UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Présentée par

Julien Henry

Thèse dirigée par **Dr. David Monniaux**
et coencadrée par **Dr. Matthieu Moy**

préparée au sein du laboratoire **Verimag**
et de l'école doctorale **MSTII**

Static Analysis by Abstract Interpretation and Decision Procedures

Thèse soutenue publiquement le **13 octobre 2014**,
devant le jury composé de :

Pr. Roland Groz
Professeur, Grenoble INP, Président
Dr. Antoine Miné
Chargé de Recherche CNRS, Rapporteur
Pr. Cesare Tinelli
Professor, University of Iowa, Rapporteur
Dr. Hugues Cassé
Maître de Conférences, Université de Toulouse, Examinateur
Pr. Andreas Podelski
Professor, University of Freiburg, Examinateur
Dr. David Monniaux
Directeur de Recherche CNRS, Directeur de thèse
Dr. Matthieu Moy
Maître de Conférences, Grenoble INP, Co-Encadrant de thèse

Remerciements

TODO This is a preliminary version

Je tiens spécialement à remercier:
I would like to thank:

- David Monniaux et Matthieu Moy, mes directeurs de thèse, pour leur encadrement pendant ces trois ans,
  David Monniaux and Matthieu Moy, my advisors, for having supervised these three years of work,
- Antoine Miné et Cesare Tinelli, pour avoir accepté de rapporter ce manuscrit,
  Antoine Miné and Cesare Tinelli, for having accepted to review this manuscript,
- Roland Groz, Hugues Cassé et Andreas Podelski, pour avoir accepté de participer au jury de ma thèse.
  Roland Groz, Hugues Cassé and Andreas Podelski, for having accepted to take part in the jury.

Abstract

Static program analysis aims at automatically determining whether a program satisfies some particular properties. For this purpose, *abstract interpretation* is a framework that enables the computation of invariants, i.e. properties on the variables that always hold for any program execution. The precision of these invariants depends on many parameters, in particular the *abstract domain*, and the iteration strategy for computing these invariants. In this thesis, we propose several improvements on the abstract interpretation framework that enhance the overall precision of the analysis.

Usually, abstract interpretation consists in computing an ascending sequence with widening, which converges towards a fixpoint which is a program invariant; then computing a descending sequence of correct solutions without widening. We describe and experiment with a method to improve a fixpoint after its computation, by starting again a new ascending/descending sequence with a smarter starting value. Abstract interpretation can also be made more precise by distinguishing paths inside loops, at the expense of possibly exponential complexity. Satisfiability modulo theories (SMT), whose efficiency has been considerably improved in the last decade, allows sparse representations of paths and sets of paths. We propose to combine this SMT representation of paths with various state-of-the-art iteration strategies to further improve the overall precision of the analysis.

We propose a second coupling between abstract interpretation and SMT in a program verification framework called *Modular Path Focusing*, that computes function and loop *summaries* by abstract interpretation in a modular fashion, guided by error paths obtained with SMT. Our framework can be used for various purposes: it can prove the unreachability of certain error program states, but can also synthesize function/loop preconditions for which these error states are unreachable.

We then describe an application of static analysis and SMT to the estimation of program worst-case execution time (WCET). We first present how to express WCET as an *optimization modulo theory* problem, and show that natural encodings into SMT yield formulas intractable for all current production-grade solvers. We propose an efficient way to considerably reduce the computation time of the SMT-solvers by conjoining to the formulas well chosen summaries of program portions obtained by static analysis.

We finally describe the design and the implementation of Pagai, a new static analyzer working over the LLVM compiler infrastructure, which computes numerical inductive invariants using the various techniques described in this thesis. Because of the non-monotonicity of the results of abstract interpretation with widening operators, it is difficult to conclude that some abstraction is more precise than another based on theoretical local precision results. We thus conducted extensive comparisons between our new techniques and previous ones, on a variety of open-source packages and benchmarks used in the community.

Résumé

L'analyse statique de programme a pour but de prouver automatiquement qu'un programme vérifie certaines propriétés. L'*interprétation abstraite* est un cadre théorique permettant de calculer des invariants de programme. Ces invariants sont des propriétés sur les variables du programme vraies pour toute exécution. La précision des invariants calculés dépend de nombreux paramètres, en particulier du *domaine abstrait* et de l'ordre d'itération utilisés pendant le calcul d'invariants. Dans cette thèse, nous proposons plusieurs extensions de cette méthode qui améliorent la précision de l'analyse.

Habituellement, l'interprétation abstraite consiste en un calcul de point fixe d'un opérateur obtenu après convergence d'une *séquence ascendante*, utilisant un opérateur appelé *élargissement*. Le point fixe obtenu est alors un invariant. Il est ensuite possible d'améliorer cet invariant via une *séquence descendante* sans élargissement. Nous proposons une méthode pour améliorer un point fixe après la séquence descendante, en recommençant une nouvelle séquence depuis une valeur initiale choisie judiscieusement. L'interprétation abstraite peut également être rendue plus précise en distinguant tous les chemins d'exécution du programme, au prix d'une explosion exponentielle de la complexité. Le problème de satisfiabilité modulo théorie (SMT), dont les techniques de résolution ont été grandement améliorée cette décennie, permettent de représenter ces ensembles de chemins implicitement. Nous proposons d'utiliser cette représentation implicite à base de SMT et de les appliquer à des ordres d'itération de l'état de l'art pour obtenir des analyses plus précises.

Nous proposons ensuite de coupler SMT et interprétation abstraite au sein de nouveaux algorithmes appelés *Modular Path Focusing* et *Property-Guided Path Focusing*, qui calculent des *résumés* de boucles et de fonctions de façon modulaire, guidés par des traces d'erreur. Notre technique a différents usages: elle permet de montrer qu'un état d'erreur est inatteignable, mais également d'inférer des préconditions aux boucles et aux fonctions.

Nous appliquons nos méthodes d'analyse statique à l'estimation du temps d'exécution pire cas (WCET). Dans un premier temps, nous présentons la façon d'exprimer ce problème via *optimisation modulo théorie*, et pourquoi un encodage naturel du problème en SMT génère des formules trop difficiles pour l'ensemble des solveurs actuels. Nous proposons un moyen simple et efficace de réduire considérablement le temps de calcul des solveurs SMT en ajoutant aux formules certaines propriétés impliquées obtenues par analyse statique.

Enfin, nous présentons l'implémentation de Pagai, un nouvel analyseur statique pour LLVM, qui calcule des invariants numériques grâce aux différentes méthodes décrites dans cette thèse. Nous avons comparé les différentes techniques implémentées sur des programmes open-source et des benchmarks utilisés par la communauté.

Contents

Chapter 1

Introduction

1.1 Context

The last decades have seen the emergence and the impressive expansion of embedded computing systems. They range from small portable devices such as smartphones, digital watches, MP3 players, etc., to very large and complex systems, like in avionics, automotive or medical devices. These systems all share the property that failures can not be tolerated: for a smartphone produced in millions of units, it may be dramatically expensive to fix; on avionics or medical devices, it may threaten human lives; on nuclear power plants it involves high environmental risks. In addition, the specifications for these systems are highly constrained, in terms of energy consumption, maximal execution time, etc., which makes the validation even more challenging.

Nowadays, around six billion processor chips are produced every year, and 98% of them end-up in embedded systems [JPC14]. At the same time, these systems become even more complex: the size of the embedded software may contain several millions of lines of code, which increases the risk of encountering bugs. History already gave examples of bugs with dramatic impacts: one can classically cite the Ariane 5 crash in 1996 due to an integer overflow [Ari96], or the death of 8 people in the National Cancer Institute of Panama in 2000 due to a bug in the graphical user interface of the radiation therapy machine.

This motivates the need of a very reliable validation process for these safety-critical systems, that provides trustworthy guarantees that the software fulfills its specification. In some cases, this process is imposed by certification authorities, e.g. the DO-178C standard for airborne systems. The approaches for validating softwares can be divided in two categories:

- *Testing*, or *dynamic Analysis*, which consists in observing the behavior of the software given some well-chosen sample inputs. This approach can detect bugs but is not exhaustive, since all inputs cannot be tested in general. It thus does not prove the system has no bugs.
- *Static Analysis* is the class of validation techniques which in contrast are performed on a static representation of the program. Some of these techniques mathematically prove that the code does not have bugs without executing it, while others are able to detect bugs. This thesis fits into the significant research effort for constructing efficient and precise static analysis algorithms and techniques.

1.2 Static Analysis

Static program analysis aims at automatically determining whether a program satisfies some particular properties. This is an important research topic, since it tackles a problem which is

known to be undecidable (Rice's theorem): in other words, it is impossible to design a static analysis that proves any non-trivial property on any program both exactly and automatically.

An important property of software validation tools is *soundness*: they have to take into account every possible behavior of the software. In the case the tool is sound, it allows to mathematically prove the absence of bug in the analyzed program. In this thesis, we do not talk about unsound analysis techniques, even though they are also useful and used in practice for finding bugs. Bug finders include the industrial tools Coverity[1] and Polyspace Bug Finder[2].

Sound and automatic software verification tools mostly rely on the following approaches:

- *Model Checking* [CES86]: it consists in exploring the possible states of the program during its execution. Since the number of states can be infinite or extremely large, *symbolic model checking* considers large numbers of states at the same time and represents them compactly. *Bounded model checking* only considers the execution traces up to a certain length, in order to keep the state space tractable. The analysis is then unsound since some traces are ignored. State-of-the-art model checking techniques abstract the program behavior into a simpler model, and iteratively refine this abstraction until the given property is proved correct in this model. SPIN [Hol97], BLAST [BHJM07], CBMC [CKL04] or Java PathFinder[3] are examples of model checkers.

- *Abstract Interpretation* [CC77, CC92] computes an over-approximation of the set of reachable program states. It aims at discovering invariants of a particular well-chosen shape – referred to as *abstract domain* – that allow to prove the properties of interest. Industrial tools like Astrée [BCC+03] or Polyspace [Pol] can prove the absence of runtime errors on large embedded softwares written in C. Nowadays, Abstract interpretation is used beyond the scope of embedded softwares: Microsoft Visual Studio IDE incorporates a static analyzer [Log11] for automatically checking correctness specifications for .NET bytecode.

- *Symbolic Execution* [Kin76] consists in executing the program while keeping input variables symbolic rather than assigning them a value. In this way, one can derive path invariants, i.e. properties on the variables that are always true for a given path. This method would not scale if applied naively because of an exponential blowup in the numbers of paths, and requires more evolved techniques for merging [KKBC12] or heuristically ignoring certain paths.

These approaches are tightly related: some model-checking based tools make use of abstract interpretation. Symbolic execution can be seen as an instance of abstract interpretation. In this thesis, we propose new abstract interpretation based techniques that integrate some kind of bounded model checking for increasing the precision of the results.

1.3 Contributions

The major drawback of abstract interpretation is over-approximation: since the real set of reachable program states cannot be computed exactly, one computes a bigger set that contains the real one. This approximate set is commonly called *abstract value*. However, this approximation is frequently too rough and not sufficient to prove the desired properties, mostly because of two reasons:

[1] https://scan.coverity.com/
[2] http://www.mathworks.fr/products/polyspace-bug-finder/
[3] http://babelfish.arc.nasa.gov/trac/jpf

(1) the abstract value is restricted to a particular pattern, called *abstract domain*, and precision is lost throughout the analysis because of the limited expressiveness of this domain.

(2) the set is obtained in finite time thanks to an *extrapolation* of the behavior of the program (called *widening*). This widening induces non-recoverable and unpredictable loss of precision throughout the analysis.

As a result, for a given program and a given property, an abstract interpretation based static analyzer can only answer "*correct*" – if the computed set is precise enough to prove it – or "*I don't know*": maybe there is a bug in the real program, or maybe the approximation is simply not precise enough to prove it. The latter case is usually called *false alarm*, or *false positive*. An analysis with too many false positives is not tolerable: human attention is needed for every alarm to check whether it is real or a false positive.

In this thesis, we tackle both problems **(1)** and **(2)**, and propose new solutions for limiting these loss of precision. We also provide a large practical contribution with the implementation of a new abstract interpretation based static analyzer called Pagai.

We propose contributions in four different areas of static analysis:

Recovering precision after computation with widening The approximate set is computed iteratively by an *ascending sequence* that performs widenings. Once an invariant is found, it can usually be made more precise by a *descending sequence*. In this thesis, we describe and experiment with a new way of performing this descending sequence, and thus improve the precision lost due to point **(2)**.

Our experiments show that this approach improves the results of abstract interpretation in practice, and have been published in [HH12]. This contribution is detailed in section 4.1.

Combination with bounded model checking Abstract Interpretation computes an invariant of a particular shape, called abstract domain, at each program location. Point **(1)** can be addressed by applying abstractions using this abstract domain only at a subset of the program locations. This can be achieved by using bounded model checking on some portions of the analyzed program, so that precision is not lost on these portions. Bounded model checking then provides the abstract interpretation engine with an iteration strategy for computing a more precise invariant. We propose to apply this idea to several state-of-the art abstract interpretation techniques to increase precision. Our work has been published in [HMM12b], and is detailed in section 4.2.

Secondly, in chapter 5, we extend this work to a modular and inter-procedural analysis that takes advantage of both bounded model checking and abstract interpretation to automatically derive useful function preconditions, while lazily analyzing each program fragments to improve scalability.

Pagai static analyzer We propose a new static analyzer called Pagai, published in [HMM12a], based on the LLVM compiler infrastructure, written in C++ with around 20.000 lines of code. It implements the various techniques presented throughout this thesis, and allows running experiments on real softwares written in C, C++, or Objective-C and compare both the precision and the cost of the various methods. It is able to derive precise numerical program invariants and prove the absence of integer overflows as well as array out of bounds accesses. It also checks for the validity of *assert* statements. It has already been used by other research teams for experimenting with their own techniques [AS13].

Application to Worst-Case Execution Time (WCET) We propose a way of applying bounded model checking for deriving precise and semantic-sensitive Worst Case Execution Time bounds for loop-free programs, taking benefits of the great research advances in the field of *Satisfiability Modulo Theories (SMT)*. We show both theoretically and experimentally that a simple encoding of the WCET problem into SMT formula is too hard to solve for state-of-the-art solvers. We then propose a method that transforms this formula into an equisatisfiable one, which is solved in a reasonable amount of time. The work has been published in [HAMM14] and is detailed in chapter 6.

1.4 Outline

This thesis is organized into three parts:

Part I gives an introduction to static analysis techniques based on abstract interpretation. It first recalls the standard framework of abstract interpretation, as well as several other approaches for program verification relevant for this thesis, in particular bounded model checking. Then, it mentions the main state-of-the-art improvements over the initial abstract interpretation algorithm, used for improving its precision.

Part II is dedicated to our theoretical contributions: we start with a new method for improving the precision after an invariant has been found. We propose a new algorithm called *Guided Path Analysis*, that takes best of two state-of-the-art iteration strategies for improving the overall precision of the analysis with the help of decision procedures. We propose an extension of this last technique for the computation of disjunctive invariants. Then, we propose a framework for a modular static analysis based on the combination of SMT and abstract interpretation. The framework generates function and loop summaries, and infers preconditions for the function so that an error state is not reachable.

Part III contains our practical contributions as well as an application of static analysis to the estimation of *Worst-Case Execution Time* (WCET) of programs. We start with a new approach for computing WCET based on bounded model checking by SMT-solving. The last part is dedicated to the presentation of our new static analyzer, Pagai, as well as experimental results.

Part I

State of the Art

Chapter 2

Program Invariants Computation by Static Analysis

2.1 Basics of Abstract Interpretation

Abstract interpretation [CC77, CC92] is a general framework for computing sound approximations of program semantics. It has already been widely used for the design of static analyzers, such as Astrée [BCC+03] or Polyspace [Pol], whose objective is to prove that the program under analysis satisfies some important properties such as "The program never dereferences a null pointer", "The program never divides by zero", or more generally "These classes of runtime errors cannot happen" and "the user-specified assertions are never violated". Most of these properties are undecidable in general because of the too high complexity of formal semantics of programming languages. However, one can work with a sound and decidable abstraction of this semantics, that can be used for computing an over-approximation of the set of reachable program states. Abstract interpretation gives a theoretical framework for designing sound-by-construction approximate semantics.

Figure 2.1 illustrates the main principle of the approach: the set R is the non-computable set of reachable states. Abstract interpretation aims at over-approximating this set by a simpler set A that contains R. This set A should be sufficiently precise to prove the absence of bugs: here, the intersection of A with a set of error states E_2 is empty, meaning that every state in E_2 is proved unreachable. Conversely, A intersects with the set E_1 while R does not: the over-approximation A is too rough to prove the unreachability of the errors in E_1.

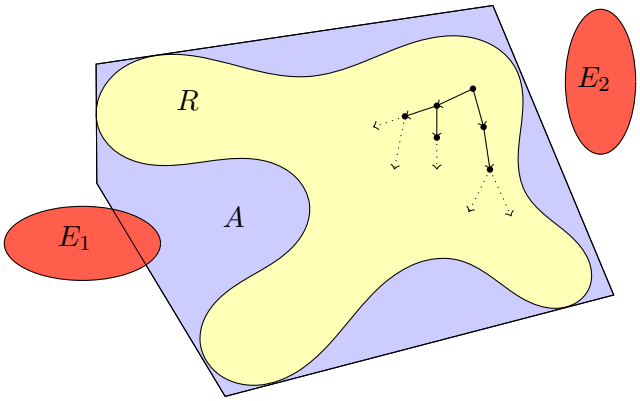


Figure 2.1: Abstract Interpretation computes over-approximations. The errors in E_2 are proved unreachable, while the non-empty intersection of A with E_1 raises a *false-alarm*.

In this section, we describe abstract interpretation as proposed in [CC77, CC92]. First, we start with usual required definitions.

2.1.1 Definitions

First, we need to introduce standard definitions needed for describing abstract interpretation.

Definition 1 (Partially ordered set)**.** A *Partially ordered set* $(\mathcal{P}, \sqsubseteq)$ is a set $\mathcal{P}$ together with a partial order $\sqsubseteq$, i.e. a binary relation verifying the properties:

- $\forall x \in \mathcal{P}, x \sqsubseteq x$ (reflexivity)
- $\forall (x, y) \in \mathcal{P}^2, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ (antisymmetry)
- $\forall (x, y, z) \in \mathcal{P}^3, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ (transitivity)

Definition 2 (Upper and Lower Bounds)**.** Let $(\mathcal{P}, \sqsubseteq)$ be a partially ordered set, and $S \subseteq \mathcal{P}$. An element $u \in \mathcal{P}$ is an *upper bound* of S if $\forall s \in S, s \sqsubseteq u$. u is the *least upper bound* of S (noted $\sqcup S$) if $u \sqsubseteq u'$ for each upper bound u' of S. Similarly, $l \in \mathcal{P}$ is a *lower bound* of S if $\forall s \in S, u \sqsubseteq s$. l is the *greatest lower bound* of S (noted $\sqcap S$) if $l' \sqsubseteq l$ for each lower bound l' of S.

Definition 3 (Lattice)**.** A *lattice* $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where each pair $(x, y) \in \mathcal{L}^2$ has a least upper bound, noted $x \sqcup y$, and a greatest lower bound, noted $x \sqcap y$. $\sqcup : \mathcal{L} \times \mathcal{L} \longrightarrow \mathcal{L}$ and $\sqcap : \mathcal{L} \times \mathcal{L} \longrightarrow \mathcal{L}$ are respectively called *join* and *meet* operators.

A *complete lattice* is a lattice for which every subset, possibly infinite, has a least upper bound and an greatest lower bound. In this case, we note $\bot \stackrel{\text{def}}{=} \sqcap \mathcal{L}$ (called *bottom*) and $\top \stackrel{\text{def}}{=} \sqcup \mathcal{L}$ (called *top*). In particular, if $\mathcal{P}(X)$ is the set of all subsets of a set X, also called *powerset*, then $(\mathcal{P}(X), \subseteq, \cup, \cap)$ is a complete lattice.

Definition 4 (Ascending Chain Condition)**.** A partially ordered set $(\mathcal{P}, \sqsubseteq)$ is said to satisfy the *ascending chain condition* if for any increasing sequence $(x_i)_{i \in \mathbb{N}}$ of elements of $\mathcal{P}$, there exist $k \in \mathbb{N}$ such that $(x_i)_{i > k}$ is stationary.

Definition 5 (Properties of Application)**.** Let F be an application $\mathcal{P}_1 \longrightarrow \mathcal{P}_2$ between two partially ordered sets $(\mathcal{P}_1, \sqsubseteq_1)$ and $(\mathcal{P}_2, \sqsubseteq_2)$. F is said to be *monotonic* if $\forall (x, x') \in \mathcal{P}_1^2, x \sqsubseteq_1 x' \Rightarrow F(x) \sqsubseteq_2 F(x')$. We call *operator* an application $F : \mathcal{P} \longrightarrow \mathcal{P}$, i.e. an application from a partially ordered set to itself. In this case, we note $F^i(x)$ the i-th iterate of F on x, such that $F^1(x) = F(x)$ and $\forall i > 1, F^i(x) = F(F^{i-1}(x))$.

Definition 6 (Fixpoint)**.** Let $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ be a lattice and $F : \mathcal{L} \to \mathcal{L}$. An element $x \in \mathcal{L}$ is called a *fixpoint* of an operator F if $F(x) = x$. Similarly, it is called a *pre-fixpoint* if $F(x) \sqsupseteq x$, and is a *post-fixpoint* if $F(x) \sqsubseteq x$. If they exists, the least fixpoint of F is noted $\text{lfp}(F)$, and the greatest fixpoint is noted $\text{gfp}(F)$.

Program Model

The behavior of a program can be formally described by a *Transition System*, that defines a set S of program states, a set $I \subseteq S$ of possible initial states, and a transition relation $R \subseteq S \times S$. If $(p, q) \in R$, we usually note $p \to q$ to represent the fact that there is a transition from p to q. With this formalism, the control structure of the program is encoded in the transition relation.

In this thesis, we often represent programs in the form of *Control Flow Graphs*, which has the benefit of explicitly encoding the control structure of the program:

Definition 7 (Control-flow Graph)**.** A *control-flow graph* is a directed graph $G = (N, I, T, \Sigma)$, where:

- N is a set of program locations, also called program points. The elements of N are the nodes of the graph.
- $I \subseteq N$ is the set of initial program locations,
- Σ is a set, (usually, the set of program states, or later an abstraction thereof),
- T is a set of transitions $(i, \tau_{i,j}, j)$ between two locations i and $j \in N$, noted $i \xrightarrow{\tau_{i,j}} j$. $\tau_{i,j}$ is an operator $\Sigma \to \Sigma$ that we call *forward transformer*.

The transition relations are typically defined from the semantics of the program instructions.

2.1.2 Reachable Program States as a Fixpoint

Let States be the set of all possible *states* of a program. Informally, the *state* of a program is the current value for each registers — including data registers, but also program counter, etc. — and related memory, i.e. stack and heap. The powerset $\mathcal{P}(\mathsf{States})$ is the set of all subsets of States. In particular, the set of reachable states is in $\mathcal{P}(\mathsf{States})$, but is not computable in general. The semantics of a program can be defined by a *transition relation* τ that maps a set of states to the next reachable states after one atomic step of execution. Starting from an initial state $x_0 \in \mathsf{States}$, we can reach the states $\{x_i \in \mathsf{States} \ / \ x_i \in \tau^i(\{x_0\})\}$. It follows that if S is a set of states, the set of states reachable after k atomic execution steps is $\tau^k(S)$. If $I \subseteq \mathsf{States}$ is the set of possible initial states of the program, then the set of reachable states is defined by:

$$\tau^*(I) \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} \tau^i(I)$$

Let $\Phi : \mathcal{P}(\mathsf{States}) \longrightarrow \mathcal{P}(\mathsf{States})$ defined by:

$$S \longmapsto I \cup \tau(S)$$

It is immediate to see that $\Phi(\tau^*(I)) = I \cup \tau(\tau^*(I)) = I \cup \bigcup_{i \geq 1} \tau^i(I) = \tau^*(I)$. Indeed, a reachable state is either an initial state, or a state reachable in one step from a reachable state. The set of reachable state is thus a fixpoint of Φ.

Φ is monotonic because τ is monotonic, and I is a pre-fixpoint of Φ, so we can use Kleene's theorem to say that the least fixpoint of Φ, noted $\text{lfp}(\Phi)$, is equal to $\lim_{k \to +\infty} \Phi^k(\bot)$. It is easy to see that this limit is actually equal to $\tau^*(I)$, so we deduce the important result that $\tau^*(I) = \text{lfp}(\Phi)$.

Example 1. Suppose we have the very simple program depicted in Figure 2.2.

```
unsigned x = 0;
while (x < 100) {
  x+=2;
}
```

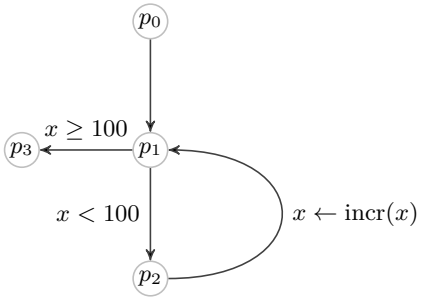


Figure 2.2: Very simple C program, and its corresponding control-flow graph.

In the control-flow graph, incr is the function defined as:

$$\begin{array}{rcl} \text{incr} : \mathbb{N} & \longrightarrow & \mathbb{N} \\ x & \longmapsto & x+2 \end{array}$$

For simplicity, we suppose here that x is a mathematical natural number instead of being bounded by `UMAX`, the greatest `unsigned` value in C. We can consider here that a program state is actually an element of $\mathsf{States} = \{p_0, p_1, p_2, p_3\} \times \mathbb{N}$. The transition relation τ can be easily derived from the graph:

$$\begin{array}{rcl} \tau(\{(p_0, x)\}) & = & \{(p_1, x)\} \\ \tau(\{(p_1, x)\}) & = & \begin{cases} \{(p_2, x)\} \text{ if } x < 100 \\ \{(p_3, x)\} \text{ if } x \geq 100 \end{cases} \\ \tau(\{(p_2, x)\}) & = & \{(p_1, x+2)\} \end{array}$$

The set of initial states is $I = \{(p_0, 0)\}$. The set of reachable states is the least fixpoint of Φ as previously defined. In this case,

$$\begin{array}{rcl} \Phi(\bot) & = & I = \{(p_0, 0)\} \\ \Phi^2(\bot) & = & \{(p_0, 0), (p_1, 0)\} \\ \Phi^3(\bot) & = & \{(p_0, 0), (p_1, 0)\} \cup \{(p_2, 0)\} \\ \Phi^4(\bot) & = & \{(p_0, 0), (p_1, 0), (p_2, 0), (p_1, 2)\} \\ \dots & & \end{array}$$

After some steps, (Φ^k) becomes stationary (which is not the case in general) and is equal to:

$$\begin{array}{rcl} \Phi^\infty(\bot) & = & \{(p_0, 0)\} \cup \\ & & \{(p_1, x), 0 \leq x \leq 100 \wedge x \equiv 0(2)\} \cup \\ & & \{(p_2, x), 0 \leq x \leq 99 \wedge x \equiv 0(2)\} \cup \\ & & \{(p_3, 100)\} \\ & = & \text{lfp}(\Phi) = \tau^*(I) \end{array}$$

From this result, we can deduce program *invariants*, i.e. properties on the reached program states that are true for every possible execution. Here, an interesting invariant is: "After the loop, variable x is equal to 100."

2.1.3 Concrete and Abstract Domains

For a given transition relation τ, the most precise property of interest would be "the set of reachable states is $\tau^*(I)$". However, Rice's theorem implies that this set is not computable in general. The fundamental reasons are:

- The elements of $\mathcal{P}(\mathsf{States})$, and in particular $\tau^*(I)$, may not be machine-representable,
- The transition relation τ and the Φ operator may not be computable,
- The least fixpoint of Φ may not be reached after a finite number of Kleene iterations.

However, since the discovered invariant is typically used for proving some property P about the program: $\forall x \in \tau^*(I), P(x)$, it is sufficient for proving this property to find a computable set $\mathcal{R}$ that includes $\tau^*(I)$ and for which $\forall x \in \mathcal{R}, P(x)$. From this set, the obtained invariant will be "the set of reachable states is included in $\mathcal{R}$" and will be precise-enough to prove the property of interest. This set $\mathcal{R}$ should be a machine-representable element of a simpler domain called

abstract domain and noted $X^\sharp$. The invariant that will be discovered is weaker than $\tau^*(I)$: it may be the case that the computed set $\mathcal{R}$ does not verify the property P while $\tau^*(I)$ does. Indeed, $\mathcal{R}$ over-approximates the real set of reachable states, and there may exist an element $x \in \mathcal{R} \backslash \tau^*(I)$ for which $P(x)$ does not hold. In this case, P is not proved correct and is called *false-positive*. The objective is then to minimize the number of *false-positives* in the analysis by computing sufficiently precise program invariants.

Definition 8 (Abstract Domain)**.** Let X be the set of elements to be abstracted. X is called *concrete domain*. We assume that $(X, \sqsubseteq, \sqcup, \sqcap)$ is a lattice, where $\sqsubseteq$ is the partial order, and $\sqcup$ and $\sqcap$ are respectively the join and meet operators. An *abstract domain* over the concrete domain X is a couple $(X^\sharp, \gamma)$, where γ is a function from $X^\sharp$ to X called *concretization function*, and $(X^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$ is a lattice satisfying the following properties:

- $\sqsubseteq^\sharp$ is a sound approximation of $\sqsubseteq$:

$$\forall x^\sharp, y^\sharp \in X^\sharp, x^\sharp \sqsubseteq^\sharp y^\sharp \Rightarrow \gamma(x^\sharp) \sqsubseteq \gamma(y^\sharp)$$

  In other words, we can also say that the concretization function γ is monotonic.

- The abstract join operator $\sqcup^\sharp : X^\sharp \times X^\sharp \longrightarrow X^\sharp$ is a sound abstraction of $\sqcup : X \times X \longrightarrow X$:

$$\forall x^\sharp, y^\sharp \in X^\sharp, \gamma(x^\sharp) \sqcup \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcup^\sharp y^\sharp)$$

- Similarly, the abstract meet operator $\sqcap^\sharp : X^\sharp \times X^\sharp \longrightarrow X^\sharp$ abstracts $\sqcap : X \times X \longrightarrow X$ such that:

$$\forall x^\sharp, y^\sharp \in X^\sharp, \gamma(x^\sharp) \sqcap \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcap^\sharp y^\sharp)$$

An element $x^\sharp \in X^\sharp$ represents an element $\gamma(x^\sharp) \in X$, and is a sound abstraction of any x such that $x \sqsubseteq \gamma(x^\sharp)$. Intuitively, we can say that $x^\sharp$ is *more precise* than $y^\sharp$ if $x^\sharp \sqsubseteq^\sharp y^\sharp$, since it represents a set which is smaller.

Example. In the previous example, we had $\mathsf{States} = \{p_0, p_1, p_2, p_3\} \times \mathbb{N}$. The set X is then $\mathcal{P}(\mathsf{States})$. A possible abstract domain would be $X^\sharp = \mathcal{I}$, where $(\mathcal{I}, \sqsubseteq, \sqcup, \sqcap)$ is the complete lattice of the intervals of $\mathbb{N}$:

- $\forall x, y, x', y' \in \mathbb{N}, [x, y] \sqsubseteq [x', y'] \Leftrightarrow x \geq x'$ and $y \leq y'$,
- $\forall x, y, x', y' \in \mathbb{N}, [x, y] \sqcup [x', y'] = [\min(x, x'), \max(y, y')]$,
- $\forall x, y, x', y' \in \mathbb{N}, [x, y] \sqcap [x', y'] = [\max(x, x'), \min(y, y')]$.

where $[x, y] = \emptyset$ when $x > y$. The concretization function γ can be defined as:

$$\begin{array}{rccl} \gamma : & \mathcal{I} & \longrightarrow & \mathcal{P}(\mathsf{States}) \\ & [x, y] & \longmapsto & \{(p_i, x_i) \in \mathsf{States}, x_i \in [x, y]\} \end{array}$$

The best abstraction of the reachable program states in this abstract domain is $x^\sharp = [0, 100]$, and the concretization of this element is $\gamma(x^\sharp) = \{(p_i, x_i) \in \mathsf{States}, x_i \in [0, 100]\}$ that includes $\tau^*(I)$. Since, for instance, $(p_3, 0)$ is included in $\gamma(x^\sharp)$ the property "After the loop, variable x is equal to 100" is no longer proved correct, because of the too rough abstraction.

2.1.4 Fixpoint Computation

We have seen that the set of reachable states of a program can be expressed as the least fixpoint of an operator Φ over the concrete domain. The idea is then to compute a fixpoint of an operator $\Phi^\sharp$ over the abstract domain that is a sound approximation of Φ, i.e.:

$$\forall x^\sharp \in X^\sharp, \Phi \circ \gamma(x^\sharp) \sqsubseteq \gamma \circ \Phi^\sharp(x^\sharp)$$

$\Phi^\sharp$ can be derived from a *forward abstract transformer* $\tau^\sharp : X^\sharp \to X^\sharp$ that satisfies the property:

$$\forall x^\sharp \in X^\sharp, \forall x, y \in X, x \sqsubseteq \gamma(x^\sharp) \wedge \tau(x) = y \Rightarrow y \sqsubseteq \gamma \circ \tau^\sharp(x^\sharp)$$

The fixed point transfer theorem [CC77] states that the least fixpoint of $\Phi^\sharp$ is a sound abstraction of the least fixpoint of Φ:

$$\gamma(\mathrm{lfp}(\Phi^\sharp)) \sqsupseteq \mathrm{lfp}(\Phi)$$

It follows that any fixpoint of $\Phi^\sharp$ is a safe abstraction of the least fixpoint of Φ. Thus, one can just concentrate on computing a fixpoint of $\Phi^\sharp$, which can be done by computing the limit of the Kleene's sequence defined inductively by:

$$\begin{cases} x_0^\sharp = & \bot \\ x_i^\sharp = & \Phi^\sharp(x_{i-1}^\sharp), \forall i > 0 \end{cases}$$

At each step, $x_i^\sharp$ is a sound abstraction of $x_i = \Phi^i(\bot)$. Moreover, $\Phi^\sharp$ is monotonic, and $(\Phi^{\sharp i}(\bot))_{i\geq 1}$ forms an increasing sequence. In the case $X^\sharp$ satisfies the *ascending chain condition*, the sequence eventually stabilizes to some limit $\tilde{x}^\sharp = \mathrm{lfp}(\Phi^\sharp)$, which is a sound abstraction of the set of reachable states. Generally, $x_0^\sharp$ is defined to be equal to $\bot$ since $\bot$ is always a prefixpoint of $\Phi^\sharp$, but $x_0^\sharp$ could be actually set to any prefixpoint of $\Phi^\sharp$.

However, most of the interesting abstract domains do not satisfy the ascending chain condition, and thus the Kleene's iteration may never converge to a fixpoint. In order to ensure the termination of the computation, one must define and use a *widening* operator:

Definition 9 (Widening Operator)**.** An operator $\nabla : X^\sharp \times X^\sharp \longrightarrow X^\sharp$ is said to be a *widening* operator if it satisfies the following properties:

- $\forall x^\sharp, y^\sharp \in X^\sharp, x^\sharp \sqsubseteq^\sharp x^\sharp \nabla y^\sharp$
- $\forall x^\sharp, y^\sharp \in X^\sharp, y^\sharp \sqsubseteq^\sharp x^\sharp \nabla y^\sharp$
- For every chain $(x_i^\sharp)_{i\in\mathbb{N}}$, the increasing chain $(y_i^\sharp)_{i\in\mathbb{N}}$ defined by :

$$\begin{cases} y_0^\sharp = & x_0^\sharp \\ y_i^\sharp = & y_{i-1}^\sharp \nabla x_i^\sharp, \forall i > 0 \end{cases}$$

  is stable after a finite time, i.e. $\exists k \in \mathbb{N}, \forall i > k, y_i^\sharp = y_k^\sharp$

In other words, if we want to compute the limit of a sequence $(x_i^\sharp)_{i\in\mathbb{N}}$ that does not stabilize after a finite number of steps, one can use a widening operator to compute the limit of another sequence $(y_i^\sharp)_{i\in\mathbb{N}}$ that converges in finite time to an overapproximation of the limit of $(x_i^\sharp)_{i\in\mathbb{N}}$.

If the abstract domain does not satisfy the ascending chain condition, we thus can compute a fixpoint of $\Phi^\sharp$ in finite time using widening:

$$\begin{cases} y_0^\sharp = & \bot \\ y_i^\sharp = & y_{i-1}^\sharp \nabla \Phi^\sharp(y_{i-1}^\sharp), \forall i > 0 \end{cases}$$

We then obtain after some iterations a post-fixpoint of $\Phi^\sharp$. Indeed, if we note $\tilde{y}^\sharp$ the limit of the sequence, we have $\tilde{y}^\sharp = \tilde{y}^\sharp \nabla \Phi^\sharp(\tilde{y}^\sharp)$, and the properties on the widening operator imply that $\Phi^\sharp(\tilde{y}^\sharp) \sqsubseteq^\sharp \tilde{y}^\sharp$. The widening operator is needed to ensure termination, at the expense of precision: for instance, one could define the widening operator to be $\forall x^\sharp, y^\sharp \in X^\sharp, x^\sharp \nabla y^\sharp = \top$, which is correct but extremely imprecise. In the case the abstract domain satisfies the ascending chain condition, a widening can still be used to trade precision for efficiency, since a widening can lead to a postfixpoint in a few steps while classical Kleene iterations may converge after a long time.

However, once we have a postfixpoint of $\Phi^\sharp$, assuming $\Phi^\sharp$ is monotonic, it is possible to continue applying the $\Phi^\sharp$ operator on it and still obtain a (post)fixpoint which is either equal or smaller than the previous one:

$$\begin{cases} z_0^\sharp = & \tilde{y}^\sharp \\ z_i^\sharp = & \Phi^\sharp(z_{i-1}^\sharp), \forall i > 0 \end{cases}$$

The sequence $(z_i^\sharp)_{i\in\mathbb{N}}$ is decreasing, and each element $z_i^\sharp$ is a post-fixpoint of $\Phi^\sharp$. Similarly to the ascending sequence, the decreasing sequence may not stabilize to a fixpoint of $\Phi^\sharp$. In practice, this decreasing sequence, also called *narrowing sequence*, reaches a fixpoint in a very few number of iterations, otherwise the sequence is stopped after a given number of steps.

Finally, the limit $\tilde{z}^\sharp$ is a sound approximation of the set of reachable states: $\mathrm{lfp}(\Phi) \sqsubseteq \gamma(\tilde{y}^\sharp)$. Figure 2.3 illustrates the sequences $(x_i^\sharp)_i$, $(y_i^\sharp)_i$ and $(z_i^\sharp)_i$ previously defined.

2.1.5 Kleene Iteration Strategies

In practice, instead of considering the set of reachable states for the entire program, it is possible to decompose it as the union of the reachable states at each program location (also called program point). Typically, there is a program location before and after each atomic instruction of the program. If the program is seen as a control-flow graph, the set of program locations is the set of nodes in the graph. The set of reachable states at a given program point depends on the reachable states of its predecessors. The operators Φ and $\Phi^\sharp$ can be respectively decomposed into $\Phi_1, \Phi_2, \ldots, \Phi_n$ and $\Phi_1^\sharp, \Phi_2^\sharp, \ldots, \Phi_n^\sharp$, where n is the number of program locations, with each $\Phi_i^\sharp$ being a sound abstraction of Φ_i. In this case, $\mathrm{lfp}(\Phi^\sharp)$ is the least solution of this system of equations:

$$\begin{cases} x_1^\sharp &= \Phi_1^\sharp(x_1^\sharp, \ldots, x_n^\sharp) \\ x_2^\sharp &= \Phi_2^\sharp(x_1^\sharp, \ldots, x_n^\sharp) \\ &\vdots \\ x_n^\sharp &= \Phi_n^\sharp(x_1^\sharp, \ldots, x_n^\sharp) \end{cases} \tag{2.1}$$

Similarly to subsection 2.1.4, Kleene iterations over this system of equations are not guaranteed to terminate, and a widening operator has to be used during the ascending sequence:

$$\begin{cases} x_{1,i+1}^\sharp &= x_{1,i}^\sharp \nabla \Phi_1^\sharp(x_{1,i}^\sharp, \ldots, x_{n,i}^\sharp) \\ x_{2,i+1}^\sharp &= x_{2,i}^\sharp \nabla \Phi_2^\sharp(x_{1,i}^\sharp, \ldots, x_{n,i}^\sharp) \\ &\vdots \\ x_{n,i+1}^\sharp &= x_{n,i}^\sharp \nabla \Phi_n^\sharp(x_{1,i}^\sharp, \ldots, x_{n,i}^\sharp) \end{cases}$$

In our notations, the components $\Phi_i^\sharp$ have n parameters, but in practice their value only depend on a small subset of them. Using a control-flow graph representation of the abstract

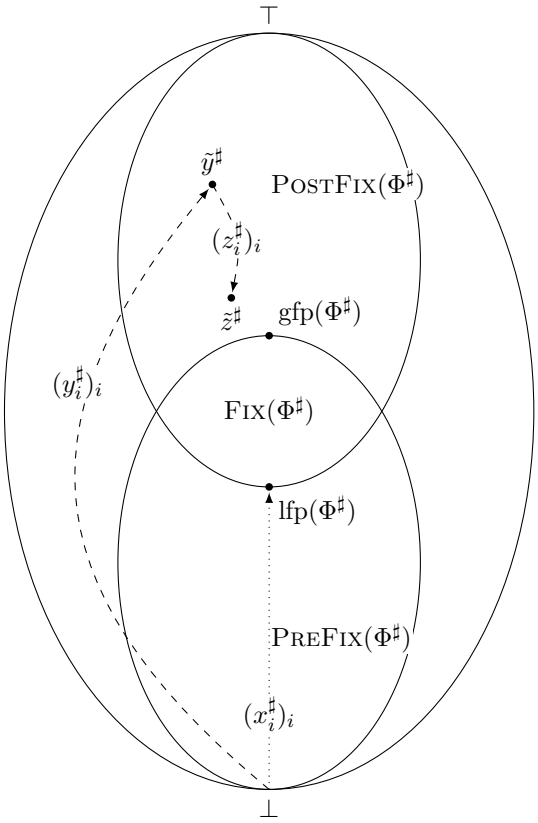


Figure 2.3: Illustration of the ascending and descending sequences in the abstract lattice. Fix($\Phi^\sharp$),PreFix($\Phi^\sharp$) and PostFix($\Phi^\sharp$) are respectively the sets of fixpoints, prefixpoints and postfixpoints of the operator $\Phi^\sharp$. $(x_i^\sharp)_i$ is the ascending sequence without widening, which is often infinite. $(y_i^\sharp)_i$ is the ascending sequence with widening. $(z_i^\sharp)_i$ is the descending sequence. Note that in practice, the decreasing sequence generally converges to an element $\tilde{z}^\sharp \in$ Fix($\Phi^\sharp$).

semantics of the program, the value $x^\sharp_{k,i+1}$ related to the node k only depends on the values of the $x^\sharp_{k',i+1}$, for which there exist a transition (k',k) in the graph. Applying a widening operator at each program location may degrade the precision of the analysis dramatically. [CC92] suggests applying widenings only at a subset P_W of these control points, and just use the least upper bound operator for the control points that are not in P_W. Breaking all cycles in the graph is sufficient for the analysis to terminate:

Theorem 1. Let $G = (N, I, T, X^\sharp)$ be the control flow-graph representing the abstract semantics of the program. Let P_W be a subset of N, such that the new graph G' obtained after removing from G the nodes in P_W has no cycle.

