widening/narrowing operators to enforce termination, may not compute MFP solution. In this section, we discuss two acceleration techniques to data-flow analysis:

1. to compute MFP solution in the domain of template polyhedra using Strategy iteration methods

2. to compute a tight over-approximation of MOP solution for a restricted class of counter systems in the domain of convex polyhedra.

Overview: Strategy iteration methods are used for computing best inductive invariants in the abstract domain with the help of mathematical programming. They are not restricted to simple loops and they are able to compute the invariant of a whole program at once thus supporting global acceleration. However, these methods are restricted to template domains, e.g. template polyhedra and quadratic templates. Similarly to Kleene iterations, the max-strategy improvement algorithm produces an ascending sequence of

pre-fixpoints that are less than or equal to the least inductive invariant we are aiming for. The pre-fixpoints are obtained through convex optimization techniques,e.g., linear programming. The algorithm converges to the least inductive invariant after at most exponentially many steps unlike the kleene iteration.

We present basic notations of abstract interpretation, numerical domains, and max-strategy iteration.

5.2 Preliminaries

A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ is called *affine* if and only if there exist $A \in \mathbb{R}^{m\times n}$ and $d \in \overline{\mathbb{R}}^m$ such that $f(x) = Ax + d$ for all $x \in \overline{\mathbb{R}}^n$. Here, we use the convention $-\infty + \infty = -\infty$. Here, we can observe that f is monotone if all the entries of A are non-negative. A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}$ is called *weak-affine* if and only if there exist $a \in \mathbb{R}^n$ and $b \in \overline{\mathbb{R}}$ such that $f(x) = a^\top x + b$ for all $x \in \overline{\mathbb{R}}^n$ with $f(x) \neq -\infty$. A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ is called if *weak-affine* if and only if there exist weak-affine mappings $f_1, ..., f_m : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}$ such that $f = (f_1, ..., f_m)^\top$. Every affine mapping is weak-affine, but not vice-versa. In this section, we are concerned with mappings that are point-wise minimums of finitely many monotone and weak-affine mappings. These mappings are in particular concave, i.e., the set of points below the graph of the function is convex.

5.3 SAT modulo linear arithmetic

The set of SAT modulo linear real arithmetic formulas Φ is defined through the following grammar:

$$e ::= \ c \mid x \mid e_1 + e_2 \mid c.e' \qquad \Phi ::= \ e_1 \leq e_2 \mid e_1 < e_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi'$$

Here, $c \in \mathbb{Z}$ is a constant, x is a real valued variable, e, e', e_1, e_2 are real-valued linear expressions, and $\Phi, \Phi_1, \Phi_2, \Phi'$ are formulas. An *interpretation* $\mathcal{I}$ for a formula Φ is a mapping that assigns a real value to every real-valued variable. We write $\mathbb{M} \models \Phi$ for "$\mathbb{M}$ is a *model* of Φ". We now inductively define a function $[\![e]\!]$ that evalues a linear expression e as follows:

$$[\![c]\!]\mathbb{M} = c$$

$$[\![x]\!]\mathbb{M} = \mathbb{M}(x)$$

$$[\![e_1 + e_2]\!]\mathbb{M} = [\![e_1]\!]\mathbb{M} + [\![e_2]\!]\mathbb{M}$$
$$[\![c.e']\!]\mathbb{M} = c.[\![e']\!]\mathbb{M}$$

5.4 Statements

Let the set **Stmt** of all statements is the set of all SAT modulo linear real arithmetic formulas without negation. Note that non-strict and strict inequality constraints are permitted. The formula $e_1 \neq e_2$ is also permitted since it is written as $e_1 < e_2 \vee e_1 > e_2$. We can transform any SAT modulo real arithmetic formula into this form in linear time by pushing negations to the leaves.

The $\mathbb{Z}$-valued variables $x_1, ..., x_n$ and $x'_1, ..., x'_n$, that may occur in the formula, play a particular role. The values of the variables $x_1, ..., x_n$ represent the values of the program variables before executing the statement, and the values of the variables $x'_1, ..., x'_n$ represent the values of the program variables after executing the statement. For convenience, we denote the vectors $(x_1, ..., x_n)^\intercal$ and $(x'_1, ..., x'_n)^\intercal$ also by $\mathbf{x}$ and $\mathbf{x}'$, respectively.

A statement is called *merge-simple* if and only if it is in disjunctive normal form, i.e., s is of the forms $s_1 \vee ... \vee s_k$, where the statements $s_1, ..., s_k$ do not use the Boolean connector $\vee$. Any statement can be rewritten into an equivalent merge-simple statement in exponential time and space using distributivity. A merge-simple statement s that does not use the Boolean connector $\vee$ at all is called *sequential*.

5.5 Program Model

We consider programs modeled as symbolic control flow graphs over a memory state space $\mathbb{R}^n$. A program CFG G is a $(L, \mathcal{R}, l^{\mathrm{i}})$, where

- L is a finite set of locations, $l^{\mathrm{i}} \in L$ is the initial location, and
- $\mathcal{R} \subseteq L \times \mathbf{Stmt} \times L$ is a finite set of control-flow edges

A state is described by a vector $\in \mathbb{R}^n$. For every statement $s \in \mathbf{stmt}$, we assign a *concrete semantics* $[\![s]\!] : 2^{\mathbb{R}^n} \to 2^{\mathbb{R}^n}$, where for any $Y \subseteq \mathbb{R}^n$, $[\![s]\!](Y)$ executing the statement s on Y.

Statements: The set of statements **stmt** is the set of all SAT modulo linear arithmetic formulas without Boolean variables and without negation as

discussed in §5.4. Here, non-strict inequalities and strict inequalities are permitted. We can (in linear time) transform any SAT modulo linear real arithmetic formula without Boolean variables into this form by pushing negations to the leaves.

The *collecting semantics* $V : L \to 2^{\mathbb{R}^n}$ of a program $G = (L, \mathcal{R}, l_0)$ is defined as the least solution of the following constraint system:

$$V[l_0] \supseteq \mathbb{R}^n \qquad V[l'] \supseteq [\![s]\!](V[l]) \text{ for all } (l, s, l') \in \mathcal{R} \tag{5.1}$$

Abstract Semantics

Let $(2^{\mathbb{R}^n}, \alpha, D^\sharp, \gamma)$ be a Galois connection with $\alpha : 2^{\mathbb{R}^n} \to D^\sharp$ and $\gamma : D^\sharp \to 2^{\mathbb{R}^n}$. The *abstract semantics* of a statement s is defined by

$$[\![s]\!]^\sharp := \alpha \circ [\![s]\!] \circ \gamma$$

Here, we have chosen to use the best abstract transformer. The *abstract semantics* $V^\sharp$ of the program G is the least solution of the following constraint system

$$V^\sharp[l_0] \sqsupseteq^\sharp \alpha(\mathbb{R}^n) \qquad V^\sharp[l'] \sqsupseteq^\sharp [\![s]\!]^\sharp(V^\sharp[l]) \text{ for all } (l, s, l') \in \mathcal{R} \tag{5.2}$$

5.6 Template Shaped Conjunctive Abstract Domain

In the considered approach, we assume that the abstract base domain is presented as a fixed, parametrized *template* $\mathbf{T}$ which is a conjunction of formulas (over the state variables and abstract parameters) from a decidable fragment of quantifier-free first-order logic describing the desired property of a program. Now we present some instances of template-shaped abstract domains.

Template Polyhedra Template polyhedra are polyhedra the shape of which is fixed by a so-called template. The domain operations can be performed efficiently with the help of linear programming (LP) solvers.

Template polyhedra over reals One can define template polyhedra over reals. We will use the following notations: Let $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$, the operators sup, $\leq$, etc are point-wisely lifted to vectors.

A template polyhedron is generated by a template constraint matrix, or short template, $\mathbf{T} \in \mathbb{R}^{m \times n}$ of which each row contains at least one non-zero

entry. The set of template polyhedra $\mathcal{P}^T$ generated by $\mathbf{T}$ is $\{X_{\mathbf{d}} | \mathbf{d} \in \overline{\mathbb{R}}^m\}$ with $X_{\mathbf{d}} = \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^n, \mathbf{T}\mathbf{x} \leq \mathbf{d}\}$ and $X_\perp = \emptyset$.An abstract value is represented by the vector of bounds $\mathbf{d}$. The analysis tries to find the smallest values of $\mathbf{d}$ representing a fixed point of the semantic equations. Here,$\top$ and $\perp$ are naturally represented by the bound vectors $\boldsymbol{\infty}$ and $-\boldsymbol{\infty}$ respectively.

The abstraction $\alpha : \wp(\mathbb{R}^n) \to \overline{\mathbb{R}}^m$ and concretization $\gamma : \overline{\mathbb{R}}^m \to \wp(\mathbb{R}^n)$ are defined below:

- Concretization: $\gamma_{\mathbf{T}}(\mathbf{d}) = \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^n, \mathbf{T}\mathbf{x} \leq \mathbf{d}\}, \mathbf{d} \in \overline{\mathbb{R}}^m$
- Abstraction: $\alpha_{\mathbf{T}}(Y) = \min\{\mathbf{d} \in \overline{\mathbb{R}}^m | \gamma_{\mathbf{T}}(\mathbf{d}) \supseteq Y\}, Y \subseteq \mathbb{R}^n$
- Join: $\mathbf{d} \sqcup_T \mathbf{d}' = (max(d_1, d'_1), ..., max(d_m, d'_m))$
- Image: The templates may vary from location to location. Let $\mathbf{T}_l$ be the template in location $l \in L$ and $\mathbf{d}_l$ be the corresponding vector of bounds. For a transition $(l, s, l') \in \mathcal{R}$, the *abstract semantics* of s, $[\![s]\!]^\sharp(\mathbf{d}_l)$ is defined by

$$[\![s]\!]^\sharp(\mathbf{d}_l) = \max\{\mathbf{T}_{l'}\mathbf{x}' | \mathbf{T}_l\mathbf{x} \leq \mathbf{d}_l \wedge s(\mathbf{x}, \mathbf{x}')\} \tag{5.3}$$

For template polyhedral domain, the constraint system (5.2) has exactly one $\overline{\mathbb{R}}^m$-valued variable $V[l]^\sharp$ for each location $l \in L$. For solving the constraint system (5.2), we can use methods corresponding to performing a classical Kleene iteration. But convergence is not always guaranteed as the abstract domain does not satisfy the ascending chain condition.

We use a technique inspired by an encoding used by *max-strategy iteration* methods [15, 17, 25]. These methods state the invariant inference problem over template polyhedra as a disjunctive linear optimisation problem. This is solved iteratively by an upward iteration in the lattice of template polyhedra:using SMT solving, a conjunctive subsystem ("strategy") whose solution extends the current invariant candidate is selected. This subsystem is then solved by an LP solver; the procedure terminates as soon as an inductive invariant is found.

We now prove a crucial result about abstract semantics of statements 5.4

Lemma 15. *Let s be a sequential statement and $\mathbf{d} \in \mathbb{R}^m$. The operator $[\![s]]^\sharp$ is a point-wise minimum of finitely many monotone and weak-affine operators. For all $\mathbf{d} \in \mathbb{R}^m$, $[\![s]\!]^\sharp(\mathbf{d})$ can be computed in polynomial time through linear programming.*

Proof. Let $i \in \{1, \ldots, m\}$.

$[\![s]\!]_i^\sharp(\mathbf{d}) = sup\{\mathbf{T}_i x' | x' \in [\![s]\!](\gamma(\mathbf{d}))\}$

$= sup\{\mathbf{T}_i x' | x' \in \mathbb{R}^n \text{ and } \exists x, \mathbf{T}x \leq \mathbf{d} \text{ and } s[x/\mathbf{x}, x'/\mathbf{x}'] \text{ is satisfiable}\}$

Since, s is a sequential statement and does not contain disjunctions, the optimization problem aims at optimizing a linear objective function w.r.t linear constraints. The optimal value of this problem can be computed in polynomial time through linear programming techniques. To check feasibility by standard techniques, we need to convert the strict inequality $e_1 < e_2$ by a non-strict inequality ($e_1 \leq e_2 - \epsilon$), where ϵ is appropriately small and can be computed in polynomial time. Provided that the optimization problem is feasible, we can replace strict equalities with non-strict equalities. Let $s[< / \leq]$ be the statement obtained from s by replacing every strict inequality relation by a non-strict inequality relation. The optimal value can now be computed using linear programming and is equal to the optimization problem above.

we will now show that $[\![s]\!]_i^\sharp$ is a point-wise minimum of finitely many monotone and weak-affine operators. Since $s[x/\mathbf{x}, x'/\mathbf{x}'][< / \leq]$ is a sequential statement and a conjunction of non-strict linear inequalities, there exist matrices A, A' and A'' and a vector b such that, for all x and x', $s[x/\mathbf{x}, x'/\mathbf{x}'][< / \leq]$ is satisfiable iff there exists a x'' such that $Ax + A'x' + A''x'' \leq b$. The vector x'' are the variables in s which are existentially quantified. The optimization problem can be rewritten as

$$[\![s]\!]_i^\sharp(\mathbf{d}) = sup\{\mathbf{T}_i x' | x' \in \mathbb{R}^n, \exists x \in \mathbb{R}^n . \exists x'' \in \mathbb{R}^q . \mathbf{T}x \leq \mathbf{d} \text{ and } Ax + A'x' + A''x'' \leq b\}$$

From Farkas' lemma, provided that the optimization problem is feasible, we have

$$[\![s]\!]_i^\sharp(\mathbf{d}) = inf\{\mathbf{d}^\top y_1 + b^\top y_2 | y_1, y_2 \geq 0, \mathbf{T}^\top y_1 + A^\top y_2 = 0, A''^\top y_2 = 0, A'^\top y_2 = \mathbf{T}_i^\top\}$$

Since $y_1 \geq 0$ for all feasible solutions of the linear programming problem, $[\![s]\!]_i^\sharp$ coincides with a point-wise infimum of monotone and affine operators on the set $\{\mathbf{d} \in \mathbb{R}^m | [\![s]\!]_i^\sharp(\mathbf{d}) > -\boldsymbol{\infty}\}$. So, for a sequential statement s, we have that $[\![s]\!]_i^\sharp$ is a point-wise infimum of monotone and weak-affine operators. This implies that $[\![s]\!]_i^\sharp$ is concave. □

Lemma 16 (Merge-Simple Statement)**.** *Let s be a merge-simple statement. The operator $[\![s]\!]^\sharp$ is a point-wise maximum of finitely many point-wise minima of finitely many monotone and weak-affine mappings. For all $\mathbf{d} \in \mathbb{R}^m$, $[\![s]\!]^\sharp(\mathbf{d})$ can be computed in polynomial time through linear programming*

Proof. Let $s \equiv s_1 \vee \ldots \vee s_k$, where $s_1, \ldots, s_k$ are sequential statements. So, we have that $[\![s]\!]^\sharp(\mathbf{d}) = [\![s_1]\!]^\sharp(\mathbf{d}) \vee \ldots \vee [\![s_k]\!]^\sharp(\mathbf{d})$, where Lemma 15 can be applied to compute $[\![s]\!]_i^\sharp(\mathbf{d})$. □

Lemma 17. *Given a template constraint matrix* $\mathbf{T}$ *and an arbitrary statement* s*, deciding* $[\![s]\!]^\sharp(\boldsymbol{\infty}) > -\boldsymbol{\infty}$ *holds, is* $\mathbf{NP} - complete$*.*

5.7 Max-Strategy Iteration

Max-strategy iteration is used for computing the least solution of a system of equations $\mathcal{E}$ of the form $\delta = \mathbf{F}(\delta)$, where δ are the template bounds. Also, $F_i, 0 \leq i \leq n$ is a finite maximum of monotonic and concave operators $\mathbb{R}^n \to \mathbb{R}$. This max-strategy improvement algorithm is guaranteed to compute the least fixed point of $\mathbf{F}$.