Then, the sequence of vectors $(x^\sharp_{1,i}, \ldots, x^\sharp_{n,i})_{i\in\mathbb{N}}$, converges in finite time to an overapproximation of the least solution of the equation system 2.1, where the $x^\sharp_{k,i}$ are defined by:

$$x^\sharp_{k,i+1} = \begin{cases} x^\sharp_{k,i} \nabla \Phi^\sharp_k(x^\sharp_{1,i}, \ldots, x^\sharp_{n,i}) & \text{if } k \in P_W \\ x^\sharp_{k,i} \sqcup^\sharp \Phi^\sharp_k(x^\sharp_{1,i}, \ldots, x^\sharp_{n,i}) & \text{otherwise} \end{cases}$$

Example. In our example of Figure 2.2, we attach to each program state $p_i, i \in \{0,1,2,3\}$ an abstract value $x^\sharp_i \in \mathcal{I}$. This will allow to compute for each program point an interval that contains all possible values for the variable `x`. We will use the abstract transformer $\Phi^\sharp : \mathcal{I}^4 \to \mathcal{I}^4$ defined as follows:

$$\Phi^\sharp \begin{pmatrix} x^\sharp_0 \\ x^\sharp_1 \\ x^\sharp_2 \\ x^\sharp_3 \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} \Phi^\sharp_0() \\ \Phi^\sharp_1(x^\sharp_0, x^\sharp_2) \\ \Phi^\sharp_2(x^\sharp_1) \\ \Phi^\sharp_3(x^\sharp_1) \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} [0,0] \\ x^\sharp_0 \sqcup \text{incr}^\sharp(x^\sharp_2) \\ x^\sharp_1 \sqcap (-\infty, 99] \\ x^\sharp_1 \sqcap [100, +\infty) \end{pmatrix}$$

with $\text{incr}^\sharp$:$\mathcal{I} \to \mathcal{I}$ being a sound abstraction of incr:

$$[x, y] \mapsto [x + 2, y + 2]$$

Since the abstract domain of intervals $\mathcal{I}$ does not satisfy the ascending chain condition – for instance, the sequence $([0, i])_{i>0}$ is strictly increasing – we have to define a widening operator for $\mathcal{I}$: $\forall [x, y], [x', y'] \in \mathcal{I}, [x^\sqcup, y^\sqcup] \stackrel{\text{def}}{=} [x, y] \sqcup [x', y']$,

$$[x, y] \nabla [x', y'] \stackrel{\text{def}}{=} \begin{cases} [x, y] & \text{if } x = x^\sqcup \wedge y = y^\sqcup \\ [x, +\infty) & \text{if } x = x^\sqcup \wedge y < y^\sqcup \\ (-\infty, y] & \text{if } x < x^\sqcup \wedge y = y^\sqcup \\ (-\infty, +\infty) & \text{if } x < x^\sqcup \wedge y < y^\sqcup \end{cases}$$

In the control-flow graph of the program, the control point p_1 and p_2 are forming a cycle (in other words, $x^\sharp_1$ and $x^\sharp_2$ are mutually dependent in the iteration process). We can define $P_W = \{p_1\}$ to be the set of widening points, which disconnects all cycles in the graph.

We compute iteratively the ascending sequence defined as follows:

$$\begin{cases} y^\sharp_{0,0} = \bot \\ y^\sharp_{1,0} = \bot \\ y^\sharp_{2,0} = \bot \\ y^\sharp_{3,0} = \bot \end{cases} \text{and } \forall i \in \mathbb{N}, \begin{cases} y^\sharp_{0,i+1} = y^\sharp_{0,i} \sqcup [0,0] \\ y^\sharp_{1,i+1} = y^\sharp_{1,i} \nabla (y^\sharp_{0,i} \sqcup \text{incr}^\sharp(x^\sharp_{2,i})) \\ y^\sharp_{2,i+1} = y^\sharp_{2,i} \sqcup (y^\sharp_{1,i} \sqcap (-\infty, 99]) \\ y^\sharp_{3,i+1} = y^\sharp_{3,i} \sqcup (y^\sharp_{1,i} \sqcap [100, +\infty)) \end{cases}$$

$$\begin{matrix} \bot \\ \bot \\ \bot \\ \bot \end{matrix} \to \begin{matrix} [0,0] \\ \bot \\ \bot \\ \bot \end{matrix} \to \begin{matrix} [0,0] \\ [0,0] \\ \bot \\ \bot \end{matrix} \to \begin{matrix} [0,0] \\ [0,0] \\ [0,0] \\ \bot \end{matrix} \to \begin{matrix} [0,0] \\ [0,+\infty) \\ [0,0] \\ \bot \end{matrix} \to \begin{matrix} [0,0] \\ [0,+\infty) \\ [0,99] \\ [100,+\infty) \end{matrix}$$

Once the sequence has stabilized, one can compute a decreasing sequence:

$$\begin{cases} z^\sharp_{0,0} &= [0,0] \\ z^\sharp_{1,0} &= [0,+\infty) \\ z^\sharp_{2,0} &= [0,99] \\ z^\sharp_{3,0} &= [100,+\infty) \end{cases} \text{ and } \forall i \in \mathbb{N}, \begin{cases} z^\sharp_{0,i+1} &= [0,0] \\ z^\sharp_{1,i+1} &= z^\sharp_{0,i} \sqcup \text{incr}^\sharp(z^\sharp_{2,i})) \\ z^\sharp_{2,i+1} &= z^\sharp_{1,i} \sqcap (-\infty, 99]) \\ z^\sharp_{3,i+1} &= z^\sharp_{1,i} \sqcap [100,+\infty)) \end{cases}$$

$$\begin{array}{l} [0,0] \\ [0,+\infty) \\ [0,99] \\ [100,+\infty) \end{array} \rightarrow \begin{array}{l} [0,0] \\ [0,101] \\ [0,99] \\ [100,+\infty) \end{array} \rightarrow \begin{array}{l} [0,0] \\ [0,101] \\ [0,99] \\ [100,101] \end{array}$$

We have reached a fixpoint of $\Phi^\sharp$ which is a overapproximation of the set of reachable states: at p_0, $\mathtt{x} = 0$, at p_1, $\mathtt{x} \in [0,101]$, at p_2, $\mathtt{x} \in [0,99]$ and at p_3, $\mathtt{x} \in [100,101]$. Note that using the abstract domain $\mathcal{I}$, a better invariant at p_1 would have been $\mathtt{x} \in [0,100]$. Even though the decreasing sequence improves the precision of the result a lot, it does not reach the least fixpoint of $\Phi^\sharp$.

Chaotic Iterations

For a given i, the $(x^\sharp_{k,i})_k$ do not depend from each other, but only from the values obtained at the $(i-1)$-th iteration. However, for a faster convergence and better precision, one could reuse the value of an already computed $x^\sharp_{k,i}$ at the place of $x^\sharp_{k,i-1}$ each time it occurs, since the $\Phi^\sharp_k$ operators are monotonic and $x^\sharp_{k,i} \sqsupseteq^\sharp x^\sharp_{k,i-1}$. In other words, one could start with a vector $(x^\sharp_1, \ldots, x^\sharp_n) \stackrel{\text{def}}{=} (x^\sharp_{1,0}, \ldots, x^\sharp_{n,0})$, define an operator Update: $X^\sharp \longrightarrow X^\sharp$ as follows:

$$\text{Update}(x^\sharp_k) \stackrel{\text{def}}{=} \begin{cases} x^\sharp_k \nabla \Phi^\sharp_k(x^\sharp_1, \ldots, x^\sharp_n) & \text{if } k \in P_W \\ x^\sharp_k \sqcup^\sharp \Phi^\sharp_k(x^\sharp_1, \ldots, x^\sharp_n) & \text{otherwise} \end{cases}$$

and successively apply Update on the $x^\sharp_k$'s, assuming every k is chosen infinitely often.

Algorithm 1 describes a classical algorithm for computing the chaotic iterations until a postfixpoint has been reached. Each abstract values $x^\sharp_p, p \in N$ is initialized to $\bot$ and updated iteratively. A set Active of active nodes is initialized to the initial states. It is used for storing a working list of states p for which $x^\sharp_p$ has to be updated. When an abstract value $x^\sharp_p$ is updated, then every $x^\sharp_q$ such that q is a successor of p in the control-flow graph, i.e. $(p,q) \in T$, has to be recomputed.

In this algorithm, line 6 chooses a node among the active nodes. One can choose any node and obtain a correct result at the end, but experience shows that the precision of the resulting abstract values as well as the number of iterations before stabilization highly depends on this choice. The problem of choosing the state, also called *iteration strategy*, has been extensively studied in the literature, for instance in [Bou92]. Most of these strategies are based on the decomposition of the control-flow graph into its strongly connected components (also noted *SCC*), for instance:

- First stabilize the first SCC before proceeding to the next, or
- First stabilize innermost loops.

Algorithm 1 Standard Abstract Interpretation with chaotic iterations

```
1: function ChaoticIterations( Control-flow graph G = (N, I, T))
2:     Active ← I                                          ▷ Set of active nodes
3:     Choose a set P_W ⊆ N of widening nodes
4:     x_p^♯ ← ⊥ for each p ∈ N
5:     while Active ≠ ∅ do
6:         Choose p ∈ Active
7:         Active ← Active \ {p}
8:         x_old^♯ ← x_p^♯
9:         if p ∈ P_W then
10:            x_p^♯ ← x_p^♯ ∇ Φ_k^♯(x_1^♯, ..., x_n^♯)
11:        else
12:            x_p^♯ ← x_p^♯ ⊔^♯ Φ_k^♯(x_1^♯, ..., x_n^♯)
13:        end if
14:        if x_old^♯ ⊏^♯ x_p^♯ then
15:            Active ← Active ∪ Successors(p)
16:        end if
17:    end while
18: end function
```

2.1.6 Galois Connection based Abstract Interpretation

In this chapter, we described the concretization-based abstract interpretation framework, where the concrete and the abstract domains are connected by a concretization function γ. However, abstract interpretation can also be introduced with other formalisms, for instance based on Galois connections [CC77]. In this formalism, the concrete domain $(X, \sqsubseteq)$ and the abstract domain $(X^\sharp, \sqsubseteq^\sharp)$ are connected by two function α and γ, such that:

- $\alpha : X \to X^\sharp$ is monotonic,
- $\gamma : X^\sharp \to X$ is monotonic,
- $\forall x \in X, \forall x^\sharp \in X^\sharp, \alpha(x) \sqsubseteq^\sharp x^\sharp \Leftrightarrow x \sqsubseteq \gamma(x^\sharp)$

Function α is called the *abstraction function*, and γ is the *concretization function*. A concrete value $x \in X$ can be abstracted by $\alpha(x)$(which is the best abstraction for x), whose concretization includes x: $\gamma \circ \alpha(x) \sqsupseteq x$. Alternatively, an abstract value $x^\sharp$ may represent a concrete value $\gamma(x^\sharp)$, whose abstraction is included in $x^\sharp$: $\alpha \circ \gamma(x^\sharp) \sqsubseteq^\sharp x^\sharp$.

If $\Phi : X \to X$ is an operator over the concrete domain, it can be abstracted by the operator $\Phi^\sharp : X^\sharp \to X^\sharp$ such that $\forall x^\sharp \in X^\sharp, \Phi^\sharp(x^\sharp) \stackrel{\text{def}}{=} \alpha \circ \Phi \circ \gamma(x^\sharp)$. Defining an abstract operator is then very simple using this formalism. However, in practice, there exist some abstract domains for which we cannot define an abstraction function α: on these domains, some elements $x \in X$ do not have a best abstraction. The classical example is the abstract domain of convex polyhedra, for which there does not exist a best abstraction of a circle.

2.1.7 Usual Numerical Abstract Domains

In this thesis, we make use of numerical abstract domains. In other words, we are interested in properties involving only the numerical variables of the program. The literature is full of useful abstract domains for such numerical properties, and we only mention here the one we used in our implementation and experiments.

- *Convex Polyhedra* [CH78, Hal79] discover invariants as conjunctions of linear constraints of the form $\sum_i c_i x_i \leq C$, where the x_i are the numerical variables of the program, and C and the C_i are constants.
- *Template Polyhedra* are convex polyhedra for which the user restricts the directions of the faces, i.e. the c_i's constants are fixed during the analysis.
- *Octagons* [Min04] are template polyhedra where the linear constraints are of the form $\pm x_i \pm x_j \leq C$.
- *Intervals*, or *boxes*, is the class of template polyhedra with constraints of the form $\pm x_i \leq C$.

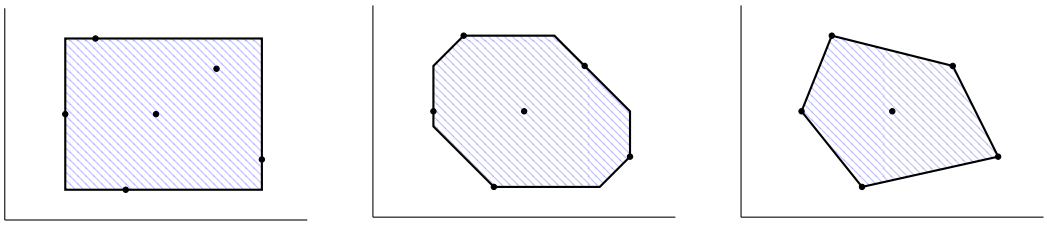


Figure 2.4: Usual numerical abstract domains. The set of points in black is abstracted by an overapproximating abstract value (in shadings).

2.2 Some Relevant Model Checking Approaches

2.2.1 Bounded Model Checking

Basics of Satisfiability Solving Modulo Theories

The Boolean satisfiability decision problem (usually called SAT) is the canonical NP-complete problem: given a propositional formula (e.g. $(a \vee \neg b) \wedge (\neg a \vee b \vee \neg c)$), decide whether it is satisfiable. If so, output a satisfying assignment, i.e. a valuation for each variables such that the formula is true. Despite an exponential worst-case complexity, very efficient algorithm have been developed and current Boolean satisfiability solvers solve many useful SAT problems of industrial size. The majority of the state-of-the art solvers are based on the DPLL algorithm. [KS08, BHvMW09] give a detailed description of this algorithm.

SAT was extended to support atoms from a theory $\mathcal{T}$ in addition to propositional literals. The resulting decision problem is called *satisfiability modulo theory* (SMT). For instance, the theories of linear integer arithmetic (LIA) and linear rational arithmetic (LRA) have atoms of the form $a_1 x_1 + \cdots + a_n x_n \bowtie C$ where $a_1, \ldots, a_n, C$ are integer constants, $x_1, \ldots, x_n$ are variables (interpreted over $\mathbb{Z}$ for LIA and $\mathbb{R}$ or $\mathbb{Q}$ for LRA), and $\bowtie$ is a comparison operator $=, \neq, <, \leq, >, \geq$. LIA and LRA is highly used in the field of software verification, as well as other theories like bitvectors, arrays or uninterpreted functions.

Satisfiability modulo theories like LIA or LRA is also NP-complete. However, tools based on the DPLL($\mathcal{T}$) algorithm [KS08, BHvMW09, Tin02] solve many useful SMT problems in practice. All these tools provide a *satisfying assignment* if the problem is satisfiable. Here, we briefly explain the main principle of the DPLL($\mathcal{T}$) algorithm, which is required for understanding chapter 6, and how it can be useful for software verification.

DPLL We briefly summarize the principle of the DPLL algorithm for deciding the SAT problem. The procedure consists in the alternation two phases:

- *Decision*, that picks one Boolean variable heuristically, and decides its value to be either *true* or *false*.
- *Boolean Constraint Propagation*, that propagates the consequences of the previous decision on the other variables. It happens regularly that a conflict is encountered because of a wrong choice in the decision phase. In that case, the algorithm computes the reason of this conflict (bad choice in some decision phase) and *backtracks*, i.e. unsets the variables that have been assigned between the bad decision and the current assignment, and restart with another decision.

The algorithm terminates in the case there is no more decision step to apply (i.e. the set of variables is fully assigned) and there is no conflict – the answer is SAT –, or when the conflict is not just due to a bad decision – the answer is UNSAT –. Again, we refer the reader to [KS08, BHvMW09] for more details.

State of the art solvers implement an improvement over the DPLL algorithm, called CDCL, which provides more efficient and non-chronological backtracking features.

DPLL($\mathcal{T}$) Suppose we have an SMT formula that contains atoms of a given theory $\mathcal{T}$. $\mathcal{T}$ should be a first-order quantifier-free theory, that admits a decision procedure for conjunctions of atoms in $\mathcal{T}$, noted $\mathcal{T}$-solver. For instance, the *general simplex* method is a $\mathcal{T}$-solver for LRA, as well as *Branch-and-Bound* for LIA [KS08]: they can decide whether a conjunction of linear inequalities are satisfiable or not, and if so, can give a possible assignment for the integer/real variables. The main principle of DPLL($\mathcal{T}$) is to combine a SAT solver with a $\mathcal{T}$-solver for the chosen theory. In this short explanation of DPLL($\mathcal{T}$), we restrict ourselves to a single theory, but it is possible to combine theories using the Nelson-Oppen method.

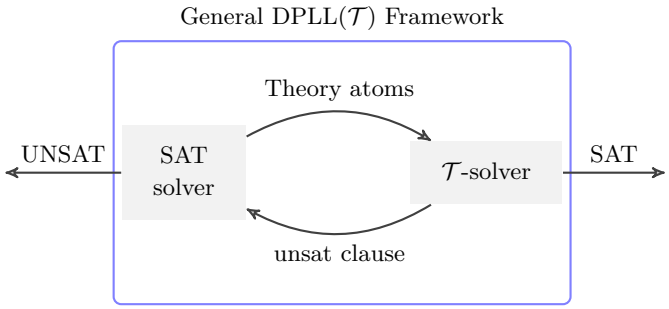


Figure 2.5: DPLL($\mathcal{T}$) framework

The DPLL SAT solver and $\mathcal{T}$-solver interact with each other by exchanging information (see Figure 2.5): the SAT solver abstracts the SMT formula into a SAT formula with the same Boolean structure, and provide to the $\mathcal{T}$-solver satisfying assignments for the Boolean abstraction. The $\mathcal{T}$-solver checks whether this assignment is satisfiable in the theory. Similarly to *Boolean Constraint Propagation*, *Theory Propagation* propagates the effects of a *decision* that are due to the theory in use. If the formula is not satisfiable in the theory, it computes an *unsat clause*, i.e. a subset of the assignment that is unsatisfiable, to prune the search space of the SAT solver. The unsat clause is a key feature of the algorithm, since it is turned back to the SAT solver as a conjunction of Boolean assignments that are together unsatisfiable, and thus

provides a second reason for backtracking (recall that the first reason is the conflicts obtained by Boolean Constraint Propagation and Theory Propagation). For this reason, it is essential that "good" (or "short") unsat clauses can be obtained for the backtracking to be efficient.

Theory propagation does not need the SAT solver to provide a full assignment for all the atoms, and is practically applied at the same time and in conjunction with Boolean Constraint Propagation in order to detect conflicts and backtrack as soon as possible.

Note that state of the art solvers are based on CDCL instead of DPLL. The combination of a CDCL SAT solver with the theory solver is however similar.

Principle of Bounded Model Checking

SMT solving has several applications to the field of software verification, in particular in *Bounded Model Checking*: it is possible to encode the semantics of the traces of bounded length, or at least an overapproximation thereof, into a logical formula that can be checked for satisfiability by an SMT solver. It can be used for finding program traces of a given length that violate some property. Given a set of initial states $I(x_0)$, a transition relation $\tau(x_i, x_{i+1})$ and a property $P(x)$, this formula has the following form:

$$I(x_0) \wedge \bigwedge_{0 \leq i \leq k-1} \tau(x_i, x_{i+1}) \wedge \bigvee_{0 \leq i \leq k} \neg P(x_i)$$

Example 2. Suppose we have the simple C program from Figure 2.6 (left), for which we want to find an error trace that violates the assertion. undet() is a function that returns a nondeterministic Boolean value.

```
unsigned x = 0;
while (undet()) {
  if (undet()) x+=2;
}
assert(x < 50);
```

$$\begin{aligned}
&x_I = 0 \\
&\wedge((x_1 = x_I + 2) \vee (x_1 = x_I)) \\
&\wedge((x_2 = x_1 + 2) \vee (x_2 = x_1)) \\
&\wedge \cdots \\
&\wedge((x_k = x_{k-1} + 2) \vee (x_k = x_{k-1})) \\
&\wedge(x_F = x_k) \wedge (x_F \geq 50)
\end{aligned}$$

Figure 2.6: Bounded model checking of a simple program

Program traces of length k (i.e. with exactly k iterations of the loop) that violate the final assertion are the models of the SMT formula from Figure 2.6 (right): x_I is the initial value for x, x_F is its final value, and x_i is its value after i loop iterations. If the formula is satisfiable, it means the error state is reachable in a program trace of length k, and the successive values for x during the program can be precisely extracted from the model returned by the SMT solver. Similarly, it is possible to encode into SMT the program traces of length *at most* k. In this simple example, the smallest k for which the formula is satisfiable is for $k = 25$, with $x_I = 0, x_1 = 2, x_2 = 4, \cdots, x_{25} = x_F = 50$, which corresponds to the traces that always enters the *then* branch of the *if* statement.

Bounded Model Checking is good for finding bugs and returning error traces. However, since some programs may have traces of unbounded length (or of exponential size), this approach alone can not prove properties to be invariant for any traces, as abstract interpretation does. In this thesis, we make use of bounded model checking in the context of loop free program

portions in section 4.2. The way of encoding the program traces into SMT is fully described later in subsection 4.2.1.

2.2.2 Predicate Abstraction & CEGAR

Predicate Abstraction [BMMR01, CKSY04] is a popular method for abstracting a program into a finite state system with respect to a finite set of predicates, that safely overapproximates the set of traces in the original program. It has already been successfully applied in industry with tools like the SLAM model checker [BMMR01]. The idea is to remove data variables from the model and only track well chosen predicates on these variables (for instance, only track the predicates $x < 0$, $x > 0$ and $x = 0$ for a variable x). The construction of the finite state system can be seen as an abstract interpretation of the program, since a lattice can be build from the set of predicates. The resulting system is then small enough to be analyzed by a model checker that checks whether the property holds. If not, it may be for two reasons: either there is indeed a bug, or the counterexample is spurious. In the latter case, the *Counter Example Guided Abstraction Refinement* [CGJ$^+$00] (CEGAR) loop refines the abstraction by introducing new predicates, in order to eliminate the spurious trace, and restarts the analysis on the refined abstraction. The newly introduced predicate can be for instance derived from *Craig Interpolants* [McM03, McM05, McM06]:

Definition 10 (Craig Interpolant)**.** A *Craig Interpolant* for an unsatisfiable conjunction $F_1 \wedge F_2$ is a formula I such that:

- $F \Rightarrow I$
- $I \wedge F_2$ is unsatisfiable
- $Vars(I) \subseteq Vars(F_1) \cap Vars(F_2)$, where $Vars(F)$ denotes the free variables in a formula F

Intuitively, the interpolant obtained from a spurious trace is a "small" reason why the trace is unfeasible. The effectiveness of this approach strongly relies on the quality of the interpolants used for producing new predicates. These predicates indeed should be sufficiently general to prune many spurious counterexamples at once. For this reason, computing good interpolants is an active research topic [AM13].

2.2.3 k-Induction

Properties on programs can also be proved using k-induction [SSS00]: one attempts to prove the property holds in a given state, supposing it holds for the k previous states. The induction proof requires two steps:

- the *base* case: prove that the property holds for the k first states. This is done by bounded model checking:
$$I(x_0) \wedge \bigwedge_{0 \leq i \leq k-1} \tau(x_i, x_{i+1}) \wedge \bigvee_{0 \leq i \leq k} \neg P(x_i)$$
- the *induction* phase: given the property holds for k successive states, proves it holds for the next state:
$$\bigwedge_{0 \leq i \leq k} [P(x_i) \wedge \tau(x_i, x_{i+1})] \wedge \neg P(x_{k+1})$$

The algorithm consists in initializing k to 0, and iteratively incrementing it when the induction phase fails. If the base case fails, it means that the property does not hold and a counterexample of length k is returned. The Kind model checker [HT08][1] has shown this approach to be powerful in practice.

This approach can be strengthened by using external invariants: instead of proving that the property is inductive – which is not always the case –, one can prove it is *inductive relative to* another property I which is known to be invariant. The induction phase becomes:

$$\bigwedge_{0 \leq i \leq k} [I(x_i) \wedge P(x_i) \wedge \tau(x_i, x_{i+1})] \wedge \neg P(x_{k+1})$$

In this thesis, we discover program invariants using advanced abstract interpretation based techniques. These invariants can then be fed to a k-induction based algorithm and thus provide an efficient combination of the two approaches.

2.2.4 IC3

IC3 [Bra11, Bra12] – or PDR, for *Property Directed Reachability* [EMB11, HB12]) – is a state of the art procedure for proving properties on finite state systems, that has recently been extended to infinite state systems through the use of SMT [CGMT14]. We briefly mention it here since it demonstrated to be very efficient in practice.

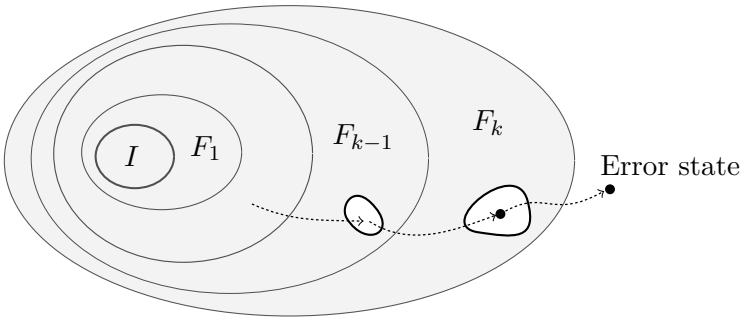


Figure 2.7

This procedure tries to compute a sufficiently precise inductive invariant F to prove a property P. It computes a sequence of sets $F_0 = I, F_1, \ldots, F_k$, where each F_i (called *frame*) is an overapproximation of the set of states reachable in at most i steps. The goal is to find some F_k that is inductive and proves the property:

$$I(x_0) \Rightarrow F_k(x_0) \tag{2.2}$$

$$F_k(x_i) \wedge \tau(x_i, x_{i+1}) \Rightarrow F_k(x_{i+1}) \tag{2.3}$$

$$F_k(x_i) \Rightarrow P(x_i) \tag{2.4}$$

For a given working frame F_k, the procedure tries to find whether there exist an error state, i.e. in $\neg P$, reachable in one step from a state in F_k. If so, F_k, $F_{k-1}, \ldots, F_1$ are refined by removing from them a set of states, called *lemmas*, that can lead to that particular error state (see Figure 2.7). In the case of infinite state systems, the frames are SMT formulas and one removes infinite sets at once through interpolation [CGMT14]. This phase continues until F_k

[1] `http://clc.cs.uiowa.edu/Kind/`

cannot reach $\neg P$ in one step. Once it is the case, a new frame F_{k+1} is created and initialized to P, and directly refined with the already discovered lemmas that are inductive. The program terminates if it turns out that F_k and F_{k+1} are equal, otherwise k has to be increased. If the property P does not hold, a counterexample of length $k+1$ can be returned.

Chapter 3

When Abstract Interpretation lacks Precision

As mentioned in section 2.1, abstract interpretation computes an over-approximation of the set of the reachable program states. A correct over-approximation would be "Every state is reachable". However, such approximation has no interest since it does not allow to prove the desired properties. Then, precision is important in order to prevent having *false-alarms*. In this chapter, we detail the various sources of imprecision in abstract interpretation, as well as the state-of-the-art techniques for obtaining precise invariants.

3.1 Sources of Imprecision

In general, abstract interpretation based on Kleene iterations with widening suffers from various loss of precision during the fixpoint computation.

1. The choice of the **abstract domain** is crucial and is guided by the properties the user wants to prove. Typically, for a given property $P(x)$ (where $x \in \mathsf{States}$) on the set of reachable states, there should exist an abstract transformer $\Phi^\sharp$ whose least fixpoint $\mathrm{lfp}(\Phi^\sharp)$ is a precise-enough approximation of $\mathrm{lfp}(\Phi)$, such that $\forall x \in \gamma(\mathrm{lfp}(\Phi^\sharp)), P(x)$.

2. Even if the abstract domain is theoretically sufficiently expressive to prove a given property $P(x)$, i.e $\forall x \in \gamma(\mathrm{lfp}(\Phi^\sharp)), P(x)$, the ascending and descending sequences may not converge to a sufficiently precise overapproximation of $\mathrm{lfp}(\Phi^\sharp)$. This is due to the use of the **widening operator**, that trades precision for efficiency and for guaranteeing termination. In addition, most of the widening operators of standard abstract domains do not have the good property of being *monotonic*: applying a widening on a more precise abstract value does not necessarily gives a smaller result; for instance, using the intervals abstract domain, $[0, 10] \nabla [0, 12] = [0, +\infty)$, and $[1, 10] \nabla [0, 12] = (-\infty, +\infty)$, while $[0, 10] \sqsubset [0, 10]$. This leads sometimes to surprising and disappointing results, where a supposedly more precise iteration strategy eventually gives worse results.

   The design of good widening operators for usual abstract domains has been studied in the literature, for instance for the domain of convex polyhedra [CH78, Hal79, BHRZ05]. In some cases, it is also possible to replace widening by abstract acceleration [GH06]. Recent work also propose alternative ways of computing the fixpoint over infinite height lattices without the need for a widening operator. These methods are called *policy iteration*. Section 3.2 gives an overview of several propositions for limiting the effects of widening in terms of loss of precision, that are relevant for the rest of this thesis.

3. The classical way of constructing the $\Phi^\sharp$ abstract transformer, as explained in subsection 2.1.5, is to define an operator $\Phi^\sharp_k$ for each control point p_k of the control-flow graph. For each control point that has several incoming transitions — which is in practice very often the case —, this operator is defined as the least upper bound of the abstract values coming from each transition. It results that least upper bound operations are applied very often during the analysis, and thus degrades precision. In section 3.3, we explain how it is possible to change the way of applying least upper bound operations to improve the precision of the final result.

3.2 Fighting bad Effects of Widenings

3.2.1 Improved Widening Operator

Intuitively, the principle of widening is to extrapolate the behavior of the first loop iteration to deduce an overapproximation of the reachable states after any number of iterations, supposing that each of them has a similar behavior. This is a very strong hypothesis, and is wrong in many cases, thus leading to a loss of precision. In the following, we briefly explain usual improvements of the widening operator: it is indeed possible to design better extrapolations, that are not just based on the effect of the first loop iteration.

Delayed widening

During the ascending sequence, each abstract values $x^\sharp_k$ is updated using the UPDATE function described in 2.1.5. This function uses the widening operator when the control point p_k is in the set P_W of widening points. However, for ensuring termination, the widening operator has to be used only infinitely often, but not necessarily at each updates of $x^\sharp_k$. For instance, we can use least upper bounds for the $n-1$ first updates of $x^\sharp_k$, and use a widening for the n-th update, where n is a user-defined parameter. In this case, one can attach to each control point $p_k \in P_W$ an integer variable $i_k \in \mathbb{N}$, and update the couple (x_k, i_k) with $(x_k, i_k) \nabla^n \Phi^\sharp_k(x^\sharp_1, \ldots, x^\sharp_n)$. ∇^n is called *delayed widening*.

Definition 11 (Delayed Widening)**.** Let $X^\sharp$ be the abstract domain and ∇ a widening operator over $X^\sharp$. Then, the operator $\nabla^n : (X^\sharp \times \mathbb{N}) \times X^\sharp \to (X^\sharp \times \mathbb{N})$ is defined by:

$$\forall x^\sharp_k, y^\sharp_k \in X^\sharp, \forall i_k \in \mathbb{N}, (x^\sharp_k, i_k) \nabla^n y^\sharp_k \stackrel{\text{def}}{=} \begin{cases} (x^\sharp_k \nabla y^\sharp_k, 0) & \text{if } i_k \geq n \\ (x^\sharp_k \sqcup^\sharp y^\sharp_k, i_k + 1) & \text{otherwise} \end{cases}$$

Figure 3.1 illustrates the use of the delayed widening with two-dimensional convex polyhedra. Intuitively, using the delayed widening ∇^n allows to extrapolate the behavior of the n last iterations instead of only the last one. It improves the precision of the analysis in practice, but the ascending sequence will require more iterations – the number of iterations for stabilizing a loop will be potentially multiplied by n –. A related idea would be to unroll the loop n times. The delayed widening approach avoids increasing the size of the graph while keeping some of the benefits of unrolling. Example 3 shows a simple example where unrolling gives better results.

Example 3 (Delayed Widening)**.** We consider the program **int** x = 10; **while**(x!=0) {x = −x;} . Then, with unrolling and the abstract domain of intervals, we can prove that the program

never terminates, since the loop header will be split into two nodes whose abstract values will be $x \in [-10, -10]$ and $x \in [10, 10]$. Without unrolling, we only obtain $x \in [-10, 10]$ at the loop header.

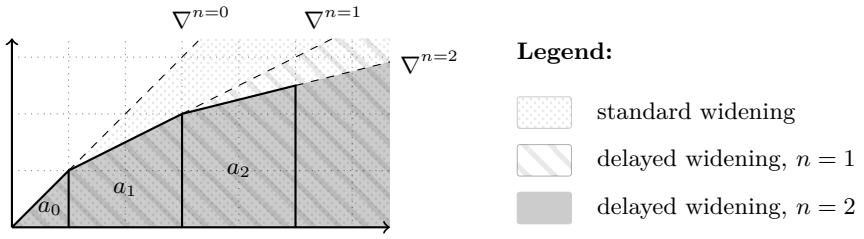


Figure 3.1: Delayed widening on convex polyhedra, with $n = 0$ (standard widening), 1 and 2. The abstract value $x^\sharp$ after k updates without widening is the union of the areas a_0 to a_{k-1}. The plain area is the abstract value after a delayed widening with $n = 2$. It is smaller than the hatched area, which is the abstract value obtained when $n = 1$. Widening without delay is the less precise.

Parma Widening

[BHRZ03, BHRZ05] proposed a general framework for the design of new widening operators from an existing one, that applies to numerical abstract domains. As for delayed widening 3.2.1, the idea is to use least upper bound operations instead of widenings in some cases, while keeping the guarantee of termination. In this approach, the choice of applying a widening or not is guided by some property on the two operands, called *limited growth*.

Definition 12 (Limited Growth)**.** Let $x_1^\sharp, x_2^\sharp \in X^\sharp$, where $X^\sharp$ is the abstract domain of convex polyhedra. Each $x_i^\sharp$ can be defined by a set of linear constraints C_i in a canonical form, or by a canonicalized system of generators (V_i, L_i, R_i). V_i, L_i and R_i are respectively the sets of vertices, lines and rays representing the polyhedron. The relation $\curvearrowright$, defined for $x_1^\sharp \sqsubseteq^\sharp x_2^\sharp$, is a partial order defined by:

$$x_1^\sharp \curvearrowright x_2^\sharp \overset{\text{def}}{\iff} \begin{cases} \dim(x_2^\sharp) > \dim(x_1^\sharp), \text{ or} \\ \operatorname{codim}(x_2^\sharp) > \operatorname{codim}(x_1^\sharp), \text{ or} \\ |C_1| > |C_2|\,, \text{ or} \\ |C_1| = |C_2| \text{ and } |V_1| > |V_2|\,, \text{ or} \\ |C_1| = |C_2| \text{ and } |V_1| = |V_2| \text{ and } \kappa(R_1) \gg \kappa(R_2) \end{cases}$$

where $|X|$ is the cardinal of a set X, and $\kappa(R_i)$ is the multiset of non null coordinates of the rays in R_i. $\gg$ is the classical order on multisets (if $\#(k, X)$ is the number of occurrence of k in X, $X_1 \gg X_2 \overset{\text{def}}{=} \exists j \in \mathbb{N}, \#(j, X_1) > \#(j, X_2) \wedge \forall k > j, \#(k, X_1) = \#(k, X_2))$.

Using this definition, it can be shown that there is no infinite sequence $(x_i^\sharp)_{i\in\mathbb{N}}$ for which $\forall i \in \mathbb{N}, x_i^\sharp \curvearrowright x_{i+1}^\sharp$. One reason is that the abstract domain $X^\sharp$ is finite-dimensional (typically, the number of dimension is bounded by the number of numerical variables in the program).

Definition 13 (Parma Widening)**.** If ∇ is a widening operator for the abstract domain $X^\sharp$, and $\sqcup^\sharp$ is the least upper bound, the operator ∇' defined by:

$$\forall x_1^\sharp, x_2^\sharp \in X^\sharp, x_1^\sharp \nabla' x_2^\sharp \overset{\text{def}}{=} \begin{cases} x_1^\sharp \sqcup^\sharp x_2^\sharp & \text{if } x_1^\sharp \curvearrowright x_2^\sharp \\ x_1^\sharp \nabla x_2^\sharp & \text{otherwise} \end{cases}$$

is a widening operator. Note that [BHRZ05] gives a slightly more complicated definition for ∇'.

In other words, the Parma widening uses the least upper bound instead of the classical widening operator whenever there is a geometric reason for the ascending sequence to be of finite length. In this way, we have $\forall x_1^\sharp, x_2^\sharp \in X^\sharp, x_1^\sharp \nabla' x_2^\sharp \sqsubseteq^\sharp x_1^\sharp \nabla' x_2^\sharp$. However, because of the non-monotonicity of the widening operator, the final result at the end of the ascending sequence is not necessarily more precise.

In this thesis, we provide some experimental results that illustrate the effectiveness of this new widening operator compared to the classical one (see subsection 4.3.2).

Since most of the numerical abstract domains are subsets of general convex polyhedra, a simplified version of Definition 12 could be used for defining the relation $\curvearrowright$.

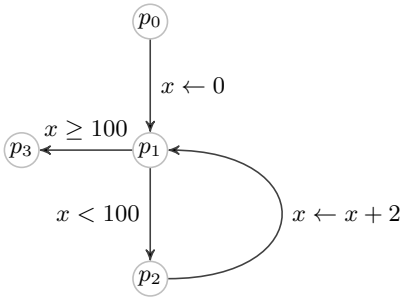
Widening with Threshold

Another usual improvement of the widening operator is the widening "up to" [Hal93, HPR97], also called widening with thresholds [BCC+03]. It consists in pre-computing a set $\mathcal{T}$ of constraints ($\mathcal{T} \subset X^\sharp$) that are likely to be invariants in a widening location (for instance, the negation of the exit condition of a loop). The widening "up to" is actually the application of the standard widening, whose result in intersected with every thresholds that are satisfied by both operands of the widening:

$$\forall x_1^\sharp, x_2^\sharp \in X^\sharp, x_1^\sharp \nabla_{\mathcal{T}} x_2^\sharp \stackrel{\text{def}}{=} (x_1^\sharp \nabla x_2^\sharp) \sqcap^\sharp \prod^\sharp \{t \in \mathcal{T}, t \sqsupseteq^\sharp x_1^\sharp \wedge t \sqsupseteq^\sharp x_2^\sharp\}$$

The precision of this widening highly depends on the way the set of thresholds is defined. In practice, a good heuristic is to propagate the constraints that are syntactically written as guards inside the loop and take the obtained constraints at the widening point as thresholds. In this way, the set of thresholds is different for each widening point. A generalized way of statically determining the threshold is described in [LCJG11]. Some other work dynamically construct the set $\mathcal{T}$ of thresholds using a counterexample guided refinement algorithm based on a forward and backward analysis [WYGI07].

Example 4 (Widening with thresholds)**.** We take the program from example 2.2 and use the intervals abstract domain. The propagation of the guard $x < 100$ to the head of the loop gives



$x < 102$. We thus define the set of thresholds $\mathcal{T} = \{x < 102\}$. The abstract value at point p_1 starts at $[0, 0]$. After one loop iteration, it is updated to $[0, 0] \nabla_{\mathcal{T}} [0, 2] = [0, 101]$ instead of $[0, +\infty]$ with the standard widening. The next iteration shows that $[0, 101]$ is an invariant, and the analysis terminates. In this case, the threshold helped discovering a precise invariant without the need of the descending sequence.

In many cases, the syntactically obtained thresholds and the descending sequence allow to recover almost the same precision, because the descending sequence has also the effect of propagating the guards to the widening point. They are however not equivalent, especially when a widening operation degrades the precision of the rest of the ascending sequence that can not be recovered by narrowings.

Abstract Acceleration

Abstract acceleration [GH06, FG10] is an alternative that computes the abstract transitive closure of certain classes of loops. Given a initial state x for the loop and a transition relation τ, the *transitive closure* of the loop is defined as $\tau^*(x) = \bigcup_{k\geq 0} \tau^k(x)$. Abstract acceleration computes an optimal approximation of τ^* in the abstract domain, in general convex polyhedra, without applying the classical Kleene iterations with widening. It is then locally more precise since it does not apply any extrapolation of the invariant candidates.

Example 5. The example code x = x0; y = y0; while (x <= 100) {x+=1; y+=2;} is a loop performing a translation of the vector of variables. The set of reachable program states can be succinctly characterized by the Presburger formula:

$$\exists k \geq 0, x = x0 + k \wedge y = y0 + 2k \wedge \forall k', (0 \leq k' \leq k) \Rightarrow x0 + k' \leq 100$$

Abstract acceleration was first restricted to loops performing translations and resets of variables. It has then be extended to other more general classes of linear loops [SJ11, JSS14].

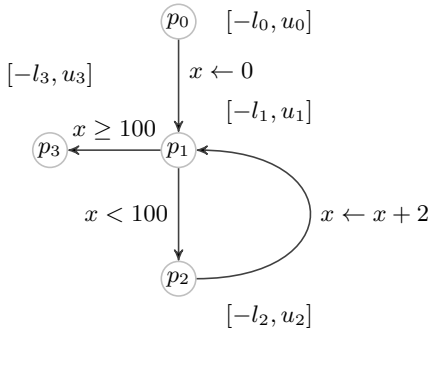
3.2.2 Policy Iteration

Policy (or *strategy*) *iteration* [CGG$^+$05, GGTZ07, GS07a, GS07b, GS11] is an alternative approach to solve the fixpoint equation that does not require any extrapolation operator like in Kleene iterations. The idea is to iteratively compute the least fixpoint of a sequence of simpler semantic equations $x = \Phi^{\sharp(i)}(x)$, such that the least fixpoint of the function $\Phi^\sharp$ is computed after a finite number of iterations. The sequence of operators $(\Phi^{\sharp(i)})_i$ is called *strategy*, and the operator $\Phi^{\sharp(i)}$ is chosen according to the computed fixpoint of $\Phi^{\sharp(i-1)}$. The fixpoint can be either approached from above or from below (max- or min-strategy iteration).

Policy iteration is currently restricted to *Template* abstract domains, i.e. a restriction of convex polyhedra for which the direction of the faces are given. This is the main drawback of the approach, since it is hard to know which directions are of interest for a particular program.

Example 6. We illustrate the method of max-strategy iteration with the example control-flow graph in Figure 3.2 and the intervals abstract domain. At each control point p_i, it is possible to characterize the interval $[-l_i, u_i]$ containing the possible values for x as a maximum involving the different incoming transitions. Then, one can chose for each *max* one of its operand and solve the resulting system using *linear programming* (LP).

We start with the initial *strategy* $l_0 = u_0 = \infty$ and $\forall i \neq 0, l_i = u_i = -\infty$ and check if it is an invariant by replacing the values in the equation system. We get $l_1 = u_1 = \max\{-\infty, 0\} = 0 \neq -\infty$. The idea is then to change the strategy by choosing $l_1 = \sup\{-x' / -l_0 \leq x \leq u_0 \wedge x' = 0\}$ and $u_1 = \sup\{x' / -l_0 \leq x \leq u_0 \wedge x' = 0\}$ — i.e. the operands of the max that give the correct result —, then solve the new system by LP. We keep updating the strategy until the resulting fixpoint is a fixpoint of the equation system. In the worst case, the sequence of strategies before convergence will enumerate the exponential number of possible assignments for the l_i's and u_i's.

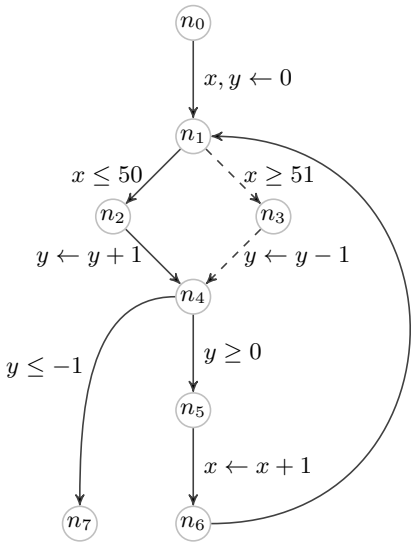


$$
\begin{aligned}
&l_0 = u_0 = \infty \\
&l_1 = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{-x'/-l_0 \leq x \leq u_0 \wedge x' = 0\}, \\ \sup\{-x'/-l_2 \leq x \leq u_2 \wedge x' = x + 2\} \end{array} \right. \\
&u_1 = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{x'/-l_0 \leq x \leq u_0 \wedge x' = 0\}, \\ \sup\{x'/-l_2 \leq x \leq u_2 \wedge x' = x + 2\} \end{array} \right. \\
&l_2 = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{-x/-l_1 \leq x \leq u_1 \wedge \wedge x < 100\} \end{array} \right. \\
&u_2 = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{x/-l_1 \leq x \leq u_1 \wedge \wedge x < 100\} \end{array} \right.
\end{aligned}
$$

Figure 3.2: Simple loop and its associate equation system

3.2.3 Guided Static Analysis

[GR07, GR06] proposed *Guided Static Analysis*, which provides an efficient framework for computing precise invariants for programs having several distinct phases with very different behaviors. The method is the following: during the ascending sequence, remember which transitions are feasible before any application of a widening operator. After applying a widening, the ascending sequence is restrained to the transitions that have been previously tagged as feasible (the formerly unfeasible transitions are ignored even if they became feasible after widening). The ascending sequence stabilizes on a invariant for the considered subprogram, and a descending sequence is computed to improve precision. Then, we restart an ascending sequence, taking into account feasible transitions that were previously ignored. Example 7 illustrates the technique on a classic example where a loop has two distinct phases. The intuition behind this technique is to avoid that the precision lost by a widening propagates until the end of the analysis. Applying a descending sequence quickly after each loop phase indeed limits this propagation.




```
x = 0;
y = 0;
while (true) {
  if (x <= 50)
    y++;
  else
    y--;
  if (y < 0) break;
  x++;
}
```

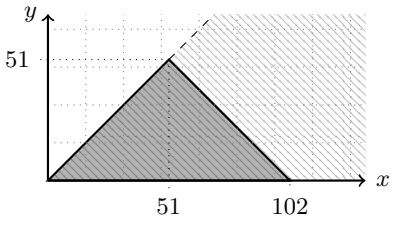


Figure 3.3: Loop with two distinct phases.