We now write our static analysis problem into a system of monotone fixpoint equations over $\overline{\mathbb{R}}$. The constraint system 5.2

$$V^\sharp[l_0] \sqsupseteq^\sharp \alpha(\mathbb{R}^n) \qquad V^\sharp[l'] \sqsupseteq^\sharp [\![s]\!]^\sharp(V^\sharp[l]) \text{ for all } (l, s, l') \in \mathcal{R}$$

has exacly one $\overline{\mathbb{R}}^m-$valued variable $V^\sharp[l]$ for each program point $l \in L$. For simplicity, we use δ for denoting the vector of bound variables appearing in syntactic expressions, and $\mathbf{d}$ for the vector carrying the actual bounds. $\delta_{l,i}$ is the bound variable corresponding to the i^{th} line of the template in location l. The equation system $\mathcal{E}$ is constructed from the abstract semantics of the program's transitions. We have:

$$\delta_{l^{\mathrm{i}}} = \boldsymbol{\infty}$$

$$\delta_{l'} = max(\{-\boldsymbol{\infty}\} \cup \{[\![s]\!]^\sharp(\delta_l) | (l, s, l') \in \mathcal{R}\}) \quad \text{for } l' \neq l^{\mathrm{i}}$$

Max-Strategies: Each max-strategy μ induces a "subsystem" $\delta = \widehat{\mathbf{F}}(\delta)$ of $\mathcal{E}$ in a sense that exactly one argument $\widehat{F}_i$ of the max operator on the right-hand side of each equation $\delta_i = max(..., \widehat{F}_i(\delta), ...)$ is chosen. Intuitively, a max-strategy picks for each maximum operator one of its operands. In our application the no. of max-strategies are exponential in the size of $\mathcal{E}$. To be precise, the cardinality is $\mathcal{O}(2^{n^2})$.

One has to compute least solution $lfp[\![\mathcal{E}]\!]$ of the system $\mathcal{E}$, where $[\![\mathcal{E}]\!]$ is defined as

$$[\![\mathcal{E}]\!](\mathbf{d}) = \max_{\mu \text{ in } \mathcal{E}} [\![\mu]\!](\mathbf{d})$$

and with $[\![\mu]\!](\mathbf{d}) = [\![\delta = \widehat{\mathbf{F}}(\delta)]\!](\mathbf{d}) = \widehat{\mathbf{F}}(\mathbf{d})$.
Max-strategy improvement $lfp[\![\mathcal{E}]\!]$ is computed by iteratively improving strategies μ until the fixed point $lfp[\![\mu]\!]$ of a strategy equals $lfp[\![\mathcal{E}]\!]$. The least fixed point $lfp[\![\mu]\!]$ of a strategy μ is computed by solving an LP problem.

$lfp[\![\mathcal{E}]\!]$ can be computed by solving the constraint system

$$\text{for each } (\delta_{l'} = [\![s]\!]^\sharp(\delta_l)) \text{ in } \mu : \mathbf{d}_{l'} \leq \mathbf{T}_{l'}\mathbf{x}' \wedge \mathbf{T}_l\mathbf{x} \leq \mathbf{d}_l \wedge s(\mathbf{x}, \mathbf{x}')$$

and the objective function $\mathsf{max} \sum_i \mathbf{d}_i$, i.e. the sum of all bounds $\mathbf{d}$.

μ' is called an *improvement* of μ w.r.t $\mathbf{d}$, i.e., $\mu' = \mathsf{max\_improve}_{\mathcal{E}}(\mu, \mathbf{d})$ iff

1. $[\![\mu']\!](\mathbf{d}) \geq [\![\mu]\!](\mathbf{d})$
2. If $(\delta_i = \widehat{F}_i(\delta))$ in μ and $(\delta_i \geq \widehat{F'_i}(\delta))$ in μ' and $\widehat{F}_i \neq \widehat{F'_i}$, then $\widehat{F'_i}(\mathbf{d}) > \widehat{F}_i(\mathbf{d})$

Note that, for a max-strategy μ and $\mathbf{d}$ such that $[\![\mu]\!](\mathbf{d}) = \mathbf{d}$, there might be several max-strategies μ' that are improvements of μ w.r.t. $\mathbf{d}$. However, different heuristics for selecting improving strategies may lead to different practical complexities.

The max-strategy improvement algorithm starts with the initial max-strategy μ_0 and successively performs the following two steps in the order until the least solution of $\mathcal{E}$ is found:

- Compute $\mu' = \mathsf{max\_improve}_{\mathcal{E}}(\mu, \mathbf{d})$
- Evaluate the max-strategy μ' w.r.t $\mathbf{d}$ to obtain a new value for $\mathbf{d}$.

For every improving strategy there is some equation where an argument of its max-operator leads to a greater bound. Since the arguments of the max-operator are required to be monotonic, the bounds are always monotonically increasing. Thus the arguments that have already been selected in previous strategies need not be considered again.

Succinct graphs

Consider the program P_3 shown below in Fig. 5.1. Here, program P3 is abstracted through the program $G_3 = (L_3, E_3, l_0)$ shown in Fig.5.2a. Most existing techniques consider a control-flow graph with transitions expressed as sequential statements only, by applying abstraction at every program point.

1: initial strategy $\mu := \{\delta_{l^{\mathrm{i}}} = \boldsymbol{\infty}, \delta_l = -\boldsymbol{\infty} \text{ for all } l \neq l^{\mathrm{i}}\}$
2: initial bounds $\mathbf{d} := \{\delta_{l^{\mathrm{i}}} = \boldsymbol{\infty}, \delta_l = -\boldsymbol{\infty} \text{ for all } l \neq l^{\mathrm{i}}\}$
3: **while d** is not a solution of $\mathcal{E}$ **do**
4: $\quad \mu' := \mathsf{max\_improve}_{\mathcal{E}}(\mu, \mathbf{d})$
5: $\quad \mu := \mu'$
6: $\quad \mathbf{d} := lfp[\![\mu]\!]$
7: **end while**
8: **return d**

Algorithm 1: Max-Strategy iteration algorithm

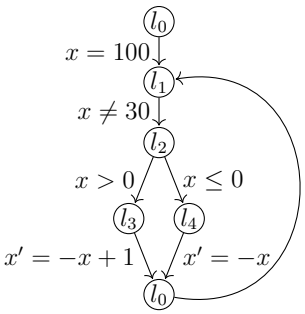
```
1 //Program P3
2 int x=100;
3 while(x!=30){
4   if(x>0)
5     x=-x+1;
6   else
7     x=-x;
8 }
```

Figure 5.1: A non-flat multi-path loop program P_3

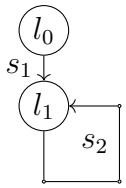
However, it is not necessary to apply abstraction at every program point, i.e., to assign an abstract value to each program point. The idea is very similar to path-focusing technique [24, 18]. Given a control flow graph with transitions having sequential statements, we compute a feedback vertex set such that removing those vertices cuts all cycles in the graph. We then construct a program where the only control nodes are those in the feedback vertex set, but the edges carry arbitrary statements instead of sequential statements.

Since all loops contain the program point l_1, $\{l_1\}$ is a feedback vertex set of G_3. Here, we rewrite the control-flow graph G_3 into a control-flow graph $G_4 = (L_4, E_4, l_0)$ shown in Fig. 5.2b. Both G_3 and G_4 are equivalent w.r.t. the collecting semantics. Let V_3 denote the collecting semantics of G_3 and V_4 denote the collecting semantics of G_4. Here, $V_3[l] = V_4[l]$ for all $l \in L_4$. W.r.t. abstract semantics, G_4 is usually more precise than G_3 because we reduced the number of merge points. Here, $V_4^\sharp[l] \sqsubseteq V_3^\sharp[l]$ for all $l \in L_4$, where $V_3^\sharp$ denotes the abstract semantics of G_3 and $V_4^\sharp$ denotes the abstract semantics

of G_4.



(a) 1a



$G_3 = (L, E, l_0)$
$E = \{(l_0, s_1, l_1), (l_1, s_2, l_1)\}$
$s_1 \equiv x = 100$
$s_2 \equiv \phi \wedge (\phi_1 \vee \phi_2)$
$\phi \equiv x > 30 \vee x < 30$
$\phi_1 \equiv x > 0 \wedge x' = -x + 1$
$\phi_2 \equiv x \leq 0 \wedge x' = -x$

(b) 1a

Figure 5.2: (a) Program graph G_3 of P_3; (b) Program G_4

This is independent of any abstract domain, and the results of program analyses on this new graph, at nodes from the feedback vertex sets, are sound invariants for the original program. Finding minimal feedback vertex set is linear when control-flow is reducible (when loops have a single entry point), and NP-complete when the graph is arbitrary. So, heuristics are used to find feedback vertex set [9]. However, our loop control-flow graphs are reducible.

Example P_3 (Max-strategy iteration) Given below is the system of equations $\mathcal{E}_{G_4}$ for G_4. Here, we use the template constraint matrix $\begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix}$

$$\begin{aligned}
\delta_{l_{0,1}} &= \infty \\
\delta_{l_{0,2}} &= \infty \\
\delta_{l_{1,1}} &= max\{[\![s_1]\!]^\sharp_1(\delta_{l_{0,1}}, \delta_{l_{0,2}}), [\![s_2]\!]^\sharp_1(\delta_{l_{1,1}}, \delta_{l_{1,2}})\} \\
\delta_{l_{1,2}} &= max\{[\![s_1]\!]^\sharp_2(\delta_{l_{0,1}}, \delta_{l_{0,2}}), [\![s_2]\!]^\sharp_2(\delta_{l_{1,1}}, \delta_{l_{1,2}})\}
\end{aligned}$$

Here, the initial strategy $\mu_0 := \{\delta_{l^i} = \boldsymbol{\infty}, \delta_l = -\boldsymbol{\infty} \text{ for all } l \neq l_0\}$
The initial bounds are $\mathbf{d^0} := \{\delta_{l^i} = \boldsymbol{\infty}, \delta_l = -\boldsymbol{\infty} \text{ for all } l \neq l_0\}$

The improvement strategy μ_1 is obained using μ_0 and $\mathbf{d}^0$:

$$\mu_1 = \begin{cases} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = sup\{x' | x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 100\} \\ \delta_{1,2} = sup\{-x' | x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 100\} \end{cases}$$

After computing least fixpoint for the strategy μ_1, that is $lfp[\![\mu_1]\!]$, the current bounds are:

$$\mathbf{d}_1 = \begin{cases} \delta_{0,1} \to \infty \\ \delta_{0,2} \to \infty \\ \delta_{1,1} \to 100 \\ \delta_{1,2} \to -100 \end{cases}$$

Here, s_2 is an arbitrary statement which is neither sequential nor merge-simple. The algorithm in [18] does not transform statement s_2 into an equivalent merge-simple statement by transforming s_2 into a disjuntive normal form as this makes the system exponential. Instead they keep exponential system implicit and obtain improvement strategies using SAT/SMT solvers. The improved strategy μ_2, which contains only one disjunct in $(x < 30 \vee x > 30)$, is obtained using μ_1 and $\mathbf{d}_1$, which is:

$$\mu_2 = \begin{cases} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = sup\{x' | x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x > 30 \wedge x > 0 \wedge x' = -x + 1\} \\ \delta_{1,2} = sup\{-x' | x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x > 30 \wedge x > 0 \wedge x' = -x + 1\} \end{cases}$$

After computing least fixpoint for the strategy μ_2, that is $lfp[\![\mu_2]\!]$, the current bounds are:

$$\mathbf{d}_2 = \begin{cases} \delta_{0,1} \to \infty \\ \delta_{0,2} \to \infty \\ \delta_{1,1} \to 100 \\ \delta_{1,2} \to 99 \end{cases}$$

With the current bounds, there is no improvement strategy for μ_2 and the algorithm terminates with the least fixpoint $\mathbf{d}_2$ for the system of equations $\mathcal{E}_{G_4}$.

Theorem 18 (Termination)**.** *Algorithm terminates after a finite number of iterations*

Sketch. Termination follows from these observations:

1. Max-strategy iteration terminates after a finite number of improvement steps, because there is a finite number of strategies and each strategy is visited at most once [14].

2. Max-strategy iteration returns the unique least fixed point w.r.t. the given system of equations [14].

□

5.7.1 Adaptation of Max-Strategy Iteration for C Programs

Template polyhedra over bitvectors One can define template polyhedra over bitvectors and use SAT/SMT solvers to perform the domain operations. We will use the following notations: Let $\overline{BV} = BV \cup \{\bot\}$ where $\bot$ is smaller than any element in BV; the operators min, $\leq$, etc are point-wisely lifted to vectors of $\overline{BV}$.

A bitvector template polyhedron is generated by a template constraint matrix, or short template, $\mathbf{T} \in BV^{m\times n}$ of which each row contains at least one non-zero entry. The set of template polyhedra $\wp^T$ generated by $\mathbf{T}$ is $\{X_{\mathbf{d}} | \mathbf{d} \in \overline{BV}^m\}$ with $X_{\mathbf{d}} = \{\mathbf{x} | \mathbf{x} \in BV^n, \mathbf{Tx} \leq \mathbf{d}\}$ and $X_\bot = \emptyset$. The width of the bitvectors needs to be extended appropriately in order to avoid overflows in the computation of $\mathbf{Tx}$

he abstraction $\alpha : 2^{BV^n} \to \overline{BV}^m$ and concretization $\gamma : \overline{BV}^m \to 2^{BV^n}$ are defined below:

- Concretization: $\gamma_{\mathbf{T}}(\mathbf{d}) = \{\mathbf{x} | \mathbf{x} \in BV^n, \mathbf{Tx} \leq \mathbf{d}\}, \mathbf{d} \in \overline{BV}^m$
- Abstraction: $\alpha_{\mathbf{T}}(Y) = \min\{\mathbf{d} \in \overline{BV}^m | \gamma_{\mathbf{T}}(\mathbf{d}) \supseteq Y\}, Y \subseteq BV^n$
- Join: $\mathbf{d} \sqcup_T \mathbf{d}' = (max(d_1, d'_1), ..., max(d_m, d'_m))$
- Image: The templates may vary from location to location. Let $\mathbf{T}_l$ be the template in location $l \in L$ and $\mathbf{d}_l$ be the corresponding vector of bounds. For a transition $(l, R, l') \in \mathcal{R}$, the *abstract semantics* of R, $[\![R]\!]^\sharp(\mathbf{d}_l)$ is defined by

$$[\![R]\!]^\sharp(\mathbf{d}_l) = \max\{\mathbf{T}_{l'}\mathbf{x}' | \mathbf{T}_l\mathbf{x} \leq \mathbf{d}_l \wedge R(\mathbf{x}, \mathbf{x}')\} \tag{5.4}$$

In the abstract domain $\langle D^\sharp, \sqsubseteq \rangle$, the abstract semantics $V^\sharp$ of the program $G = (L, E, l_0)$ is the least solution of the following constraint system:

$$V[l_0]^\sharp \sqsubseteq \alpha(BV^n) \qquad [\![s]\!]^\sharp(V[l]^\sharp) \sqsubseteq V[l']^\sharp \text{ for all } (l, s, l') \in E \tag{5.5}$$

For template polyhedral domain, the constraint system has exactly one $\overline{BV}^m$-valued variable $V[l]^\sharp$ for each location $l \in L$. For solving the constraint system (5.2), we can use methods corresponding to performing a classical Kleene iteration. Convergence is always guaranteed as the abstract domain is finite. But due to the enormous height of the lattice, we need a faster convergence acceleration that makes the computational effort independent from the number of states and loop iterations.