Example 7. In this example, we consider the program depicted in Figure 3.3. A standard strategy would apply a widening that gives for n_1 the abstract value $x = y \wedge 0 \leq x$ after one loop iteration. The transition between n_1 and n_3 is then feasible and taken into account in the analysis, leading to a huge loss of precision at the join point n_4. The resulting invariant is depicted in hatched pattern. Guided static analysis first computes a precise invariant for the graph where the dashed transitions are ignored: $x = y \wedge 0 \leq x \leq 51$ (with the help of a narrowing sequence). It then computes an invariant for the full program. The obtained invariant is in dark gray and is strictly smaller than the previous one.

3.3 Fighting bad Effects of Least Upper Bounds

As it is claimed in 3.1, it is possible to get a more precise result by delaying the least upper bound operations between two control points p and q, which is equivalent to distinguish every paths between p and q and apply a "big" least upper bound only at q.

Consider we have the portion of the control flow graph in Figure 3.4, where the $\mathcal{G}^\sharp_{i,k}$ are the guards and $\tau^\sharp_{i,k}$ the abstract transition relation. If $p_1, \cdots, p_n$ are the predecessors

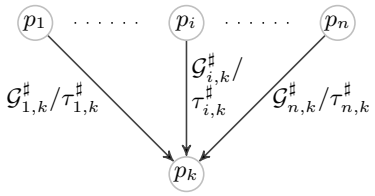


Figure 3.4

of p_k in the graph, the common definition of the abstract transformer is $\Phi^\sharp_k(x^\sharp_1, \ldots, x^\sharp_n) \stackrel{\text{def}}{=} \bigsqcup^\sharp_{i\in[1,n]} \left[\tau^\sharp_{i,k}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,k})\right]$. This definition of $\Phi^\sharp_k$ implies that for each control point having several incoming transitions, precision will be lost due to the least upper bound operator $\sqcup^\sharp$. Intuitively, the least upper bound merges the behaviors of the program traces, that may be very different depending on which predecessor the trace went through.

It is possible to delay the least upper bound operations and get a more precise result. Delaying the least upper bounds, assuming monotonicity, always gives an abstract value which is smaller than (or equal to) the one we would get without delaying. More formally, if k' is a successor of k in the graph, we have:

$$\bigsqcup^\sharp_{i\in[1,n]} \tau^\sharp_{k,k'}\left(\tau^\sharp_{i,k}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,k}) \sqcap^\sharp \mathcal{G}^\sharp_{k,k'}\right) \sqsubseteq^\sharp \tau^\sharp_{k,k'}\left(\bigsqcup^\sharp_{i\in[1,n]} \left[\tau^\sharp_{i,k}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,k})\right] \sqcap^\sharp \mathcal{G}^\sharp_{k,k'}\right)$$

Proof. We note $y^\sharp_{i,k} \stackrel{\text{def}}{=} \tau^\sharp_{i,k}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,k})$.

$$\bigsqcup^\sharp_{i\in[1,n]} \left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right) \sqsubseteq^\sharp \left(\bigsqcup^\sharp_{i\in[1,n]} y^\sharp_{i,k}\right) \sqcap^\sharp \mathcal{G}^\sharp_{i,k}$$

Then, because $\tau^\sharp_{k,k'}$ is monotonic:

$$\tau^\sharp_{k,k'}\left(\bigsqcup^\sharp_{i\in[1,n]} \left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right)\right) \sqsubseteq^\sharp \tau^\sharp_{k,k'}\left(\left(\bigsqcup^\sharp_{i\in[1,n]} y^\sharp_{i,k}\right) \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right) \tag{3.1}$$

and

$$\forall i \in [1,n], \tau^\sharp_{k,k'}\left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right) \sqsubseteq^\sharp \tau^\sharp_{k,k'}\left(\bigsqcup_{i\in[1,n]}^\sharp \left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right)\right)$$

Finally,

$$\bigsqcup_{i\in[1,n]}^\sharp \tau^\sharp_{k,k'}\left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right) \sqsubseteq^\sharp \tau^\sharp_{k,k'}\left(\bigsqcup_{i\in[1,n]}^\sharp \left(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}\right)\right) \tag{3.2}$$

since both operands are upper bounds of $\left\{\tau^\sharp_{k,k'}(y^\sharp_{i,k} \sqcap^\sharp \mathcal{G}^\sharp_{i,k}), i \in [1,n]\right\}$ and the left operand is the least of them. Equations 3.1 and 3.2 give the result. □

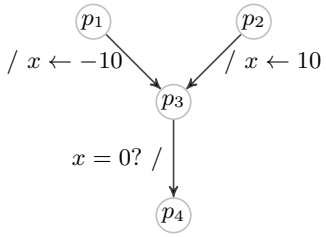


Figure 3.5

Example 8. We take a portion of a control-flow graph described in Figure 3.5. We want to compute the set of possible values for the variable x at control point p_4, using the intervals abstract domain. Suppose we have $x^\sharp_1 = x^\sharp_2 = \top$. Then, if we directly compute $x^\sharp_4$ without least upper bound at $x^\sharp_3$:

$$\bigsqcup_{i\in\{1,2\}}^\sharp \tau^\sharp_{3,4}\left(\tau^\sharp_{i,3}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,3}) \sqcap^\sharp \mathcal{G}^\sharp_{3,4}\right) = \left([-10,-10] \sqcap^\sharp [0,0]\right) \sqcup^\sharp \left([10,10] \sqcap^\sharp [0,0]\right) = \bot$$

If we first compute $x^\sharp_3$ as the least upper bound of the two incoming abstract values, and then compute $x^\sharp_4$:

$$\tau^\sharp_{3,4}\left(\bigsqcup_{i\in\{1,2\}}^\sharp \left[\tau^\sharp_{i,3}(x^\sharp_i \sqcap^\sharp \mathcal{G}^\sharp_{i,3})\right] \sqcap^\sharp \mathcal{G}^\sharp_{3,4}\right) = \left([-10,-10] \sqcap^\sharp [10,10]\right) \sqcap^\sharp [0,0] = [0,0]$$

We deduce that in many cases, it is more precise to delay the application of least upper bounds, which boils down to modify the transitions or the states in the control flow graph so that there is one single control point with several incoming transitions at the end. This is always possible for a graph without cycles, at the expense of an exponential blowup in the number of states or transitions.

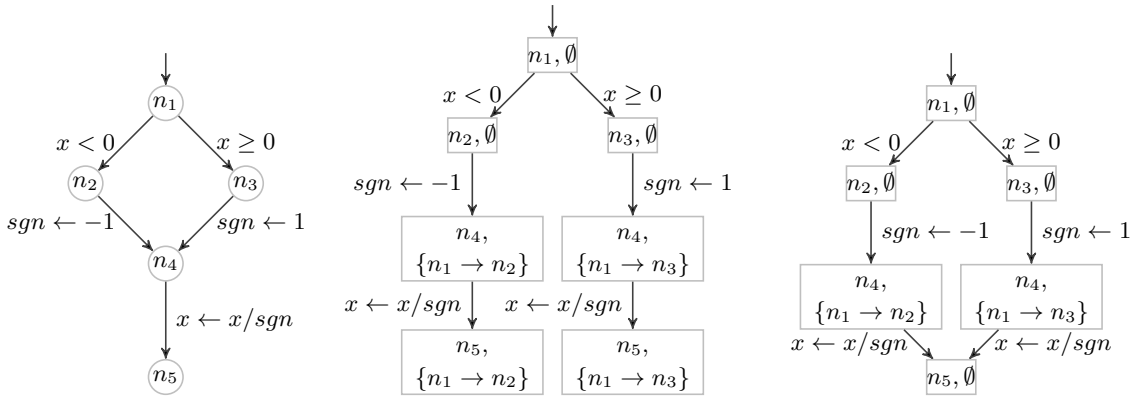
The two following approaches propose solutions to do this path distinction in a efficient way.

3.3.1 Trace Partitioning

A simple way of distinguishing paths is to remember which transitions have been taken at the conditional branches. Rival and Mauborgne [MR05, RM07] developed *Trace Partitioning*, which is an abstract domain that abstracts sets of program traces. The intuition is to duplicate program states and attach to them a set of transitions the trace went through. Then, the main

challenge is to choose which transitions we keep in this set to get a precise result and which one we forget to keep the size of the graph tractable (then, we merge states and compute least upper bounds). Indeed, each transition we keep typically multiplies the size of the new graph by 2. [MR05, RM07] give efficient heuristics for dynamically choosing the partition.

Example 9. Figure 3.6 shows a trace partitioning to prove that a program does not divide by zero. Any convex numerical abstract domain will fail on the graph 3.6a since the least upper bound at point n_4 will contain 0. Figure 3.6b fully distinguish the possible paths, and proves the property. Figure 3.6c merges the paths at point n_5 and can still prove the property.



(a) Program CFG

(b) Remember which transition was taken until n_5

(c) Remember which transition was taken until n_4, and merge at n_5

Figure 3.6: Example of trace partitioning, which proves that a division by zero (between control points n_4 and n_5) does not occur.

3.3.2 Path Focusing

The main drawback when distinguishing paths, if applied without restriction, is the exponential blowup in the number of transitions in the control-flow graph. For instance, a succession of n if-then-else statements has a number of transitions linear in n in the classical control-flow graph, but will give 2^n transitions once every paths have been distinguished. In Trace Partitioning 3.3.1, one have to use heuristics to merge paths at some points to keep the graph tractable. [MG11] proposed a different method for distinguishing every paths while keeping the memory and time consumption reasonable in most cases. The semantics of any loop-free portion of a control-flow graph, or at least an overapproximation thereof, can be encoded into an SMT formula. This formula is satisfiable if there exists a semantically feasible path from the initial state to final state of this portion. A model of this formula is then one feasible path. The construction of this formula is detailed in section 4.2. It is then possible to apply abstract interpretation over the control-flow graph where all paths between loop headers have been expanded, while keeping the representation of this graph succinct using an SMT formula. If this formula is appropriately conjoined with the invariant candidates for the initial and final states of the portion, the model (if satisfiable) exhibits a path that violates the invariant candidate, which is chosen to be the next focus path in the Kleene's ascending sequence. In the contrary, if the formula is unsatisfiable, it means there exist no path between the two control points that violates the invariant candidates. The idea behind this technique is to benefit from the efficiency of modern SMT solvers to avoid an exhaustive exploration of every syntactically

feasible paths (in practice, SMT solvers will prune a large states space when encountering inconsistent predicates). It is often the case that a lot of program paths are unfeasible, or do not make the invariant computation progress (typically, a loop whose semantics is the identity function). In section 4.2, we detail our extensions of this Path Focusing technique.

Part II

Contributions

Chapter 4

How to get Precise Invariants by Abstract Interpretation

This chapter presents our different contributions to abstract interpretation that improve the overall precision of the analysis. Our work can be decomposed into three parts: in section 4.1, we present a novel method for improving a fixpoint after the classical descending sequence; in section 4.2, we propose efficient iteration strategies powered by satisfiability modulo theories that improve the fixpoint obtained after the ascending sequence; section 4.3 finally presents some results when using more expressive abstract domains.

4.1 Improving the Descending Sequence

The abstract interpretation framework is aimed at computing an over-approximation of the least fixpoint of an operator in a chosen abstract lattice. As detailed in subsection 2.1.4, this fixpoint (or post-fixpoint) computation relies on the *ascending sequence*, where the convergence to a post-fixpoint is guaranteed by a *widening* operator ∇. Finally, a *descending*, or *narrowing* sequence is typically computed to recover precision. With the same notations as in subsection 2.1.4, where $\tilde{y}^\sharp$ is a post-fixpoint of $\Phi^\sharp$, the descending sequence with a narrowing operator Δ_N — with some chosen $N > 0$ — consists in computing the limit of the sequence $(z_i^\sharp)_{i\geq 0}$, that converges in finite time:

$$\begin{cases} & z_0^\sharp = \tilde{y}^\sharp \\ \forall i > 0 & z_i^\sharp = z_{i-1}^\sharp \Delta_N \Phi^\sharp(z_{i-1}^\sharp) = \begin{cases} \Phi^\sharp(z_{i-1}^\sharp) & \text{if } i < N \\ z_{i-1}^\sharp & \text{otherwise} \end{cases} \end{cases}$$

This descending sequence may in some cases recover the precision lost by the ascending sequence, intuitively because it gathers information about the end of the execution with the guards that the widening ignored, but also fails on very simple examples. We first illustrate this problem with a simple example.

4.1.1 Motivating Example

We propose in Figure 4.1 a very simple example for which the standard descending sequence fails. We use the abstract domain of intervals for the variable i. Let us first remove line 4 in this C code: in that case, we obtain the most simple possible loop: i starts in $[0, 0]$ in p_2, one loop iteration gives $i \in [1, 1]$ before entering p_2, and widening yields $i \in [0, +\infty)$ at p_2, which is a post-fixpoint. By iterating one step further without widening, we get $i \in [0, 0] \sqcup [1, 100] =$

```
1 int i = 0;
2 while (i < 100) {
3   i++;
4   while (undet()) {}
5 }
```

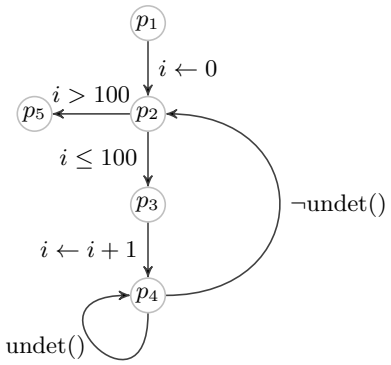


Figure 4.1: Simple example for which the decreasing sequence fails to recover precision

$[0, 100]$. Now, in the presence of the inner loop at line 4, the $\Phi^\sharp$ operator is the following:

$$\Phi^\sharp \begin{pmatrix} x_1^\sharp \\ x_2^\sharp \\ x_3^\sharp \\ x_4^\sharp \\ x_5^\sharp \end{pmatrix} = \begin{pmatrix} \Phi_1^\sharp() \\ \Phi_2^\sharp(x_1^\sharp, x_4^\sharp) \\ \Phi_3^\sharp(x_2^\sharp) \\ \Phi_4^\sharp(x_3^\sharp, x_4^\sharp) \\ \Phi_5^\sharp(x_2^\sharp) \end{pmatrix} = \begin{pmatrix} (-\infty, +\infty) \\ [0,0] \sqcup x_4^\sharp \\ x_2^\sharp \sqcap (-\infty, 100] \\ incr^\sharp(x_3^\sharp) \sqcup x_4^\sharp \\ x_2^\sharp \sqcap [101, +\infty) \end{pmatrix}$$

When applying this operator on the post-fixpoint $\tilde{y}^\sharp$ reached after the ascending sequence, we get the same post-fixpoint, which is consequently a fixpoint:

$$\Phi^\sharp(\tilde{y}^\sharp) = \Phi^\sharp \begin{pmatrix} (-\infty, +\infty) \\ [0, +\infty) \\ [0, 100] \\ [1, +\infty) \\ [101, +\infty) \end{pmatrix} = \begin{pmatrix} (-\infty, +\infty) \\ [0,0] \sqcup [1, +\infty) \\ [0, +\infty) \sqcap (-\infty, 100] \\ [1, 101] \sqcup [1, +\infty) \\ [0, +\infty) \sqcap [101, +\infty) \end{pmatrix} = \tilde{y}^\sharp$$

The problem comes from $\Phi_4^\sharp$, which prevents the value $x_4^\sharp$ from decreasing because of the least upper bound with $x_4^\sharp$ itself. Intuitively, narrowing gives a precise invariant for p_3, but the propagation of the guard $i \leq 100$ stops at p_4 since the self loop prevents the invariant at p_4 from being intersected with $i \leq 101$. Such self loop without any effect on the program may occur often in practice, for instance when the CFG is obtained from the product of two threads. Several techniques in the literature solve this problem, for instance by slicing the program according to variable i [MG12], or using smart widening as described in subsection 3.2.1. In this section, we propose a different approach that works with the usual widening operator and do not modify the CFG, but modifies the decreasing sequence. It can be usefully combined with these other approaches.

4.1.2 Improving a Post-Fixpoint

Principle

Classically, abstract interpretation computes a post-fixpoint using Kleene iterations, by starting from a initial abstract value. In practice, one always chooses $\bot$ as the initial abstract value. In

this section, we claim that starting from $\bot$ is not always the best choice and picking another initial abstract value could in some cases lead to a more precise post-fixpoint.

More formally described, the ascending sequence computes the limit of the following sequence:

$$\begin{cases} y_0^\sharp = & \bot \\ y_i^\sharp = & y_{i-1}^\sharp \nabla \Phi^\sharp(y_{i-1}^\sharp), \forall i > 0 \end{cases}$$

that converges to a post-fixpoint $\tilde{y}^\sharp$ that we will note in this section $\Phi^{\sharp\,\nabla}(\bot)$ to emphasize the fact that the chosen initial value is $\bot$. However, for any chosen initial value $i^\sharp$, the ascending sequence will eventually reach a post-fixpoint of $\Phi^\sharp$. It is then possible to generalize this ascending sequence by defining the operator over the abstract domain $X^\sharp$ noted $\Phi^{\sharp\,\nabla}$, so that $\forall i^\sharp \in X^\sharp, \Phi^{\sharp\,\nabla}(i^\sharp)$ is the limit of the sequence

$$\begin{cases} y_0^\sharp = & i^\sharp \\ y_i^\sharp = & y_{i-1}^\sharp \nabla \Phi^\sharp(y_{i-1}^\sharp), \forall i > 0 \end{cases}$$

Similarly, we define $\Phi^{\sharp\,\nabla\Delta}(i^\sharp)$ as the limit of the descending sequence initialized with $\Phi^{\sharp\,\nabla}(i^\sharp)$. Starting from a different initial value has an interest since $\Phi^{\sharp\,\nabla}$ and $\Phi^{\sharp\,\nabla\Delta}$ are not increasing in general: with some well chosen initial value, one may reach after the ascending sequence a smaller (or incomparable) post-fixpoint than the one we get when starting from $\bot$. As a consequence, the result of the descending sequence which is performed from this post-fixpoint may also be smaller (or incomparable).

Figure 4.2 illustrates the way of improving a fixpoint using two different ascending/descending sequences: one first computes a fixpoint $\tilde{z}^\sharp$ as usual by starting from $\bot$, with an ascending sequence $(y_i^\sharp)_i$ and a decreasing sequence $(z_i^\sharp)_i$. A second fixpoint $\tilde{v}^\sharp$ is computed by starting from an element $i^\sharp \neq \bot$. Then, the greatest lower bound $\tilde{m}^\sharp = \Phi^{\sharp\,\nabla\Delta}(\bot) \sqcap \Phi^{\sharp\,\nabla\Delta}(i^\sharp)$ of the two obtained fixpoints is a safe overapproximation of the least fixpoint $\mathrm{lfp}(\Phi^\sharp)$.

In this section, we describe a way of smartly choosing this initial value $i^\sharp$ based on a known (post-)fixpoint, so that a new ascending/descending sequence may reach a more precise fixpoint.

Resetting an Ascending Sequence

Intuition In the previous example from subsection 4.1.1, narrowing fails for p_4. p_4 has two incoming edges $\tau^\sharp_{3\to4}$ and $\tau^\sharp_{4\to4}$. However, it is easy to see that $\tau^\sharp_{3\to4}([0,100]) = [1,101]$ is an invariant for p_4, and is strictly smaller than the one previously obtained. The fact it is strictly smaller gives the intuition that one could use it for trying to improve the invariant at p_4. Consider now a new ascending sequence, starting with $x_3^\sharp = [0,100]$ instead of $\bot$, and $\bot$ for the other control points:

p_1 :	$\bot$		$(-\infty,+\infty)$		$(-\infty,+\infty)$		$(-\infty,+\infty)$
p_2 :	$\bot$		$\bot$		$[0,101]$		$[0,101]$
p_3 :	$[0,100]$	$\to$	$[0,100]$	$\to$	$[0,100]$	$\to$	$[0,100]$
p_4 :	$\bot$		$[1,101]$		$[1,101]$		$[1,101]$
p_5 :	$\bot$		$\bot$		$\bot$		$[101,101]$

The analysis yields the most precise invariant in the intervals domain. What we learn from this example is that starting a new ascending sequence with some well chosen initial abstract value can lead to a better invariant in the end. We shall now see how to choose this initial abstract value.

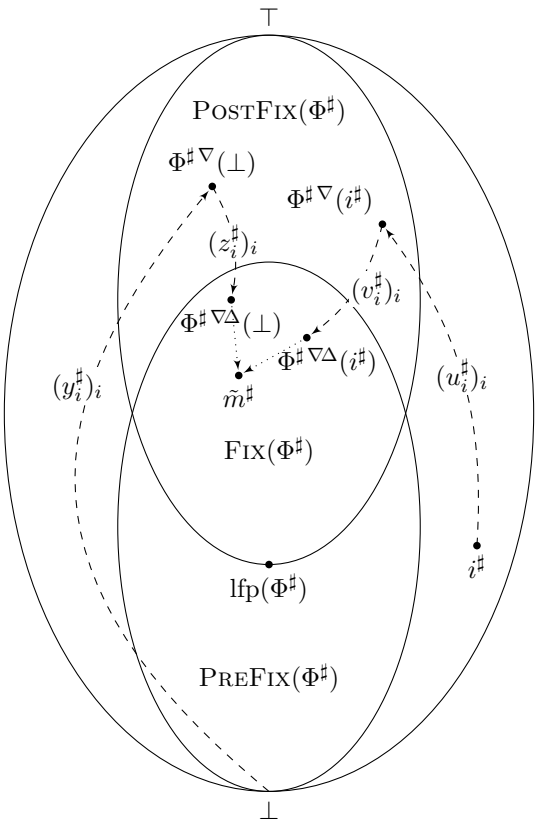


Figure 4.2: Illustration of the ascending and descending sequences over the abstract lattice

Projection of a fixpoint according to a set of seeds For the new ascending sequence to be more precise, one should judiciously choose a subset $S \subseteq P$ of the control points called *seeds*. For any abstract value $x^\sharp = (x_1^\sharp, \ldots, x_n^\sharp) \in X^\sharp$, we define the operator $\Downarrow$: $X^\sharp \times \mathcal{P}(P) \to X^\sharp$:

$$x^\sharp \Downarrow S \stackrel{\text{def}}{=} (s_1^\sharp, \ldots, s_n^\sharp), \text{ where } \forall i, s_i^\sharp \stackrel{\text{def}}{=} \begin{cases} x_i^\sharp & \text{if } p_i \in S \\ \bot & \text{otherwise} \end{cases}$$

The usual ascending/descending sequence reaches the (post-)fixpoint $\Phi^{\sharp\,\nabla\Delta}(\bot)$. We then define a set $S_\bot$ from this fixpoint and compute $\Phi^{\sharp\,\nabla\Delta}(\Phi^{\sharp\,\nabla\Delta}(\bot) \Downarrow S_\bot)$. Our improved solution is the greatest lower bound of the two computed fixpoints. The choice of $S_\bot$ is crucial for improving the final result: the final result will be a safe approximation for any choice of $S_\bot$, but only a smart choice of $S_\bot$ may lead to a precise result. For instance, if we choose $s_\bot = \emptyset$, it means we restart the ascending sequence from $\bot$ and will get exactly the same result as before. If we choose $S_\bot = P$, we will start from abstract values that are already invariants and we also get the same final result. The choice of seed points is described in details in subsection 4.1.3.

4.1.3 Choice of Seed Points

We suppose first that the control-flow graph has one single widening point p_i. This assumption will be relaxed further. The abstract value at p_i depends on those at the predecessors of p_i. If we recursively decompose the CFG into its Strongly Connected Components (SCC), each predecessor is either in the same SCC, either in a previous SCC. We note $\tilde{z}^\sharp \stackrel{\text{def}}{=} \Phi^{\sharp\,\nabla\Delta}(\bot)$. When applying standard chaotic iteration strategies (see subsection 2.1.5), the first non-$\bot$ abstract value at p_i comes from the control points in the previous SCCs, and we call it $z_i^{\sharp 0}$.

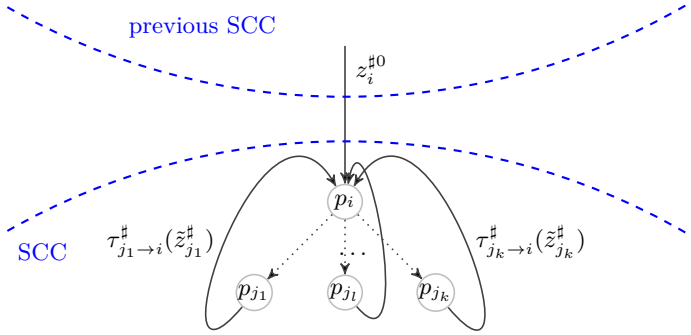


Figure 4.3

Let $p_{j_1}, \ldots, p_{j_k}$ be the predecessors of p_i that are in the same SCC, as it is illustrated by Figure 4.3. Then, p_{j_l} is a good candidate for being a seed point if the image of $\tilde{z}^\sharp_{j_l}$ by $\tau^\sharp_{j_l\to i}$ is strictly smaller than $\tilde{z}^\sharp_i$. It may happen for two reasons:

- either some "initial states" in $z_i^{\sharp 0}$ have been lost on the paths from p_i to p_{j_l}. This case is not interesting since the states in $z_i^{\sharp 0}$ won't be shown to be unreachable. For example, a transition that performs a translation like x = x+1; over an abstract value $0 \leq x$, will give $1 \leq x$, which is smaller. However, if the value 0 comes from the previous SCC (i.e. is in $z_i^{\sharp 0}$), it will be reintegrated anyway in the first step of the new ascending iteration.
- or, $\tau^\sharp_{j_l\to i}(\tilde{z}^\sharp_{j_l})$ collected some properties that are later lost by a least upper bound before reaching p_i. This is a good choice for a seed point.

More generally, one could choose a seed point p_j which is not a direct predecessor of the widening point. In that case, $\tau^\sharp_{j\to i}$ is not a simple transition relation, but actually computes the image of an abstract value by the set of paths between p_j and p_i.

Definition 14 (Seed Control Points, case of one single widening point)**.** With the previous notations, we define the set of *Seed Control Points* $S_\perp \subseteq P$, so that $p_j \in S_\perp$ if and only if the following conditions hold:

- $z_i^{\sharp 0} \sqcup \tau^\sharp_{j\to i}(\tilde{z}^\sharp_j) \sqsubset \tilde{z}^\sharp_i$ (C1)
- $\tau^\sharp_{j\to i}(\tilde{z}^\sharp_j) \not\sqsubseteq z_i^{\sharp 0}$ (C2)
- At least one successor of p_j has several predecessors (C3)

(C1) says that the image from p_j gives a strictly smaller invariant, not only because it removed initial states. (C2) comes from the fact it is useless to propagate again any subset of $z_i^{\sharp 0}$, since the ascending iteration will start with an initial state bigger than $z_i^{\sharp 0}$ anyway. (C3)

finally avoids adding too many seed points, and choose only points that precede merging point (with several incoming transitions) since they are the only one loosing information.

Remark: Depending on the iteration strategy, the first non-$\bot$ value $z_i^{\sharp 0}$ — the one coming from the analysis of the previous SCC — may not always be the same. Our method for choosing the set of *seed* points may thus give a different set $S_\bot$. With the strategy of first stabilizing previous SCCs, we somehow get the "greatest" possible $z_i^{\sharp 0}$, which prevents from choosing too many *seed* points.

The case of a strongly connected components with several widening points only requires generalizing the previous definition:

Definition 15 (Seed Control Points)**.** For a given fixpoint $\tilde{z}^\sharp$, the control point p_j is in the set of *Seed Control Points* $S_\bot$ if and only if the following conditions hold:

- There exists a widening point p_i such that:
  - $z_i^{\sharp 0} \sqcup \tau^\sharp_{j\to i}(\tilde{z}_j^\sharp) \sqsubset \tilde{z}_i^\sharp$
  - $\tau^\sharp_{j\to i}(\tilde{z}_j^\sharp) \not\sqsubseteq z_i^{\sharp 0}$
  - every paths between p_j and p_i do not go through a widening point.
- At least one successor of p_j has several predecessors.

Back to the Example

We illustrate the choice of seed points on our motivating example from subsection 4.1.1. When starting from $\bot$, the analysis reaches the fixpoint

$$\Phi^{\sharp\,\nabla\Delta}(\bot) = \Phi^\sharp \begin{pmatrix} (-\infty, +\infty) \\ [0, +\infty) \\ [0, 100] \\ [1, +\infty) \\ [101, +\infty) \end{pmatrix}$$

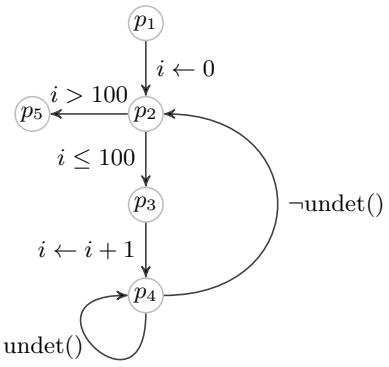


Figure 4.4

p_4 is a widening point, and has two incoming transitions from the same SCC. During the first ascending iteration, the first non-empty abstract value attached to p_4 is $z_4^{\sharp 0} = [0, 0]$. p_3 and p_4 are the candidates for being a seed point since they are predecessors of p_4. p_3 is attached the value $[0, 100]$, and p_4 the value $[0, +\infty)$. Then,

- for p_4: (C1) says: $z_4^{\sharp 0} \sqcup \tau^{\sharp}_{4\to 4}([0, +\infty)) = [0, +\infty) \not\sqsubseteq [0, +\infty)$: the condition is not verified, so p_4 is not chosen.
- for p_3: (C1) holds: $z_4^{\sharp 0} \sqcup \tau^{\sharp}_{3\to 4}([0, 100]) = [0, 100] \sqsubset [0, +\infty)$. (C2) also holds: $\tau^{\sharp}_{3\to 4}([0, 100]) = [1, 101] \not\sqsubseteq [0, 0]$. Then, p_3 is a good candidate and is chosen as a seed point.

Finally, one seed point has been chosen, and a new ascending iteration is computed with the initial value $(\bot, \bot, [0, 100], \bot, \bot)$.

The new ascending sequence reaches the expected precise fixpoint.

Improvements of the Technique

We briefly suggest some improvements of the method:

First Improvement During the second ascending sequence, one should intersect at each step the current abstract value with the previously computed fixpoint $\Phi^{\sharp\,\nabla\Delta}(\bot)$. The invariant obtained after the ascending sequence may then be smaller and could save the analysis to compute a descending sequence. This improvement has been implemented and our experiments in subsection 4.1.5 have been run with these settings.

Second Improvement Our approach consists in recomputing a new ascending/descending sequence with the knowledge of a previously computed fixpoint. However, depending on which seed points have been chosen, one can only recompute this ascending/descending sequence on the subgraph that is likely to be improved. More precisely:

- one should not recompute a SCC if these two conditions hold:
  - it does not contain any seed point;
  - the abstract value at the entry point is not improved by the new analysis of the previous SCC.
- for a given SCC that contains seed points, one should only recompute the control points that may improve the invariant at a widening point. Intuitively the seed points won't be improved since they are used as starting point. It follows that any control point from which any path to a widening point also goes through a seed point won't have any effect on the overall precision. Finally, one should only consider those program points for which there exist a path to a widening point that does not crosses a seed point.

4.1.4 A More Illustrative Example

Our motivating example is well handled by our approach, but would also be with other techniques, such as widenings with thresholds [Hal93, HPR97, LCJG11, WYGI07] or program slicing [MG12]. Here, we propose another example for which other state-of-the art technique do not get a precise invariant, while our approach does. The example in C and its representation as a CFG are depicted in Figure 4.5. The figure also illustrates the reachable concrete states for (i, j) at control point p_4.

```
i = 0;
while (i < 4) {
  j = 0;
  while (j < 4) {
    i++;
    j++;
  }
  i = i-j+1;
}
```

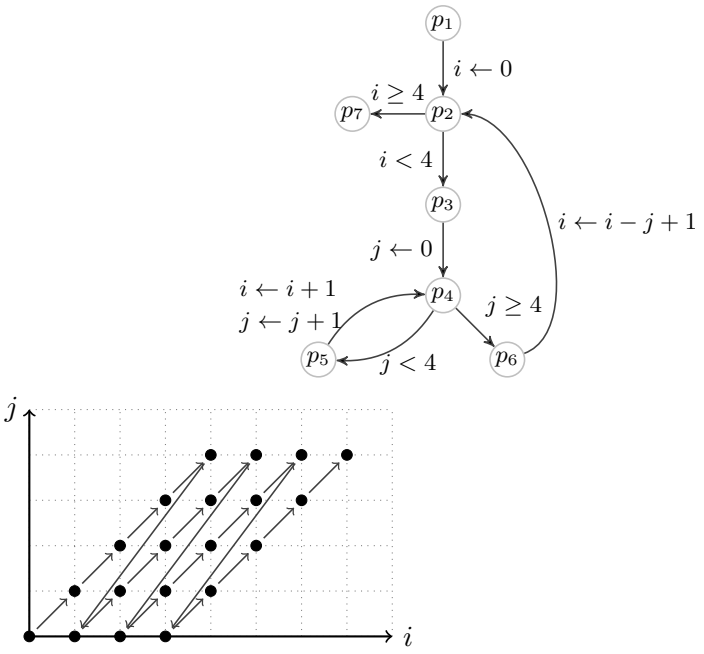


Figure 4.5: Example

We detail the execution of the analysis with the domain of convex polyhedra (see Figure 4.6):

- after the first ascending sequence, we get $0 \leq j \leq i$ for p_4;
- the descending sequence only recovers the upper bound for j, for the same reason as our motivating example. The reached fixpoint is thus $0 \leq j \leq i \wedge j \leq 4$;
- we search for candidate seed points: p_5 and p_3 are the predecessors of p_4. The abstract value coming from p_3 is $0 \leq i \leq 3 \wedge j = 0$ and satisfies all the criteria. The one coming from p_5 is $1 \leq j \leq i \wedge j \leq 4$ and does not satisfy criterion (C1). The set of seed point is then $\{p_3\}$;
- we restart an ascending sequence with the abstract value at p_4 initialized at $0 \leq i \leq 3 \wedge j = 0$. We reach the post-fixpoint $0 \leq j \leq i \leq j + 3$;
- the final descending sequence leads to the fixpoint $0 \leq j \leq i \leq j + 3$, which is strictly smaller than the previous one. Note that if we apply the first improvement described previously, this fixpoint is reached without the need of the last descending sequence.

Finally, our approach catches the additional property $i \leq j + 3$, which is not obtained with the classical approach. This constraint is not syntactically present in the program, and thus cannot be discovered with other techniques such like widening with thresholds.

4.1.5 Experiments

Our improved descending sequence has been implemented in our tool Pagai, which is precisely described in chapter 7.

The major limitation of the implementation is that the choice of seed points is limited to direct predecessors of the loop headers. Then, our experimental results should be seen as initial results that could probably be greatly improved with the fully general choice of seed points.

Our results on a panel of GNU programs are depicted in Table 4.1. We compare the precision of the abstract values obtained at the loop headers of the various benchmarks, both with our

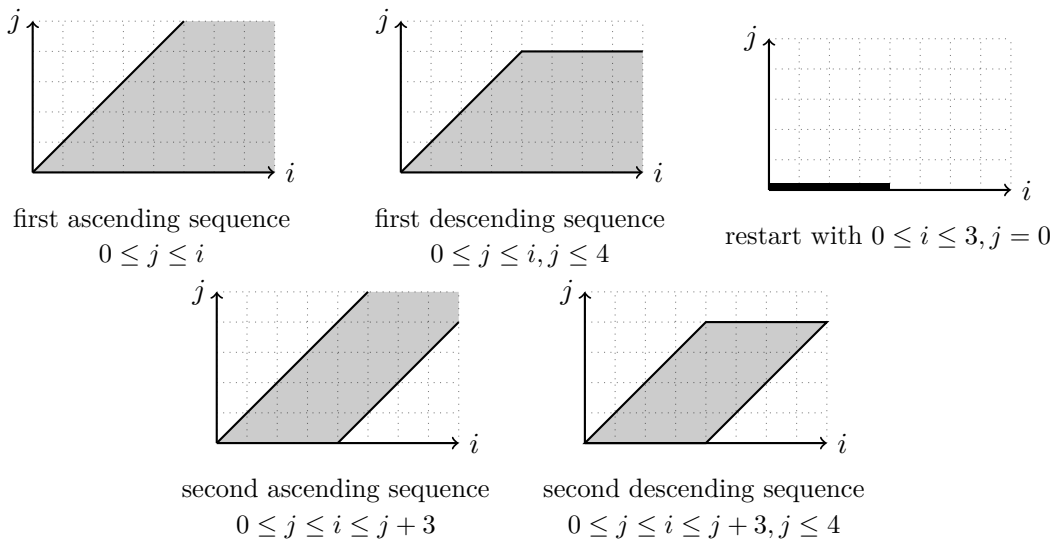


Figure 4.6: Analysis results at point p_4, for the example in Figure 4.5

improved descending iteration (noted **N**) and with a classical abstract interpretation (noted **S**). We also compare the time needed for computing the final invariant. In the "Comparison" column, we detail the number of control points where the improved descending sequence leads to a strictly smaller abstract value ($\sqsubset$), or does not improve the final result ($=$). The "Functions" column gives information on the analyzed source code: #total is the number of functions, #seeds is the number of functions where seed points have been identified, and $\sqsubset$ gives the number of functions where the invariant is improved. The last column "Time eq" gives the timing consumption for analyzing the functions that are not improved.

Benchmark	Comparison		Time (seconds)		Functions				Time eq (seconds)	
	$\sqsubset$	$=$	N	S	#total	#seeds	$\sqsubset$	$=$	N	S
libgsl	127	2724	1074.3	424.9	3888	624	81	3807	1038.2	406.8
grep	19	273	9.7	5.6	372	29	7	365	3.8	2.6
libsuperlu_4.3	9	674	16.1	8.9	187	68	6	181	11.7	6.5
tar	12	536	9.5	6.6	1038	27	9	1029	4.6	3.8
libglpk	98	2406	60.5	35.2	1495	323	53	1442	31.6	18.7
libgmp	23	199	11.6	7.9	299	47	17	282	9.2	6.5
gnugo	366	1302	87.2	56.2	2962	257	150	2812	25.0	19.4
libjpeg	38	396	6.4	3.9	335	80	20	315	4.9	3.2
sed	3	82	4.1	2.2	196	3	2	194	0.8	0.7
gzip	21	209	10.2	5.6	212	15	9	203	4.8	3.0
wget	26	431	16.5	9.4	707	38	15	692	6.7	4.8
libpng16	32	358	7.7	5.5	498	32	15	483	4.9	3.5
libsuperlu	9	674	13.7	7.9	187	68	6	181	9.9	5.8

Table 4.1: Experimental results, and comparison with standard abstract interpretation.

4.2 Improving the Ascending Sequence

In this section, we explain how it is possible to get precise results by combining abstract interpretation with bounded model checking by SMT solving (already introduced in section 2.2.1).

4.2.1 Program Encoding into SMT Formula

In this section, we detail a sound way of encoding the semantics of a program into an SMT formula, whose models are feasible program traces. In this chapter, we suppose that the program is described by a control-flow graph in Static Single Assignment (SSA) form:

Definition 16 (Static Single Assignment Form)**.** A program is said to be in *Static Single Assignment Form* if and only if any variable is only assigned in one program location.

Note that this is not a restriction, since any non-SSA program can be translated into SSA form. This can be achieved by renaming the variables assigned several time, and introducing special instruction called Φ-instructions. These Φ-instructions are attached to the nodes of the CFG, and allow to declare an SSA-variable whose value depends on the incoming control point. For instance, in Figure 4.7, variable x.2 is assigned to x.0 if the program execution goes through the *then* branch of the *if-then-else*, and is assigned to x.1 if it goes through the *else* branch. Efficient algorithms for SSA translations and analysis are known[Boi10].

```
if (c) x = 0;
else x = 1;

y = x+1;
```

$\Longrightarrow$

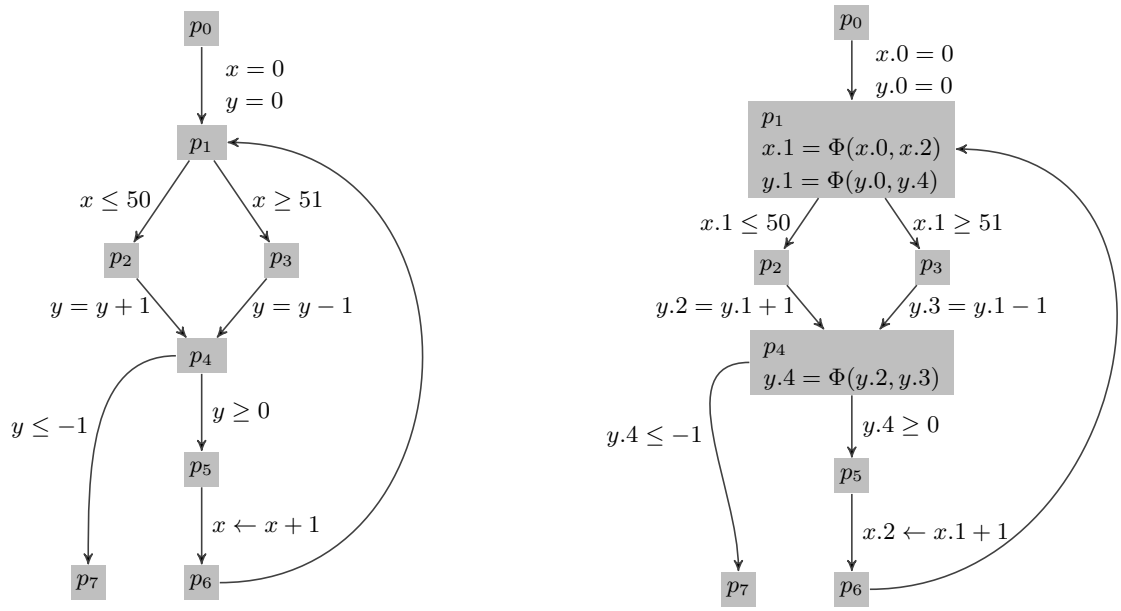
```
if (c.0) x.0 = 0;
else x.1 = 0;

x.2 = Phi (x.0, x.1);
y.0 = x.2 + 1;
```

Figure 4.7: SSA transformation for a simple program fragment

In practice, SSA transformation need to introduce Φ-instructions at the control flow merges, that may typically come from loops or *if-then-else* statements. Figure 4.8 illustrates SSA transformation for a simple program containing a loop: the Φ-instruction $x.1 = \Phi(x.0, x.1)$, that we will note $x.1 = \Phi([x.0, p_0], [x.1, p_6])$ for clarity, assigns to $x.1$ the value of $x.0$ if the execution trace comes from p_0, and $x.1$ if it comes from p_6.