We use a technique inspired by an encoding used by *max-strategy iteration* methods [15, 17, 25]. These methods state the invariant inference problem over template polyhedra as a disjunctive linear optimisation problem. This is solved iteratively by an upward iteration in the lattice of template polyhedra:using SMT solving, a conjunctive subsystem ("strategy") whose solution extends the current invariant candidate is selected. This subsystem is then solved by an LP solver; the procedure terminates as soon as an inductive invariant is found. Since we deal with finite domains (bit-vectors) we can use *binary search* as optimisation method instead of an LP solver [10].

5.8 Acceleration for simple loops using convex polyhedra

In this section, we discuss abstract acceleration of simple loops that aims at computing the best-correct approximation of the effect of simple loops in the domain of convex polyhedra [21]. We now present basic notions of abstract domain of convex polyhedra and its domain operations. A convex polyhedron is respresented in two ways:

- Generator Representation: This representation encodes the polyhedron P as the convex hull of:
  - A finite set $\mathcal{V} \subset \mathbb{R}^n$ of vertices v_i
  - A finite set $\mathcal{R} \subset \mathbb{R}^n$ representing rays. r_i
- Constraint representation that encodes polyhedron as a conjunction of linear constraints $\mathbf{Ax} \leq \mathbf{b}$, i.e., intersection of halfspaces with $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, and $\mathbf{b} \in \mathbb{R}^m$.

Both representations can be converted into one another by using the double description method [13].

Domain operations. The set of convex polyheda ordered by inclusion forms a lattice with $\top$ and $\bot$ denoting the polyhedron $\mathbb{R}^n$ and $\emptyset$. Some domain operations can be performed more efficiently using the generator representation only, others based on the constraint representation, and some use both. A convex polyhedron p is *included* in convex polyhedron q iff all vertices of p are in q, and all rays of p are rays of q. A convex polyhedron p is *empty* iff p has no vertex. The *meet* between two polyhedra p and q is computed by the conjunction of their constraint systems. The union of two convex polyhedra is in general not convex and the convex hull ($\sqcup$) is computed by the union of their generators. For projection, the algorithm of *Fourier-Motzkin elimination* is used to eliminate variables from a system of linear equalities using the constraint representation. The *Minskowski sum* of two polyhedra p and q is defined as the set $\{x_1 + x_2 | x_1 \in p, x_2 \in q\}$. The *time elapse* operation is defined as $p \nearrow q = \{x_1 + t x_2 | x_1 \in p, x_2 \in q, t \in \mathbb{R}^{\geq 0}\}$

5.8.1 Linear Accelerability of Linear Transformations

Definition 19 (Linear accelerability)**.** *A transition $R : \mathbf{x}' = \mathbf{M}\mathbf{x} + \mathbf{d}$ is linearly accelerable iff its reflexive and transitive closure R^* can be written as a finite union of sets*

$$R^* = \lambda X. \bigcup_i \{\hat{\mathbf{M}}_i \mathbf{x} + k\hat{\mathbf{d}}_i | k \geq 0, \mathbf{x} \in X\}$$

The class of linearly accelerable linear transformations is larger than the Presburger-based linear relations with finite monoid. The accelerability based on finite monoid creterion is merely based on the matrix $\mathbf{M}$. We now give an alternative characterization based on the homogenous form of affine transformations. Any transformation of dimension n can be written as a linear transformation $\begin{pmatrix} \mathbf{x}' \\ x'_{n+1} \end{pmatrix} = \begin{pmatrix} \mathbf{M} & \mathbf{d} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ x_{n+1} \end{pmatrix}$ of dimension $n+1$ and with $x_{n+1} = 1$ in the initial set X.

We now compute the Jordan normal form $\overline{\mathbf{J}} \in \mathbb{C}^n$ of $\mathbf{M}' = \begin{pmatrix} \mathbf{M} & \mathbf{d} \\ \mathbf{0} & 1 \end{pmatrix} \in \mathbb{R}^n$, where $\overline{\mathbf{J}} = Q^{-1}\mathbf{M}'Q$ with a nonsingular matrix $Q \in \mathbb{C}^n$. $\overline{\mathbf{J}}$ consists of block diagonal matrix consisting of Jordan blocks $\mathbf{J}_i$ associated with the eigenvalues $\lambda_i \in \mathbb{C}$ of $\mathbf{C}'$.

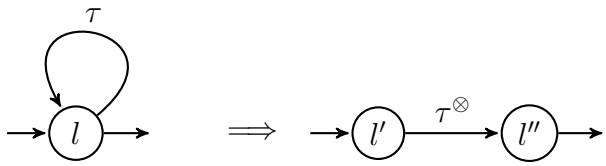


Figure 5.3: a) Self-loop transition b) Accelerated transition

The linear acclerability of a Jordan block w.r.t its size and its eigenvalue will be examined below

Theorem 20 (Accelerable linear transformations)**.** *A linear transformation* $R : \mathbf{x}' = \mathbf{M}'\mathbf{x}$ *is linearly accelerable iff each of the Jordan blocks* $\mathbf{J}$ *of its Jordan form* $\overline{\mathbf{J}}$ *safisfy the following criteria:*

- $\mathbf{J}$ *is of size 1 and its associated eigenvalue* $\lambda \in \{0\}$ *or* $\lambda \in \{e^{i2\pi\frac{p}{q}} | p, q \in \mathbb{N}\}$
- $\mathbf{J}$ *is of size 1 and its associated eigenvalue* $\lambda \in \{0\}$ *or* $\lambda \in \{e^{i2\pi\frac{p}{q}} | p, q \in \mathbb{N}\}$*, and in the latter case the variable associated with the second dimension of the block has only a finite number of values in* X *in the Jordan basis.*
- $\mathbf{J}$ *is of size greater than* 2 *and its associated eigenvalue* $\lambda = 0$*.*

Abstract Acceleration of Simple Loops

The basic idea of abstract acceleration is to replace a loop transition τ by a meta transition $\tau^{\otimes}$, where we compute a formula over-approximating the set $\tau^*(X) = \bigcup_{k\geq 0} \tau^k(X)$ using a single polyhedron as shown in Fig.5.3. Here, we want to compute a "close" approximation to the convex hull of $\tau^*(X)$ that is $\gamma(\tau^{\otimes}(\alpha(X))) \supseteq \tau^*(X)$.