Each instruction present in the control-flow graph has a semantics that can be safely overapproximated by a logical formula in the chosen theory. Note that certain theories have been specially invented so that it is possible to encode in SMT the exact semantics of some instructions: one can mention for instance the theory of bitvectors [KS08, BCF+07] or floating point numbers[RW10]. The SSA instruction may have side effects in the memory, and can only define the value of one single SSA scalar variable. In this thesis, side effects are not tracked in the SMT encoding. It follows that instructions loading from the memory (*load*) are strongly overapproximated by nondeterministic choices, and instructions that store into the memory (*store*) are approximated to *nop* operations (do nothing). However, a simple pointer analysis over the SSA program enables in practice to remove most of the *load* and *store* instructions through the use of scalar registers.



(a) Non SSA control flow graph.

(b) SSA translated control flow graph.

Figure 4.8: SSA transformation of a simple control flow graph

Loop-free Program

In the simple case where the program has no cycle, a trace between the initial state and the final state goes through each control point and each transition at most once. The control structure can then be easily encoded using Boolean variables corresponding to the transitions and the control points. We detail here how to construct an SMT formula whose models are feasible program traces.

Definition 17. Let $P = \{p_1, p_2, \ldots, p_m\}$ be the set of control points of the program, and τ the set of transitions. We define for each $p_i \in P$ a Boolean variable b_i, and for each transition $(p_i, p_j, C_{i,j}) \in \tau$ a Boolean variable $t_{i,j}$. The condition $C_{i,j}$ is a Boolean SSA-variable defined either by an instruction, a function parameter, etc. or by a nondeterministic choice.

Then, it is possible to construct a formula whose models are program traces, by setting these Boolean variables in the following way:

- For each p_i in P, if p_i is the initial state, $b_i = true$. Otherwise, $b_i = \bigvee_{j/(p_i,p_j,C_{i,j})\in\tau} t_{i,j}$
- For each $(p_i, p_j, C_{i,j}) \in \tau$, $t_{i,j} = b_i \wedge C_{i,j}$.

Note that it would be simple to eliminate the b_i's in order to reduce the number of Boolean variables in the formula. For convenience, we keep both in this thesis to simplify the encoding of some instructions.

Here, a program state is identified by a tuple of values for the SSA scalar variables $\{x_1, x_2, \ldots, x_n\}$. For the particular SSA scalar variable x_k assigned by an instruction I_k, such that $I_k(x_1, \ldots, x_{k-1})$ is the set of possible values for x_k, we can say that the SMT formula $I_k^\sharp(x_1, \ldots, x_{k-1}, x_k)$ is a safe overapproximation of the instruction semantics if

$$\forall x_1, \ldots, x_k, \quad x_k \in I_k(x_1, \ldots, x_{k-1}) \Rightarrow I_k^\sharp(x_1, \ldots, x_{k-1}, x_k) \text{ is } true$$

Since the program variable is assigned only once, the possible assignments for the corresponding SMT variable in the formula will overapproximate the possible values for that variable in any program execution.

Example 10.

1. A Φ-instruction of the form $x.2 = \Phi([b.0, x.0], [b.1, x.1])$ can be precisely encoded by $(b.0 \Rightarrow x.2 = x.0) \wedge (b.1 \Rightarrow x.2 = x.1)$.
2. A floating point operation $x = y + z$ may be encoded by $y + z - \varepsilon \leq x \leq y + z + \varepsilon$ with some ε such that $|\varepsilon| \leq \varepsilon_r |y + z|$, according to the semantics of floating points, if the theory is Linear Rational Arithmetic (LRA).

Finally, the resulting formula is:

$$\rho_P = \left[\bigwedge_{i \in 1..m} b_i = \dots\right] \wedge \left[\bigwedge_{(p_i, p_j, C_{i,j}) \in \tau} t_{i,j} = \dots\right] \wedge I_1^\sharp(x_1) \wedge I_2^\sharp(x_1, x_2) \wedge \dots \wedge I_n^\sharp(x_1, \dots, x_n)$$

Handling Overflow

A sound way of encoding arithmetic operations over integer types for some usual programming languages (e.g. C) would be to use the theory of bitvectors to represent elements of integer types. However, efficiently manipulating bitvectors is hard since the integers will be considered as arrays of Booleans, and the semantics of the usual operations over the integers will have a complicated definition. Here, we suggest a different approach, using the theory of Linear Integer Arithmetic (LIA): we introduce a Boolean variable ofl whose valuation will be *true* if the operation overflows. Then, the SMT encoding of the C unsigned instruction $x = y + z$ will be:

$$(\text{ofl} = (y + z \geq \texttt{UMAX})) \wedge (\neg\, \text{ofl} \Rightarrow x = y + z) \wedge (\text{ofl} \Rightarrow x = y + z - \texttt{UMAX} - 1)$$

where `UMAX` is the greatest element of the corresponding type. In the case the overflow is *undefined behavior* in the source language (e.g. C signed integers), one can create an *error* control point and branch to it whenever the ofl Boolean variable is *true*.

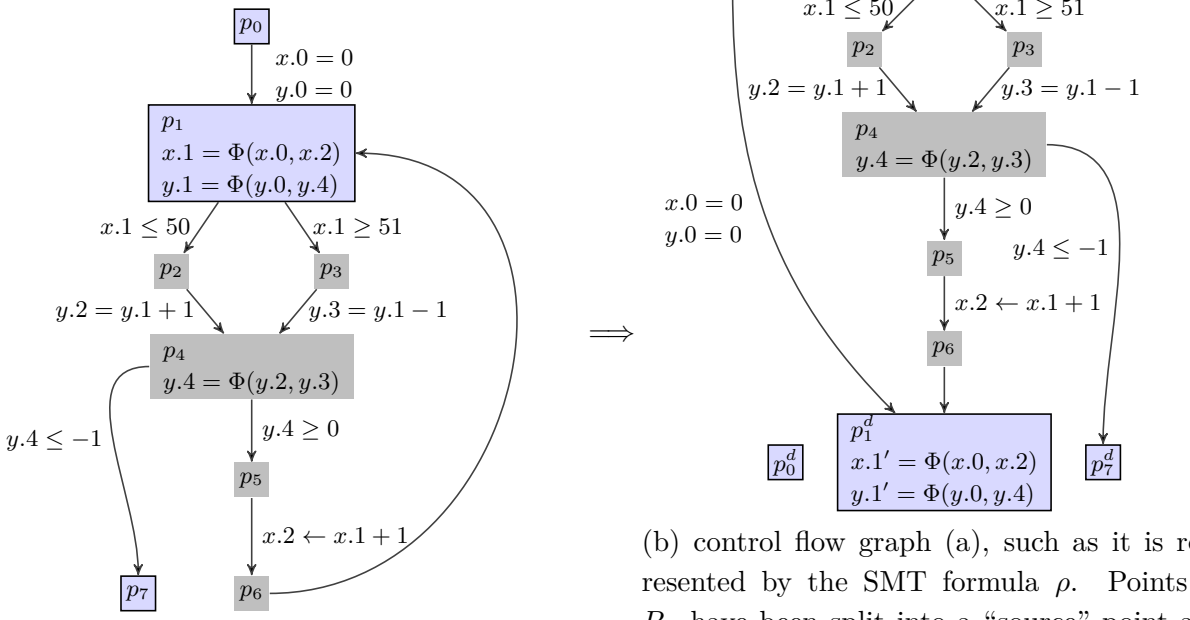
Program with Loops

The previous encoding provides a formula whose models are program traces between the initial and the final control point (or possibly an *error* control point) in the case the control-flow graph (CFG) has no cycle. However, it is not possible to construct such formula if the program contains loops, since the control points inside the loop could be reached several time. In this case, a standard approach is *Bounded Model Checking*, that consists in removing the loop using unrolling up to a given depth D. After unrolling, the CFG is loop-free and the previous encoding applies. However, models for the obtained formula will only be traces that iterate over the loop at most D times, forgetting about longer feasible traces.

Here, our approach is not using loop unrolling: we slightly change our previous encoding so that the models are feasible loop-free subtraces between two control points. The motivation of doing so is described in the following sections. The starting and ending points of these subtraces are chosen in a set of *cutting points* defined as follows:

Definition 18 (Cutting points)**.** Let $G = (P, \tau)$ be the control-flow graph representing a program, and $P_W \subseteq P$ be the set of widening points. A set $P_R \subseteq P$ is a *correct* set of *Cutting points* if and only if $P_W \subseteq P_R$ and P_R contains the initial and final control points.

A minimal set of cutting points can be computed in linear time [Sha79] for reducible graphs. Then, the SMT formula that we will construct will express program traces between two cutting points. We will use a similar encoding as in the loop-free case for a modified graph that has no cycle. Figure 4.9 depicts an example of SSA control flow graph with a loop, as well as the modified graph where the cutting points have been split.



(a) CFG in SSA form. $P_R = \{p_0, p_1, p_7\}$

(b) control flow graph (a), such as it is represented by the SMT formula ρ. Points in P_R have been split into a "source" point and a "destination" point, and Φ-variables associated to points in P_R have been primed.

Figure 4.9: Program CFG, and loop free transformed graph being encoded by our SMT formula

We recast the encoding of the control structure with Boolean variables:

Definition 19. Let $P = \{p_1, p_2, \ldots, p_m\}$ be the set of control points of the program, and τ the set of transitions. Let $P_R \subseteq P$ be a set of *Cutting points*. We define for each $p_i \in P$ P_R a Boolean variable b_i, and for each transition $(p_i, p_j, C_{i,j}) \in \tau$ a Boolean variable $t_{i,j}$. For each $p_i \in P_R$, we define two Boolean variables b_i^s and b_i^d. Informally, b_i^s will be evaluated to true if the trace starts at p_i, while b_i^d is evaluated to true if the traces ends at p_i. The control structure of the program is encoded in SMT in the following way:

- For each $(p_i, p_j, C_{i,j}) \in \tau$, $t_{i,j} = \begin{cases} b_i^s \wedge C_{i,j} & \text{if } p_i \in P_R \\ b_i \wedge C_{i,j} & \text{otherwise} \end{cases}$
- For each p_i in $P \backslash P_R$, $b_i = \bigvee_{j/(p_i,p_j,C_{i,j})\in\tau} t_{i,j}$
- For each p_i in P_R, $b_i^d = \bigvee_{j/(p_i,p_j,C_{i,j})\in\tau} t_{i,j}$
- There is exactly one Boolean variable among the set $\{b_i^s\}_{p_i \in P_R}$ which evaluates to *true*. This property can be expressed by the subformula

$$\left[\bigvee_i b_i^s\right] \wedge \bigwedge_{p_k \in P_R} \left[b_k^s \Rightarrow \bigwedge_{i \neq k} \neg b_i^s\right]$$

When the program has cycles, it may be possible to get a loop-free trace for which the starting and the ending control point are the same. In that case, one should distinguish the values for the Φ-instructions defined in control points in P_R before and after the effect of the loop. The value before, e.g. x_k, is part of the precondition that will be later conjoined to the formula – and thus is not assigned in the formula –, and the value after, noted x'_k, is defined by the Φ-instruction using the formula $I^\sharp_k(x_1, \ldots, x_{k-1}, x'_k)$ (recall that $I^\sharp_k$ is the safe overapproximation of the effects of the instruction). In other words, each use of the SSA variable x_k is encoded with the x_k variable in the formula, while the definition of the SSA Φ-instruction x_k actually defines the primed variable x'_k. To conclude, the SMT encoding of the instruction defining x_k is:

$$J^\sharp_k(x_1, \ldots, x_{k-1}) = \begin{cases} I^\sharp_k(x_1, \ldots, x'_k) \text{ if } x_k \text{ is a } \Phi\text{-instruction defined in a point in } P_R \\ I^\sharp_k(x_1, \ldots, x_k) \text{ otherwise} \end{cases}$$

Finally, the SMT formula we get for the program P is the following:

$$\begin{aligned} \rho_P = \quad & \left[\bigwedge_i b_i = \ldots\right] \wedge \left[\bigwedge_i b^d_i = \ldots\right] \wedge \left[\bigwedge_{(p_i,p_j,C_{i,j})\in\tau} t_{i,j} = \ldots\right] \\ \wedge \quad & \left[\bigvee_i b^s_i\right] \wedge \bigwedge_{p_k \in P_R} \left[b^s_k \Rightarrow \bigwedge_{i\neq k} \neg b^s_i\right] \\ \wedge \quad & J^\sharp_1() \wedge J^\sharp_2(x_1) \wedge \cdots \wedge J^\sharp_n(x_1, \ldots, x_{n-1}) \end{aligned}$$

4.2.2 Guided Path Analysis

In this section, we adopt the notations and the SMT formula defined in subsection 4.2.1.

Guided static analysis [GR07], as described in subsection 3.2.3, applies to the transition graph of the program, and is a way of limiting the loss of precision due to widenings. We now present a new technique applying this analysis on the expanded graph from [MG11] (see subsection 3.3.2), where all the paths between loop headers are expanded, thus avoiding control flow merges in many control points.

The combination of these two techniques aims at first discovering a precise inductive invariant for a subset of traces between two *cutting* points, by the mean of ascending and descending Kleene sequences. When an inductive invariant has been found, we add new feasible paths to the subset and compute an inductive invariant for this new subset, starting with the results from the previous analysis. In other words, our technique considers an ascending sequence of subsets of the paths between two points in P_R. We iterate the operations until the whole program (i.e all the feasible paths) has been considered. The result will then be an inductive invariant of the entire program.

Formally, if $\tau^\sharp$ is the abstract transition relation of the expanded control flow graph, we will work with a finite sequence of transition relations $(\tau^\sharp_i)_{i\leq n}$ such that $\forall x^\sharp \in X^\sharp, \tau^\sharp_0(x^\sharp) = \bot$, $\tau^\sharp_n = \tau^\sharp$, and $\forall x^\sharp \in X^\sharp, \forall i < n,\ \tau^\sharp_i(x^\sharp) \sqsubseteq \tau^\sharp_{i+1}(x^\sharp)$. The sequence of operators $(\Phi^\sharp_i)_{i\leq n}$ is defined by:

$$\forall x^\sharp \in X^\sharp, \Phi^\sharp_i(x^\sharp) = I^\sharp \sqcup \tau^\sharp_i(x^\sharp)$$

We then iteratively compute a fixpoint $\tilde{x}^\sharp_i$ of the $\Phi^\sharp_i$ starting from the computed fixpoint $\tilde{x}^\sharp_{i-1}$ of $\Phi^\sharp_{i-1}$. $\tau^\sharp_i$ is defined according to $\tilde{x}^\sharp_{i-1}$, so that $\tau^\sharp_i(\tilde{x}^\sharp_{i-1}) = \tau^\sharp(\tilde{x}^\sharp_{i-1})$.

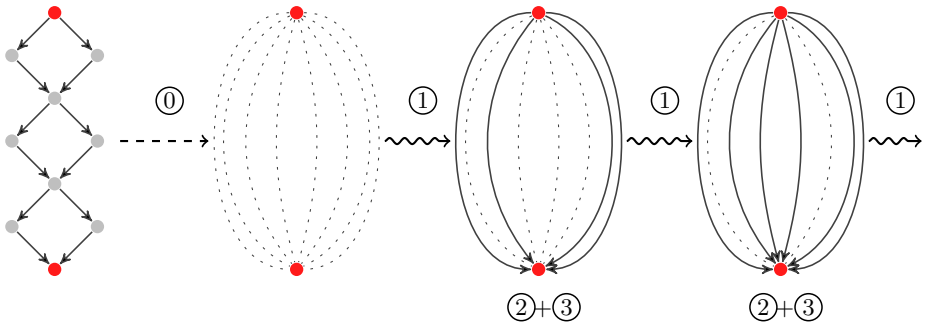


Figure 4.10: Ascending sequence of subsets of paths between two *Cutting* points.

Algorithm

Our new technique performs guided static analysis on the expanded CFG that distinguishes every paths between the *Cutting* points. The initial step ⓪ (see Figure 4.10) consists in succinctly representing the expanded CFG by the SMT formula ρ_P previously defined.

We initialize the current working subset of paths, noted $\mathcal{P}$, to the empty set. In practice, this set $\mathcal{P}$ can be stored in memory using a compact representation, such as binary decision diagrams. We also maintain two sets of control points:

- A' contains the points in P_R that may be the starting points of new feasible paths. Typically, each time the abstract value attached to a control point is updated, there may exist a new feasible path starting from it.
- A is the work list that contains the *active* points in P_R on which we apply the ascending iterations. Each time the abstract value of a control point p is updated, p is inserted in both A and A'.

Our algorithm is fully described in Algorithm 2. We distinguish three phases in the main loop of the analysis, illustrated by Figure 4.10. When entering ①, our technique has already computed a precise inductive invariant for the program restricted to the paths in $\mathcal{P}$. Since $\mathcal{P}$ is initially empty, an inductive invariant X_i^s for each control point p_i is its set of initial states I_{p_i}.

① **Add new paths:** we start finding a new relevant subset $\mathcal{P} \cup \mathcal{P}'$ of the graph. $\mathcal{P}$ is the previous subset, and $\mathcal{P}'$ is the set of newly feasible paths. consists in choosing this set $\mathcal{P}'$. Either the previous iteration or the initialization lead us to a state where there are no more paths in the previous subset $\mathcal{P}$ that make the abstract values of the successors grow. Narrowing iterations preserve this property. However, there may exist such paths in the entire graph, that are not in $\mathcal{P}$. This phase computes these paths and adds them to $\mathcal{P}'$. This phase is described in more details later and corresponds to lines 9 to 13 in Algorithm 2.

② **Ascending iterations:** given a new subset $\mathcal{P}$, we search for paths in the set $\mathcal{P}$, i.e included in the working subgraph, that lead to a state outside of the current candidate invariants.. Each time we find such a path, we update the abstract value of the destination point of the path. This is the phase explained in the next paragraph, and corresponds to lines 16 to 20 in Algorithm 2.

③ **Descending iterations:** we perform narrowing iterations the usual way (line 21 in algorithm 2) and reiterate from step 1 unless there are no more points to explore, i.e. $A' = \emptyset$.

Algorithm 2 Guided static analysis on implicit multigraph

```
1:  A' ← {p|P_R/I_p ≠ ∅}
2:  A ← ∅
3:  P ← ∅                                          ▷ Paths in the current subset
4:  for all p_i ∈ P_R do
5:      X_i^s ← I_{p_i}
6:  end for
7:  while A' ≠ ∅ do
8:      P' ← ∅                                     ▷ Set of new paths
9:      while A' ≠ ∅ do
10:         Select p_i ∈ A'
11:         A' ← A' \ {p_i}
12:         ComputeNewPaths(p_i)                   ▷ Updates A, P' and possibly A'
13:     end while
14:     P ← P ∪ P'
15:                                                ▷ ascending iterations on P
16:     while A ≠ ∅ do
17:         Select p_i ∈ A
18:         A ← A \ {p_i}
19:         PathFocusing(p_i)                      ▷ Updates A and A'
20:     end while
21:     Narrow
22: end while
23: return {X_i^s, i ∈ P_R}
```

Step ② corresponds to the application of path-focusing [MG11] to the elements of A until A is empty. When A becomes empty, it means that an invariant for the current subgraph has been reached. As proposed by [GR07], we can do some narrowing iterations. These narrowing iterations allow to recover precision lost by widening, *before* computing and taking into account new feasible paths. Thus, our technique combines both the advantages of *Guided Static Analysis* and *Path-Focusing*.

The order of steps is important: narrowing has to be performed before adding new paths, or some spurious new paths would be added to $\mathcal{P}$. Also, starting with the addition of new paths avoids wasting time doing the ascending iterations on an empty graph.

Ascending Iterations by Path-Focusing

For computing an inductive invariant over a subgraph (step ②), we use the Path-focusing algorithm from [MG11] with special treatment for self loops (line 19 in algorithm 2).

In order to find which path to focus on, we construct an SMT formula $f(p_i)$, whose model when satisfiable is a path that starts in $p_i \in P_R$, goes to a successor $p_j \in P_R$ of p_i, such that the image of the current abstract value X_i^s attached to p_i by the path abstract transformer is not included in the current X_j^s. Intuitively, such a path makes the abstract value X_j^s grow,

and thus is an interesting path to focus on for reaching the fixpoint. We loop until the formula becomes unsatisfiable, meaning that the analysis of p_i is finished.

If we note $Succ(i)$ the set of indices j such that $p_j \in P_R$ is a successor of p_i in the expanded multigraph, and X_i^s the abstract value associated to p_i :

$$f(p_i) = \underbrace{\rho_P}_{\text{program encoding}} \wedge \overbrace{b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge X_i^s}^{\text{path starts from } b_i^s \text{ in } X_i^s} \wedge \bigvee_{j \in Succ(i)} \overbrace{(b_j^d \wedge \neg\, \text{primed}(X_j^s))}^{\text{path ends in } b_j^d \text{ outside } X_j^s} \wedge \underbrace{\mathcal{P}}_{\text{path is in } \mathcal{P}}$$

where $\text{primed}(X_j^s)$ is the encoding of the X_j^s abstract value into an SMT formula where the Φ-variables defined in p_j are renamed into their primed counterparts. For the sake of simplicity, we use the same notation X_i^s for both the abstract value and its encoding into an SMT formula. This implies that any element of the abstract domain can be exactly expressed within the chosen theory in SMT. In other words, the theory used in the formula should be at least as expressive as the abstract domain. In the case of usual numerical abstract domains (e.g. intervals, octagons, polyhedra), the corresponding formula is simply the conjunction of the constraints defining the abstract value.

The difference with [MG11] is that we do not work on the entire transition graph but on a subset of it. Therefore we conjoin the formula with the current set of working paths $\mathcal{P}$, expressed as a Boolean formula, where the Boolean variables are the *reachability predicates* of the control points. We can easily construct this formula from the binary decision diagram using dynamic programming, and avoid an exponentially sized formula. In other words, we force the SMT solver to give us a path included in the set $\mathcal{P}$. Note that it would also be possible to encode a new ρ formula for each working subset of paths, instead of using the encoding of the entire program conjoined with $\mathcal{P}$; it is unclear whether it makes any difference in practice. On the first hand, a larger formula may compromise performance of the SMT solver. On the second hand, always using the same formula may take more benefits from incremental solving capabilities of state-of-the-art solvers.

If the formula is satisfiable, the SMT-solver returns a model that provides a program trace between b_i^s and one of the b_j^d. The trace $p_{i_1} \to \cdots \to p_{i_k}$ – where k is the length of the trace, $i_1 = i$ and $i_k = j$ – is directly given by the Boolean variables assigned to *true* in the model. We then update the invariants candidate X_j^s with $X_j^s \sqcup \tau^\sharp_{i_{k-1} \to i_k} \circ \cdots \circ \tau^\sharp_{i_1 \to i_2}(X_i^s)$ (with potentially a widening if $p_j \in P_W$).

It is possible that the trace is actually a self-loop, i.e. $p_i = p_j$. In that particular case, it is possible to apply *abstract acceleration* [GH06, Gon07] or a local *widening/narrowing* phase.

We successively update the invariant candidates until the SMT formula becomes unsatisfiable. In that case, the X_j^s are correct invariants modulo the precondition X_i^s.

Adding new Paths

Our technique computes the fixpoint iterations on an ascending sequence of subgraphs, until the complete graph is reached. When the analysis of a subgraph is finished, meaning that the abstract values for each control points has converged to an inductive invariant for this subgraph, the next subgraph to work on has to be computed (step ①).

This new subgraph is the union of the paths of the previous one with a set $\mathcal{P}'$ of new paths that become feasible regarding the current abstract values. The paths in $\mathcal{P}'$ are computed one after another, until no more path can make the invariant grow. This is line 12 in Algorithm 2. We also use SMT solving to discover these new paths, by subtly recasting the SMT formula

given to the SMT solver: we now simply check for the satisfiability of $f'(p_i)$ where $f'(p_i)$ is $f(p_i)$ where the conjunct $\mathcal{P}$ is removed. Since we already know that $f(p_i) \wedge \mathcal{P}$ is unsatisfiable, none of the paths given by the SMT solver will be in $\mathcal{P}$.

Also, to prevent the algorithm from adding too many paths at a time, and in particular to prevent the algorithm from trying to re-add the same path infinitely many times, we use another abstract value associated to the control point p_j, noted X_j^d, which is distinct from X_j^s, and initialized to X_j^s right before computing new paths. In the SMT formula, we associate to p_j the abstract value X_j^s when p_j is the starting point of the path, and X_j^d when it is its destination point. We thus check the satisfiability of the formula $f'(p_i)$, where:

$$f'(p_i) = \overbrace{\rho_P \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge X_i^s}^{\text{same as in } f(p_i)} \wedge \bigvee_{j \in Succ(i)} \overbrace{(b_j^d \wedge \neg\, \text{primed}(X_j^d)}^{\text{same as in } f(p_i) \text{ but } X_j^d \text{ instead of } X_j^s)}$$

This SMT actually means "Does there exist a path in the entire program that starts in b_i^s, with the starting variables having a value in X_i^s, that goes to a successor b_j^d of b_i^s, with the new values for the variables being outside the current candidate invariant X_j^s ?"

X_j^d is updated when the point p_j is the target of a new path. This way, further SMT queries do not compute other paths with the same source and destination if it is not needed (because these new paths would not make X_j^d grow, hence would not be returned by the SMT solver). The algorithm is described in Algorithm 3.

Algorithm 3 Construction of the new subgraph

```
1:  procedure ComputeNewPaths(p_i)
2:      for each p_j successor of p_i do X_j^d ← X_j^s
3:      end for
4:      while true do
5:          res ← SmtSolve[f'(p_i)]
6:          if res = unsat then
7:              break
8:          end if
9:          Extract the path π from p_i to p_j from the model
10:         X_j^d ← X_j^d ⊔ π(X_i^s)
11:         P' ← P' ∪ {π}
12:         A ← A ∪ {p_i}
13:     end while
14: end procedure
```

Variant The algorithm 3 for adding new paths can be slightly modified in order to reduce the number of subgraphs in the sequence. Indeed, if $p_1 \to p_2$ becomes feasible, and $p_2 \to p_3$ become feasible as well because $\pi_{2\to3} \circ \pi_{1\to2}(X_1^s) \neq \bot$, then one could insert the path $\pi_{2\to3}$ instantaneously instead of waiting for the next iteration. In some cases, in may degrade the precision of the analysis, but will be faster in general because it reduces the number of subgraphs $\mathcal{P}$ to process. The new algorithm is the same as Algorithm 3, by replacing line 10 by $X_j^s \leftarrow X_j^s \sqcup \pi(X_i^s)$, removing line 2 and inserting $A' \leftarrow A' \cup \{p_i\}$ after line 12. In the example page 67, we illustrate our approach using this variant.

Termination

The termination of the analysis is subject to a property between the abstraction used in the abstract interpretation, i.e. the abstract domain and the chosen forward abstract transformer $\tau^\sharp$, and the overapproximations done in the SMT encoding. Intuitively, for a path $p_{i_1} \to p_{i_2} \to p_{i_k}$ returned by the SMT solver that gives a point M outside the invariant candidate $X^s_{i_k}$ (i.e. $M \not\sqsubseteq X^s_{i_k}$), the image of $X^s_{i_1}$ by the corresponding abstract transformer $\tau^\sharp_{i_1 \to i_k} = \tau^\sharp_{i_{k-1} \to i_k} \circ \cdots \circ \tau^\sharp_{i_1 \to i_2}$ has to contain the point M, otherwise the SMT solver may keep returning the same path infinitely many times. In other words, the SMT formula representing the program should be at least as precise as the abstract transformer in the abstract interpretation. An alternative is to incrementally insert blocking clauses into the SMT formula whenever a trace is proved impossible in the abstract interpretation, as it is done by [HSIG10].

Termination of this algorithm is then guaranteed, because:

- The subset of paths $\mathcal{P}$ strictly increases at each loop iteration, and is bounded by the finite number of paths in the entire graph. The set $\mathcal{P}'$ always verifies $\mathcal{P} \cap \mathcal{P}' = \emptyset$ by construction, which guarantees that $\mathcal{P}'$ will eventually be empty after a finite number of loop iterations. When $\mathcal{P}'$ is empty after computing the new paths, it follows that A and A' are empty as well and the main loop terminates.
- The ascending sequence by *Path-Focusing* terminates because of the properties of widening, which is applied infinitely often.

Example

We illustrate our approach with the program described in 4.11. This program implements a construct commonly found in control programs (in automotive or avionics for instance): a rate or slope limiter. In the real program its input is the result of previous computation steps, but here we consider that the input is nondeterministic within $[-10000, +10000]$.

Suppose first that the nested loop at line 8 is commented. In this case, *Path-focusing* works well because all the paths starting at the loop header are actually self loops. In such a case, the technique performs a widening/narrowing sequence or accelerates the loop, thus yields a precise invariant. However, in many cases, there also exist paths that are not self loops, in which case *Path-focusing* applies only widening and loses precision. Alternatively, *Guided Static analysis* behaves badly because of a least upper bound computed in the end of the first *if* statement at line 5.

```
1 void rate_limiter() {
2   int x_old = 0;
3   while (1) {
4     int x = input(-100000, 100000);
5     if (x > x_old+10) x = x_old+10;
6     if (x < x_old-10) x = x_old-10;
7     x_old = x;
8     while (condition()) {wait();}
9 }
```

Figure 4.11: Rate Limiter Example

We choose P_R as the set of loop headers of the function, plus the initial state. In this case, we have three elements in P_R.

The main loop in the expanded multigraph has then 4 distinct paths going to the header of the nested loop. The initial control point is noted p_1, the one corresponding to line 3 is noted p_3, and the inner loop header is noted p_8.

Guided static analysis yields, at line 3, x_old $\in (-\infty, +\infty)$. Path-focusing also finds x_old $\in (-\infty, +\infty)$. Now, let us see how our technique performs on this example.

Figure 4.12 shows the sequence of subset of paths during the analysis. The points in P_R are noted p_i, where i is the corresponding line in the code: for instance, p_3 corresponds to the header of the main loop.

1. The starting subgraph is depicted in Figure 4.12 Step 1. In the beginning, this graph has no transitions: $\mathcal{P} = \emptyset$.

2. We compute the new feasible paths that have to be added into the subgraph. We first find the path from p_1 to p_3 and obtain at p_3 x_old $= 0$. The image of x_old $= 0$ by the path that goes from p_3 to p_8, and that goes through the *else* branch of each *if-then-else*, is $-10 \leq$ x_old ≤ 10. This path is then added to our subgraph. Moreover, there is no other path starting at p_3 whose image is not in $-10 \leq$x_old≤ 10. Finally, since the abstract value associated to p_8 is $-10 \leq$x_old≤ 10, the path from p_8 to p_3 is feasible and is added into P. The final subgraph is depicted in Figure 4.12 Step 2.

3. We then compute the ascending iterations by path-focusing. At the end of these iterations, we obtain $-\infty \leq$x_old$\leq +\infty$ for both p_3 and p_8.

4. We now can apply narrowing iterations, and recover the precision lost by widening: we obtain $-10000 \leq$x_old≤ 10000 at points p_3 and p_8.

5. Finally, we compute the next subgraph. The SMT-solver does not find any new path that makes the abstract values grow, and the algorithm terminates.

Our technique gives us the expected invariant x_old$\in [-10000, 10000]$. Here, only 3 paths out of the 6 have been considered during the analysis. In particular, the self loop at line 8 is never chosen and thus the narrowing behaves better. In practice, depending on the order the SMT-solver returns the paths, other feasible paths could have been added during the analysis.

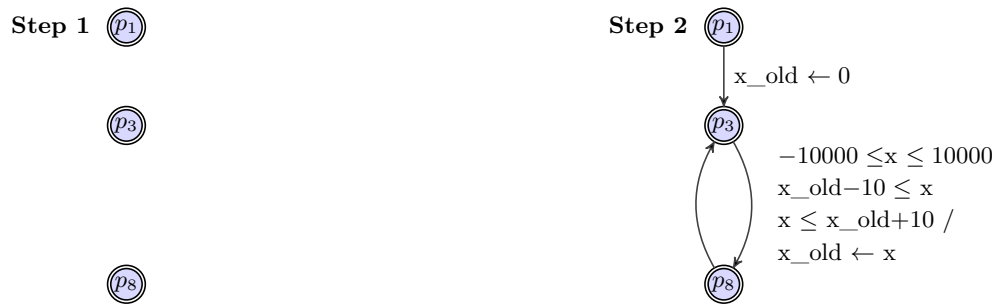


Figure 4.12: Ascending sequence of subgraphs

In this example, we see that our technique actually combines best of *Guided Static Analysis* and *Path Focusing*, and is more precise than the intersection of the invariants computed by the two techniques separately.

Improvements

Here, we described reasonable improvements of our approach, that have been studied but not implemented nor experimented in this thesis.

Dynamic Choice of the Cutting Points The more elements in the set of *Cutting points* P_R, the more least upper bound operations will be computed. An improvement of our method consists in the use of a different set of Cutting points for each working subgraph. The set P_R is still used for computing the new paths, but once a subgraph $\mathcal{P}$ has been chosen, one can define a correct set of Cutting points $P_R^{\mathcal{P}}$ for this particular subgraph. In general, this set $P_R^{\mathcal{P}}$ may be included in P_R, thus avoiding least upper bounds at some control points in $P_R/P_R^{\mathcal{P}}$. Having a different set of Cutting points implies that a new $\rho_{\mathcal{P}}$ formula has to be computed each time, and thus may degrade the performance of the analysis in terms of computation time. Indeed, keeping the same ρ_P formula in the whole analysis takes high benefits of incremental SMT solvers, that remember learned clauses between two "only slightly different" SMT checks. This benefit is lost when modifying the ρ_P formula, since the structure of the formula differs. This improvement has not been implemented in this thesis and its behavior compared to the unimproved approach is not known.

Focusing on a *very different* Path In the *Path-Focusing* algorithm from [MG11], the SMT queries are used for focusing on one path in the expanded graph for which there is a reachable state outside the current invariant candidate. There may exist a very important number of such paths, and the number of iterations before convergence highly depends on the order in which the paths are returned by the SMT solver, used as black-box. Suppose for instance we apply *Path-Focusing* on the graph depicted in Figure 4.13, where there are N paths, noted $\pi_1, \ldots, \pi_N$, with N very big, and for each $i \leq N$, π_i increments a variable x by i. x is initialized to 0 and the abstract domain is intervals. In the final control point, the invariant should be $x \in [1, N]$. This invariant may be reached in 2 iterations by focusing to π_1 and π_N only. However, the SMT solver may also return successively π_1, π_2, etc. until π_N, thus leading to N iterations. This is a problem since N can be very big – e.g. a program with a succession of 10 *if-then-else* leads to 2^{10} distinct paths in the expanded graph –. We show later in chapter 6 that this problem occurs in real-life and useful applications, such as the estimation of Worst-Case Execution Time (WCET) or energy consumption. Here, the use of a widening is not a solution, since the narrowing phase would lead to the same problem.

It is possible to work around this problem by using a variant of *Optimization Modulo Theory* [ST12, LAK+14]: informally, instead of searching for a path that just enlarges the abstract value, one can search for a path that gives a point *far* from the current abstract value.

We give here a heuristic for usual numerical abstract domains (e.g. intervals, octagons, polyhedra), illustrated by Figure 4.14 with a convex polyhedron. In these domains, abstract values can be represented by a finite number of linear inequalities. When the abstract value is not an invariant, it means there is at least one these inequalities which is not invariant. In *Path-Focusing*, the assignment of the numerical variables in the model, noted $\mathcal{M}_1$, given by the SMT solver directly gives the set of falsified constraints. One can pick one of these constraints $(C_0) : a_1x_1 + a_2x_2 + \cdots + a_kx_k \leq C$, and compute the distance d_1 between (C_0) and the model $\mathcal{M}_1$, that assigns the x_i's to $\mathcal{M}_1(x_i)$, using the usual formula:

$$d_1 = \frac{|a_1.\mathcal{M}_1(x_1) + \cdots + a_k.\mathcal{M}_1(x_k) - C|}{\sqrt{a_1^2 + \cdots + a_k^2}}$$

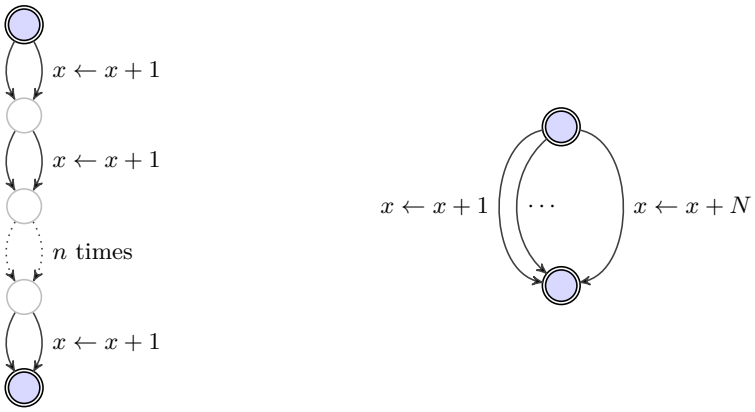


Figure 4.13: In the left, a program with 2^n distinct paths. The forward transformers of these paths are of the form $x \leftarrow x + i, i \in [0, n]$. In the right side, a simpler control flow graph with N paths, where the i-th path is labelled with $x \leftarrow x + i$.

Then, one can transform (C_0) into $(C_1) : a_1x_1 + a_2x_2 + \cdots + a_kx_k \leq C + d_1$, and use SMT solving to find a new model that does not satisfy (C_1) by checking the satisfiability of the following formula:

$$\rho_P \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s \wedge X_i^s \wedge b_j^d \wedge \neg primed(C_1) \wedge \mathcal{P}$$

If the formula is *sat*, we get a new model $\mathcal{M}_2$ whose distance from C_0 is at least twice bigger than $\mathcal{M}_1$, and we can iterate by constructing a constraint C_2 and restart.

If the formula is *unsat*, we focus on the path given by the satisfiable assignment of the previous formula.

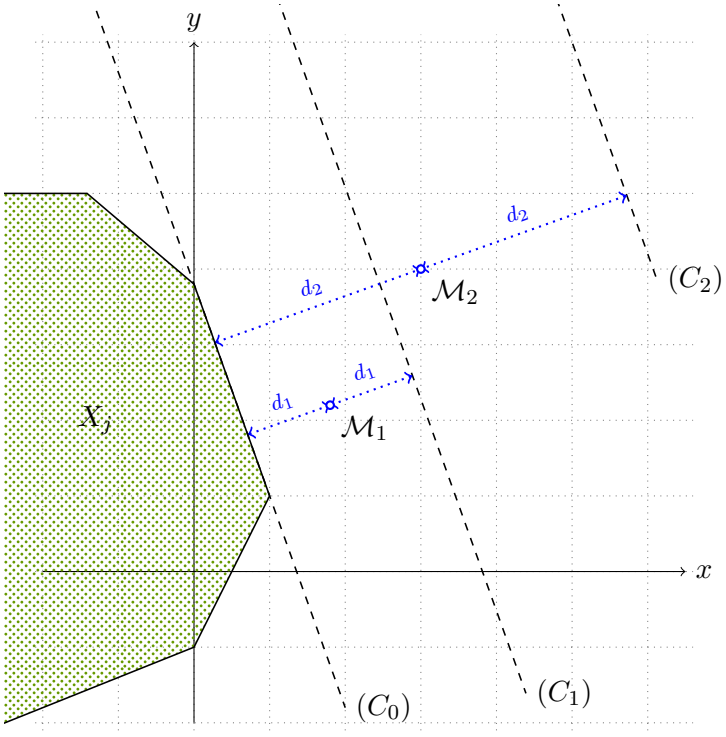


Figure 4.14: Search for a model *far* from the face (C_0) of X_j.

For the example in Figure 4.13 with N paths, this approach guarantees that the *Path-Focusing* will focus on at most $\log(N)$ paths instead of N. It is unclear whether this optimization makes sense in practice: future work includes experimenting with this approach to see if it improves the overall analysis in time.

4.3 Using a more Expressive Abstract Domain

4.3.1 Guided Path Analysis with Disjunctive Invariants

While many (most?) useful program invariants on numerical variables can be expressed as conjunctions of inequalities and congruences, it is sometimes necessary to introduce disjunctions: the invariant is then not only a single abstract value but a union thereof, which increases the expressiveness. For instance, the loop **for** (**int** i=0; i<n; i++) {...} verifies $0 \leq i \leq n \vee (i = 0 \wedge n < 0)$ at the loop header. For this very simple example, a simple syntactic transformation of the control structure (into **int** i=0; **if** (i<n) **do** {...} **while** (i<n)) is sufficient, but in more complex cases, more advanced analyses are required [BSIG09, Jea03, SDDA11, MB11]; in intuitive terms, they discover *phases* or *modes* in loops.

[GZ10] proposed a technique for computing disjunctive invariants, by distinguishing all the paths inside a loop. In this section, we propose to improve this technique by using SMT queries to find interesting paths, the objective being to avoid an explicit exhaustive enumeration of an exponential number of paths.

For each control point p_i, we compute a disjunctive invariant $\bigvee_{1 \leq j \leq m_i} X_{i,j}$. We denote by n_i the number of distinct paths starting at p_i. To perform the analysis, one chooses an integer $\delta_i \in [1, m_i]$, and a mapping function $\sigma_i : [1, m_i] \times [1, n_i] \mapsto [1, m_i]$. The mapping function determines which disjunct of the abstract value to join with, depending on the disjunct we started from as well as the path being taken. The k-th path starting from p_i is denoted $\tau_{i,k}$. The image of the j-th disjunct $X_{i,j}$ by the path $\tau_{i,k}$ is then joined with $X_{i,\sigma(j,k)}$. Initially, the δ_i-th abstract value contains the initial states of p_i, and all other abstract values contain $\emptyset$.

For each control point $p_i \in P_R$, m_i, δ_i and σ_i can be defined heuristically. For instance, one could define σ_i so that $\sigma_i(j, k)$ only depends on the last transition of the path, or else construct it dynamically during the analysis.

Our method improves this technique in two ways :

- Instead of enumerating the whole set of paths, we keep them implicit and compute them only when needed.
- At each loop iteration of the original algorithm [GZ10], one has to compute for each disjunct of the invariant candidate **and** for each path an image by the corresponding abstract forward transformer. Yet, many of these images may be redundant: for instance, there is no point enumerating disjuncts and paths that yield abstract values already included in the current candidate invariant. In our approach, we compute such an image only if it makes the resulting abstract value grow.

Our improvement consists in a modification of the SMT formula we solve in 4.2.2. We introduce in this formula Boolean variables $\{d_j, 1 \leq j \leq m\}$, so that we can easily find in the model which abstract value of the disjunction of the source point has to be picked to make the invariant candidate of the destination grow. The resulting formula that is given to the SMT solver is defined by $g(p_i)$:

$$g(p_i) = \overbrace{\rho_P \wedge b_i^s \wedge \bigwedge_{\substack{j \in P_R \\ j \neq i}} \neg b_j^s}^{\text{same as in } f(p_i)} \wedge \overbrace{\bigvee_{1 \leq k \leq m_i} (d_k \wedge X_{i,k} \wedge \bigwedge_{l \neq k} \neg d_l)}^{\text{starting value is in the } k\text{-th disjunct}}$$
$$\wedge \bigvee_{j \in Succ(i)} \underbrace{(b_j^d \wedge \bigwedge_{1 \leq k \leq m_i} (\neg primed(X_{j,k})))}_{\substack{\text{there is a } b_j^d \text{ for which} \\ \text{the final values are not in any disjunct}}}$$

When the formula is satisfiable, we know that the index j of the starting disjunct that has to be chosen is the one for which the associated Boolean value d_j is *true* in the model. Then, we can easily compute the value of $\sigma_i(j,k)$, thus know the index of the disjunct to join with.

In our algorithm, the initialization of the abstract values slightly differs from algorithm 2 line 5, since we now have to initialize each disjunct. Instead of Line 5, we initialize $X_{i,k}$ with $\bot$ for all $k \in \{1,..,m_i\} \setminus \{\delta_i\}$, and X_{i,δ_i} with $\leftarrow I_{p_i}$.

Furthermore, the Path-Focusing algorithm (line 19 from algorithm 2) is enhanced to deal with disjunctive invariants, and is detailed in algorithm 4.

The *Update* function can classically assign $X_{i,\sigma_i(j,k)} \nabla (X_{i,\sigma_i(j,k)} \sqcup \tau_{i,k}(X_{i,j}))$ to $X_{i,\sigma_i(j,k)}$, or can integrate the special treatment for self loops proposed by [MG11], with widening/narrowing sequence or acceleration.

Algorithm 4 Disjunctive invariant computation with implicit paths

```
1: while true do
2:     res ← SmtSolve [g(p_i)]
3:     if res = unsat then
4:         break
5:     end if
6:     Compute the path τ_{i,k} from res
7:     Take j ∈ {l | d_l = true}
8:     Update(X_{i,σ_i(j,k)})
9: end while
```

We experimented with a heuristic of dynamic construction of the σ_i functions, adapted from [GZ10]. For each control point $p_i \in P_R$, we start with one single disjunct ($m_i = 1$) and define $\delta_i = 1$. M denotes an upper bound on the number of disjuncts per control point.

The σ_i functions take as parameters the index of the starting abstract value, and the path we focus on. Since we dynamically construct these functions during the analysis, we store their already computed image into a compact representation, such as Algebraic Decision Diagrams (ADD) [BFG+97]. These ADD's extend Binary Decision Diagrams by attaching to the terminal nodes not only a Boolean value, but an element of an arbitrary domain. In our case, this element is an integer value expressing the index of the disjunct to join with, depending on the focused path and the starting disjunct.

$\sigma_i(j,k)$ is then constructed on the fly only when needed, and computed only once. When the value of $\sigma_i(j,k)$ is required but undefined, we first compute the image of the abstract value $X_{i,j}$ by the path indexed by k, and try to find an existing disjunct of index j' so that the least upper bound of the two abstract values is exactly their union (using SMT-solving). If such an index exists, then we set $\sigma_i(j,k) = j'$. Otherwise:

- if $m_i < M$, we increase m_i by 1 and define $\sigma_i(j,k) = m_i$,
- if $m_i = M$, the simplest heuristic is to set $\sigma_i(j,k)$ to M, even though this choice is clearly not optimal. Our current implementation is using this setting for simplicity, and we already improve results on many examples. We can suggest probably better heuristics that we would like to try and experiment with, for instance:
  - join with the abstract value that has the most constraints in common.
  - join with a disjunct that does not increase the dimension of the abstract value.

The main difference with the original algorithm [GZ10] is that we construct $\sigma_i(j,k)$ using SMT queries instead of enumerating a possibly exponential number of paths to find a solution.

4.3.2 Comparison between usual Numerical Abstract Domains

An evident way of improving precision of abstract interpretation is the use of a more expressive abstract domain. For computing numerical invariants, it is intuitively more precise to use for instance octagons rather than intervals for inferring invariants. However, because of the non monotonicity of the widening operator, it may happen that a supposedly more precise abstract domain gives worse results than a simpler one. For this reason, one should experiment with the various abstract domains on real code and see how they behave in practice in terms of precision and timing consumption. We propose in this section some experimental results obtained with various usual numerical abstract domains, with our analyzer Pagai (full details on the tool are in chapter 7).

We compared by inclusion the results of standard abstract interpretation with the domains of intervals (**BOX**), octagons (**OCT**), convex polyhedra (**PK** and **PPL**), the reduced product of convex polyhedra and linear congruences (**PKGRID**), and finally convex polyhedra with Parma widening (**PPL_B**). Recall that Parma widening is explained page 39. All these abstract domains are provided by the APRON library [JM09]. Note that **PK** and **PPL**[BHZ] are two different implementations of convex polyhedra. Parma widening is only available with the second implementation, hence only the comparison between **PPL** and **PPL_B** makes sense.

Our experiments were conducted on a panel of well known GNU projects with the original abstract interpretation algorithm [CC77, CH78]. Some of the projects contain many arithmetical computations, in particular scientific libraries like libgsl, libglpk or libsuperlu. We compared the precision of abstract domains by pair and give the results in Table 4.2. In a given column "D1 // D2", where D1, D2 $\in$ { **BOX**, **OCT**, **PK**, **PKGRID**, **PPL**, **PPL_B**}:

- $\sqsubset$ gives the number of loop headers where the invariant obtained with D1 is strictly more precise than the one obtained with D2,
- $\sqsupset$ counts the number of loops headers where D2 is strictly more precise than D1,
- $=$ counts the number of loops headers where D1 and D2 have the same result,
- $\neq$ counts the number of loops headers where D1 and D2 have incomparable result, meaning that none invariant includes the other.

In the table, D1 is always more expressive (thus supposedly equally or more precise) than D2: the $\sqsubset$ should then intuitively contain numbers much greater than in column $\sqsupset$ and $\neq$.

Let us briefly comment the results: on these benchmarks, many loop headers are actually simple and a static analysis with intervals is sufficient. This explains the high number of cases

Benchmark	PK // OCT				PK // BOX				OCT // BOX				PKGRID // PK				PPL_B // PPL			
	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$
libgsl	421	81	2327	17	1276	36	1486	48	1149	1	1681	20	106	0	2744	1	-	-	-	-
grep	11	5	276	0	43	8	236	5	46	5	238	3	4	0	288	0	26	1	265	0
libsuperlu	84	10	587	2	301	2	377	3	306	0	377	0	7	0	676	0	158	2	523	0
tar	21	3	524	0	54	6	487	1	47	4	497	0	15	1	532	0	28	1	519	0
libglpk	168	62	2235	39	666	42	1718	78	690	22	1783	9	60	3	2441	0	221	19	2245	19
libgmp	44	14	158	6	82	9	119	12	71	0	149	2	23	0	197	2	26	3	193	0
gmugo	86	110	1451	21	419	65	1120	64	459	6	1196	7	-	-	-	-	330	21	1292	25
libjpeg	48	3	382	1	134	2	298	0	130	0	304	0	13	0	421	0	61	0	373	0
sed	5	0	80	0	10	1	74	0	10	1	74	0	1	0	84	0	3	0	82	0
gzip	26	9	189	6	54	2	164	10	45	2	177	6	5	0	225	0	31	8	182	9
wget	28	15	411	3	59	14	380	4	58	1	398	0	13	0	442	2	55	2	400	0
libpng16	29	5	350	6	107	2	277	4	100	0	289	1	52	0	337	1	56	1	332	1
TOTAL	971	317	8970	101	3205	189	6736	229	3111	42	7163	48	299	4	8387	6	995	58	6406	54

Benchmark	PK // OCT				PK // BOX				OCT // BOX				PKGRID // PK				PPL_B // PPL			
	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$
libgsl	15%	3%	82%	1%	45%	1%	52%	2%	40%	0%	59%	1%	4%	0%	96%	0%	-	-	-	-
grep	4%	2%	95%	0%	15%	3%	81%	2%	16%	2%	82%	1%	1%	0%	99%	0%	9%	0%	91%	0%
libsuperlu	12%	1%	86%	0%	44%	0%	55%	0%	45%	0%	55%	0%	1%	0%	99%	0%	23%	0%	77%	0%
tar	4%	1%	96%	0%	10%	1%	89%	0%	9%	1%	91%	0%	3%	0%	97%	0%	5%	0%	95%	0%
libglpk	7%	2%	89%	2%	27%	2%	69%	3%	28%	1%	71%	0%	2%	0%	97%	0%	9%	1%	90%	1%
libgmp	20%	6%	71%	3%	37%	4%	54%	5%	32%	0%	67%	1%	10%	0%	89%	1%	12%	1%	87%	0%
gmugo	5%	7%	87%	1%	25%	4%	67%	4%	28%	0%	72%	0%	-	-	-	-	20%	1%	77%	1%
libjpeg	11%	1%	88%	0%	31%	0%	69%	0%	30%	0%	70%	0%	3%	0%	97%	0%	14%	0%	86%	0%
sed	6%	0%	94%	0%	12%	1%	87%	0%	12%	1%	87%	0%	1%	0%	99%	0%	4%	0%	96%	0%
gzip	11%	4%	82%	3%	23%	1%	71%	4%	20%	1%	77%	3%	2%	0%	98%	0%	13%	3%	79%	4%
wget	6%	3%	90%	1%	13%	3%	83%	1%	13%	0%	87%	0%	3%	0%	97%	0%	12%	0%	88%	0%
libpng16	7%	1%	90%	2%	27%	1%	71%	1%	26%	0%	74%	0%	13%	0%	86%	0%	14%	0%	85%	0%
TOTAL	9%	3%	87%	1%	31%	2%	65%	2%	30%	0%	69%	0%	3%	0%	96%	0%	13%	1%	85%	1%

Table 4.2: Comparison between usual numerical abstract domains. Cells with '-' indicate timeout for at least one of the two compared domains. The first table indicates the exact number of control points, while the second table gives the proportion in percentage.

Benchmark	BOX	PKEQ	PPL	PPL_B	PKGRID	PK	OCT
libgsl	7.5	36.9	-	-	219.5	179.7	19.0
grep	1.8	5.3	9.5	19.9	28.8	5.5	2.3
libsuperlu	1.9	6.4	13.5	28.1	75.6	9.3	3.9
tar	2.4	7.7	10.2	12.7	32.8	6.3	1.8
libglpk	11.6	41.4	56.7	89.3	223.4	38.7	15.1
libgmp	2.2	3.8	21.1	21.0	59.4	8.3	3.7
gnugo	9.6	45.9	86.9	183.2	-	73.9	16.6
libjpeg	1.3	4.8	6.1	9.9	34.2	6.6	2.5
sed	0.6	1.4	5.8	8.7	13.8	3.5	0.8
gzip	0.9	5.2	12.5	20.6	37.6	5.8	1.6
wget	2.0	6.6	15.5	26.7	68.5	10.2	4.3
libpng16	1.4	5.9	10.6	20.0	43.6	9.5	2.6
TOTAL	43.2	171.3	>248.4	>440.1	>837.2	357.3	74.2

Table 4.3: Timing comparison between numerical abstract domains. Results are shown in seconds. Cells with '-' indicate timeout or missing data.

where **BOX** equals more complex domains like **OCT** or **PK**. For more complicated loops, both octagons and polyhedra are powerful. However, one should note that convex polyhedra quite regularly yields worse results than intervals (up to 4% of the cases) or octagons (up to 7%).

Another noticeable result is the very good results of the Parma widening (see the column **PPL_B // PPL**), that improves on the standard polyhedra widening in up to 23% of the cases, while behaving worse very rarely. More generally, one can see that standard polyhedra widening regularly yields less precise result than simpler domains (**OCT**, **BOX**): on gnugo, **OCT** is even more precise in overall. This is a somehow disappointing result that motivates the use of smarter widening operators or better iteration strategies when using this domain.

Table 4.3 gives the timing consumption of the analysis with the different domains. As expected, the more expressive the domain, the more costly it is in practice. We should note that **OCT** has reasonable overhead compared to **BOX** (around $\times 2$). Polyhedra cost around $5\times$ more than octagons, and $9\times$ more than intervals. Finally, we should also notice than even though Parma widening is good in terms of precision, it induces an significant overhead compared to classical widening (around $\times 2$).

To conclude, it is interesting to see how well the various abstract domains behave on real examples, since their theoretical complexity and their actual performance in practice may be substantially different. It is also important to compare their precision on real examples, since their widening operators are non-monotonic. These experiments have been conducted on chosen GNU projects: we guess that the results may significantly vary depending on the input programs. In particular, we did not compare the abstract domains on safety critical software (e.g. avionics code), because of a lack of this kind of benchmarks. Another interesting experiments include comparison of the abstract domains with other analysis techniques, e.g. guided path analysis instead of standard abstract interpretation.

4.4 Conclusion

In this chapter, we proposed several methods for improving the precision of state-of-the-art abstract interpretation-based algorithms. We first showed that it is possible to improve an already discovered fixpoint by restarting a new ascending/descending sequence with a smart choice of the initial abstract value. This new approach gives interesting results on real code, in particular in the presence of nested loops. To our knowledge, this was the first attempt in the literature for improving the result **after** the fixpoint has been reached: existing approaches were only focusing on precision **during** fixpoint computation. This work motivated other work in the same line [AS13].

We also presented an original SMT-based method, called Guided Path Analysis, aimed at avoiding least upper bound operations at many program points, combined with a smart iteration strategy that limits the bad effects of widenings. Given an abstract domain, this algorithm yields in general much better results than standard abstract interpretation with the same abstract domain. In this thesis, we only used SMT solving to represent precisely loop-free program paths. We should note that the SMT encoding we are using could also be used for automating the construction of abstract forward transformers. For instance, [TR12b] uses a very similar algorithm to ours for deriving invariants without the need to define the abstract transformers. Experimental results that show the efficiency of this contribution will be detailed further in chapter 7.

All these analysis are intra-procedural. This is an important limitation in the sense that rough approximations are applied at function call sites, and lead to many false alarms. In addition, SMT based approaches may suffer from scalability problems in case the analyzed function has a huge size (e.g. after inlining). We propose in the next chapter to deal with these two issues.

Chapter 5

Modular Static Analysis

In the previous chapter, we proposed an intraprocedural static analysis by abstract interpretation, that uses bounded model checking for limiting the loss of precision due to least upper bound operations. The intraprocedural issue can be partially addressed by function inlining, at the expense of a blowup in the size of the code. When such inlining is applied, experiments show that the resulting SMT formulas have a large size and are hard to solve even by the state-of-the art solvers.

A classical way of doing interprocedural analysis is to compute *summaries* for functions, that provide relations between their input and output variables [SP81, GT07, Cla77]. These summaries are then used whenever the function is called.

In this chapter, we propose a framework for a modular and interprocedural summary-based analysis of a program, given some properties to check. The main idea is to compute rough overapproximations of the different parts of the program, and improve these overapproximations incrementally until we find sufficiently precise invariants to prove the given properties.

We first decompose the program into *blocks*, defined in section 5.1, which abstract parts of the program, such as loops or function calls. We obtain a loop-free program on which we can use model checking by SMT-solving to discover traces that violate the properties of interest. Once such a violating trace is discovered, we try to prune it by recursively refining the abstractions of the blocks the trace goes through. We can use state-of-the art abstract interpretation techniques for computing the *summaries* of abstracted blocks, also called *input/output relations*.

The organization of the chapter is summarized in Figure 5.1: First, in section 5.1, we detail our decomposition of program into abstraction blocks to be analyzed separately. This section gives the preliminaries for the next sections. In section 5.2, we illustrate on an example the principle of the algorithms described in the chapter, in particular our Property Guided Path Focusing from section 5.5. In section 5.3, we describe our simpler extension of the path focusing algorithm for computing summaries. This section is written as a preliminary of section 5.4, where we improve the algorithm so that the computed summaries are reusable in other calling contexts. These algorithms are intended to compute precise summaries for every program parts with any particular objective. Then, we propose in section 5.5 an algorithm for verifying a given user-defined property, using successive refinements of the summaries which are guided by error traces. This last algorithm is theoretically described in this thesis and its implementation is part of future work. Finally, section 5.6 explains how to infer preconditions to functions (or loops) based on this last algorithm.

This chapter therefore presents three contributions:

- an algorithm, called *Modular Path Focusing*, for computing both precise and general *summaries* in section 5.4,
- in section 5.5, an extension of this algorithm, called *Property-Guided Path Focusing*, that

aims at proving user-given properties by lazily computing summaries,

- a procedure for generating function/loop preconditions, based on *Property-Guided Path Focusing*.

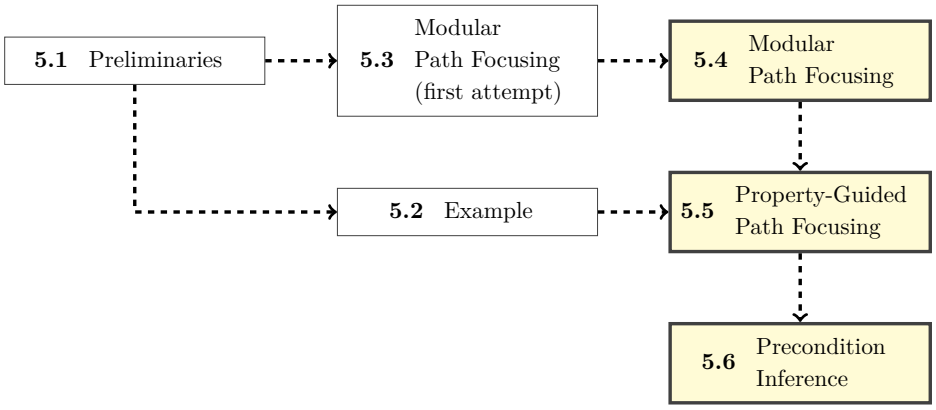


Figure 5.1: Organization of the chapter, and dependencies between sections

The implementation and experimentations of the techniques described here are current work: at present, only section 5.3 has been implemented in our tool Pagai, which is not sufficient to conduct interesting experiments.

5.1 Block Decomposition

We suppose that the program is defined by a set of functions/procedures, each of them described by their control flow graph.

Instead of doing the analysis over these graphs, we summarize some parts (subgraphs) in order to analyze them modularly. We thus obtain new graphs, where the nodes are called *blocks*, with some of these blocks being the abstraction of a subgraph of the original one. The decomposition is recursive: e.g. a function with nested loops will be composed of abstraction blocks that themselves contain abstraction blocks.

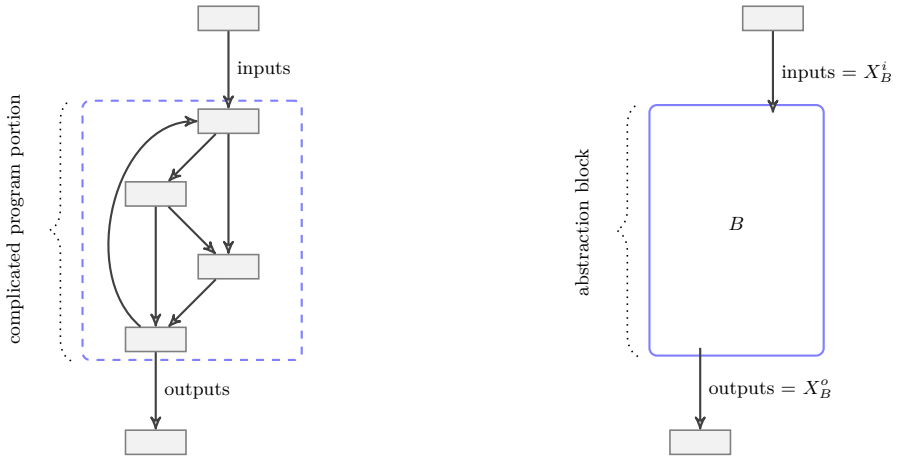


Figure 5.2: Abstracting a program part into a *Block*, with same inputs and outputs

In practice, a block may abstract:

- a basic block.
- a loop or a function. In this case, the block actually abstracts a graph of blocks (as in Figure 5.2).
- a function call.
- an abstraction of a complex program instruction, such as non-linear and/or floating point operation.
- an abstraction of several paths. Such abstraction is not mandatory, but improves scalability since the complex behavior of every paths will be summarized by a simple formula.

The nodes of the control flow graph are basic blocks. These basic blocks are lists of instructions. The semantics of each instruction has to be overapproximated by a logical formula for its SMT encoding (as we already explained in subsection 4.2.1), and an abstract forward transformer used in the abstract interpretation. Given an abstract domain, some instructions are simply assigned a "good" abstract transformer: for instance, with convex polyhedra, one can reasonably define an abstract transformer for an addition instruction over integers. We call them *simple* instructions. Conversely, other instructions could be assigned different abstract transformers: for instance, the transformer for a function call can be defined in multiple ways, depending on which technique we use to compute the *summary* of the function. Another example could be the multiplication instruction, for which the abstract transformer could be defined from different linearization techniques [Min04, STDG12].

For these *non-simple* instructions, there are several reasonable abstract transformers that may not be comparable in terms of precision. For that reason, a static analysis could start with an initial abstract transformer for the instruction, and compute a more precise transformer during the analysis if needed. In our implementation, we consider that the only non-simple instruction is the function call. The other instructions are assigned a chosen abstract transformer that never changes during the analysis.

We suppose that these *non-simple* instructions are isolated in dedicated basic blocks, for which the instruction is the only single one. This is not a restriction, since any CFG can be simply transformed to fit this property by just splitting the basic blocks appropriately.

We define our notion of *blocks*:

Definition 20 (Block)**.** Given a control-flow graph of basic blocks, a *block* is either:

- a basic block that only contains *simple* instructions. In this case, the block is called **concrete**.
- an **abstraction** block, that can be:
  - a basic block containing only a *non-simple* instruction (e.g a function call, followed by an unconditional branch.
  - a strongly connected component of the graph, that contains more than one basic block. In practice, this block abstracts the effect of a loop in the program.
  - a subgraph that contains every basic block between a chosen block and one of its dominators. Recall that a basic block d is a dominator of b if every path going through b also goes through d.

Notations We define here the notations used throughout this chapter. Each abstraction block B has a set of **input variables**, noted X_B^i, as well as a set of **output variables**, noted X_B^o. We will note $X_B^{i/o}$ the set $X_B^i \cup X_B^o$. We always suppose that $X_B^i \cap X_B^o = \emptyset$. This is not a restriction, since we can use an "SSA-like" transformation that provides this guarantee.

For instance, if B abstracts a function, the set X_B^i typically contains the global variables and function arguments. X_B^o contains the returned value, the global variables and one variable per argument passed by reference. The global variables should be considered both as input and output, since they may be modified inside the function. Note that a pre-analysis could eliminate some of these global variables if the function is shown to only read/store certain fragments of the memory.

In this chapter, we define and use logical formulas and abstract values with different dimensions: some abstract values have as dimensions only input variables, some others have both input and output variables, etc. For clarity, we note in parenthesis the dimensions of the abstract values (or free variables in SMT formulas): for instance, $A(X_B^i)$ is an abstract value with X_B^i as dimensions, i.e. the input variables of block B. In some cases where there is no ambiguity in the dimensions, they may not be explicitly written.

Once the notion of block is defined, as well as the input and output variables for a block, we can define the notion of input/output relation:

Definition 21 (Input/Output Relation (I/O))**.** Let B be a block. An *input/output relation* (also called *summary*) is a formula $R_B(X_B^{i/o})$ involving the variables in $X_B^{i/o}$, that safely abstracts the behavior of the program when going through B. This I/O relation can either be a correct over-approximation for any input, or for a particular *Context* noted $C_B(X_B^i)$. In the last case, the formula $C_B(X_B^i) \Rightarrow R_B(X_B^{i/o})$ holds.

Example 11 (absolute value)**.** Suppose we have a function that takes as input a variable x and returns its absolute value $r = |x|$. Examples of input/output relations for the function are:

$$\begin{aligned} x \geq 0 &\Rightarrow r = x \\ x < 0 &\Rightarrow r = -x \\ \text{true} &\Rightarrow r \geq x \wedge r \geq -x \end{aligned}$$

5.1.1 Graph of Blocks

In this chapter, our algorithms operate on control flow graphs of blocks instead of basic blocks. We shall thus describe our procedure for building the graph of blocks from the usual CFG of the program.

Let G be a control flow graph, where the nodes are basic blocks. We built the associate control flow graph of blocks in the following way:

- Compute the strongly connected components (we call them SCCs) of the graph (using Tarjan's algorithm for instance).
- for each SCC,
  - if it contains one single block, this basic block is a block according to definition 20.
  - if it contains several basic blocks, recursively construct a block from this component of the graph. The block is then the abstraction of the graph of nested blocks derived from this SCC.
- We build the graph of blocks from the control flow graph, where each SCC is replaced with its corresponding block. There is a transition between two blocks if there is a basic block in the first SCC that has a transition to a block in the second SCC.

Figure 5.3 illustrates the decomposition into blocks for a simple control flow graph.

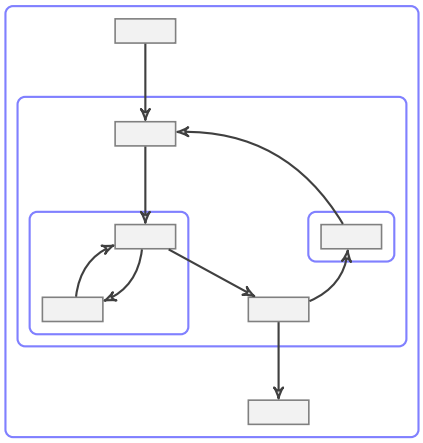


Figure 5.3: Correct decomposition of a control flow graph into Blocks. The main block abstracts the function. Two blocks are abstracting loops. The smaller block abstracts a basic block that contains a *non-simple* instruction.

Abstraction of Several Paths

A code without loops but many paths (typically, a large number of successive or nested if-then-else) may lead in the further SMT encoding to formulas that make the solver blow up. This issue is explained in detail in chapter 6. In this case, we can create blocks that abstract a set of paths, in order to compute input/output relations that safely overapproximate the behavior of any paths. These relations will possibly prevent the SMT solver from enumerating many paths that are all unfeasible for a similar reason.

Definition 22 (immediate dominator)**.** The immediate dominator of a block B, noted $dom(B)$, is a dominator of B such that there is no block between $dom(B)$ and B that dominates B.

We can construct new abstraction blocks in the following way:

- We choose a block B in the graph that has several incoming transitions
- We compute the immediate dominator $dom(B)$ of this block.
- We create the block that contains block B, its dominator, and every block B' for which there is a path from $dom(B)$ to B that goes through B'.

Relation with Horn Clause Encoding Our decomposition of a program into blocks is closely related to the idea of encoding a program into Horn clauses [GLPR12]. Each non-interpreted function would correspond to a block, and choosing the set of blocks is related to the choice of the uninterpreted functions in the Horn clauses. Our approach can then be seen as a technique operating on Horn clauses (which may either prove the desired property or say "I don't know"), and could probably be combined with existing approaches [HBdM11, BMR13].

5.2 Introductory Example

We take as an example the C program in Figure 5.4, that we decompose into blocks as explained in the previous section. We are interested in proving the assert statement $y \geq 0$ of the function f. This example illustrates the algorithm described later in section 5.5.

```
1 int input(int min, int max) {
2   int x;
3   assume(x >= min && x <= max);
4   return x;
5 }
6
7 int f(int N) {
8   assume(0 <= N && N <= 10);
9   int x;
10  int y = input(0,100);
11
12  while (true) {
13    assert(y >= 0);
14    x = input(10-N,10+N);
15    if (x > y)
16      y++;
17    if (x < y)
18      y--;
19  }
20 }
```

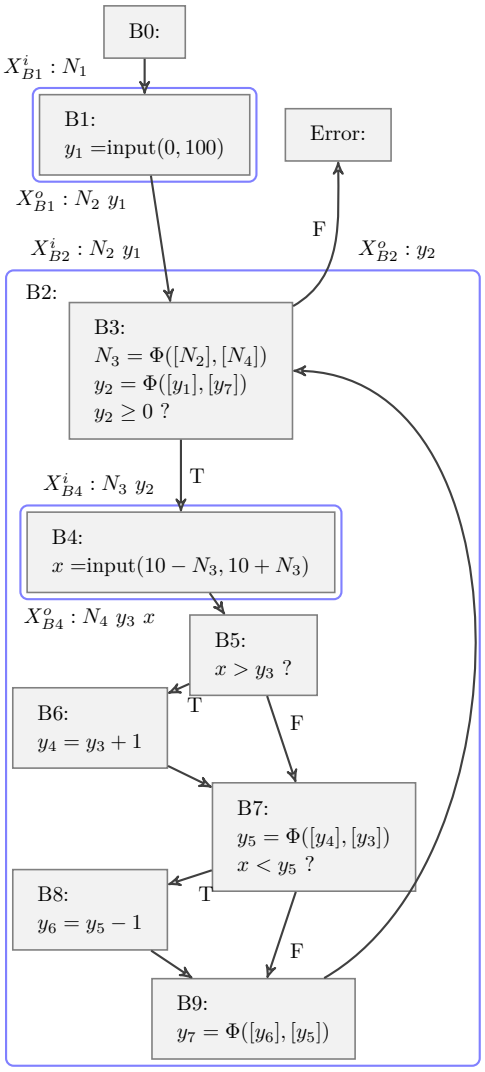


Figure 5.4: An example C program, with its decomposition into blocks. Each block is labeled, from $B0$ to $B9$, and $Error$.

The abstraction blocks are here $B1$, $B2$ and $B4$ (see Figure 5.4). Their input/output variables are $X^i_{B1} = \{N_1\}$, $X^o_{B1} = \{N_2, y_1\}$, $X^i_{B2} = \{N_2, y_1\}$, $X^o_{B2} = \{y_2\}$, $X^i_{B4} = \{N_3, y_2\}$, $X^o_{B4} = \{N_4, y_3, x\}$.

Initially, each abstraction block is assigned a rough input/output relation. Basically, the initial relations are only expressing the equalities between the input and output variants of the same variables if not modified inside the block. The input function is initially supposed to return a nondeterministic integer, since it has not been analyzed yet. The initial relations are then:

$$R_{B1}(N_1, N_2, y_1) \stackrel{\text{def}}{=} (N_1 = N_2)$$
$$R_{B2}(N2, y_1, y_2) \stackrel{\text{def}}{=} true$$
$$R_{B4}(N_3, y_2, N_4, y_3, x) \stackrel{\text{def}}{=} (N_3 = N_4 \wedge y_2 = y_3)$$

Note that some of the variables have been duplicated to verify the required property, that for each block B, $B^i \cap B^o = \emptyset$: for instance, variable N (resp. y_2) corresponds to N_3 at the entry point of block B_4, and to N_4 (resp. y_3) at its exit point.

We construct the SMT formula associated to function f and check with an SMT solver whether the `Error` state is reachable, in a similar way as it is done in bounded model checking and section 4.2. We find the error trace:

$$B0 \overset{N_1=10}{\longrightarrow} B1 \overset{y_1=42,N_2=10}{\longrightarrow} B2 \overset{y_2=-1}{\longrightarrow} Error \tag{5.1}$$

The error trace would possibly be unfeasible with more precise relations for the nested blocks. Thus, one can try to refine the input/output relation of block $B1$. A simple analysis of the input function gives the relation $min \leq ret \leq max$, where ret is the returned value. Applied to our case in block $B1$, we get $0 \leq y_1 \leq 100$. The relation R_{B1} can then be refined: $R_{B1}(N_1, N_2, y_1) \overset{\text{def}}{=} (N_1 = N_2 \wedge 0 \leq y_1 \leq 100)$. However, this new relation does not cut the trace since it includes the model assignment $y_1 = 42$. Since the error trace also goes through $B2$, we can refine the relation for $B2$ as well in order to obtain one that does not intersect with $y_1 = 42 \wedge N_2 = 10 \wedge y_2 = -1$. We recursively call our procedure and try to cut the "subtrace" inside $B2$, taking into account the context where it is run: $N_2 = 10 \wedge y_1 = 42$.

Since $B2$ has a nested abstraction block, we can reuse the relation for the input function for obtaining a precise relation for block $B4$:

$$R_{B4}(N_3, y_2, N_4, y_3, x) \overset{\text{def}}{=} (N_3 = N_4 \wedge y_2 = y_3 \wedge 10 - N_3 \leq x \leq 10 + N_3)$$

We compute a new relation for $B2$ by Path Focusing or Guided Path Analysis [MG11, HMM12b] with convex polyhedra, and find at the loop header $10 - N_2 \leq y_2 \leq y_1$, which is a correct invariant in the case where $(y_1 \geq 10 + N_2) \wedge (N_2 \leq 10)$, which is a superset of our context. This generalization of context is explained in section 5.4.

We then deduce a new input/output relation for block $B2$, which cuts the error trace (5.1):

$$[(y_1 \geq 10 + N_2) \wedge (N_2 \leq 10)] \Rightarrow [10 - N_2 \leq y_2 \leq y_1]$$

and go back to the analysis of the whole function. We update the formula by asserting this new relation and get a new error trace:

$$B0 \overset{N_1=10}{\longrightarrow} B1 \overset{y_1=5,N_2=10}{\longrightarrow} B2 \overset{y_2=-1}{\longrightarrow} Error$$

In this trace, the assignments $y_1 = 5$ and $N_2 = 10$ do not fit in the left-hand side of our previously computed relation for $B2$. We then analyze $B2$ again, in the new context. It gives the input/output relation:

$$[0 \leq y_1 \leq 10 + N_2] \Rightarrow [(0 \leq y_1 \leq 10 + N_2) \wedge (0 \leq y_2 \leq 10 + N_2)]$$

After going back to the analysis of the whole function, we get an *unsat* formula and there is no more error trace.

Suppose now that line 8 is only assume(0 <= N). In this case, we would get a new *sat* formula that we could not cut in the case $N > 10$. However, we can still prove the property under the assumption $N \leq 10$. The method for generating correct preconditions is detailed in section 5.6.

5.3 Modular Path Focusing

In this section, we present a new algorithm for computing procedure or loop summaries, based on abstract interpretation and SMT. The algorithm is in essence an extension of Path Focusing or Guided Path Analysis, to interprocedural settings with summaries.

5.3.1 Algorithm

We start with a program $\mathcal{P}$ (with input variables $X^i_{\mathcal{P}}$ and output variables $X^o_{\mathcal{P}}$), for which we want to compute a precise input/output relation between $X^i_{\mathcal{P}}$ and $X^o_{\mathcal{P}}$. This program can be seen as a block that itself contains nested blocks. We can assume some additional properties about the input variables (called *Context*).

To each block B representing an abstraction of a subgraph, we associate a partial mapping function $\text{Inv}_B : \mathcal{A}(X^i_B) \longrightarrow \mathcal{A}(X^{i/o}_B)$, where $\mathcal{A}$ is the abstract domain in use, and X^i_B, $X^{i/o}_B$ the dimensions of the abstract values. Inv_B maps properties about the input variables of the block, e.g. a context $C(X^i_B)$, to an overapproximated input/output relation $R(X^{i/o}_B)$ between input and output variables, safe if $C(X^i_B)$ holds. This is a partial function which is dynamically constructed during the analysis. When a block is encoded into an SMT formula, its nested abstraction blocks are replaced with the input/output relations defined by their corresponding Inv_B functions.

Here, we extend Path Focusing techniques to be modular and interprocedural: we compute an invariant at the block header and at each terminating control point (control points that have outgoing transitions outside the block). To draw a parallel with the previous chapter, the set P_R of points where we compute an invariant is now only the block header and the terminating control points: in the particular case where the block abstracts a loop, the block header is actually the loop header, thus the property of being acyclic when disconnecting the points in P_R still holds by construction. The input/output relations of the block are the invariants discovered at these terminating control points. We first encode the block as an SMT formula; its nested blocks are replaced with the input/output relations defined by their corresponding Inv functions. We use an SMT solver to find a path between the block header and a terminating control point (or the block header itself if the block abstracts a loop), for which the current abstract value associated with one successor is not an invariant. Once we found such a trace, we get the set of abstraction blocks the trace goes through, and their corresponding calling context, which is given by the model returned by the SMT solver. If one of the already computed input/output relation can be used in this context, we use it directly, otherwise we have to compute a relation by calling our analysis recursively on the nested block. The algorithm (precisely described in Algorithm 5) operates as follows:

① Initialize the abstract values: the initial state gets the calling context $A_s = C$ and the termination states get $\bot$.

② Using an SMT query, find a path from the entry point to a termination point *succ* for which the current associated abstract value is not invariant. This path is noted τ, the variable assignments in the model are given by a function $\mathcal{M}$. From the path τ, we can deduce a list of abstraction blocks the focusing trace goes through. We note them $B_1, \ldots, B_n$.

③ The computation of $\tau(A_s)$ requires input/output relations for the nested blocks $B_1, \ldots, B_n$, that are given by their Inv partial function. The algorithm for computing such $\tau(A_s)$ is detailed just after in subsection 5.3.2. Here, one needs for each block B_k a relation $R_k(X^{i/o}_{B_k})$ which is correct in the context $\tau_k(A_s)$, where τ_k denotes the path in τ that starts from the entry point but stops at the entry of B_k (see Figure 5.5). This relation should either be computed if there is no appropriate relation in Inv_{B_k}, or reused if there is already in $\text{Dom}(\text{Inv}_{B_k})$ a context greater than $\tau_k(A_s)$.

④ Finally, one can compute the image $\tau_k(A_s)$ using the appropriate relations for the nested abstracted blocks, and update A_{succ} accordingly. In the particular case of a loop, i.e. $succ = s$, abstract acceleration or a local widening/narrowing sequence can be applied, as it is done in [MG11] and subsection 4.2.2.

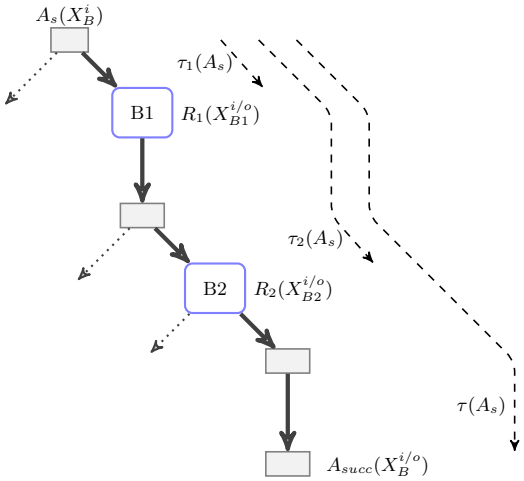


Figure 5.5: A path returned by the SMT solver, that goes through two abstraction blocks B_1 and B_2.

A prototype implementation of algorithm 5 is available in the Pagai tool, which is described in chapter 7. The main drawback of this procedure is that new relations have to be recomputed each time the calling context changes. In section 5.4, we propose a way of computing precise input/output relations for contexts that are more general (i.e. larger) than the initial one $C(X_B^i)$.

Remark: An important source of imprecision in Path Focusing from [MG11] is the case of nested loops, that prevent from applying direct narrowing or abstract acceleration on self-loops (one considers paths between two different loop headers) and thus loose precision due to widenings. Here, our approach generally improves precision since inner loops are abstracted within a block, so that every loops are actually self-loops.

Algorithm 5 Modular Path Focusing, illustrated by Figure 5.5

1: **procedure** ModularPathFocusing(Block B, Context $C(X_B^i)$)
2: $A_s(X_B^i) \leftarrow C(X_B^i)$ ▷ starting abstract value
3: For all $succ$, $A_{succ}(X_B^{i/o}) \leftarrow \bot$ ▷ final abstract values
4: $\rho_B \leftarrow$ EncodeToSMT(B) ▷ SMT encoding of block B
5: **while** SMTsolve $\Big[b_s \wedge A_s(X_B^i) \wedge \rho_B \wedge \bigvee_{succ} (b_{succ} \wedge \neg A_{succ}(X_B^{i/o}))\Big]$ **do** (Is there a path in B from b_i to some b_{succ} going outside the current A_{succ}?)
6: Path τ, model $\mathcal{M} \leftarrow$ getModel()
7: $(B_1, \ldots, B_n) \leftarrow$ AbstractedBlocks(τ) ②
8: **for all** $B_i \in (B_1, \ldots, B_n)$ **do**
9: **if** Dom(Inv$_{B_i}$) does not contain $\tau_i(A_s)$ **then**
10: ModularPathFocusing($B_i, \tau_i(A_s)$)
11: **end if** ③
12: **end for**
13: $A_{tmp}(X_B^{i/o}) \leftarrow \tau(A_s(X_B^i))$
14: **if** $succ = s$ **then** ▷ self-loop
15: Update A_{tmp} with widening/narrowing
16: **end if** ④
17: $A_{succ} \leftarrow A_{succ} \sqcup A_{tmp}$
18: **end while**
19: Add Inv$_B : C(X_B^i) \mapsto (A_{succ})$ ▷ the new relation is inserted into Inv$_B$
20: **end procedure**

5.3.2 Computing $\tau(A_s)$

In line 13 of Algorithm 5, one has to compute the abstract value resulting from the transformation of the initial abstract value through the path. When the path crosses nested abstraction blocks, one needs to use their corresponding available input/output relations. During the computation, there is a map Inv$_{B_k}$ for each nested block B_k, that contains the list of available relations for this block. The abstract value $\tau(A_s(X_B^i))$ can then be derived from every relation whose context is satisfied. It results that it is possible to combine several input/output relations for the same nested block. We found it clearer to detail in this thesis the algorithm for computing the abstract value after traversing an abstraction block (Algorithm 6): it takes as input the block to traverse and the abstract value just before, and returns the abstract value just after. In Figure 5.5, this function would be called for instance with input $I = \tau_1(A_s)$, block $B1$, and would return the abstract value just after traversing $B1$.

5.4 Generalizing Correct Contexts for Input/Output Relations

A key point in a modular and interprocedural static analysis is to compute input/output relations that are *reusable* in different calling contexts [Moy08]. In the extreme settings, recomputing a relation at each call sites would be expensive and similar to function inlining, while computing one single relation correct in any context would not be precise enough. In this section, we propose a way of computing relations in sufficiently general contexts while not sacrificing precision.

Algorithm 6 Compute the abstract value after crossing an abstraction block. This algorithm is used by any algorithm that computes $\tau(A_s)$.

1: **function** ComputeTransform(Input $I(X_B^i)$, Block B)
2: $\quad Temp(X_B^{i/o}) \leftarrow I(X_B^i)$ ▷ change dimensions
3: $\quad$ **for all** $C(X_B^i) \in \mathrm{Dom}(\mathrm{Inv}_B)$ s.t $I(X_B^i) \sqsubseteq C(X_B^i)$ **do** } intersect with any correct relation
4: $\quad\quad Temp(X_B^{i/o}) = Temp(X_B^{i/o}) \sqcap \mathrm{Inv}_B(C(X_B^i))$
5: $\quad$ **end for**
6: $\quad I'(X_B^o) \leftarrow P_{X_B^o}(Temp(X_B^{i/o}))$ ▷ Project into output variables only
7: $\quad$ **return** $I'(X_B^o)$
8: **end function**

Example 12 (Simple Example)**.** Suppose we have the following program fragment, and this fragment is chosen to be an abstraction block:

```
if (x > 0)
  x++;
else
  x--;
```

If we note x^i (resp. x^o) the variable x before (resp. after) the block, correct input/output relations would be for instance:

$$\text{true} \Rightarrow x^i - 1 \leq x^o \leq x^i + 1 \tag{5.2}$$

$$x^i > 0 \Rightarrow x^o = x^i + 1 \tag{5.3}$$

$$x^i \leq 0 \Rightarrow x^o = x^i - 1 \tag{5.4}$$

Equation 5.3 and Equation 5.4 clearly show that analyzing a block with a smaller context yields in the end a more precise invariant compared to the one we would obtain without any precondition (Equation 5.2). The reason is that some paths are unfeasible under certain preconditions. Suppose now that the analysis of this block is triggered by our modular path focusing algorithm with the input context $x^i > 10$. The analysis of the block would find the relation $x^i > 10 \Rightarrow x^o = x^i + 1$. This result is not more precise than Equation 5.3, however its context is less general. In this section of the thesis, we explain how to infer 5.3 instead of the less general relation, while not sacrificing the precision, e.g. we do not want to infer only 5.2.

Our approach is the following: even though we compute a relation in a calling context $C(X^i)$ for the input variables, we start the computation of the new input/output relation of a block with $A_s(X^i) = \top$, where $A_s(X^i)$ is the abstract value attached to the block header, as if we were computing a relation correct for any context. Intuitively, the idea is to compute an input/output relation "as precise as" the one we would obtain when starting with $A_s(X^i) = C(X^i)$, but in a sense more general, i.e. true for a larger set of input variables, noted $GC(X^i)$ (then, $C(X^i) \sqsubseteq GC(X^i)$).

During the fixpoint computation, we collect the properties implied by $C(X^i)$ that we use to improve the precision of our invariant, incrementally building an SMT formula over the variables in X^i, that we will note precond(X^i). precond(X^i) is initialized to *true* and tightened during the analysis with new constraints over the input variables that are required to be precise. Once the input/output relation has been computed, precond(X^i) is an arbitrary formula, and the computed relation is correct under this precondition. One can then derive a simpler precondition $GC(X^i) \sqsupseteq C(X^i)$ in our abstract domain, which is included in precond(X^i).

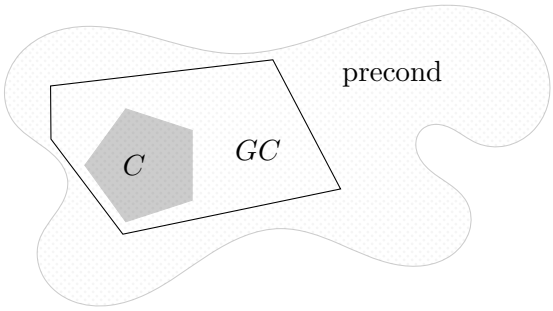
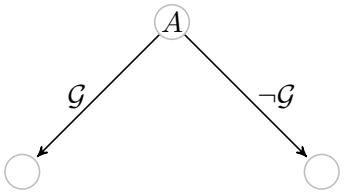


Figure 5.6: $GC(X^i), C(X^i)$ abstract values, and the precond(X^i) formula

Intuition, with Simple Transition

For getting the intuition behind our approach, suppose we have a program location, and its attached abstract value A, and two transitions guarded by some constraints $\mathcal{G}$ and $\neg\mathcal{G}$:



The abstract value A is a relation between variables in X^i, i.e. inputs, and local temporary variables X^l (typically, live variables). For clarity, we note it $A(X^{i/l})$.