Theorem 21 (Translations)**.** *Let* τ *be a translation* $G \to \mathbf{x}' = \mathbf{x} + \mathbf{d}$*, then for every convex polyhedron* X*,*

$$\tau^{\otimes}(X) = X \sqcup \tau((X \sqcap G) \nearrow \{\mathbf{d}\})$$

Proof. $\mathbf{x}' \in \bigcup_{k\geq 1} \tau^k(X)$

$$\begin{aligned}
&\iff \mathbf{x}' \in \tau(\bigcup_{k\geq 0} \tau^k(X)) \\
&\iff \exists k \geq 0, \exists \mathbf{x}_0 \in X, \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + k\mathbf{d} \wedge G(\mathbf{x}_0) \\
&\qquad \wedge \forall k' \in [1, k] : G(\mathbf{x}_0 + k'\mathbf{d}) \\
&\iff \exists k \geq 0, \exists \mathbf{x}_0 \in X, \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + k\mathbf{d} \wedge G(\mathbf{x}_0) \wedge G(\mathbf{x}_k) \\
&\iff \exists \alpha \geq 0, \exists \mathbf{x}_0 \in X, \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + \alpha\mathbf{d} \wedge G(\mathbf{x}_0) \\
&\iff \exists \mathbf{x}_0 \in X \sqcap G, \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k \in (\{\mathbf{x}_0\} \nearrow \{vd\}) \\
&\iff \mathbf{x}' \in \tau((X \sqcap G) \nearrow \{\mathbf{d}\})
\end{aligned}$$

□

Theorem 22 (Translations/resets)**.** *Let τ be a translation with resets $G \to \mathbf{x}' = \mathbf{C}\mathbf{x} + \mathbf{d}$, then for every convex polyhedron X,*

$$\tau^{\otimes}(X) = X \sqcup \tau(X) \sqcup \tau((\tau(X) \sqcap G) \nearrow \{\mathbf{C}\mathbf{d}\})$$

Proof. The formula trivially holds for $k = 0, 1$. So, for $k \geq 2$,
$\mathbf{x}' \in \bigcup_{k\geq 2} \tau^k(X)$

$$\begin{aligned}
&\iff \mathbf{x}' \in \tau(\bigcup_{k\geq 0} \tau^k(\tau(X))) \\
&\iff \exists k \geq 0, \exists \mathbf{x}_0 \in \tau(X), \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + k\mathbf{C}\mathbf{d} \wedge G(\mathbf{x}_0) \\
&\qquad \wedge \forall k' \in [1, k] : G(\mathbf{x}_0 + k'\mathbf{C}\mathbf{d}) \\
&\iff \exists k \geq 0, \exists \mathbf{x}_0 \in \tau(X), \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + k\mathbf{C}\mathbf{d} \wedge G(\mathbf{x}_0) \wedge G(\mathbf{x}_k) \\
&\iff \exists \alpha \geq 0, \exists \mathbf{x}_0 \in \tau(X), \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k = \mathbf{x}_0 + \alpha\mathbf{C}\mathbf{d} \wedge G(\mathbf{x}_0) \\
&\iff \exists \mathbf{x}_0 \in \tau(X) \sqcap G, \exists \mathbf{x}_k : \mathbf{x}' \in \tau(\mathbf{x}_k) \wedge \mathbf{x}_k \in (\{\mathbf{x}_0\} \nearrow \{\mathbf{C}\mathbf{d}\}) \\
&\iff \mathbf{x}' \in \tau((\tau(X) \sqcap G) \nearrow \{\mathbf{C}\mathbf{d}\})
\end{aligned}$$

□

The structure of the formula is due to the fact that a translation with resets to *constants* iterated N times is equivalent to the same translation with resets, followed by a pure translation iterated $N - 1$ times.

Finite Monoid Transitions

Let $\tau : \mathbf{Ax} \leq \mathbf{b} \rightarrow \mathbf{x}' = \mathbf{Cx} + \mathbf{d}$ such that $\langle \mathbf{C} \rangle$ form a finite monoid. Here, as a conseqequence of the definition §2.5.3, we have that $\exists q > 0 : \mathbf{C}^{2q} = \mathbf{C}^q$ where $\tau^*(X)$ can be rewritten as

$$= \bigcup_{0 \leq j \leq q-1} (\tau^q)^*(\tau^j(X))$$

From the above equation it is sufficient to accelerate τ^q, which equals

$$\tau^q = \bigwedge_{0 \leq i \leq q-1} \left(\mathbf{AC}^i\mathbf{x} + \Sigma_{0 \leq j \leq i-1}\mathbf{C}^j\mathbf{d} \leq \mathbf{d}\right) \rightarrow \mathbf{x}' = \mathbf{C}^q\mathbf{x} + \Sigma_{0 \leq j \leq q-1}\mathbf{C}^j\mathbf{d} \quad (5.6)$$

Here, we denote the left-hand side of the arrow as $\mathbf{A}'\mathbf{x} \leq \mathbf{b}'$ and the right-hand side as $\mathbf{x}' = \mathbf{C}'\mathbf{x} + \mathbf{d}'$. From the above finite monoid criterion and its periodicity, we can prove that $\mathbf{C}^q$ is diagonizable and all eigenvalues of $\mathbf{C}^q$ are in $\{0, 1\}$.

Let $\mathbf{C}' = \mathbf{C}^q$

Lemma 23 (Translation with resets in the eigenbasis)**.** *A translation* $\tau : \mathbf{Ax} \leq \mathbf{b} \rightarrow \mathbf{x}' = \mathbf{C}'\mathbf{x} + \mathbf{d}$ *where* $\mathbf{C}' = \mathbf{C}'^2$ *is a translation with resets* τ' *in the eigenbasis of* $\mathbf{C}'$

$$\tau' : \mathbf{AQx} \leq \mathbf{b} \rightarrow \mathbf{x}' = \mathbf{Q}^{-1}\mathbf{C}'\mathbf{Qx} + \mathbf{Q}^{-1}\mathbf{d}$$

where $\mathbf{Q}^{-1}\mathbf{C}'\mathbf{Q} = diag(\lambda_1, \ldots, \lambda_n)$ *and* λ_i *the eigenvalues of* $\mathbf{C}'$*.*

Now, we have abstract acceleration for finite monoid transitions using the following theorem.

Theorem 24 (Finite monoid transformation)**.** *Let* τ *be a transition* $G \wedge \mathbf{x}' = \mathbf{Cx} + \mathbf{d}$ *where* $\exists q > 0 : \mathbf{C}^{2q} = \mathbf{C}^q$*, then for every convex polyhedron* X

$$\tau^{\otimes}(X) = \bigsqcup_{0 \leq j \leq q-1} (\tau^q)^{\otimes}(\tau^j(X))$$

is an over-approximation of $\tau^*(X)$*, where* τ^q *is defined in eq.* (5.6) *and* $(\tau^q)^{\otimes}$ *is computed using Lemma 23.*

5.9 Abstract Acceleration of Multiple Loops

In §5.8.1 we discussed techniques for abstract acceleration of simple loops, which means that the loops are non-flat. For each such simple loops if they are accelerable, we replace the loop transition with an accelerated meta transition $\tau^{\otimes}$ making it loop-free. Widening is not required when these simple loops are accelerable. In the case of nested loops, which are non-flat systems, these abstract loop acceleration techniques can be applied to innermost loops. Here, innermost loops are replaced with accelerated transitions. Widening is necessary for outer loops

5.10 Widening and acceleration

Widening is still needed in the case of non-accelerable transitions, outer loops of nested loops and to guarantee convergence when there are multiple self-loops in the same control location. The crucial property that seperates acceleration from widening is the property of *monotonicity* that makes the analysis more robust and predictible. Widening operators in general are not monotonic.

Chapter 6

Discussion

6.1 Completeness in Abstract Interpretation

Completeness in abstract interpretation is a powerful notion and can be spotted everywhere. An abstract transfer function $f^\sharp$ is a complete complete abstract interpretation of f on the domain $D^\sharp$, when $\alpha \circ f = f^\sharp \alpha$ holds. Completeness intuitively encodes the greatest precision for $f^\sharp$. That is the abstract behaviour of $f^\sharp$ on $D^\sharp$ exactly matches the abstraction of the concrete behaviour of f. Interestingly, as a consequence, we have fixpoint completeness for f. That is, if $f^\sharp$ is complete for f, then from the fixpoint transfer theorem, we have that $\alpha \circ \texttt{lfp}(f) = \texttt{lfp}(f^\sharp) \circ \alpha$. If a program property q is exactly representable in the abstract domain $D^\sharp$ and if the abstract semantics on $D^\sharp$ for a program P is complete, then answering the query q in the abstract is the same as answering q in the concrete. This is to say that the abstraction $D^\sharp$ introduces no false positives in answering q in P. If a complete static analysis of P raises an alarm in answering the query q then this alarm is surely real.