If $A(X^{i/l}) \sqcap \mathcal{G}(X^l) = \bot$ or $A(X^{i/l}) \sqcap \neg\mathcal{G}(X^l) = \bot$, it means that one of the two transitions is unfeasible, without the need of extra information or assumption. Otherwise, some properties implied by $C(X^i)$ could make one of the transitions become unfeasible, and thus prevent from loosing precision due to a later least upper bound operation. This happens if:

1. $(A(X^{i/l}) \sqcap C(X^i)) \sqsubseteq \neg\mathcal{G}(X^l)$, or equivalently $A(X^{i/l}) \sqcap C(X^i) \sqcap \mathcal{G}(X^l) = \bot$:

   This is the case where the transition guarded by $\mathcal{G}$ is actually not feasible if we suppose the context. In this case, the projection over X^i of $\neg\mathcal{G}(X^{i/l}) \sqcap A(X^{i/l})$ (denoted $P_{X^i}(\neg\mathcal{G} \sqcap A)$ in Figure 5.7) gives a formula over the input variables, for which the transition guarded by $\mathcal{G}(X^{i/l})$ would not be feasible.

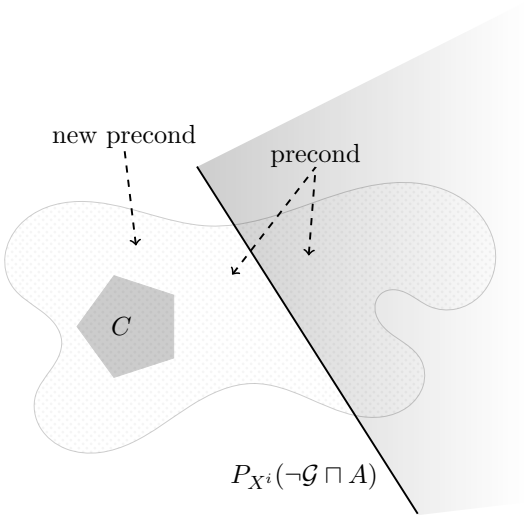


Figure 5.7: The set representing precond (dotted area) is intersected with $P_{X^i}(\neg\mathcal{G} \sqcap A)$ to contain only those points for which the transition is unfeasible.

We could then continue our analysis, assuming that this formula is part of the precondition we have for the input variables, and that the transition cannot be taken. Formally, we assign $\text{precond}(X^i) \leftarrow \text{precond}(X^i) \wedge P_{X^i}(\neg\mathcal{G}(X^l) \sqcap A(X^{i/l}))$.

2. or, $(A(X^{i/l}) \sqcap C(X^i)) \sqsubseteq \mathcal{G}(X^l)$:

   This is the symmetrical case, where the transition guarded by $\neg\mathcal{G}$ is not feasible in the context. Similarly, we restrict the precondition to a smaller one: $\text{precond}(X^i) \leftarrow \text{precond}(X^i) \wedge P_{X^i}(\mathcal{G}(X^l) \sqcap A(X^{i/l}))$.

In this way, one can prune unfeasible transitions by using the calling context, and construct in parallel the formula $\text{precond}(X^i)$ for which the obtained input/output relation is correct. However, $\text{precond}(X^i)$ is not necessarily an element of the abstract domain. Once the fixpoint is reached, the precond formula represents a set of abstract states greater than the initial context. Finally, one could search for an element $GC(X^i)$ in the abstract domain, included in our discovered precondition, and that itself includes $C(X^i)$. This last step is described later in page 90.

Updating the Context when Discovering Unfeasible Paths

In general, in the settings of path focusing, we are not only interested in showing that a particular transition is unfeasible, but an entire path. Here, we explain how the previous idea for simple transitions extends to paths.

Suppose we have a path τ that is feasible with the current $\text{precond}(X^i)$, but is not in the input context : $\tau(A(X^{i/l}) \sqcap C(X^i)) = \bot$. τ is given by the model returned by the SMT solver, when checking the satisfiability of the formula:

$$\overbrace{\left[b_s \wedge A(X^{i/l}) \wedge \rho \wedge \bigvee_{succ} (b_{succ} \wedge \neg A_{succ}(X^{i/o}))\right]}^{\text{same as in Algorithm 5: is there a path...?}} \wedge \text{precond}(X^i)$$

Let $\mathcal{G}_1(X^l), \mathcal{G}_2(X^l), \ldots, \mathcal{G}_n(X^l)$ be the guards the path τ goes through. We apply quantifier elimination over the formula

$$\exists\ x_1, x_2, \ldots, x_k,\ A(X^{i/l}) \wedge \bigwedge_{1 \leq i \leq n} \mathcal{G}_i(X^l)$$

where $\{x_1, x_2, \ldots, x_k\} \stackrel{\text{def}}{=} X^l$.

This formula represents the set of input states for which the path is feasible. If we consider the case where $A(X^{i/l})$ is a convex polyhedron and the $(\mathcal{G}_i)_{1 \leq i < n}$ are linear inequalities, this formula is actually a conjunction of linear inequalities. In this case, we can use already existing algorithms for projection, like Fourier-Motzkin elimination in LRA, or Omega test and Cooper's method in LIA ([KS08] give details on these algorithms).

The quantifier elimination gives us an equisatisfiable formula $F(X^i)$ that we negate and add to precond(X^i):

$$\text{precond}(X^i) \leftarrow \text{precond}(X^i) \wedge \neg F(X^i)$$

The analysis continues with the new precond(X^i), for which the path τ has become unfeasible. Finally, for each path we were able to remove, an associate formula has been added to precond(X^i). In practice, this added formula is likely to prune several paths at the same time.

Algorithm 7 Modular Path Focusing with Context Generalization. ②, ③ and ④ is the corresponding code from Algorithm 5.

1: **procedure** ModularPathFocusing(Block B, Context $C(X^i)$)
2: $A_s(X^i_B) \leftarrow \top$ ▷ starting abstract value
3: For all *succ*, $A_{succ}(X^{i/o}_B) \leftarrow \bot$ ▷ final abstract values
4: $GC(X^i) \leftarrow \top$ ▷ generalized context
5: $\rho_B \leftarrow$ EncodeToSMT(B) ▷ SMT encoding of block B
6: **while** SMTsolve $\Big[b_s \wedge A_s(X^i_B) \wedge \rho_B \wedge \bigvee_{succ} (b_{succ} \wedge \neg A_{succ}(X^{i/o}_B)) \Big]$ (Is there a path in B from b_i to some b_{succ} going outside the current A_{succ}?) **do**
7: ②
8: ③
9: $A_{tmp}(X^{i/o}_B) \leftarrow \tau(A_s(X^i_B))$
10: **if** $A_{tmp} \neq \bot \ \wedge\ \tau(A_s(X^i_B) \sqcap C(X^i_B)) = \bot$ **then**
11: Update precond(X^i_B)
12: Continue
13: **end if**
14: ④
15: **end while**
16: $GC(X^i_B) \leftarrow$ GeneralizeContext$(C(X^i_B), GC(X^i_B))$
17: Add $\text{Inv}_B : GC(X^i_B) \mapsto A_{succ}(X^{i/o}_B)$
18: **end procedure**

(τ is not feasible in the context C)

From the Obtained Precondition Formula to $GC(X^i)$

In the end, precond(X^i) is a conjunction of added formulas but may not be a convex polyhedron, we thus can search for a convex polyhedron implied by $C(X^i)$ and that implies precond(X^i), using SMT-solving. From the model returned by the SMT-solver, which is a single point, we

can deduce a convex polyhedron that also satisfies the formula using the method described by [Mon10]. In a few words, from a satisfying assignment of the formula, one can derive the set of atoms in the formula evaluated to *true*. In LIA or LRA, it gives a set of linear inequalities. One can then try to iteratively remove some of these inequalities and see whether the new polyhedron still implies the formula. Note that, since we try to find a polyhedron that contains the context C, we first remove the atoms that are not implied by C.

Algorithm 8

```
1: function GeneralizeContext(Context C(X^i), Formula precond(X^i))
2:     while true do
3:         m(X^i) ← SMT-query [precond(X^i)]
4:         Generalize m(X^i) into a convex polyhedron M(X^i)        ▷ see [Mon10] for details
5:         if C(X^i) ∧ ¬M(X^i) is unsat then
6:             return M(X^i)
7:         else
8:             precond(X^i) ← precond(X^i) ∧ ¬M(X^i)
9:         end if
10:    end while
11:    return C(X^i)
12: end function
```

Our algorithm is detailed in Algorithm 8, and is called once for each call to procedure ModularPathFocusing, once an inductive invariant has been found.

Example 13 (Context Generalization)**.** Suppose we have $X^i = \{x_0, y_0\}$, $X^l = \{x, y\}$, $C(X^i) = (x_0 > 20)$, $A(X^{i/l}) = x \geq x_0 + 2 \wedge y \geq y_0$, and a path with guards $\mathcal{G}(X^l) = (y \leq 0 \wedge x \leq 10)$.

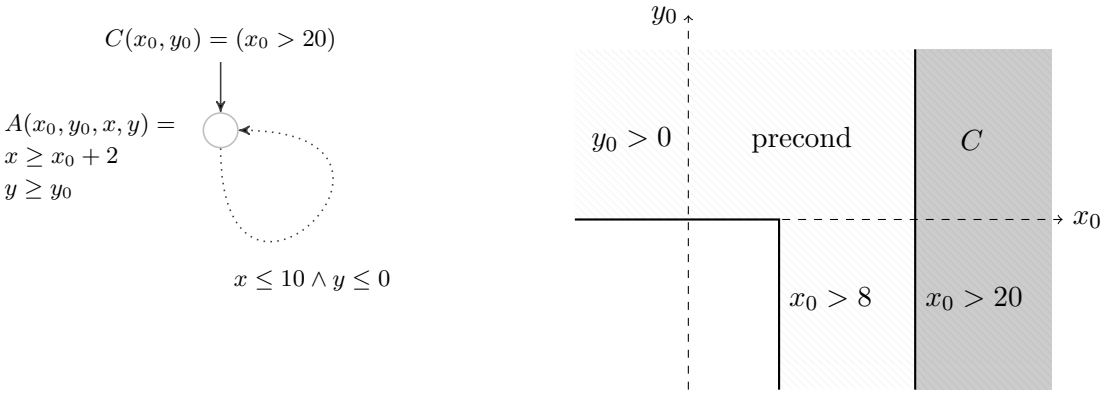


Figure 5.8: In the left, the control flow graph that illustrates the infeasible path. In the right, a graphical representation of precond and the input context C.

The context $C(X^i)$ verifies $C(X^i) \wedge A(X^{i/l}) \wedge cond(X^l) \equiv \bot$. We apply quantifier elimination over the formula (polyhedral projection):

$$\exists\ x\ y,\ (y \leq 0) \wedge (x \leq 10) \wedge (x \geq x_0 + 2) \wedge (y \geq y_0)$$

We obtain:

$$(x_0 \leq 8) \wedge (y_0 \leq 0)$$

We then add the negation of this formula to our current precond(X^i):

$$\text{precond}(X^i) \leftarrow \text{precond}(X^i) \wedge ((x_0 > 8) \vee (y_0 > 0))$$

If, in the end, precond(X^i) $\equiv ((x_0 > 8) \vee (y_0 > 0))$, we can search for a convex polyhedron implied by $C(X^i)$ and that implies precond(X^i):

- SMT-query of $assert(\text{precond}(X^i))$ gives the model $(x_0, y_0) = (10, 0)$.
- We generalize the model and obtain $x_0 > 8$.
- SMT-query $C(X^i) \wedge \neg(x_0 > 8)$ is *unsat*. We thus can choose $GC(X^i) = (x_0 > 8)$ as a correct generalized context.

Note that the first SMT query could have returned another model whose generalization is $y_0 > 0$, which is not implied by $C(X^i)$. In this case, several SMT-queries are required.

5.5 Property-guided Modular Analysis

In this section, we start with a program $\mathcal{P}$ (with input variables $X^i_{\mathcal{P}}$ and output variables X^o_P), for which we want to prove a property $P(X^{i/o}_{\mathcal{P}})$.

Our algorithm abstracts the program $\mathcal{P}$ as a block, and tries to prove (as lazily as possible) that there is no trace in the block that negates $P(X^{i/o})$. For doing this, it tries first to prove P with rough abstractions of the nested abstraction blocks, and refines these abstractions modularly and incrementally until they are sufficiently precise to prove the property.

Conceptually, the idea is the same as in subsection 5.3.1, but here we try to avoid computing costly relations for program parts that are not relevant for proving a particular property. As in subsection 5.3.1, we dynamically construct a partial function $\text{Inv}_B : \mathcal{A}(X^i_B) \longrightarrow \mathcal{A}(X^{i/o}_B)$ that maps a calling context $C(X^i_B)$ to an overapproximated input/output relation $R(X^{i/o}_B)$ for the block, correct if $C(X^i_B)$ holds. When an error trace is discovered using bounded model checking by SMT-solving, the algorithm computes a new input/output relation for some subcomponents and update the Inv partial function of the current block, until the error trace is proved unfeasible. Our algorithm is illustrated by Figure 5.9. In the following subsections, we detail the different steps of this algorithm.

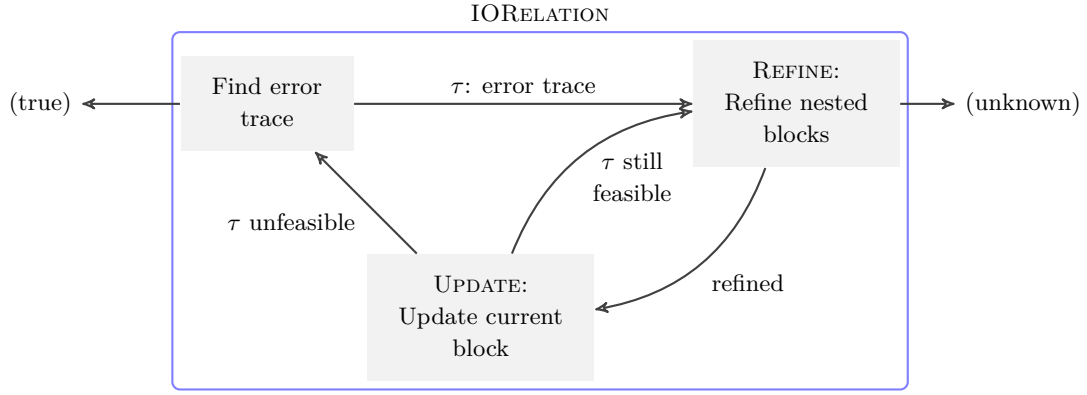


Figure 5.9: Principle of the algorithm

Algorithm 9 Modular Static Analysis using SMT

```
1: function IORelation(Block B, Property P(X_B^{i/o}), Context C(X_B^i))
2:     while true do
3:         ρ_B ← EncodeToSMT(B)                          ▷ SMT encoding of block B
                  Does there exist an error trace that violates P ?
4:         res ← SMTsolve(C(X^i) ∧ ρ_B ∧ ¬P(X^{i/o}))
5:         if res is unsat then
6:             return (true)                              ▷ No error trace, P holds
7:         end if
8:         Path τ, model M ← getModel()
9:         (B_1, ..., B_n) ← AbstractedBlocks(τ)
10:        Refine(B_1, ..., B_n, M)                       ▷ see 5.5.2
11:        Update(B, C(X_B^i))                            ▷ see 5.5.3
12:        if Update has not improved Inv_B then
13:            return (unknown)
14:        end if
15:    end while
16: end function
```

5.5.1 IORelation

The main procedure of our algorithm consists in solving the following problem:

> "Given a block B, with inputs X_B^i and outputs X_B^o, an initial context $C(X_B^i)$ and its Inv_B function, does there exist a path for which the property $P(X_B^{i/o})$ does not hold ?"

This problem is encoded into SMT and checked using an SMT solver. If ρ_B is the encoding of the program semantics into formula, the problem consists in checking the satisfiability of the formula $C(X_B^i) \wedge \rho_B \wedge \neg P(X_B^{i/o})$. If such a path does not exists, the algorithm terminates and returns *true*, meaning that the property is proved. Otherwise, one tries to refine the input/output relation of the block, by first refining the abstraction blocks the error trace goes through: this is the purpose of the Refine and Update functions, that are described later.

Using our notations, the trace τ goes through nested abstraction blocks $B_1, \ldots, B_n$, and the assignments of all SSA variables are given by a map $\mathcal{M}$. We note $\mathcal{M}(X)$ the formula expressing the assignments of the variables in X in this model. The way of showing that the trace is unfeasible is to find a new relation $R_B(X_B^{i/o})$ such that $\mathcal{M}(X_B^i) \Rightarrow R_B(X_B^{i/o})$ and $R_B(X_B^{i/o}) \Rightarrow \neg\mathcal{M}(X_B^o)$. To help find this relation, one tries to find for some nested block B_k a relation R_{B_k} with the similar property, i.e. $\mathcal{M}(X_{B_k}^i) \Rightarrow R_{B_k}(X_{B_k}^{i/o})$ and $R_{B_k}(X_{B_k}^{i/o}) \Rightarrow \neg\mathcal{M}(X_{B_k}^o)$. In other words, if we find a nested block for which the trace is locally unfeasible, then the trace is unfeasible.

Procedure Refine takes the list of nested abstraction in the trace and refines these abstractions until the trace becomes unfeasible. Procedure Update aims at updating the partial function Inv_B of the current block, that maps contexts (abstract values over X_B^i) to input/output relation (abstract values over $X_B^{i/o}$).

Function IORelation queries the SMT solver repeatedly to find paths inside the block that negate the property $P(X_B^{i/o})$. Each time we discover an error trace, we try to show it unfeasible by refining one (or more) of the traversed abstraction blocks. We can therefore update the formula ρ_B, since the partial functions Inv of some abstraction blocks have changed.

We also recompute a new input/output relation for this block, that may be more precise since the subcomponents have been refined. If the new relation is precise enough to cut the error trace, one can iterate and search for another error trace.

At some point, it may be the case that we can not refine abstraction blocks anymore, while the error trace is still feasible. In this case, the algorithm answers "unknown", in the sense that the property P is not proved correct.

In this approach, we can strengthen the invariant incrementally when the previous one is not sufficient, and avoid being unnecessarily precise on the part of the code that is not relevant.

5.5.2 Refine

1: **function** REFINE(Blocks $B_1, \ldots, B_n$, Model $\mathcal{M}$)
2: $iscut \leftarrow false$ ▷ becomes true if the trace given by the model is cut
3: **for all** $B_k \in \{B_1, \ldots, B_n\}$ **do**
4: $(iscut) \leftarrow$ IORELATION(B_k, $\underbrace{\neg\mathcal{M}(X^o_{B_k})}_{\text{Property: cannot reach the model of the outputs}}$, $\overbrace{\mathcal{M}(X^i_{B_k})}^{\text{Context: model of the inputs}}$)
5: **if** $iscut$ **then**
6: **return** ▷ trace is unfeasible, no need to refine further
7: **end if**
8: **end for**
9: **end function**

Once the SMT-solver has found an error trace, we get a model $\mathcal{M}$ that maps each variable to value, as well as the list of abstraction blocks the trace goes through. We note $\mathcal{M}(X)$ the conjunction of the assignments in the model $\mathcal{M}$ for the variables in a set X. REFINE recomputes the input/output relations for some of these blocks in order to "cut" the trace (so that the trace becomes unfeasible). For computing these input/output relations, we create a calling context $\mathcal{M}(X^i_{B_k})$ for each block B_k, and the objective property, $\mathcal{M}(X^o_{B_k})$, being the error state we want to prove unreachable. Here, both the input context and the error state are single points given by the model of the error trace.

The Inv functions of the nested blocks are then successively strengthened with a recursive call to our IORELATION algorithm. We can avoid refining every block as far as we proved the output of one nested blocks to be unfeasible. Since a trace often goes through several abstraction blocks, the order we choose for refining them may influence the effectiveness of the method. We discuss later some heuristics for choosing the refinement order.

5.5.3 Update

1: **function** UPDATE(Block B, Context $C(X^i)$)
2: $GC(X^i_B), R_B(X^{i/o}_B) \leftarrow$ compute the relation and its generalized context
3: ▷ uses for instance MODULARPATHFOCUSING (Algorithm 7)
4: Insert into $\text{Inv}_B : GC(X^i_B) \mapsto R_B(X^{i/o}_B)$
5: **end function**

UPDATE$(B, C(X_B^i))$ computes an input/output relation for block B that is correct at least in the context $C(X_B^i)$, as well as a correct context for this relation. This is exactly the purpose of our modular path focusing algorithm, that we described in section 5.3. However, in these new settings, there is no need for the modular path focusing algorithm to recursively call itself for nested abstraction blocks, since it is here handled by the REFINE function. The algorithm we use is then kept unchanged, but we remove step ③ from algorithm 7 (page 90), that was corresponding to the recursive call to the analysis procedure.

Now, recall that the objective is to prove a given property. Modular path focusing is a quite expensive technique that makes use of possibly many SMT queries before reaching a fixpoint. In our framework, it is possible to compute the fixpoints using cheapest techniques, such as those explained in chapter 2 and chapter 3, and only apply modular path focusing if the previous technique lacks precision. For instance, one can use standard abstract interpretation from [CC77, CH78] for computing the relations, considering as dimensions of the abstract values both the input and the output variables. In this case, the context generalization is done in the same way it is described in section 5.4 for simple transitions.

Our framework can then mix several abstract interpretation based techniques. A heuristic would be to run first the cheapest technique, see if the invariant it provides is sufficient for cutting the error trace, and if not run a more precise technique.

5.5.4 Some Heuristics for Refinement

When an error trace is found, our framework tries to prove it unfeasible by refining some program portions crossed by the trace. There are multiple ways of improving the input/output relations: either recursively refine one of the nested abstraction block, or run a more powerful analysis technique for the block itself, with the same abstractions of the nested blocks. In addition, when the error trace traverse several abstraction blocks, it is unclear which one has to be refined in priority for cutting the trace.

We propose here a reasonable heuristic for choosing the order of refinement, that should be experimentally compared with others. Such experiments are part of future work.

Suppose the implementation provides several analysis techniques, ordered by "*practical complexity*", from the cheapest technique, noted T_1, to the more expensive one, noted T_k. We label each computed relation with the technique that was used for getting it. When an error trace is discovered, the REFINE procedure has the list of nested abstraction blocks $B_1, \ldots, B_n$. One can deduce for each block what is the greatest technique that has already been run and can be used (we call it *level* of the block).

- If every block has the same level, we successively analyze every block with the next level,
- Otherwise, let T be the level of the greatest block. The blocks are then pushed in a priority queue, that takes first the block with the smallest technique (with respect to complexity). We successively refine every block until every block reaches the level T.

In both cases, we stop any analysis as soon as the trace is cut. Finally, the UPDATE function runs the same technique as the cheapest nested block.

5.5.5 SMT Encoding of Blocks

In all this chapter, we rely on an SMT encoding of the semantics of abstraction blocks. In subsection 4.2.1, we detailed a similar encoding for control-flow graphs. The difference here

is that we encode a graph of blocks instead of basic blocks, where some of the blocks are abstractions defined by input/output relations.

Non-abstracted blocks (i.e. blocks that are actually basic blocks) are encoded as before, while the encoding of abstraction blocks B uses of the Inv_B function:

$$\bigwedge_{C(X_B^i)\in \text{Dom}(\text{Inv}_B)} (C(X_B^i) \Rightarrow \text{Inv}_B(C(X_B^i)))$$

For blocks that abstract several paths without loops, we use their Inv partial function to plug a summary of these paths, as well as their classical encoding. This last encoding is needed to be fully precise, and the summary will serve the SMT-solver to perform faster by acting in some cases as a smart Craig interpolant [McM05]. Intuitively, the summary will help the SMT solver to prune many unfeasible paths at once, instead of doing a possible exhaustive enumeration of every paths. This phenomenon is precisely explained later in section 6.4.

If the block is an abstraction of a loop, the block header is a loop header by construction. In the exact same way as in subsection 4.2.1, we split this block header into two distinct blocks, the first one with only incoming transitions coming from that block, and the second one with only the outgoing transitions. In this way, the resulting control flow graph has no cycle and the SMT encoding is possible.

Models of this encoded formula are the feasible paths between the entry point of the blocks and an exit point. In case of a loop, the paths correspond to one single iteration of the loop.

From a practical point of view, this encoding has to be carefully implemented, since it is not sufficient to rely on an standard SSA control flow graph of basic blocks. Indeed, recall that the graph of block has to satisfy the following condition: input and output variables of each abstraction block have an empty intersection. For instance, in our introductory example of section 5.2 (see Figure 5.4), the block $B1$ has the variable N both as input and output, and thus one should duplicate this variable into N_1 for the input, and N_2 for the output. Another example is given by the variable y in block $B4$. As a conclusion, a classical SSA transformation of the control flow graph is not sufficient to guarantee the property, and a new "SSA-like" pass has to be done on the graph of blocks.

5.6 Generation of Preconditions

Using this framework, we can generate "interesting" preconditions at the header of blocks (representing loops, functions, etc.). Indeed, suppose we want to find a sufficient precondition for a block B such that the property $P(X_B^{i/o})$ is true, and that implies $C(X_B^i)$. If the algorithm IOrelation proves $P(X_B^{i/o})$ in the context $C(X_B^i)$, we can use the resulting $GC(X_B^i)$ (or the precond formula) as a sufficient precondition.

However, this technique requires the knowledge of an initial context where P is correct. If we do not know such context, we can:

1. Use SMT-solving to find a path in B verifying the property.
2. Use the model $\mathcal{M}(X_B^i)$ as the initial context for running the algorithm IOrelation
3. The algorithm may return *true* and give a precondition $GC(X_B^i)$ that implies $\mathcal{M}(X_B^i)$.

We can apply the three last points iteratively to discover classes of inputs for which we have P. To do so, we just have to add in the SMT query the negation of the already-discovered preconditions to prevent the solver to give a model that is in the same class as a previous one.

1: **function** GeneratePrecondition(Block B, Property $P(X^{i/o})$)
2: $\rho_B \leftarrow$ EncodeToSMT(B)
3: $\text{Inv}_B \leftarrow \{\}$
4: **while** SMTsolve($\rho_B \wedge P(X_B^{i/o}) \wedge \bigwedge_{C(X_B^i) \in \text{Dom}(\text{Inv}_B)} \neg C(X_B^i)$) **do**
5: $\mathcal{M} \leftarrow$ getModel()
6: IORelation(B, $\neg P(X_B^{i/o})$, $\mathcal{M}(X_B^i)$) ▷ updates Inv_B
7: Recompute ρ_B
8: **end while**
9: **end function**

The size of the control flow graph is finite, so is the number of classes, and the iterations are guaranteed to terminate. However, we can stop the iterations at any moment and get a correct precondition. In these settings, the correct precondition is expressed as a disjunction of convex polyhedra.

5.7 Conclusion

In this chapter, we proposed a new analysis framework called *Modular Path Focusing*. This new algorithm improves the preceding techniques described in this thesis in several ways: it improves scalability when analyzing large-sized functions by abstracting program fragments with *summaries*. It also improves precision in the sense that the analysis it provides in inter-procedural. We claim that our procedure efficiently computes summaries that are are correct in a wide context through our context generalization method, which is to our knowledge an original idea. It enables both a faster analysis — since it avoids several useless analysis of the same program portion — and a procedure for inferring *interesting* preconditions.

As described in this chapter, our procedure for generalizing the input contexts makes use of quantifier elimination, which is known to be expensive in practice. We think that it would be possible in future work to replace this quantifier elimination with interpolation for scalability reason, or use some kind of approximate quantifier elimination [KGC14].

As usual with static analysis techniques, one should conduct experiments on real code and benchmarks in order to see how efficient the algorithm is in practice; the implementation of all the algorithms presented in this chapter are currently under development in our tool Pagai, but are not yet ready and thus we do not have experimental results so far. We hope to have first results soon.

This chapter motivates interesting future work: early experiments conducted with some students showed that it is possible to greatly improve the efficiency of the SMT queries with the help of summaries: in the case where the SMT formula encodes a portion with a huge number of paths, one can avoid exploring many paths by conjoining with the formula the summary of well chosen program fragments. These summaries can be computed by the algorithms described in this chapter. We think that this idea could be an efficient way of using abstract interpretation to improve SMT. In the next chapter, we apply this idea to the particular case of Worst-Case Execution Time (WCET) estimation, for which the encoding into an SMT problem gives very challenging formulas that current solvers are not able to deal with.

Part III

Implementation and Application

Chapter 6

Worst-Case Execution Time Estimation

6.1 Motivation

In embedded systems, it is often necessary to ascertain that the worst-case execution time of a program (WCET) is less than a certain threshold. This is in particular the case for synchronous reactive control loops (infinite loops that acquire sensor values, compute appropriate actions and update, write them to actuators, and wait for the next clock tick) [CRT08]: the WCET of the loop body ("step") must be less than the period of the clock.

Computing a safe upper bound for the WCET of a program on a modern architecture requires a combination of low-level, micro-architectural reasoning (regarding pipeline and cache states, buses, cycle-accurate timing) and higher-level reasoning (program control flow, loop counts, variable pointers). A common approach is to apply a form of abstract interpretation to the micro-architecture, deduce worst-case timings for elementary blocks, and reassemble these into the global WCET according to the control flow and maximal iteration counts using integer linear programming (ILP) [TFW00, W$^+$08].

One pitfall of this approach is that the reassembly may take into account paths that cannot actually occur in the real program, possibly overestimating the WCET. This is because this reassembly is mostly driven by the control-flow structure of the program, and (in most approaches) ignores semantic conditions. For instance, a control program may (clock-)enable certain parts of the program according to modular arithmetic with respect to time (Figure 6.1); these arithmetic constraints entail that certain combinations of program fragments cannot be active simultaneously. If such constraints are not taken into account (as in most approaches), the WCET will be grossly over-estimated.

In this chapter, we propose a solution to take such *semantic constraints* into account, in a fully automated and very precise fashion. Specifically, we consider the case where the program for which WCET is to be determined contains only loops for which small static bounds can be determined (note that our approach can also be applied to general programs through summarization as we do in chapter 5, see section 6.9). This is very commonly the case for synchronous control programs, such as those found in aircraft fly-by-wire controls [SWDD09]. Programs of this form are typically compiled into C from high-level data-flow synchronous programming languages such as Simulink, Lustre or Scade. (Simulink™ is a block diagram

```
if (clock % 4==0) { /* A */ }
/* unrelated code */
if (clock % 12==1) { /* B */ }
```

Figure 6.1: A program with clock-enabled, mutually incompatible sections A and B.

environment for multidomain simulation and model-based design from The Mathworks, and SCADE™ is a model-based development environment dedicated to critical embedded software, from Esterel Technologies, derived from the academic language LUSTRE[CRT08]).

We compute the WCET of such programs by expressing it as the solution of an *optimization modulo theory* problem. Optimization modulo theory is an extension of satisfiability modulo theory where the returned solution is not just an arbitrary solution, but one maximizing some objective; in our case, solutions define execution traces of the program, and the objective is their execution time.

Expressing execution traces of programs as solutions to an SMT problem has already been described in 4.2.1. In the case of optimization, we select a particular integer variable in the formula and search for its maximal (or minimal) possible value for which there exists a satisfiable assignment. The SMT solver has to disprove the existence of solutions greater than the maximum to be returned — in our case, to disprove the existence of traces of execution time greater than the WCET. Unfortunately, all currently available SMT solvers take unacceptably long time to conclude on naive encodings of WCET problems. This is because all these solvers implement variants of the DPLL($\mathcal{T}$) approach [KS08] (see section 2.2.1), which has exponential behavior on so-called "diamond formulas", which appear in naive encodings of WCET on sequences of if-then-elses.

Computing or proving the WCET by direct, naive encoding into SMT therefore leads to intractable problems, which is probably the reason why, to our best knowledge, it has not been proposed in the literature. However, we show how an alternate encoding, including "cuts", makes such computations tractable.

In this chapter, we propose the following contributions:

1. The computation of worst-case execution time (WCET), or an over-approximation thereof, by optimization modulo theory. The same idea may also be applicable to other similar problems (e.g. number of calls to a memory allocator). Our approach exhibits a worst-case path, which may be useful for targeting optimizations so as to lower WCET [ZKW+06].

2. The introduction of "cuts" into the encoding so as to make SMT-solving tractable. The same idea may extend to other problems with an additive or modular structure, and is a special case of summaries described in chapter 5.

6.2 Traditional Approach for Estimating Worst-Case Execution Time

Let us first summarize the classical approach to static timing analysis. For more details, the reader may read for instance to [TFW00, W+08]. Figure 6.2 shows the general timing analysis workflow used in a large part of WCET tools including industrial ones such as AiT[1] or academic ones such as OTAWA[2] [BCRS10] or CHRONOS[3] [LLMR07]. For the sake of simplicity, we shall restrict ourselves to single-processor platforms with no bus-master devices except for the CPU.

The analysis considers the object code. The control flow graph is first reconstructed from the binary. Then, a *value analysis* (e.g. abstract interpretation with domain of intervals) extracts memory addresses, loop bounds and simple infeasible paths [GESL06]. Such an analysis may be

[1] `http://www.absint.com/ait/`
[2] `http://www.otawa.fr`
[3] `http://www.comp.nus.edu.sg/~rpembed/chronos/`

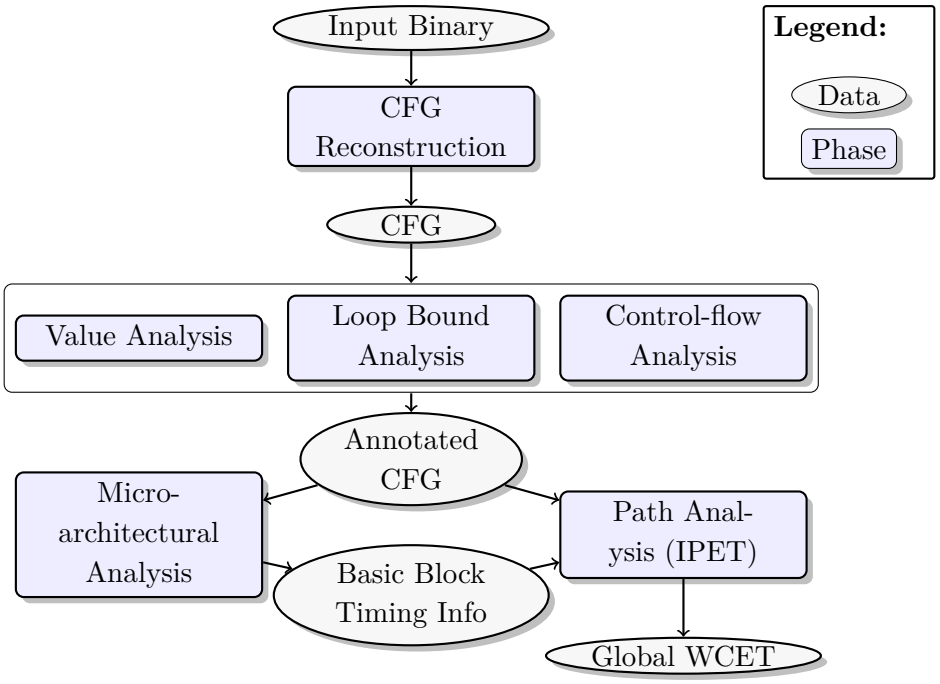


Figure 6.2: WCET analysis workflow

performed on the binary or the source files. In the latter case, it is necessary to trace object code and low-level variables to the source code, possibly using the debugging information provided by the compiler. This semantic and addressing information help the micro-architectural analysis, which bounds the execution time of basic blocks taking into account the whole architecture of the platform (pipeline, caches, buses,...)[EJ02, Rei09]. The most popular method to derive this architecture analysis is abstract interpretation with specific abstract domains. For instance, a pipeline abstraction represents sets of detailed pipeline states, including values for registers or buffers [EJ02]; while a cache abstraction typically tracks which value may or must be in each cache line [Rei09].

The last step of the analysis uses the basic block execution timings and the semantic information to derive the global WCET of the entire program, usually, in the "implicit path enumeration technique" (IPET) approach, as the solution of an integer linear program (ILP) [LM97]. The ILP variables represent the execution counts, along a given trace, of each basic block in the program. The ILP constraints describe the structure of the control flow graph (e.g. the number of times a given block is entered equals the number of times it is exited), as well as maximal iteration counts for loops, obtained by value analysis or provided by the user. Finally, the execution time to be maximized is the sum of the basic blocks weighted by their local worst-case execution time computed by the micro-architectural analysis.

The obtained worst-case path may however be *infeasible* semantically, for instance, if a condition tests $x < 10$ and later the unmodified value of x is again tested in a condition $x > 20$ along that path. This is because the ILP represents mostly syntactic information from the control-flow graph. This weakness has long been recognized within the WCET community, which has devised schemes for eliminating infeasible worst-case paths, for instance, by modifying the control-flow graph before the architecture analysis [NRM04], or by adding ILP constraints [HW02, GESL06, RMPVC13]. Infeasible paths are found via pattern matching of conditions [HW02] or applying abstract execution [GESL06]; these methods focus on paths

made infeasible by numeric constraints. These approaches are limited by the expressiveness of ILP constraints as used in IPET: they consider only "conflict conditions" (exclusive conditional statements: "if condition a is true then condition b must be false").

On a loop-free program, the ILP approach is equivalent to finding the longest path in the control-flow graph, weighted according to the local WCET of the basic blocks. Yet, again, this syntactic longest path may be infeasible. Instead of piecemeal elimination of infeasible paths, we propose encoding the set of feasible paths into an SMT formula, as done in bounded model-checking; the success of SMT-solving is based on the ability of SMT solvers to exclude whole groups of spurious solutions by learning lemmas.

Loop-free programs without recursion may seem a very restricted class, but in safety-critical control systems, it is common that the program consists in one big infinite control loop whose body must verify a WCET constraint, and this body itself does not contain loops, or only loops with small static bounds (say, for retrieving a value from an interpolation table of known static size), which can be unrolled in the analysis (note that the final executable does not require inlining). Such programs typically eschew more complicated algorithms, if only because arguing for their termination or functional correctness would be onerous with respect to the stringent requirements imposed by the authorities. Complicated or dynamic data structures are usually avoided [LM97, ch. II]. This is the class of programs targeted by e.g. the Astrée static analyzer [CCF+05].

Our approach replaces the path analysis by ILP (and possibly refinement for infeasible paths) by optimization modulo theory. The control-flow extraction and micro-architectural analysis are left unchanged, and one may thus use existing WCET tools.

6.3 Using Bounded Model Checking to Measure Worst-Case Execution Time

Bounded model checking, as described in section 2.2.1, is an approach for finding software bugs, where traces of length at most n are exhaustively explored. In most current approaches, the set of feasible traces of length n is defined using a first-order logic formula, where, roughly speaking, arithmetic constraints correspond to tests and assignments, control flow is encoded using Booleans, and disjunctions correspond to multiple control edges. This encoding is described in subsection 4.2.1, in the case the program is expressed as a control-flow graph. If constructs occur in the source program that cannot be translated exactly into the target logic (e.g. the program has nonlinear arithmetic but the logic does not), they may be safely over-approximated by nondeterministic choice.

Let us now see how to encode a WCET problem into SMT. The first step is to unroll all loops up to statically determined bounds. In a simple model (which can be made more complex and realistic, see section 6.9), each program block i has a fixed execution time $c_i \in \mathbb{N}$ given by the micro-architectural analysis. The total execution time $T(\tau)$ of a program trace τ is the sum of the execution times of the blocks encountered in the trace. This execution time can be incorporated into a "conventional" encoding for program traces in two ways, that we call *counter encoding* and *sum encoding*.

Definition 23 (Sum Encoding)**.** If Booleans $b_i \in \{0, 1\}$ record which blocks i were reached by

the execution trace τ, then the *Sum Encoding* if defined as:

$$T(\tau) = \left(\sum_{i|b_i=true} c_i \right) = \left(\sum_i b_i c_i \right) \tag{6.1}$$

Alternatively, the micro-architectural analysis may attach a cost to transitions instead of program blocks. The sum encoding is then done similarly, with Booleans $t_{i,j} \in \{0,1\}$ recording which of the transitions have been taken by an execution trace τ.

$$T(\tau) = \left(\sum_{(i,j)|t_{i,j}=true} c_{i,j} \right) = \left(\sum_{(i,j)} t_{i,j} c_{i,j} \right) \tag{6.2}$$

Definition 24 (Counter Encoding)**.** The program can be modified by adding a time counter as an ordinary variable, which is incremented in each block by the timing value of the block. Then, the final value of the counter at the end of the program is the execution time. In the case the costs are attached to transitions, such encoding is also possible: the cost variable in encoded as a Φ-instruction in each basic block with several incoming edges.

The choice to attach costs either to blocks or transitions is discussed later in subsection 6.6.2.

The problem is now how to determine the WCET $\beta = \max\ T(\tau)$. An obvious approach is binary search [ST12], maintaining an interval $[l, h]$ containing β. l and h may be respectively initialized to zero and a safe upper bound on worst-case execution time, for instance one obtained by a simple "longest path in the acyclic graph" algorithm, or simply taking the sum of all the c_i's. Algorithm 10 describes the algorithm for optimizing an integer variable *cost* representing the execution time.

Algorithm 10

```
1: function OptimizeCost(ρ, variable cost)        ▷ cost is assigned to T(τ) in ρ
2:     l ← 0, h ← Σ_i c_i
3:     while l ≠ h do
4:         m ← ⌈(l+h)/2⌉
5:         res ← SmtSolve[ρ ∧ cost ≥ m]
6:         if res = unsat then
7:             h ← m − 1
8:         else
9:             M ← getModel(), l ← M(cost)
10:        end if
11:    end while
12:    return h
13: end function
```

6.4 Intractability: Diamond Formulas

When experiencing with the encoding described in section 6.3, it turns out that solving the resulting formula takes far too much time for all state-of-the-art SMT-solvers, and most of the time does not terminate within hours or even days when analysing medium-size programs. In this section, we demonstrate both theoretically and experimentally why this encoding is not suitable for modern SMT-solvers based on the DPLL($\mathcal{T}$) framework.

Consider a program consisting in a sequence of n fragments where the i-th fragment has the form depicted in Figure 6.3.

```
if (b_i) { /* block of cost x_i */
  THEN1 /* time cost 2, not changing b_i */
} else {
  ELSE1 /* time cost 3, not changing b_i */
}
if (b_i) { /* block of cost y_i */
  THEN2 /* time cost 3 */
} else {
  ELSE2 /* time cost 2 */
}
```

Figure 6.3: Program fragment with maximal cost equal to 5.

The $(b_i)_{1\leq i\leq n}$ are Booleans. A human observer easily concludes that the worst-case execution time is $5n$, by analyzing each fragment separately, since it is easy to see that the constraint $x_i + y_i \leq 5$ holds.

This examples motivates the use of SMT instead of the traditional IPET approach using ILP: here, IPET would not be able to bound the WCET to $5n$ and would actually only bound it to $6n$, unless the user manually encodes into the ILP problem all the constraints of the form THEN1 + ELSE2 ≤ 1 and THEN2 + ELSE1 ≤ 1. For instance, the first constraint means that the branch THEN1 and ELSE2 are mutually exclusive. We explain here that the SMT approach yields the expected result $5n$, but suffers from scalability issues due to the huge number of paths in the program.

Using the "sum encoding" for creating the SMT problem, the timing analysis is expressed as

$$T = \max\left\{ \sum_{i=1}^{n} x_i + y_i \;\middle|\; \bigwedge_{i=1}^{n} (x_i = ite(b_i, 2, 3)) \wedge (y_i = ite(b_i, 3, 2)) \right\} \tag{6.3}$$

Given a bound m, an SMT-solver will have to solve for the unknowns $(b_i), (x_i), (y_i)_{1\leq i\leq n}$ the following constraint:

$$\begin{aligned} &((b_1 \wedge x_1 = 2 \wedge y_1 = 3) \vee (\neg b_1 \wedge x_1 = 3 \wedge y_1 = 2)) \wedge \ldots \\ &\qquad ((b_n \wedge x_n = 2 \wedge y_n = 3) \vee (\neg b_n \wedge x_n = 3 \wedge y_n = 2)) \wedge \\ &\qquad\qquad x_1 + y_1 + \cdots + x_n + y_n \geq m \end{aligned} \tag{6.4}$$

In the DPLL($\mathcal{T}$) and CDCL frameworks (see section 2.2.1) SMT is implemented as a combination of a SAT solver, which searches within a Boolean state space (here, amounting to $b_1, \ldots, b_n \in \{0,1\}^n$, but in general arithmetic or other theory predicates are also taken into account) and a decision procedure for conjunctions of atomic formulas from a theory $\mathcal{T}$. In this context, the combination of propositional and theory reasoning proceeds by sending clauses constructed from the predicates syntactically present in the original formula to the propositional solver. Once $b_1, \ldots, b_n$ have been picked, Equation 6.4 simplifies to a conjunction

$$x_1 = \alpha_1 \wedge y_1 = \beta_1 \wedge \ldots \wedge x_n = \alpha_n \wedge y_n = \beta_n \wedge x_1 + y_1 + \cdots + x_n + y_n \geq m \tag{6.5}$$

where the α_i, β_i are constants in $\{2, 3\}$ such that for each i, $\alpha_i + \beta_i = 5$. Such a formula is satisfiable if and only if $m \leq 5n$.

Assume now $m > 5n$. All combinations of $b_1, \ldots, b_n$ lead to unsatisfiable constraints, thus Equation 6.4 is unsatisfiable. If we exhaustively explore all these combinations, it is equivalent to exploring 2^n paths in the control flow graph, computing the execution time for each and comparing it to the bound. Let us now see how an SMT-solver will behave in practice with this formula. The SMT-solver, when exploring the Boolean state space, may detect that the current Boolean choices (say, $b_3 \wedge \neg b_5 \wedge b_7$) lead to an arithmetic contradiction, without picking a value for all the Booleans. The SMT-solver extracts a (possibly smaller) contradiction, also called *unsatisfiable core*, for instance $b_3 \wedge \neg b_5$. It adds the negation of this contradiction to the Boolean constraints as a *theory clause*, and restarts Boolean solving. The hope is that there exist short contradictions that enable the SMT-solver to prune the Boolean search space. Yet, in our case, there are no such short contradictions: if one leaves out *any* of the conjuncts in conjunction 6.5, the conjunction becomes satisfiable. Note the asymmetry between proving satisfiability and unsatisfiability: for satisfiability, one can always hope that clever heuristics will lead to one solution, while for unsatisfiability, the prover has to close all branches in the search.

The difficulty of Equation 6.4 or similar "diamond formulas" is well-known in SMT circles. It boils down to the SMT-solver working exclusively with the predicates found in the original formulas, without deriving new useful ones such as $x_i + y_i \leq 5$. All state-of-the-art solvers that we have tried have exponential running time in n when solving Equation 6.4 for $m = 5n$ (Figure 6.5).

The problem here does not come from the optimization modulo theory built on top of the SMT framework, but from the resolution techniques for solving SMT themselves: the special version of MathSAT 5, called Opti-MathSAT, which was kindly made available to us by the authors [ST12], implements the binary search approach internally. It suffers from the same exponential behavior as noted in the figure: in its last step, it has to prove that the maximum obtained truly is maximum.

The difficulty increases exponentially as the upper bound on the WCET to be proved becomes closer to the actual WCET. This phenomenon is clearly illustrated in Figure 6.6. Suppose we want to prove there exists no trace longer than $m = 5n + k$, for some $k > 0$. In that case, it is possible to derive a simpler *unsatisfiable core* rather than keeping the whole set of clauses.

As illustrated in Figure 6.4, it is sufficient for the first k portions to keep only the two clauses $x_k \leq 3$ and $y_k \leq 3$, and still get an unsatisfiable set of clauses. Thus, the problem is simpler since the blocking clause prunes 2^k traces at the same time instead of a single one.

There have been proposals of alternative approaches to DPLL($\mathcal{T}$), where one would directly solve for the numeric values instead of solving for Booleans then turning theory lemmas into Boolean constraints [Cot09, Cot10, MKS09, BDdM08, dMJ13]; but no production solver implements them.[4]

In the following, we incorporate so-called "cuts" in the encoding of the formula, in order to compute the WCET orders of magnitude faster than with the already described encoding.

[4]Dejan Jovanovic was kind enough to experiment with some of our formulas in his experimental solver [dMJ13], but the execution time was unacceptably high. We stress that this field of workable alternatives to DPLL($\mathcal{T}$) is still new and it is too early to draw conclusions.

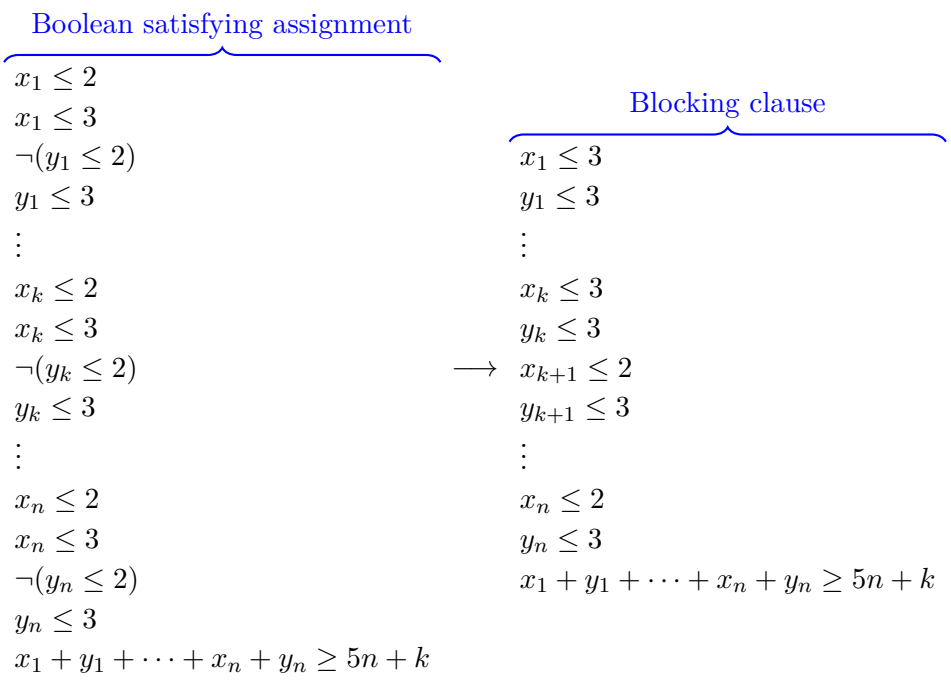


Figure 6.4: From a Boolean satisfying assignment to a blocking clause

6.5 Adding Cuts

Experiments with both sum encoding and counter encoding applied to the "conventional" encoding of programs into SMT from section 6.3 were disappointing: the SMT solver was taking far too much time. Indeed, these encodings lead to formulas with diamonds, that are intrinsically hard to solve by current solvers for the reasons explained before in section 6.4. The solver then explores a very large number of combinations of branches (exponential with respect to the number of tests), thus a very large number of partial traces $\tau_1, \ldots, \tau_n$, even though the execution time of these partial traces is insufficient to change the overall WCET.

Consider the control-flow graph in Figure 6.7; let $t_1, \ldots, t_7$ be the WCET of blocks $1 \ldots 7$ established by micro-architectural analysis (for the sake of simplicity, we neglect the time taken for decisions). Assume we have already found a path from start to end going through block 6, taking β time units; also assume that $t_1 + t_2 + \max(t_3, t_4) + t_5 + t_7 \leq \beta$. Then it is useless for the SMT-solver to search for paths going through decision 2, because none of them can have execution time longer than β; yet that is what happens if using a naive encoding with all current production SMT-solvers (see section 6.4). If instead of 1 decision we have 42, then the solver may explore 2^{42} paths even though there is a simple reason why none of them will increase the WCET.

Our proposal is simple: to the original SMT formula (from "counter encoding" or "sum encoding"), conjoin constraints expressing that the total execution time of some portions of the program is less than some upper bound (depending on the portion). This upper bound acts as an "abstraction" or "summary" of the portion (e.g. here we say that the time taken in P_2 is at most $t_2 + \max(t_3, t_4) + t_5$), and the hope is that this summary is sufficient for the SMT-solver in many cases. Two problems remain: how to select such portions, and how to compute this upper bound.

Note that these extra constraints are implied by the original formula, and thus that conjoin-

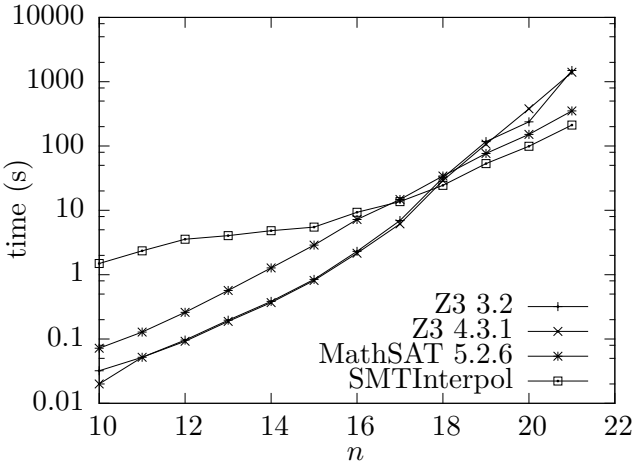


Figure 6.5: Intractability of diamond formulas obtained from timing analysis of a family of programs with very simple functional semantics. Execution times of various state-of-the-art SMT-solvers on Equation 6.4, for $m = 5n$ (the hardest), showing exponential behavior in the formula size n. The CPU is a 2 GHz Intel Core 2 Q8400.

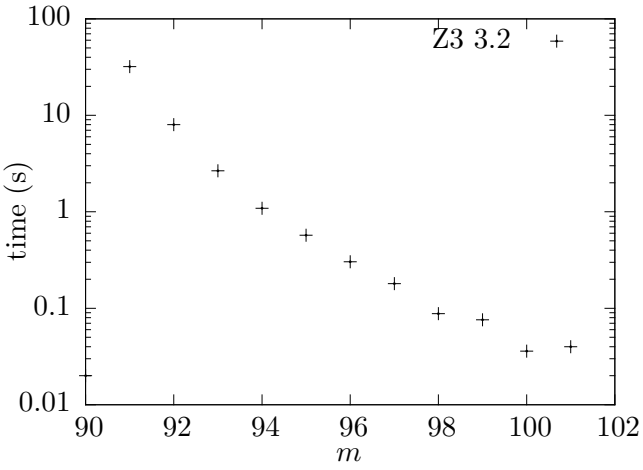


Figure 6.6: The complexity of proving the absence of traces longer than m decreases exponentially as m increases above the length of the longest trace, here, $n = 18$, and the real WCET is $5n = 90$.

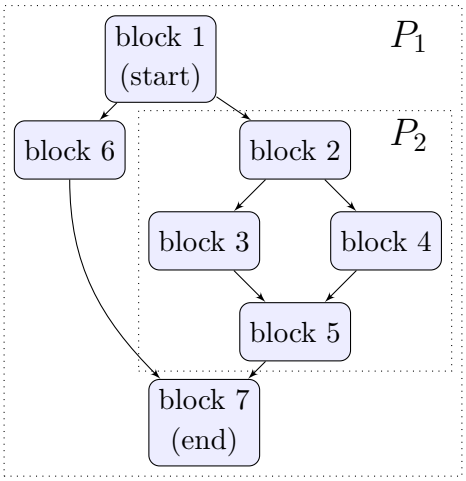


Figure 6.7: Two portions P_1 and P_2 of a program obtained as the range between a node with several incoming edges and its immediate dominator

ing them to it does not change the solution set or the WCET obtained, but only the execution time of the analysis. Such constraints are often called "cuts" in operation research, thus our terminology.

If we consider the program from Figure 6.3, for which we could not find an unsatisfiable core sufficiently general to prune sets of traces at the same time. Suppose that for each program fragment, we conjoin to the initial SMT formula the linear constraints $x_k + y_k \leq 5$ for all $k \leq n$. These constraints do not change the models of the initial formula. However, it turns out that there exists now unsatisfiable cores much more general than previously:

$$\overbrace{\begin{array}{l} x_1 \leq 2 \\ x_1 \leq 3 \\ \neg(y_1 \leq 2) \\ y_1 \leq 3 \\ x_1 + y_1 \leq 5 \\ \vdots \\ x_n \leq 2 \\ x_n \leq 3 \\ \neg(y_n \leq 2) \\ y_n \leq 3 \\ x_n + y_n \leq 5 \\ x_1 + y_1 + \cdots + x_n + y_n > 5n \end{array}}^{\text{Boolean satisfying assignment}} \longrightarrow \overbrace{\begin{array}{l} x_1 + y_1 \leq 5 \\ \vdots \\ x_n + y_n \leq 5 \\ x_1 + y_1 + \cdots + x_n + y_n > 5n \end{array}}^{\text{Blocking clause}}$$

In this ideal case, every traces in the program are pruned in one single step. The solving time with any production-grade solver drops from "non-terminating after one night" to "a few seconds" for large values of n. In the following, we propose ways of choosing these portions for deriving interesting cuts that may prune large sets of traces at once.

6.5.1 Selecting Portions

We propose different criteria for selecting the portions and deriving appropriate cuts.

Control-Structure Criterion

The choice of a portion of code to summarize follows source-level criteria: for instance, a procedure, a block, a macro expansion. If operating on a control-flow graph, a candidate portion can be between a node with several incoming edges and its *immediate dominator*, if there is non trivial control flow between the two (Fig. 6.7). the notion of immediate dominator is defined as follows:

Definition 25 (Immediate Dominator)**.** A *dominator* D of a block B is a block such that any path reaching B must go through D. The *immediate dominator* of a block B is the unique $I \neq B$ dominator of B such that I does not dominate any other dominator $D \neq B$ of B. For instance, the immediate dominator of the end of a cascade of if-then-else statements is the beginning of the cascade.

Definition 26 (Immediate Post Dominator)**.** A block P is said to *post dominate* a block B if all path to the final block that goes through B also goes through P. Similarly, the *immediate post dominator* of a block B is the postdominator of B that does not postdominate any postdominator of B (intuitively, it is the "closest" post dominator of B).

On structured languages, this means that we add one constraint for the total timing of every "if-then-else" or "switch" statement (recall that loops are unrolled, if needed into a cascade of "if-then-else"). This is the approach that we followed in our experimental evaluation (section 6.6).

Semantic Criterion

Let us however remark that these portions of code need not be contiguous: with the sum encoding, it is straightforward to encode the fact that the total time of a number of instruction blocks is less than a bound, even though these instructions blocks are distributed throughout the code. This is also possible, but less easy, with the counter encoding (one has to encode an upper bound on the sum of differences between starting and ending times over all contiguous subsets of the portion). This means that it is possible to consider portions that are semantically, but not syntactically related. For instance, one can consider for each Boolean, or other variable used in a test, a kind of "slice" of the program that is directly affected by this variable (e.g. all contents of if-then-elses testing on this variable) and compute an upper bound for the total execution time of this slice. In Figure 6.1, we consider the variable clock and compute an upper bound on the total time of A and B, since their execution is affected by this variable. In Figure 6.3, we consider the set of variables $(b_i)_{0 \leq i \leq n}$, and slice the program according to each of these Booleans. The resulting sliced graphs give us the appropriate portions.

We describe in Algorithm 11 the procedure that computes the set of semantic portions. We use the notion of *Dependency Graph* of a variable.

Definition 27 (Dependency Graph)**.** Let G denote a control-Flow Graph in SSA form. The *Dependency Graph* of G is a graph where the nodes are the program variables. There is a directed edge from v to v' if and only if the definition of v depend on the value of v'. This can happen for two reasons:

- v is the result of an instruction that takes v' as operand,
- or, v is a Φ-instruction in a basic block b, and the branch condition in the immediate dominator of b has v' as operand.

In the case G has no cycles, the dependency graph of has no cycle.

We call *Dependency Graph of a variable* v the same graph where we only keep the edges and nodes reachable from the node v.

Since the dependency graph of a program variable may be too large, we can limit its depth up to a given length k.

Algorithm 11

```
1: function PORTION(CFG G, BasicBlock t)
2:     succ ← ImmediatePostDominator(t)
3:     return {b ∈ G / ∃ a path from t to succ going through b}
4: end function
5: function SEMANTICPORTIONS(CFG G, Depth k)
6:     P ← ∅                                          ▷ Set of semantic portions
7:     V ← set of variables involved in tests.
8:     for all variable v ∈ V do
9:         S ← Dependency graph of v up to depth k
10:        T ← {b ∈ G / b has a conditional branch involving a variable in S}
11:        for all T' ⊆ T s.t. there exists a path in G traversing all blocks in T' do
12:            P ← P ∪ {⋃_{t∈T'} PORTION(t)}
13:        end for
14:    end for
15:    return P
16: end function
```

6.5.2 Obtaining Upper Bounds on the WCET of Portions

Let us now consider the problem of computing, given a portion, an upper bound on its WCET. There are several ways of deriving this upper bound, either by a simple syntactic analysis or by a more refined semantic analysis.

Syntactic Upper Bound

In the case of a contiguous portion, an upper bound may be obtained by a simple syntactic analysis: the longest syntactic path is used as a bound (even though it might be unfeasible). This approach may be extended to non-contiguous portions. Let us denote by P the portion. For each block b, let t_b be the upper bound on the time spent in block b (obtained from micro-architectural analysis), and let w_b be an unknown denoting the worst time spent inside P in paths from the start of the program to the beginning of b. If $b_1, \ldots, b_k$ are the predecessors of b, then $w_b = \max(w_{b_1} + t_{b_1}.\chi_P(b_1), \ldots, w_{b_k} + t_{b_k}.\chi_P(b_k))$ where $\chi_P(x)$ is 1 if $x \in P$, 0 otherwise. If the costs are attached to transitions instead of blocks, the computation of w_b can be done similarly. This system of equations can be easily solved in (quasi) linear time by considering the w_b in a topological order of the blocks (recall that we consider loop-free programs).

This syntactic approach gave excellent results in most benchmarks, and thus is the one we use in our experiments detailed in section 6.6.

Semantic Upper Bound

It may be the case that the cuts derived by the syntactic approach are not sufficiently precise to prune many traces. Having more precise upper bounds indeed improves our chance to cut traces as soon as possible. Another more precise approach is then to compute a semantic-sensitive upper bound for each portions. We propose two ways of deriving such upper bound by recursively calling the WCET procedure:

1. Once the portion is isolated, encode the semantics of this portion as an SMT formula, and compute the WCET of the portion using the same algorithm used for the entire program.

2. The second approach operates on the SMT formula of the entire program, and does not build a different formula per portion. Instead of directly optimizing the total cost variable of the program, we successively optimize the integer variables expressing the cuts, in order of portion size. Typically, the size of the portion is its number of contained basic blocks or transitions. These smaller problems are much easier to solve than the initial one, since only the SMT variables relative to the small portion will be relevant. This allows to strengthen the cuts with smaller upper bounds, and helps the analysis of the bigger portions. This approach can be seen as a "divide and conquer" algorithm. Note that this approach is not similar to the first one, since it takes into account possible inputs and outputs of the portion in the context of the entire program. Thus, the resulting upper bound may be smaller.

6.5.3 Example

Let us now see a short, but complete example, extracted from a control program composed of an initialization phase followed by an infinite loop clocked at a precise frequency. The goal of the analysis is to show that the WCET of the loop body never exceeds the clocking period. For the sake of brevity, we consider the "rate limiter" program that we already used in 4.2.2. The code run at every clock tick is depicted in Figure 6.8.

```
// returns a value between min and max
extern int input(int min, int max);

void rate_limiter_step() {
  int x_old = input(-10000,10000);
  int x = input(-10000,10000);
  if (x > x_old+10)
    x = x_old+10;
  if (x < x_old-10)
    x = x_old-10;
  x_old = x;
}
```

Figure 6.8: Rate Limiter Example

This program is compiled to LLVM bitcode [LA04]. Then bitcode-level optimizations are applied, resulting in an LLVM loop-free control-flow graph. From this graph we generate a

first-order formula including cuts (Figure 6.9, above). Its models describe execution traces along with the corresponding execution time *cost* given by the "sum encoding". Here, costs are attached to the transitions between each pairs of blocks. These costs are supposed to be given. subsection 6.6.2 will describe in full details how we use the OTAWA tool to derive such precise costs for each transitions.

The SMT encoding of the program semantics (Figure 6.9, below) is done as described in subsection 4.2.1. In our encoding, each transition t_i_j have a cost c_i_j given by OTAWA. For instance, the block `entry` is given the Boolean b_0, the block `if.then` is given the Boolean b_1, and the transition from `entry` to `if.then` is given the Boolean t_0_1 and has a cost of 15 clock cycles. The cuts are derived as follows: `if.end` has several incoming transitions and its immediate dominator is `entry`. The longest syntactic path between these two blocks is equal to 21. The cut will then be $c\_0\_1 + c\_1\_2 + c\_0\_2 \leq 21$. There is a similar cut for the portion between `if.end` and `if.end6`. Finally, we can also add the constraint $cost \leq 43$ since it is the cost of the longest syntactic path. While this longest syntactic path has cost 43 (it goes both through `if.then` and `if.then4`), our SMT-based approach shows there is no semantically feasible path longer than 36 clock cycles.

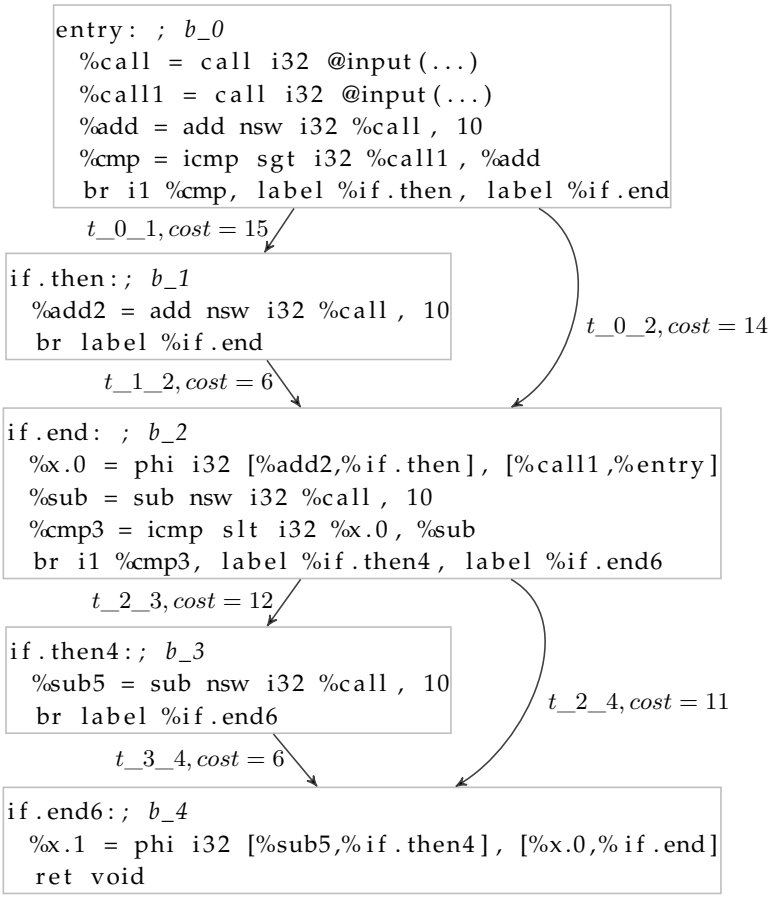
6.5.4 Relationship with Craig Interpolants

A *Craig interpolant* for an unsatisfiable conjunction $F_1 \wedge F_2$ is a formula I such that $F_1 \Rightarrow I$ and $I \wedge F_2$ is unsatisfiable, whose free variables are included in the intersection of those of F_1 and F_2.

In the case of a program $A; B$ consisting of two portions A and B executed in sequence, the usual way of encoding the program yields $\phi_A \wedge \phi_B$ where ϕ_A and ϕ_B are, respectively, the encodings of A and B. The free variables of this formula are the inputs $i_1, \ldots, i_m$ and outputs $o_1, \ldots, o_n$ of the program, as well as all temporaries and local variables. Let $l_1, \ldots, l_p$ be the variables live at the edge from A to B; then the input-output relationship of the program, with free variables $i_1, \ldots, i_m, o_1, \ldots, o_n$ is F :

$$\exists l_1, \ldots, l_p (\exists \ldots \phi_A) \wedge (\exists \ldots \phi_B)$$

When using the counter encoding, one of the input variables $i_1, \ldots, i_m$ actually refers to the initial time cost (hence equals 0), and one of the output variables $o_1, \ldots, o_n$ is the final cost of the trace, which is the value we try to optimize. Similarly, one of the variables $l_1, \ldots, l_p$ corresponds to the execution time of the trace before entering block B. Let us now assume without loss of generality that o_1 is the final time and l_1 is the time when control flow from A to B in the counter encoding. The SMT formulas used in our optimization process are of the form $F \wedge o_1 \geq \beta$. The cut for portion A is of the form $l_1 \leq \beta_A$, that for portion B of the form $o_1 - l_1 \leq \beta_B$. Then, if the cut for portion A is used to prove that $F \wedge o_1 \geq \beta$ is unsatisfiable, then this cut is a Craig interpolant for the unsatisfiable formula $(\phi_A) \wedge (\phi_B \wedge o_1 \geq \beta)$ (similarly, if the cut for portion B is used, then it is an interpolant for $\phi_B \wedge (\phi_A \wedge o_1 \geq \beta)$. Our approach may thus be understood as preventively computing possible Craig interpolants so as to speed up solving. The same intuition applies to the sum encoding (up to the creation of supplementary variables). Computing "interesting" Craig interpolants that prune large sets of traces is an active research topic. Here, we claim that the interpolants we preemptively produce will help the SMT solver in many cases. This claim is stressed by experimental results on real programs.



$$
\begin{aligned}
& -10000 \le call \le 10000 \\
\wedge\ & -10000 \le call1 \le 10000 \\
\wedge\ & add = (call + 10) \\
\wedge\ & t\_0\_1 = (b\_0 \wedge (call1 > add)) \\
\wedge\ & t\_0\_2 = (b\_0 \wedge \neg(call1 > add)) \\
\wedge\ & b\_1 = t\_0\_1 \\
\wedge\ & add2 = (call + 10) \\
\wedge\ & t\_1\_2 = b\_1 \\
\wedge\ & b\_2 = (t\_0\_2 \vee t\_1\_2) \\
\wedge\ & b\_2 \Rightarrow (x.0 = ite(t\_1\_2, add2, call1)) \\
\wedge\ & sub = (call - 10) \\
\wedge\ & t\_2\_3 = (b\_2 \wedge (x.0 < sub)) \\
\wedge\ & t\_2\_4 = (b\_2 \wedge \neg(x.0 < sub)) \\
\wedge\ & b\_3 = t\_2\_3 \\
\wedge\ & sub5 = (call - 10) \\
\wedge\ & t\_3\_4 = b\_3 \\
\wedge\ & b\_4 = (t\_2\_4 \vee t\_3\_4) \\
\wedge\ & b\_4 \Rightarrow (x.1 = ite(t\_3\_4, sub5, x.0)) \\
& \text{; seach for a trace from entry to if.end6} \\
\wedge\ & b\_0 = b\_4 = true
\end{aligned}
$$

$$
\text{timing}\left\{\begin{array}{ll}
 & \text{; } t\_0\_1 \text{ has cost 15 if taken, else 0} \\
\wedge & c\_0\_1 = ite(t\_0\_1, 15, 0) \\
\wedge & c\_0\_2 = ite(t\_0\_2, 14, 0) \\
\wedge & c\_1\_2 = ite(t\_1\_2, 6, 0) \\
\wedge & c\_2\_3 = ite(t\_2\_3, 12, 0) \\
\wedge & c\_2\_4 = ite(t\_2\_4, 11, 0) \\
\wedge & c\_3\_4 = ite(t\_3\_4, 6, 0) \\
\wedge & cost = (c\_0\_1 + c\_0\_2 + c\_1\_2 + c\_2\_3 \\
 & \qquad + c\_2\_4 + c\_3\_4)
\end{array}\right.
$$

$$
\text{cuts}\left\{\begin{array}{ll}
 & \text{; between entry and if.end} \\
\wedge & (c\_0\_1 + c\_1\_2 + c\_0\_2) \le 21 \\
 & \text{; between if.end and if.end6} \\
\wedge & (c\_3\_4 + c\_2\_4 + c\_2\_3) \le 22 \\
\wedge & cost \le 43
\end{array}\right.
$$

Figure 6.9: LLVM control-flow graph of the rate_limiter_step function, and its encoding as an SMT formula with cuts.

6.6 Experimental Results

We experimented our approach for computing the worst-case execution time on benchmarks from several sources, referenced in Table 6.1. `nsichneu` and `statemate` belong to the Mälardalen WCET benchmarks set [GBEL10][5], being the largest of the set (w.r.t. code size). `cruise-control` and `digital-stopwatch` are generated from SCADE™ designs. `autopilot` and `fly-by-wire` come from the Papabench benchmark [NCS+06] derived from the Paparazzi free software suite for piloting UAVs (`http://paparazzi.enac.fr/`). `tdf` and `miniflight` are industrial avionic case-studies.

We use the infrastructure of the PAGAI static analyzer to produce an SMT formula corresponding to the semantics of a program expressed in LLVM bitcode. This part is fully detailed later in chapter 7. Once the SMT formula is constructed, we enrich it with an upper timing bound for each basic block.

Finally, we conjoin to our formula the cuts for the "sum encoding", i.e., constraints of the form $\sum_{i \in S} c_i \leq B$, where the c_i's are the cost variables attached to the basic blocks. There is one such "cut" for every basic block with several incoming edges, according to the control-structure criterion described in section 6.5.1. The bound B is the weight of the maximal path through the range, be it feasible or infeasible, and is computed as described in section 6.5.2 (we do not apply the semantic criterion for computing precise upper bounds).

Our current implementation keeps inside the program the resulting formulas statements and variables that have no effect on control flow and thus on WCET. Better performance could probably be obtained by slicing away such irrelevant statements. Furthermore, some paths are infeasible because of a global invariant of the control loop (e.g. some Booleans a and b activate mutually exclusive modes of operations, and $\neg a \vee \neg b$ is an invariant); we have not yet integrated such invariants, which could be obtained either by static analysis of the program, either by analysis of the high-level specification from which the program is extracted [AMR13].

We use Z3 [dMB08] as an SMT solver and a binary search strategy to maximize the *cost* variable modulo SMT.

6.6.1 Results with Bitcode-Based Timing

The problem addressed in this article is not architectural modeling and low-level timing analysis: we assume that worst-case timings for basic blocks are given by an external analysis. Here we report on results with a simple timing basis: the time taken by an LLVM bitcode block is its number of instructions; our goal here is to check whether improvements to WCET can be obtained by our analysis with reasonable computation costs, independently of the architecture.

As expected, the naive approach (without adding cuts to the formula) does not scale at all, and the computation has reached our timeout in all of our largest benchmarks. Once the cuts are conjoined to the formula, the WCET is computed considerably faster, with some benchmarks needing less than a minute while they timed out with the naive approach.

Our results (Table 6.2, first part) show that the use of bounded model checking by SMT solving improves the precision of the computed upper bound on the worst-case execution time, since the longest syntactic path is in most cases not feasible due to the semantics of the instructions. As usual with WCET analyzes, it is difficult to estimate the absolute quality of the resulting bound, because the exact WCET is unknown (perhaps what we obtain is actually the WCET, perhaps it overestimates it somewhat).

[5] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Benchmark name	LLVM #lines	LLVM #BB
statemate	2885	632
nsichneu	12453	1374
cruise-control	234	43
digital-stopwatch	1085	188
autopilot	8805	1191
fly-by-wire	5498	609
miniflight	5860	745
tdf	2689	533

Table 6.1: Table referencing the various benchmarks. LLVM #lines is the number of lines in the LLVM bitcode, and LLVM #BB is its number of Basic Blocks.

On the `autopilot` software, our analysis reduces the WCET bound by 69.7%. This software has multiple clock domains, statically scheduled by the `periodic_task()` function using switches and arithmetic constraints. Approaches that do not take functional semantics into account therefore consider activation patterns that cannot occur in the real system, leading to a huge overestimation compared to our semantic-sensitive approach.

6.6.2 Results with Realistic Timing

The timing model used in the preceding subsection is not meant to be realistic. We therefore experimented with realistic timings for the basic blocks, obtained by the Otawa tool [BCRS10] for an ARMv7 architecture.

The difficulty here is that Otawa considers the basic blocks occurring in binary code, while our analysis considers the basic blocks in the LLVM bitcode. Since we apply the analysis to the bitcode obtained after all machine-independent optimization passes (including all passes that perform reorganizations such as loop unrolling), the blocks in LLVM bitcode are close to those in the binary code. However, they are not *exactly* those in the binary code, because code generation in the LLVM backend may slightly change the block structure (for instance, out-of-SSA may add move instructions over control edges, which may translate into supplemental control blocks). The matching of binary code to LLVM bitcode is thus imperfect in some cases and we had to resort to one that safely overestimates the execution time. Figure 6.11 gives an overview of the general workflow for deriving the appropriate costs of LLVM basic blocks. When the matching is perfect, the worst-case execution time obtained by the whole program analysis of Otawa is equal to the "syntactic worst case" approach that we improve upon using SMT-solving; when it is imperfect our "syntactic worst case" may be worse than the one given by Otawa. Because of this matching, we assign costs to transitions instead of basic blocks as usual, since it reduces the overapproximation in the WCET of each LLVM basic block. Figure 6.10 shows the superposition of the ARMv7 and LLVM control flow graphs, as well as the cost for each ARM block BBi given by Otawa. In this special case, the matching does not need overapproximation.

In order to solve these issues, the solution would be to generate the SMT formulas not from LLVM bitcode, but directly from the binary code; unfortunately a reliable implementation needs to address a lot of open questions, and as such, it falls into our future plans.

The results are given in Table 6.2 (second half). The improvement is comparable to our results with Bitcode-based timings. However, the resulting SMT formulas are harder to deal

Benchmark name	WCET bounds			Analysis time (in seconds)		#cuts
	syntactic/OTAWA	max-SMT	diff	with cuts	without cuts	
Bitcode-based timings (in number of LLVM instructions)						
statemate	997	951	4.6%	118.3	$+\infty$	143
nsichneu	9693	5998	38.1%	131.4	$+\infty$	252
cruise-control	123	121	1.6%	0.1	0.1	13
digital-stopwatch	332	302	9.0%	1.0	35.5	53
autopilot	4198	1271	69.7%	782.0	$+\infty$	498
fly-by-wire	2932	2792	4.7%	7.6	$+\infty$	163
miniflight	4015	3428	14.6%	35.8	$+\infty$	251
tdf	1583	1569	0.8%	5.4	343.8	254
Realistic timings (in cycles) for an ARMv7 architecture						
statemate	3297	3211	2.6%	943.5	$+\infty$	143
nsichneu* (1 iteration)	17242	<13332**	22.7%	3600**	$+\infty$	378
cruise-control	881	873	0.9%	0.1	0.2	13
digital-stopwatch	1012	954	5.7%	0.6	2104.2	53
autopilot	12663	5734	54.7%	1808.8	$+\infty$	498
fly-by-wire	6361	5848	8.0%	10.8	$+\infty$	163
miniflight	17980	14752	18.0%	40.9	$+\infty$	251
tdf	5789	5727	1.0%	13.0	$+\infty$	254

Table 6.2: *max-SMT* is the upper bound on WCET reported by our analysis based on optimization modulo theory, while *syntactic*/OTAWA is the execution time of longest syntactic path (provided by OTAWA when using realistic timings). *diff* is the improvement brought by our method. The analysis time for *max-SMT* is reported with and without added cuts; $+\infty$ indicates timeout (1 hour). *#cuts* is the number of added cuts.
*) `nsichneu` has been simplified to one main-loop iteration (instead of 2), and has been computed with cuts refinement as described in subsection 6.6.2.
**) Computation takes longer that 1 hour. A safe bound of 13332 is already known after this time.

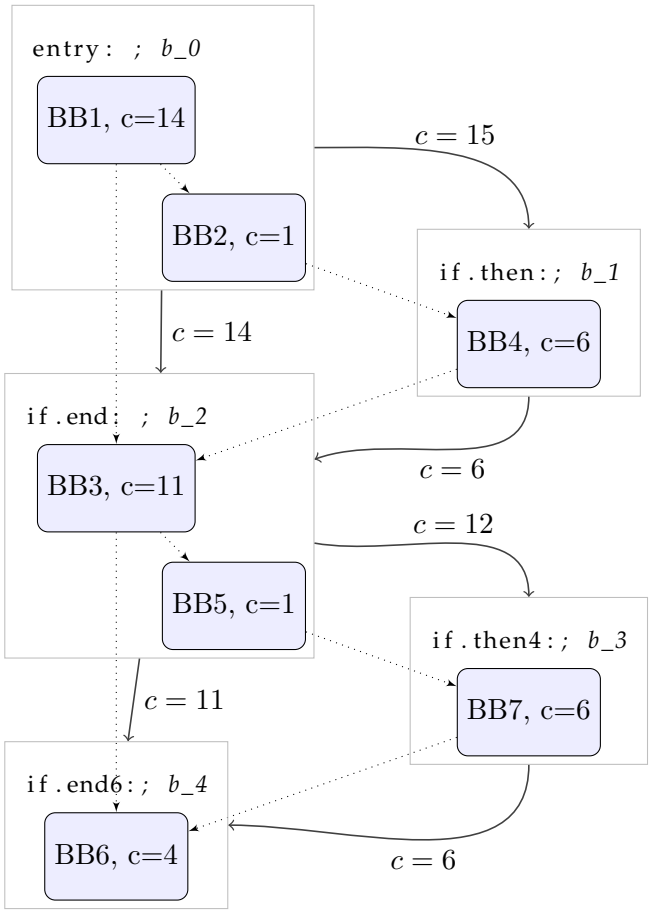


Figure 6.10: ARM control-flow graph (with 7 blocks) + traceability to LLVM IR control-flow graph for the example from subsection 6.5.3 and Figure 6.9

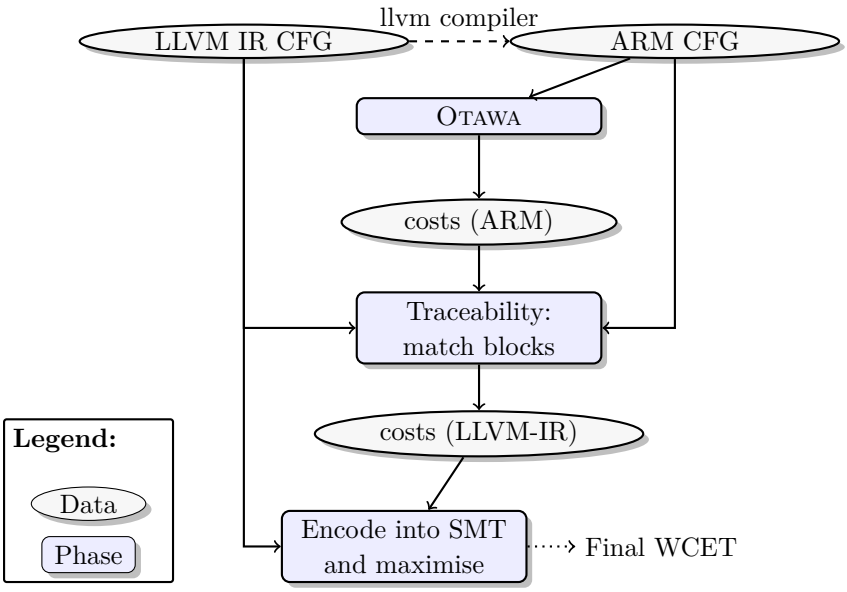


Figure 6.11: General workflow for deriving timings using Otawa.

with by the SMT solver. While the `nsichneu` benchmark is fully handled by our approach when using bitcode-based timing, it is much harder when using the realistic metric. We had to improve our implementation in two ways:

1. We extract cuts for larger portions of the program: we take the portions from our previous cuts (between merge points and their immediate dominators) and derive new cuts by recursively grouping these portions by two. We then have cuts for one half, one quarter, etc. of the program.
2. Instead of directly optimizing the total cost variable of the program, we successively optimize the variables expressing the "cuts" (in order of portion size). This allows to strengthen the cuts with smaller upper bounds, and helps the analysis of the bigger portions.

In this benchmark, all the biggest paths are unfeasible because of inconsistent semantic constraints over the variables involved in the tests. Better cuts could be derived if we were not restricted to contiguous portions in the implementation. The computation time is around 6.5 hours to get the exact WCET (13298 cycles), but we could have stopped after one hour and get a correct upper bound of 13332 cycles, which is already very close to the final result.

6.7 Related Work

The work closest to ours is from [CJ11]. They perform symbolic execution on the program, thereby unrolling an exponentially-sized execution tree (each if-then-else construct doubles the number of branches). This would be intolerably expensive if not for the very astute subsumption criterion used to fold some of the branches into others already computed. More specifically, their symbolic execution generalizes each explored state S to a first-order formula defining states from which the feasible paths are included in those starting from S; these formula are obtained from *Craig interpolants* extracted from the proofs of infeasibility.

In our approach, we also learn formula that block infeasible paths or paths that cannot lead to paths longer than the WCET obtained, in two ways: the SMT-solver learns blocking clauses

by itself, and we feed "cuts" to it. Let us now attempt to give a high-level view of the difference between our approach and theirs. Symbolic execution [CS13] (in depth-first fashion) can be simulated by SMT-solving by having the SMT-solver select decision literals [KS08] in the order of execution of the program encoded into the formula; in contrast, general bounded model checking by SMT-solving will assert predicates in an arbitrary order, which may be preferrable in some cases (e.g. if $x \leq 0$ is asserted early in the program and $x + y \geq 0$ very late, after multiple if-then-elses, it is useful to be able to derive $y \geq 0$ immediately without having to wait until the end of each path). Yet, an SMT-solver based on DPLL(T) does not learn lemmas constructed from new predicates, while the approach in [CJ11] learns new predicates on-the-fly from Craig interpolants. In our approach, we help the SMT-solver by preventively feeding "candidate lemmas", which, if used in a proof that there is no path longer than a certain bound, act as Craig interpolants, as explained in subsection 6.5.4. Our approach therefore leverages both out-of-order predicate selection and interpolation, and, as a consequence, it seems to scale better. For example, the specialized `nsichneu` benchmark, designed "to test the scalability of an analyzer", is fully handled only by our approach when using a source-based timing metric.

Two recent works — [BKKZ13] and its follow-up [KKZ13] — integrate the WCET path analysis into a counterexample guided abstraction refinement loop. As such, the IPET approach using ILP is refined by extracting a witness path for the maximal time, and testing its feasibility by SMT-solving; if the path is infeasible, an additional ILP constraint is generated, to exclude the spurious path. Because this ILP constraint relates all the conditionals corresponding to the spurious witness path, excluding infeasible paths in this way exhibits an exponential behavior we strove to avoid. Moreover, our approach is more flexible with respect to (1) the class of properties which can be expressed, as it is not limited by the ILP semantics and (2) the ability to incorporate non-functional semantics (which is unclear whether [BKKZ13] or [KKZ13] can).

[Met04] proposed an approach where the program control flow is encoded into a model along with either the concrete semantics of a simple model of instruction cache, or an abstraction thereof. The WCET bound is obtained by binary search, with each test performed using the VIS model-checker[6]. [HS09] proposed a similar approach with the model-checker UPPAAL.[7] In both cases, the functional semantics are however not encoded, save for loop bounds: branches are chosen nondeterministically, and thus the analysis may consider infeasible paths. [DOT+10] encode into UPPAAL precise models of a pipeline, instruction cache and data cache, but again the program is modeled as "data insensitive", meaning that infeasible paths are not discarded except when exceeding a loop bound.

[Hol08] considers a loop (though the same approach can also be applied to loop-free code): the loop is sliced, keeping only instructions and variables that affect control flow, and a global "timing" counter T is added; the input-output relation of the loop body is obtained as a formula in linear integer arithmetic (Presburger arithmetic); some form of *acceleration* is used to establish a relation between T, some induction variables and some inputs to the program. Applied to loop-free programs, this method should give exactly the same result as our approach. Its main weakness is that representations of Presburger sets are notoriously expensive, whereas SMT scales up (the examples given in the cited article seem very small, taking only a few lines and at most 1500 clock cycles for the entire loop execution); also, the restriction to Presburger arithmetic may exclude many programs, though one can model constructs outside of Presburger arithmetic by nondeterministic choices. Its strong point is the ability to precisely deal with loops, including those where the iteration count affects which program fragments are active.

[6] `http://vlsi.colorado.edu/~vis/`

[7] `http://www.uppaal.org/`

A recent approach to program verification is to express the semantics of programs into Horn clauses over unknown predicates, which represent the inductive invariants sought [BMR13]. These clauses naturally express groupings of program instructions. Algorithms for solving such Horn clauses are typically based on Craig interpolation, which generates new predicates; we saw in section 6.4 that the absence of generation of new predicates in DPLL($\mathcal{T}$) SMT-solvers is the reason why diamond formulas cause exponential cost. We experimented with the "unstable" version of the Z3 SMT-solver, which can solve Horn clauses, on encodings of the diamond WCET problems from section 6.4:

- a "flat" encoding, which directly translates the sequence of $2n$ if-then-else statements
- a "grouped" encoding, which first groups into a single predicate the blocks "A" and "B" associated with b_i, for all i.