6.2 MOP solution vs MFP solution

The best precise approximation of concrete reachable states in the domain $D^\sharp$ is $\alpha(\texttt{lfp}(\tau))$. Since convex polyhedra are closed under affine transformations, we have fixpoint completeness that is $\alpha(\bigcup_{k\geq 0} \tau^k) = \bigsqcup_{k\geq 0} \tau^k$. The latter formula is known as *Merge-Over-All-Paths (MOP)* solution, which is

$$X_0 \qquad X_1 = X_0 \sqcup \tau^\sharp(X_0) \qquad X_2 = X_0 \sqcup \tau^\sharp(X_0) \sqcup \tau^\sharp(\tau^\sharp(X_0)) \qquad \cdots$$

This is in contrast with the standard approach in abstract interpretation using Kleene iteration. The least fixed point computed using this approach is the *Minimum-Fixed point (MFP)* solution, which is computed as

$$X'_0 = X_0 \qquad X_1 = X'_0 \sqcup \tau^\sharp(X'_0) \qquad X_2 = X'_0 \sqcup \tau^\sharp(X'_0 \sqcup \tau^\sharp(X'_0)) \qquad \cdots$$

In the case of abstract acceleration, it should yield a tight over-approximation of the *MOP* solution. It can be seen that *MOP* solution is more precise than *MFP* solution, as in general $\tau^\sharp$ does not distribute over $\sqcup$. So, abstract acceleration is in general more precise than kleene iteration even when assuming convergence without widening.

6.3 Techniques studied in this thesis

We studied Presburger acceleration, where we identified accelerable relations for which transitive closure is effectively Presburger definable. For Presburger linear functions we have fixpoint completeness as the reachability sets are Presburger definable. Later we studied practical algorithms for synthesizing disjunctive loops summaries using SMT solvers. Acceleration can be viewed as a loop summarization method, in the sense that it aims at finding a relation between initial states and the future states that enter the loop head. While Presburger acceleration is complete for Flattable systems, the latter is not complete although it handles flattable loops. We also studied *Strategy iteration* methods for computing the *MFP* solution in the abstract domain of template polyhedra. Towards the end, we studied abstract loop acceleration for computing a tight over-approximation of *MOP* solution in the domain of convex polyhedra.

Chapter 7

Experiments

We have implemented our acceleration techniques in Veriabs as a feature extension. We particularly implemented two different acceleration technqiues, which are: 1) synthesis of precise disjunctive loop summaries using Z3 SMT solver as discussed in §4.1 and; 2) techniques for abstract acceleration of simple loops in the domain of convex polyhedra using the methods discussed in §5.8.1.

7.1 Veriabs Background

VeriAbs, is a reachability verifier for C programs that evolved as a portfolio verifier to verify wide variety of programs since its participation in International Competition on Software Verification (SV-COMP) 2017. It incorporates a portfolio of verification techniques with a focus of handling different types of programs and implements various strategies, where each strategy is a set of techniques applied in a specific sequence. It demonstrated its effectiveness in the competitions when compared with state-of-the-art model-checkers.

7.1.1 Tool Usage

VeriAbs is a command line tool that works on the GNU/Linux operating system. The executable is available for download at `https://gitlab.com/sosy-lab/sv-comp/archives-2020/raw/svcomp20/2020/veriabs.zip`, and the installation instructions are given in VeriAbs/INSTALL.txt. For a 64-bit

architecture VeriAbs accepts the -64 bit option, it considers a 32-bit architecture by default.

It checks that the function __VERIFIER_error() is never called in the input program. The user needs to specify this in a property file with a prp extension in the following format: CHECK(init(main()), LTL(G!call(__VERIFIER_error()))). A sample command to check the program ex.c with the property file a.prp is shown in the fig. 7.1

```
:~>./scripts/veriabs --property-file a.prp ex.c
```

Figure 7.1: Sample command

VeriAbs displays the verification result on the standard output. The usage of VeriAbs is shown in the Fig. 7.2. The experimental setup we used to run the benchmarks is the following: GNU/Linux OS; Resource limits of 15GB RAM, 15 min CPU time and 8 CPU cores. The results can be found in the SV-COMP'17,18,19 competition reports [5, 4, 3].

7.1.2 Participation in SV-Comp Competition

We succesfully participated in SV-Comp competition from 2017-2020 and performed well in many categories. In particular, we outperformed state-of-the-art model-checkers like CBMC, CPAChecker, UAtomizer, etc., in the ReachSafety category, one of the largest diverse programs category of SV-COMP comprising of 4079 tasks. Veriabs secured Gold in this category in 2019 and 2020 with a silver in 2018. The results can be found using the following links.

```
USAGE:

veriabs --property-file <.prp file> <source file>

To run VeriAbs on source file.

Option                          Meaning
--property-file <.prp file>     : The property specification file.
<source file>                   : The (preprocessed) source file.

Sample command:
VeriAbs/scripts/veriabs --property-file loops/ALL.prp test.i
```

Figure 7.2: Veriabs Usage

SV-Comp 2017 Competition: `https://sv-comp.sosy-lab.org/2017/results/results-verified/veriabs.2017-01-14_0946.results.sv-comp17.ReachSafety-Loops.xml.bz2.merged.xml.bz2.table.html`

SV-Comp 2018 Competition: `https://sv-comp.sosy-lab.org/2018/results/results-verified/veriabs.2017-12-02_1804.results.sv-comp18.ReachSafety-Loops.xml.bz2.merged.xml.bz2.table.html`

SV-Comp 2019 Competition: `https://sv-comp.sosy-lab.org/2019/results/results-verified/veriabs.2018-12-10_1650.results.sv-comp19_prop-reachsafety.ReachSafety-Loops.xml.bz2.merged.xml.bz2.table.html`

7.2 Acceleration of Loops: Experiments and Results

We implemented the techniques discussed in §4.1 and §5.8 as features extension in Veriabs. We tested the methods on the sugcategory of *Loops* in *ReachSafety* category of SV-Comp benchmarks. While the refinements using both approaches are always sound, the technique of Presburger Acceleration for non-flat loops is sound and complete as the acceleration is exact.

To answer the experimental questions, we also ran our implementation, as well as the state-of-the-art software model checkers like CPAChecker (version 1.9), Ultimate Automizer (version 0.1.25), and CBMC (version 5.12), on several suites of benchmark programs containing both true and false assertions. In Table 1, we report for each suite (i) the number of programs for which the anslyzer was able to prove all assertions, (ii) the number of timeout, out-of-memory, and unknown results. Timings were taken on a GNU/Linux OS machine with resource limits of 15GB RAM and 15 min CPU time and 8 CPU cores. The programs come from the subcategory of Loops in the ReachSafety category of SV-COMP17, SV-COMP18, and SV-COMP19. In total there are 193 loops from the following micro-categories of Loops: `1.` loop-acceleration `2.` loop-invgen `3.` loop-new `4.` loops, and `5.` loops-crafted-1.

In Table 2, we particularly analyze an interesting class of programs in the benchmark suite "loops-crafted-1" comprising 17 non-flat loops that are flat-table and precisely accelerable. Veriabs has outperformed state-of-the-art model-checkers in this particular category which highlights its power in verifying loops. The strength of exact acceleration, which is implemented in Veriabs, helped in verifying unsafe programs in this category.

Benchmark Suite		Veriabs		CPAChecker		UAutomizer		CBMC	
	#Programs	S	T/M/U	S	T/M/U	S	T/M/U	S	T/M/U
SV/loop-acceleration	35	**35**	0/0/0	22	13/0/0	17	18/0/0	21	0/14/3
SV/loop-invgen	29	**21**	8/0/0	16	13/0/0	25	4/0/0	2	0/5/22
SV/loop-new	11	**8**	2/0/1	6	5/0/0	4	7/0/0	6	0/1/4/0
SV/loops	66	**61**	1/0/4	57	9/0/0	50	14/0/2	43	0/10/0
SV/loops-crafted-1	52	**52**	0/0/0	2	46/1/3	16	36/0/0	7	43/2/0
Total	**193**	**177**	11/0/5	147	86/1/3	112	79/0/2	79	43/32/29

Table 7.1: Experimental Results on SV-COMP benchmarks: (S) - Solved, (T/M/U) - Timeout/Out of Memory/Unknown. Column 2 shows the total number of programs in each benchmark suite. Columns 3-10 show analysis results under 4 different conditions using state-of-the-art model checkers like Veriabs, CPAChecker, Ultimate Automizer, and CBMC. For each configuration, the left column indicates the number of programs in which all assertions were proven by the analyzer, and the right column is a triple T/M/U where T is the number of timeouts, M is the number of out-of-memory exceptions, and U for unknown result.

7.2.1 Does Refinement Allow More Assertions to be Proven?

In short the answer is "the acceleration techniques allows the analyzer to prove more assertions in practice by 15%". Note that we explicitly took into the account the programs which Veriabs fails with its current array of portfolio techniques like explicit state model checking, bounded model checking, program induction, abstract interpretation with widening, etc. The classical techniques like explicit state model checking and bounded model checking succeed on a majority of the programs, and a wide range of further techniques are required to analyse the rest. Many loops in these benchmarks, which are precisely accelerable, already succeed using the existing techniques which provide the right abstraction to prove safety assertions and refute false assertions. Without considering these loops, the technqiues implemented from this thesis allowed analyzer to prove more assertions in practice by 15%.

7.2.2 How Does Veriabs With Acceleration techniques Compare with State-of-the-Art Model Checkers?

CPAChecker: CPAchecker integrates most of the state-of-the-art technologies for software model checking, such as counterexample-guided abstraction

Benchmark type		Veriabs		CPAChecker		UAutomizer		CBMC	
	#Programs	S	T/M	S	T/M	S	T/M	S	T/M
Non-Flat (Safe)	11	**11**	0/0	0	11/0	1	10/0	2	0/9
Non-Flat (Unsafe)	5	**5**	0/0	0	5/0	0	5/0	0	0/5
Total	16	**16**	0/0	0	16/0	1	15/0	2	0/14

Table 7.2: Accelerable loops which are flattable in "loops-crafted-1" suite

refinement (CEGAR), lazy predicate abstraction, interpolation-based refinement, etc. Even with its power of using such techniques, it was out of memory for many flattable loops which are precisely accelerable as shown in Table 2. Its CEGAR based approach [6] for path programs fail to synthesize the *path programs* required to prove the safety assertion, and it gets stuck in its CEGAR loop until it eliminates all the spurious counterexample paths to prove the assertion.

Ultimate Automizer: Ultimate Automizer is a software verifier that uses a powerful technique Trace abstraction [22] that generalize proofs for traces to proofs for larger parts for the program. Its generalization box which generalize error paths to automata fail to synthesize the right automata abstraction to prove the safety assertions. This ultimately ends up in the basic CEGAR loop and thus enumerates single error paths (potentially infinitely many) to prove safety assertions until it covers all feasible program traces. Their approach timed out on many flattable loops as shown in Table 2.

CBMC: CBMC is a software verifier that uses a powerful technique of bounded model-checking [23] to prove and refute assertions. Although it succeeds well on accelerable loops due to sufficient unwinding depth, it neverthless fails on many flattable loops which are precisely accelerable as shown in the Table 2 and run out of resources: both time and memory.

Chapter 8

Conclusion

In this thesis, we studied various acceleration techniques, both exact and abstract, to improve the precision of the analysis of loops. We implemented a subset of these techniques in the tool Veriabs which improved its precision in analyzing the loops. This study helped understand the problem of incompleteness in static analyzers in the analysis of loops, where incompleteness often lead to false alarms and verification becomes a challenge. We revisited the theory of acceleration for an easy integration of acceleration techniques in the existing framework of software model-checking. We implemented and integrated acceleration techniques in the verification framework of Veriabs and compared it with start-of-the-art model checkers.

Bibliography

[1] Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. International Journal on Software Tools for Technology Transfer **10**(5), 401–424 (2008)

[2] Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: International Symposium on Automated Technology for Verification and Analysis. pp. 474–488. Springer (2005)

[3] Beyer, D.: Software verification with validation of results. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 331–349. Springer (2017)

[4] Beyer, D.: Automatic verification of c and java programs: Sv-comp 2019. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 133–155. Springer (2019)

[5] Beyer, D.: Advances in automatic software verification: Sv-comp 2020. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 347–367. Springer (2020)

[6] Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Acm Sigplan Notices. vol. 42, pp. 300–309. ACM (2007)

[7] Boigelot, B.: Symbolic methods for exploring infinite state spaces. Ph.D. thesis, Universite de Liege, Liege, Belgium (1998)

[8] Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using qdds. Formal Methods in System Design **14**(3), 237–255 (1999)

[9] Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Formal Methods in Programming and their Applications. pp. 128–141. Springer (1993)

[10] Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: International Static Analysis Symposium. pp. 145–161. Springer (2015)

[11] Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1407–1412. IEEE (2015)

[12] Finkel, A., Leroux, J.: How to compose presburger-accelerations: Applications to broadcast protocols. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 145–156. Springer (2002)

[13] Fukuda, K., Prodon, A.: Double description method revisited. In: Franco-Japanese and Franco-Chinese Conference on Combinatorics and Computer Science. pp. 91–111. Springer (1995)

[14] Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: European symposium on programming. pp. 300–315. Springer (2007)

[15] Gawlitza, T.M., Seidl, H.: Precise relational invariants through strategy iteration. In: Computer Science Logic. Lecture Notes in Computer Science, vol. 4646, pp. 23–40. Springer (2007)

[16] Gawlitza, T.M., Seidl, H.: Precise relational invariants through strategy iteration. In: Computer Science Logic. Lecture Notes in Computer Science, vol. 4646, pp. 23–40. Springer (2007)

[17] Gawlitza, T.M., Monniaux, D.: Improving strategies via smt solving. In: European Symposium on Programming. pp. 236–255. Springer (2011)

[18] Gawlitza, T.M., Monniaux, D.: Invariant generation through strategy iteration in succinctly represented control flow graphs. Logical Methods in Computer Science **8**(3) (2012)

[19] Gawlitza, T.M., Seidl, H.: Solving systems of rational equations through strategy iteration. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(3), 11 (2011)

[20] Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: International Static Analysis Symposium. pp. 144–160. Springer (2006)

[21] Gonnord, L., Schrammel, P.: Abstract acceleration in linear relation analysis. Science of Computer Programming **93**, 125–153 (2014)

[22] Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: International Conference on Computer Aided Verification. pp. 36–52. Springer (2013)

[23] Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014)

[24] Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: International Static Analysis Symposium. pp. 369–385. Springer (2011)

[25] Monniaux, D., Schrammel, P.: Speeding up logico-numerical strategy iteration. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 8723, pp. 253–267. Springer (2014)

[26] Presburger, M.: Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. In: Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs (1929)

[27] Tarski, A., et al.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955)

[28] Xie, X., Chen, B., Liu, Y., Le, W., Li, X.: Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 61–72 (2016)