The "flat" encoding, again, yields solving times in $\sim 2^n$. The "grouped" encoding yields instantaneous answers. This confirms our intuition that groups of associated statements should be abstracted as a unit, be it using interpolants as in Z3 or using precomputed bounds as in our approach. Unfortunately we have not had the time to attempt encoding large programs into Horn clauses so as to check whether the good performance on toy diamond examples translates to good performance on larger programs. We however do not see how the possibility that we have of keeping bounds of non-contiguous portions of code would easily translate into Horn clauses.

6.8 From our Modular Static Analysis to WCET Computation

Our approach for computing WCET using SMT has a tight relation with our modular static analysis described in chapter 5. In this section, we explain the relation between the input/output relations derived by our modular static analysis and the cuts derived in the particular case of WCET computation.

The computation of cuts mainly depends on the choice of portions of the control flow graph. We detailed two criteria for choosing the portions, one of them coming from the control structure. The cuts derived from these portions are actually input/output relations involving the variable expressing the cost with the counter encoding. Since the control flow graph has no cycle, the input/output relation can be simply and precisely computed with our modular path focusing detailed in section 5.3, only considering the cost variable as input and output variable for the block.

Then, our WCET approach using SMT can be seen as an instance of our modular static analysis with particular choices of program blocks and input/output variables. This tight relation brings several insights, that we would like to investigate in future work:

Deriving Stronger Cuts

In our approach, with the counter encoding, the cuts only involve the variable expressing the cost. This restriction may sometimes be a problem and better cuts may be derived from input/output relations involving all the live variables at the entry/exit points of the portions. Such relations can be computed by our modular static analysis framework from chapter 5: one should instrument the analyzed program with a variable *cost* which is incremented in each block by a given cost, as in the counter encoding. In the case of loop-free program portions, widenings are not required, and an upper bound for the cost variable can always be found. While our current cuts only bound the portion cost with a constant value, the one obtained by

this new approach would depend in general on the input and output variables of the program. In other words, instead of deriving cuts of the form $cost^o - cost^i < C$, where $cost^i$ (resp. $cost^o$) refers to the cost variable in the entry (resp. exit) point of the portion, we will get cuts of the form $C(cost^i, X^i) \Rightarrow R(cost^i, cost^o, X^{i/o})$, where X^i are the other input variables and $X^{i/o}$ are both the input and output variables.

These more precise cuts would perform much better in the case where the cost of the portion highly depends on some other program variables, e.g. if the local WCET of the portion is 5 if $x > 0$, and 50 otherwise.

However, if we use the modular path focusing with the settings of section 5.3, we loose the dichotomy strategy for optimizing the cost variable, which is important for preventing the enumeration of an exponential number of path. This issue can be solved by searching path *far* from the current candidate invariant as we explained in page 69.

Programs with Loops

It is possible to compute a precise upper bound of the WCET in the case of program with loops using the counter encoding. It is also possible with the sum encoding, by replacing the Boolean variables, used in the loop-free case for tracking which block was traversed, into integer variables, that track how many time each block is traversed. Then, one can again use our modular static analysis over the resulting program and obtain in the end a relation involving the cost variable. The loop can then be abstracted by the obtained relation and one finally obtains a loop-free control-flow graph. However, it is not guaranteed that an upper bound for the cost variable will be found, because of the overapproximation during the analysis, especially widenings. Future work include implementation and experiments with this approach.

6.9 Extensions and Future Work

The "counter encoding" is best suited for code portions that have a single entry and exit point (in which case they express the timing difference between entry and exit). In contrast, the "sum encoding" may be applied to arbitrary subsets of the code, which do not in fact need to be connected in the control-flow graph. One may thus use other heuristic criteria, such as usage of related variables.

A model based on worst-case execution times for every block, to be reassembled into a global worst-case execution time, may be too simplistic: indeed, the execution time of a block may depend on which blocks were executed beforehand, or, for finer modeling, on the value of pointer variables (for determining cache status).

A very general and tempting idea, as suggested earlier in MDD-based model-checking [Met04], in symbolic execution and bounded model checking by [CR13, CJ11], in combined abstract interpretation and SAT-solving [BCR13] is to integrate in the same analysis both the non-functional semantics (e.g. caches) and the functional semantics; in our case, we would replace both the micro-architectural analysis (or part of it) and the path analysis by a single pass of optimization modulo SMT. Because merely encoding the functional semantics and a simplistic timing model already led to intractable formulas, we decided to postpone such micro-architectural modeling until we had solved scalability issues. We intend to integrate such *non-functional* aspects into the SMT problem in future work.

Detailed modeling of the cache, pipeline, etc. may be too expensive to compute beforehand and encode into SMT. One alternative is to iteratively refine the model with respect to the

current "worst-case trace": to each basic block one attaches an upper bound on the worst-case execution time, and once a worst-case trace is obtained, a trace analysis is run over it to derive stronger constraints.

We have discussed obtaining a tight upper bound on the worst-case operation time of the program from upper bounds on the execution times of the basic blocks. If using lower bounds on the worst-case execution times of the basic blocks, one may obtain a lower bound on the worst-case execution time of the program. Having both is useful to gauge the amount of over-approximation incurred. Also, by applying minimization instead of maximization, one gets bounds on best-case execution time, which is useful for some scheduling applications [Wil06].

On a more practical angle, our analysis is to be connected to analyses both on the high level specification (e.g. providing invariants) and on the object code (micro-architectural timing analysis); this poses engineering difficulties, because typical compilation framework may not support sufficient tracing information.

Our requirement that the program should be loop-free, or at least contain loops with small constant bounds, can be relaxed through an approach similar to that of [CJ11]: the body of a loop can be summarized by its WCET and possibly some summary for the scalar variables of the program, then this entire loop can be considered as a single block in an analysis of a larger program.

6.10 A Different and Mixed Approach for Estimating WCET

We mention here some work that is currently in progress as part of the ANR W-SEPT[8] project.

Recall that in the standard approach, given worst-case timings for every basic blocks, one computes the WCET for the entire program by maximizing a cost function in integer linear programming (ILP), where the control flow of the program is encoded into constraints, together with maximal iteration counts for the loops. One then obtains as a result an upper bound on the WCET. This upper bound is semantics-insensitive, since it considers infeasible paths when solving the ILP problem. The usual workaround is to identify some infeasible paths either manually of with simple pre-analysis, and deduce some ILP constraints that prevent considering these paths when optimizing the cost function.

While we proposed solutions for replacing this ILP problem by Satisfiability Modulo Theory since the beginning of this chapter, we suggest here a way of improving the ILP approach by code instrumentation. The principle is the following:

1. Instrument the control flow graph with one integer counter per basic block, initialize them to zero, and increment them by one in their corresponding block.
2. Run a path-focusing based static analysis for deriving precise invariants involving the counters.
3. The invariant obtained in the last basic block gives relations between the various counters, e.g. linear constraints when using convex polyhedra. One can easily derive ILP constraints from these linear (in)equalities. These new constraints may express path infeasibility and thus improve the precision of the final WCET estimation fully automatically.

Example 14 (Mixed ILP + Path Focusing approach)**.** Figure 6.12 illustrates this idea on a motivating example, where an expensive program portion inside a loop is only visited once. This

[8] `http://wsept.inria.fr/`

example is a simplified version of a program given in [Par93, page 39]. If the ILP problem does not encode the constraint $counter2 \leq 1$, the obtained WCET would be highly overestimated. This approach allows to derive good ILP constraints, in a fully automatic manner, and still use the efficiency of SMT through path-focusing based analysis techniques.

```
i = 0;
while (i < 100) {
  if (i == 10) {
    // expensive computation
  }
  i++;
}
```

```
// instrumented-version
i = 0;
while (i < 100) {
  counter1++;
  if (i == 10) {
    counter2++;
    // expensive computation
  }
  i++;
}
// invariant: i = counter1 = 100 ∧ 0 ≤ counter2 ≤ 1
```

Figure 6.12: A simple example, and its instrumented version with two counters counter1 and counter2. The invariant in comments is the one obtained with our tool Pagai with guided path analysis from subsection 4.2.2.

6.11 Conclusion

We have shown that optimization using satisfiability modulo theory (SMT) is a workable approach for bounding the worst-case execution time of loop-free programs (or programs where loops can be unrolled). To our knowledge, this is the first time that such an approach was successfully applied.

Our approach computes an upper bound on the WCET, which may or may not be the actual WCET. The sources of discrepancy are

1. the micro-architectural analysis (e.g. the cache analysis does not know whether an access is a hit or a miss),
2. the composition of WCET for basic blocks into WCET for the program, which may lose dependencies on execution history[9],
3. the encoding of the program into SMT, which may be imprecise (e.g. unsupported constructs replaced by nondeterministic choices).

We showed that straightforward encodings of WCET problems into SMT yield problems intractable by all current production-grade SMT-solvers ("diamond formulas"), and how to work around this issue using a clever encoding. We believe this approach can be generalized to other properties, and lead to fruitful interaction between modular abstraction and SMT-solving.

While our redundant encoding brings staggering improvements in analysis time, allowing formerly intractable problems to be solved under one minute, the improvements in the WCET

[9]This does not apply to some simple micro-controller architectures, without cache or pipeline states, e.g. Atmel AVR™ and Freescale™ HCS12.

upper bound brought by the elimination of infeasible paths depend on the structure of the program being analyzed. The improvement on the WCET bound of some industrial examples (23%, 55%...) is impressive, in a field where improvements are often of a few percents. This means that, at least for certain classes of programs, it is necessary to take infeasible paths into account. At present, certain industries avoid using formal verification for WCET because it has a reputation for giving overly pessimistic over-estimates; it seems likely that some of this over-estimation arises from infeasible paths.

Our approach to improving bounds on WCET blends well with other WCET analyses. It can be coupled with an existing micro-architectural analysis, or part of that analysis may be integrated into our approach. It can be combined with precise, yet less scalable analyzes [KKZ13, Hol08] to summarize inner loops; but may itself be used as a way to summarize the WCET of portion of a larger program.

Chapter 7

The PAGAI Static Analyser

Many of the static analysis techniques developed throughout this thesis have been implemented in a new prototype of static analyser called PAGAI [HMM12a]. This tool aims at comparing various state-of-the-art techniques based on abstract interpretation in terms of precision and scalability. Indeed, many methods in static analysis have an exponential cost in the worst-case, but behave better in practice on most benchmarks. In addition, since the widening operator is non monotonic, a "smart" technique which is supposed to give precise results may finally give a less-precise invariant due to some widenings. For these two reasons, experimenting with the various techniques with a robust implementation is required.

However, it is difficult to compare the precision of two different tools, since their input language may be different, as well as the hypothesis and the overapproximations of diverse kinds they perform. PAGAI provides implementation of several techniques sharing the maximum of code, so that the comparisons between the different approaches are fully relevant. It can take as input large industrial source code and has already been used by other research teams for experimenting with new techniques [AS13]. Front-ends for many analysis tools put restrictions (e.g. no backward goto instructions, no pointer arithmetic...), often satisfied by safety-critical embedded programs, but not by generic programs; our tool suffers no such restrictions, though it may in some cases apply coarse abstractions which may possibly yield weak invariants.

The tool is implemented using the C++ language and has a tight integration within the LLVM compiler framework [LA04]. It is able to analyse program expressed as LLVM Intermediate Representation language[1], that can be obtained from C, C++, Objective-C and Objective-C++ code using the frontend CLANG[2], as well as other languages supported by the GCC compiler frontends through the DRAGONEGG[3] GCC plugin, for instance Ada or Fortran.

At present, PAGAI checks user-specified safety properties provided through assertions using the standard C/C++ `assert(condition)` macro, as well as various undefined behaviors (overflows in arithmetic operations and array out-of-bound accesses). The tool will attempt proving that the assertion failure or an undefined behavior is unreachable and, if unsuccessful, provides a warning message. PAGAI does not at present include bounded model checking or path exploration techniques for reconstructing an actual failure trace, thus such a warning message should be interpreted as a possible assertion failure. Executing traces falsifying assertions are considered to terminate when executing the macro; thus, user-specified assertions may be used to guide the analyzer by providing invariants that it was not able to synthesize by itself.

It also allows user-specified assumptions, through the `assume(condition)` macro.

PAGAI allows independent selection of the abstract domain (using the APRON library [JM09]) and the iteration strategy. For the strategies that require checking the satisfiability

[1] `http://llvm.org/docs/LangRef.html`
[2] `http://clang.llvm.org`
[3] `http://dragonegg.llvm.org`

of SMT formulas, PAGAI can interact with Microsoft Z3[4] [dMB08] and Yices[5] [DdM06] using their standard C/C++ API, as well as any interactive SMT solver understanding the SMTlib-2 standard [BST10a, BST10b], such as MathSat[6] [CGSS13], CVC3[7] [BT07] or CVC4[8] [BCD+11], through a pipe interface.

7.1 Examples of Usage

We briefly illustrate PAGAI on a very simple example:

```
int main() {
    int x=1; int y=1;
    while(input()) {
        int t1 = x;
        int t2 = y;
        x = t1 + t2;
        y = t1 + t2;
    }
    assert(y >= 1);
    return 0;
}
```

7.1.1 Command-Line Output

PAGAI takes as input a C file or LLVM IR (.bc) and outputs an annotated version of the initial C file. These annotations provide to the user the inferred invariants (in purple), as well as alarms (in red) and proved properties (in green). Note that alarms are either true of false positives.

In Figure 7.1, the addition in x = t1 + t2 may give an integer overflow. Indeed, in the case the input() never returns 0, x and y will be incremented by a strictly positive integer at each loop iteration. After a certain number of iterations, t1 + t2 will be strictly greater than the maximal element of type **int** in the C language, in general $2^{31} - 1$. PAGAI then raises an alarm, and continues the analysis, considering it does not overflow. The second assignment y = t1 + t2 is thus safe under the assumption that the previous addition did not overflow.

7.1.2 LLVM IR Instrumentation

PAGAI invariants are inserted into the LLVM bitcode so that they can be used by external tools. The simplest way of adding information into an LLVM bitcode file is to use metadata. Metadata can be understood as a tree, where the leaf nodes contain an information (a string, an integer, etc.). In the textual format of the bitcode, each subtree is labeled with an exclamation point followed by an identifier, e.g. !24, !dbg, etc. This subtree is defined in the end of the bitcode by its list of childrens, e.g. !23 = metadata !{metadata !24, metadata !25} is metadata that combines the subtree !24 and !25.

PAGAI uses this infrastructure to add the invariants in the LLVM IR. An example of PAGAI LLVM output is given in Figure 7.2. Typically, loop headers basic blocks are assigned metadata

[4] http://z3.codeplex.com
[5] http://yices.csl.sri.com
[6] http://mathsat.fbk.eu
[7] http://www.cs.nyu.edu/acsys/cvc3
[8] http://cvc4.cs.nyu.edu

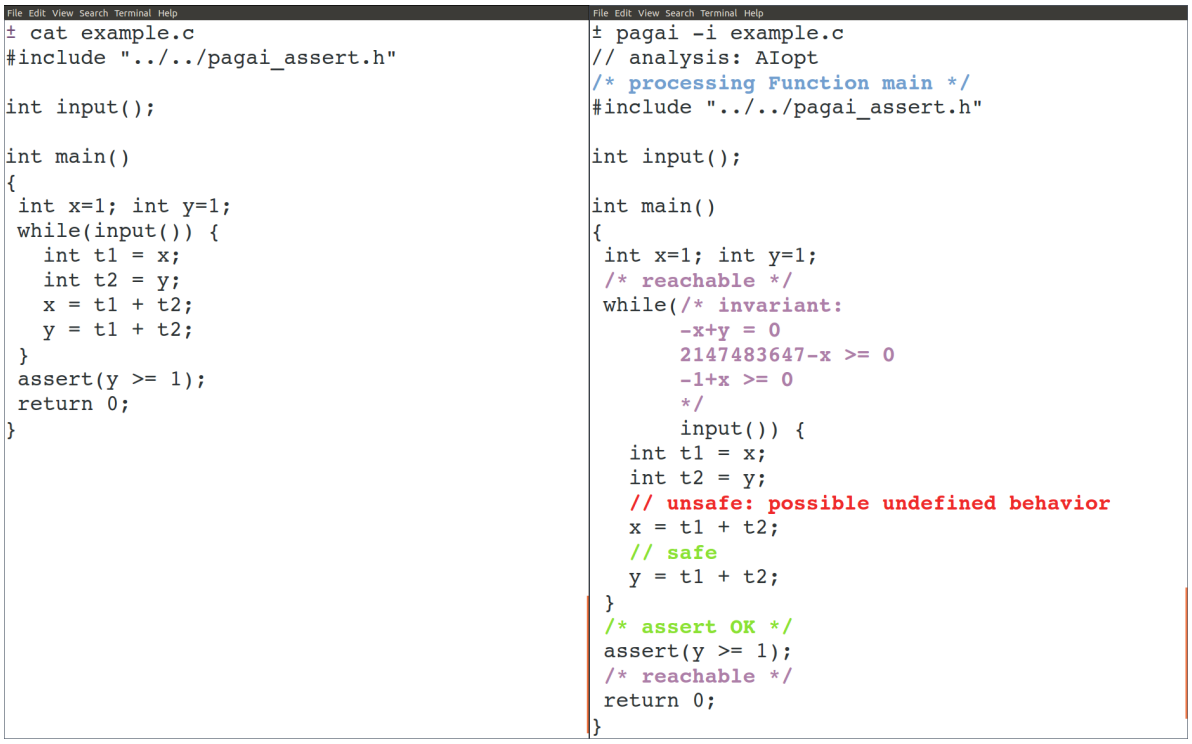


Figure 7.1: Screenshot of the PAGAI command-line output.

!pagai.invariant (see metadata !23), composed of a tuple of dimensions — here, !24, i.e. ("x","y") —, followed by the list of constraints. For instance, !26 is an equality constraint, and says that $0 + -1 * x + 1 * y = 0$. !28 is the inequality constraint $2147483647 + -1 * x + 0 * y \geq 0$.

LLVM does not allow instructions variables in metadata, one thus need to use string metadata (for instance !20 for variable y), and link them to its corresponding LLVM instruction through the !pagai.var metadata attached to its definition.

7.2 Infrastructure and Implementation

In this section, we detail some important implementation and design choices of PAGAI. Figure 7.3 gives an overview of the architecture and the external dependencies.

Static Analysis Techniques as LLVM Passes

The APRON library provides an implementation for several numerical abstract domains: intervals, octagons, convex polyhedra, etc. All of these abstract domain can be used through the AbstractClassic class (see Figure 7.4). Now, some of the analysis techniques require more evolved abstract domains, in particular the analysis with disjunctive invariants and the Lookahead Widening from [GR06]. Then, the AbstractDisj class manipulates lists of Abstract, and AbstractGopan class manipulates pairs of AbstractClassic.

The various analysis techniques are implemented as LLVM passes. These passes share the maximum amount of code inside an abstract class AIPass (see Figure 7.4). We briefly enumerate the implemented analysis:

- **AISimple** implements standard abstract interpretation, i.e. the usual chaotic iteration strategy detailed in subsection 2.1.5. This implementation can be used either with

```
...
while.cond:
  %y.0 = phi i32 [ 1, %entry ], [ %4, %cont2 ], !pagai.var !20
  %x.0 = phi i32 [ 1, %entry ], [ %0, %cont2 ], !pagai.var !21
  %call = call ... @input ..., !dbg !22, !pagai.invariant !23
  %tobool = icmp ne i32 %call, 0, !dbg !22
  br i1 %tobool, label %while.body, label %while.end, !dbg !22
...
!20 = metadata !{metadata !"y"}
!21 = metadata !{metadata !"x"}
...
!23 = metadata !{metadata !24, metadata !25}
!24 = metadata !{metadata !21, metadata !20}
!25 = metadata !{metadata !26, metadata !28, metadata !30}
!26 = metadata !{metadata !27, metadata !"==", metadata !"0"}
!27 = metadata !{metadata !"0", metadata !"-1", metadata !"1"}
!28 = metadata !{metadata !29, metadata !">=", metadata !"0"}
!29 = metadata !{metadata !"2147483647", metadata !"-1", metadata !"0"}
!30 = metadata !{metadata !31, metadata !">=", metadata !"0"}
!31 = metadata !{metadata !"-1", metadata !"1", metadata !"0"}
...
```

Figure 7.2: Example of PAGAI invariant inserted in LLVM IR as metadata.

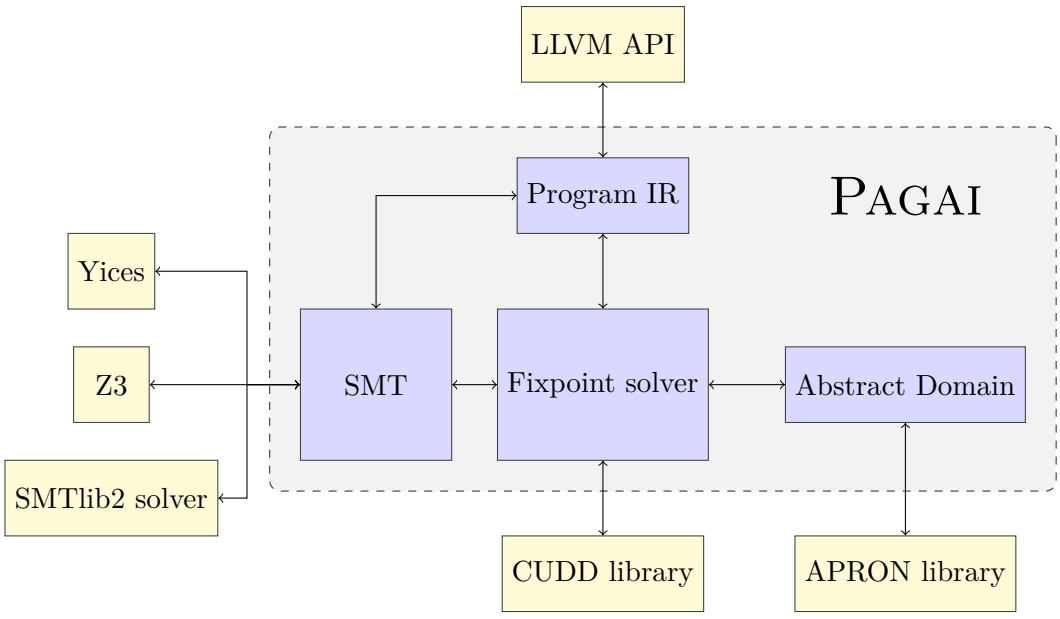


Figure 7.3: Infrastructure

standard abstract domains (**AIClassic**) or in the Lookahead Widening [GR06] settings (**AIGopan**).

- **AIguided** implements the Guided Static Analysis iteration strategy [GR07].
- **AIopt** implements our Guided Path Analysis technique from subsection 4.2.2.
- **AIpf** implements Path Focusing from subsection 3.3.2.
- **AIdis** implements our extension to Path Focusing techique with disjunctive invariants, described in subsection 4.3.1.

- **AIopt_incr** (resp. **AIpf_incr**) executes first **AIClassic**. Then it runs **AIopt** (resp. **AIpf**) by intersecting each abstract value with the invariant discovered by the first technique. It also encodes these invariants in the SMT formula (In some cases, it helps the SMT solver to decide satisfiability faster).

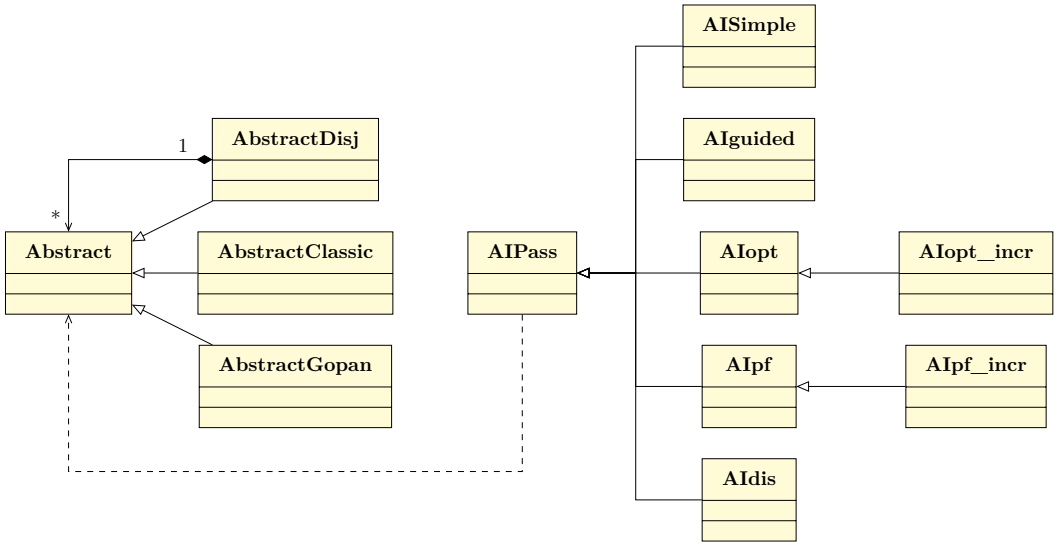


Figure 7.4: Class hierarchy of the Analysis Passes and the Abstract values

7.2.1 Static Analysis on SSA Form

LLVM bitcode is in *static single assignment* (SSA) form: a given scalar variable is given a value at a single syntactic point in the program. In concrete terms, an assignment x=2∗x+1; gets translated into a definition $x_2 = 2x_1 + 1$, with distinct variables x_1 and x_2 corresponding to the same original variable x at different points in the program. One may see conversion to SSA form as a static pre-computation of some of the symbolic propagations proposed by [Min06] to enhance the precision of analyses.

A static analysis on an SSA form control flow graph has to gracefully handle a larger number of different variables compared to a non-SSA form. A simple implementation of abstract interpretation would track every variables at each program point, and thus every abstract values share exactly the same dimensions. In this section, we detail how PAGAI considers only some subsets of the variables during the analysis, while still being able to derive fully precise invariants involving every SSA variable.

SSA introduces ϕ-functions at the header of a basic block to define variables whose value depends on which incoming edge was last taken to reach this block. For instance, for
if (...) { x = 2∗x+1; } else { x= 0; }, then x_2 is defined as $\phi(2x_1 + 1, 0)$.

In this framework, each arithmetic operation (+, −, ×, /) defines a variable; its operands themselves may not be representable as arithmetic functions, either because they are defined using ϕ-functions, loads from memory, return values from function calls, or other numerical operations (e.g. bitwise operators) that are not representable with our class of basic arithmetic operations. We may vary the class of arithmetic operations, for instance, by restricting ourselves to linear ones.

Dimensions of the Abstract Values

The SSA form motivates a key implementation decision of our tool: only those variables $v_1, \ldots, v_n$ that are not defined by arithmetic operations are retained as coordinates in the abstract domain (e.g. as dimensions in polyhedra), assuming they are live at the associated control point.

For instance, assume that x, y, z are numerical variables of a program, x is defined as $x = y + z$, and x, y, z are live at point p. Instead of having x as a dimension for the abstract value at point p, we only have y and z. All the properties for x can be directly extracted from the abstract value attached to p and the relation $x = y + z$. This is an optimization in the sense that there is redundant information in the abstract value if both x, y and z are dimensions of the abstract value. Now, suppose that $x = y + z$ but only x and z are live at p. Then, we have to keep track of y and z in the abstract value, so that we can still derive a fully precise invariant for x.

The classical definition of liveness can be adapted to our case:

Definition 28 (Liveness by linearity)**.** A variable v is *live by linearity* at a control point p if and only if one of these conditions holds:

- v is live in p.
- There is a variable v', defined as a linear combination of other variables $v_1, v_2, \ldots, v_n$, so that $\exists i \in \{1, \ldots, n\}, v = v_i$, and v' is live by linearity in p.

Finally, a variable is a dimension in the abstract domain if and only if it is live by linearity and it is not defined as a linear combination of program variables. In other words, any live variable is either directly a dimension of the abstract value, or is defined as a linear combination of variables in dimension. In addition, this linear combination can be syntactically derived from the instructions definition, and thus can be recovered immediately after the fixpoint has been reached. If one uses the abstract domain of polyhedra or linear equalities, these definitions are already elements of the abstract domain and can simply be intersected with the obtained invariant.

Having few variables as dimensions of an abstract value has two benefits:

- Abstract domain libraries often perform worse with higher dimensions for some abstract domains like polyhedra; The extra dimensions we avoid express linear equalities between variables, which should therefore cost little assuming some sparse representation of the constraints. However, several libraries, including APRON, compute with *dense* vectors and matrices, which means that any increase in dimensions slows computations.
- In practice, certain widening operators have worse precision when the number of dimensions is large[MG12]:

  Example 15. We illustrate this last point with the small example from Figure 7.5 and the abstract domain of convex polyhedra, for which an analysis gives worse precision on the two variables x and y in the presence of a third variable c: If we run the analysis while ignoring the variable c (or if we comment line 8), PAGAI yields at the loop header the invariant $y \leq x + 1 \wedge x \geq 0 \wedge x \leq y$. Now, if c is also considered as dimension of the abstract value at the loop header, and finally project the obtained invariant onto x and y, the obtained invariant is $y \leq x + 1 \wedge x \leq y$. Taking into account the c variable then degraded the precision of the invariant because of the widening.

```
1 void f() {
2   int x = 0;
3   int y = 0;
4   int c = 0;
5   while (x <= y) {
6       if (x < y) {
7         x+=2;
8         c+=x;
9       }
10      y++;
11  }
12 }
```

Figure 7.5

Abstract Values & Abstract Forward Transformers

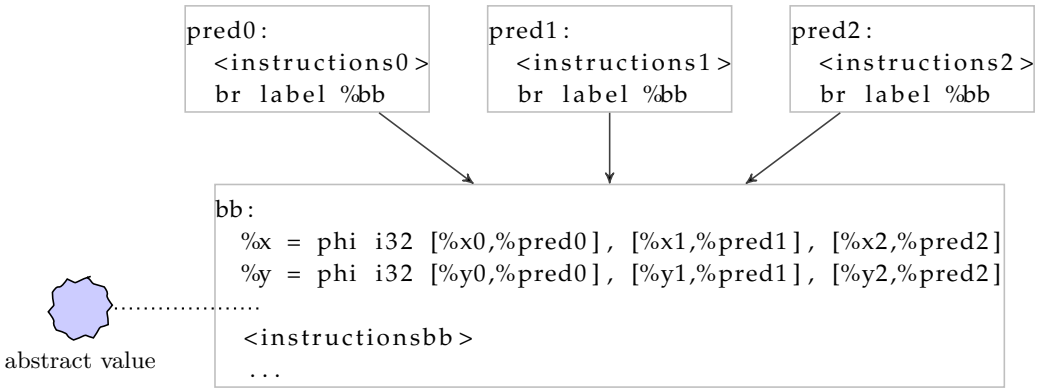
PAGAI computes abstract values for the control-flow graph of the LLVM IR, i.e. a graph of basic blocks. Each basic block is a sequence of LLVM instructions. This sequence of instructions starts with Φ-instructions (possibly none), followed by only non-Φ instructions (see Figure 7.6). The fixpoint iterations in abstract interpretation basically compute images of the abstract values by the abstract forward transformers associated to the graph transitions (or paths in the case of path-focusing techniques). In our case, there is no need to assign a different abstract value after each instruction; it is sufficient to use one single abstract value per basic block. For the SMT-based techniques that are described in the previous chapters of the thesis, we assign an abstract value only at a subset of the basic blocks (e.g. loop headers).

Parallel assignments We choose to assign to each basic block one single abstract value which is an invariant **just after** the Φ-instructions assignments (see Figure 7.6). This choice is somehow intuitive, since the Φ instructions are not *real* instructions but only assign the appropriate value depending on the incoming block — thus, it can be seen as a *store* operation in the end of the incoming block —. It follows that we define abstract forward transformers that approximate the semantics of the sequence of non-Φ instructions at once.

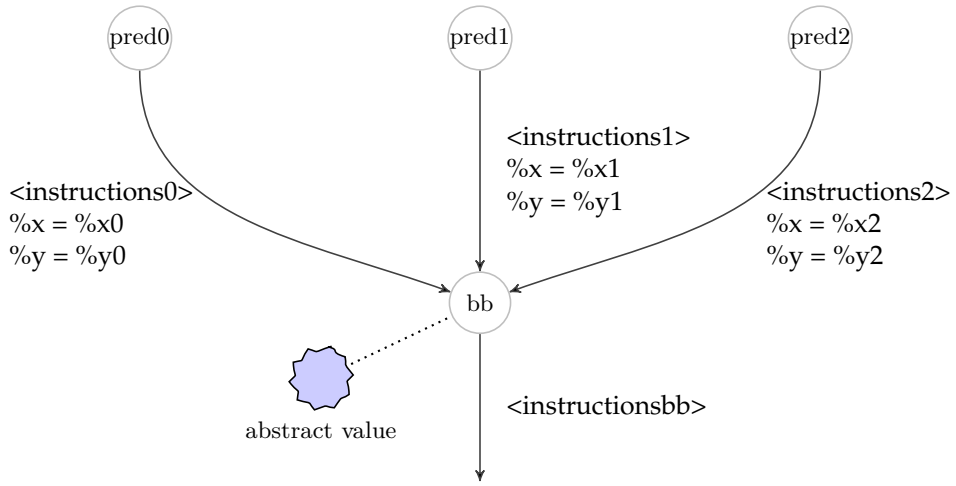
A basic block therefore amounts to a *parallel assignment* operation between live-by-linearity variables $(v_1, \ldots, v_n) \mapsto (f_1(v_1, \ldots, v_n), \ldots, f_k(v_1, \ldots, v_n))$; such operations are directly supported by the APRON library. As suggested by [Min06], this approach is more precise than running abstract operations for each program line separately: for instance, for y=x; z=x−y; with precondition $x \in [0, 1]$, a line-by-line interval analysis obtains $y \in [0, 1]$ and $z \in [-1, 1]$ while our "en bloc" analysis symbolically simplifies $z = x - x = 0$ and thus $z \in [0, 0]$.

Abstract Transformer Due to the SSA form, the dimensions of the abstract values between two basic blocks are (very) different. Then, computing the image of an abstract value over the variables in X should result in a new abstract value over different variables, that we note X'. Moreover, some intermediate instructions may use *temporary* variables that are not dimensions in the starting block nor in the successor block. We note this set of variables T.

In the case of a basic block which is a loop header, its Φ-variables will be both in X and X' — this is the case for the induction variables —. One should distinguish between the values



(a) A basic block containing two Φ-instructions



(b) Equivalent CFG fragment, where instructions are moved to edges. Φ-instructions are replaced by simple assignments in the incoming edges.

Figure 7.6: Semantics of Φ-instructions in LLVM, that motivates our choice for positionning abstract values.

of these Φ-variables in the previous and the next loop iteration. Since these Φ-assignments are the last instructions executed in the transition (or path), every other instructions will refer to their old values.

Finally, the resulting abstract image is computed from the starting abstract value A in several steps:

1. Change the dimensions of $A(X)$ to $A(X \cup T \cup X')$. The variables in $T \cup X'$ are then unconstrained.
2. Assign variables in $T \cup X'$ according to the instructions in the path. The Φ-variables of the destination basic block of the path (or transition) are temporarily not assigned their new value, since one still have to intersect with the guards.
3. Intersect with the set of guards the path goes through. These guards are constraints over the variables in $X \cup T \cup X'$.
4. Assign the new values to the Φ-variables of the successor block.

5. Project the resulting abstract value (over $X \cup T \cup X'$) onto X' only

For the techniques that make use of SMT to succinctly represent sets of paths, the parallel assignments are much larger — we assign every variables defined by instructions in the path at once — and thus are even more precise and efficient, as well as the intersection with the guards, which only requires one single call to the meet operation in the abstract domain.

7.2.2 Analysis and Transform Passes

PAGAI uses (and sometimes implements) several analysis and transform passes required by the invariant computation passes. Here, we simply mention a few of them.

Mem2Reg Our analysis currently only operates over the scalar variables of the LLVM IR and thus cannot directly cope with arrays or memory accessed through pointers. We therefore run it after the "memory to registers" (**-mem2reg**) optimization pass, provided by LLVM, which lifts most memory accesses to scalar variables. The remaining memory reads (i.e. *load* instructions) are treated as nondeterministic choices, and writes (i.e. *store* instructions) are ignored. This is a sound abstraction if memory safety is assumed; in other words, the program under analysis should respect the C99[ISO99] standard for type aliasing: two elements of different type may never alias. The mem2reg optimization pass also assumes memory safety, as well as, possibly, the absence of other undefined behaviors as defined by the C standard. This is the price of using the front-end from a generic compiler: C compilers have the right to assume that undefined behaviors do not occur, including in preprocessing and optimization phases.

Moreover, the mem2reg optimization often fails to lift *load* and *store* of global variables. For this reason, we implemented an LLVM optimization pass that is run before mem2reg, called **-globaltolocal**: it transforms global variables into local variables through the LLVM *alloca* instruction, in case these global variables are marked as *internal*. This transformation is correct only if the LLVM IR contains one single function definition, which is the function to analyze. Consequently, this transformation is applied only if the last assumption holds.

Liveness by linearity Before the invariant computation passes, one needs to compute the dimensions of the abstract values at each basic block. We implemented a pass that computes this set of live-by-linearity variables as a fixpoint, in a similar way it is standardly done for computing live variables.

Overflow intrinsics LLVM includes intrinsic functions — i.e. a special function whose name is detected by the code generator for a special treatment — for checking whether an arithmetic operation overflows. These functions allows to return both the result of the arithmetic operation, together with a Boolean that indicates if the overflow happens. This is useful for generating code that cleanly aborts when an overflow occurs during execution. These intrinsic functions are enabled when compiling with Clang and argument `-ftrapv` or `-fsanitize=undefined`. We implemented an optimization pass that replaces these instrinsic functions with a standard arithmetic operation, followed by tests for checking whether the resulting value has overflown. Basically, if the result is a N-bit integer, we check whether it is greater than $2^{N-1} - 1$ or less than -2^{N-1}. These tests would always evaluate to false when executing the code, but not in the analysis, where the integer variables are considered as integers in $\mathbb{Z}$.

Example 16. This example illustrates our program transformation in the case of a @llvm.sadd.with.overflow.i32 intrinsic function. The initial basic block that contains the intrinsics is the following:

```
%call = call i32 bitcast (i32 (...)* @input to i32 ()*)()
%0 = call { i32, i1 } @llvm.sadd.with.overflow.i32(i32 %call, i32 42)
%1 = extractvalue { i32, i1 } %0, 0
%2 = extractvalue { i32, i1 } %0, 1
%3 = xor i1 %2, true
br i1 %3, label %cont, label %handler.add_overflow
```

The intrinsics return both the result of the operation in register %1, as well as a Boolean flag in %2 which evaluates to *true* if it overflowed. In the later case, we branch to a special trap handler basic block. Our LLVM pass transforms this basic block into:

```
%call = call i32 bitcast (i32 (...)* @input to i32 ()*)()
%0 = add i32 %call, 42
%1 = icmp sgt i32 %0, 2147483647
%2 = icmp slt i32 %0, -2147483648
%3 = or i1 %1, %2
br i1 %3, label %handler.add_overflow, label %cont
```

Expansion of (dis)equality tests It is well known in abstract interpretation based techniques that guards of the form (x != y) induce imprecision since they represent non-convex sets, while usual abstract domains such as intervals or polyhedra can only represent convex sets. For this reason, standard abstract interpretation with widening fails to find a precise invariant for simple programs like function F in Figure 7.7. Even though this particular example can be managed by existing workarounds, e.g. widening with threshold, some other are not: for instance, function G in Figure 7.7 is not handled precisely by existing approaches, and requires a path-sensitive analysis catching the 3 cases $(x = 0)$, $(x > 0 \wedge x < 1000)$, and $(x = 1000)$. In particular, for the path-sensitive analysis to be precise, the path where the second case hold should be different from the path where $x < 0 \vee x > 1000$, which is not the case in the given program.

```
void F() {
  int i = 0;
  while (i != N) {
    i++;
  }
}
```

```
void G() {
  int x = 0;
  int d = 1;
  while(1) {
    if (x == 0) d=1;
    if (x == 1000) d=-1;
    x +=d;
  }
}
```

Figure 7.7: Examples of code with $=$ and $\neq$ guards.

In PAGAI, we implemented a new LLVM optimization pass, **-expandequalities** (see Figure 7.8), which is run previous to our analysis. It inserts new control flow so that the paths for the three cases $x < y$, $x = y$ and $x > y$ are different. These three cases have convex guards, thus are precisely managed by our abstract domains. Even though this transform pass

complicates the LLVM IR, it only introduces a couple of Booleans in our SMT encoding, and therefore does not compromise the efficiency of the static analysis. Note that we could have done another more natural transformation, by removing the (dis)equality test and replacing it with a cascade of if-then-else with $<$ and $>$ guards; however, this transformation would duplicate the $\neq$ branch and would not give better results.

```
if (x ⋈ y) {                    if (x < y) {}
  // THEN branch                if (x > y) {}
} else {                        if (x ⋈ y) {
  // ELSE branch        ⟶         // THEN branch
}                                } else {
                                   // ELSE branch
                                 }
```

Figure 7.8: Our code transformation, illustrated at the C level, where $\bowtie \in \{==, !=\}$

7.2.3 LLVM for Static Analysis

LLVM provides many facilities for writing a static analyzer. Indeed, However, the LLVM internal representation sometimes lacks information useful for static analysis, since its principal target is code generation. We mention here some issues due to the use of LLVM, that compromise soundness or precision of Pagai.

Types

In LLVM, some type informations for the variables are lost. A **signed int** and an **unsigned** variable at the C level would have the same type i32 in the LLVM IR. Signedness is a property of operators, rather than types, which means that one would have to look at the instructions that are using a particular variable as operand to guess its "signedness". For instance, the presence of an instruction <result> = add nuw <ty> <op1>, <op2>, which contains the nuw flag (meaning *no unsigned wrap*), somehow give an indication that <op1> and <op2> are used as unsigned values. Pagai does not use this information since its reliability is limited. Consequently, Pagai does not know by default that a unsigned variable is always positive, and thus leads to imprecision.

An alternative would be to use debugging information, which contain the C types for the C variables, and use it in the analysis. This technique is however not satisfactory since the analyzed code does not always contain such debug information.

Undefined Behaviors

The C99 standard[ISO99] defines *undefined behavior* like this:

> *"Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner [...]"*

In particular, the use of uninitialized variables in C has undefined behavior, and LLVM has a specific class for uninitialized variables called UndefValue. Pagai considers these undefined values as nondeterministic choices, which is not fully correct: an instruction undef + 1 would

not necessarily return the value incremented by one, but may also "*crash or generate incorrect results*" as described by the standard.

The second related problem is that the Clang front-end, used prior to our analysis passes, may provide an internal representation free of undefined behaviors, even though the initial C code had some, provided the front-end has performed its own choices to deal with them. Compiling into LLVM IR with an option like **-Wall** would help to detect these undefined behaviors.

From LLVM Invariants to C Invariants

PAGAI computes invariants at the LLVM IR level. Consequently, these invariants involve LLVM values, which have no direct mapping to actual variables at the C level. It is not straightforward to map PAGAI invariants from the LLVM level to the C level. We found it interesting to explain in this thesis the procedure to do it in a correct way.

When compiling a C program into LLVM with debugging informations enabled, the obtained IR contains:

- metadata that contain the names for the C variables, their type, etc.
- intrinsic instructions, called llvm.dbg.value, whose semantics is "From now on, the C variable x has the value of the LLVM variable v".

While these informations are convenient for moving information from C to LLVM and later to assembly code (usually in DWARF format, used by debuggers like gdb for e.g. printing the current value of the C variable 'x'), it requires more work for moving information in the other direction, from LLVM to C.

An example of LLVM IR with debug information is given in Figure 7.9: the figure only displays the instructions used for tracking the C variable 'x'. For instance, the instruction llvm.dbg.value ('%3',...,' x') in B0 says that 'x' is actually equal to the LLVM value %3 at this control point.

Suppose PAGAI has computed an invariant at control point B7. This invariant is likely to have the LLVM variable x.0 in its dimensions. For translating this invariant into in the C-level, one should know the current value of the C variable 'x' at control point B7. To do so, we build the data structure depicted in Figure 7.10:

- We use a depth first search algorithm that explores the control flow graph backwards from B7, and backtracks an llvm.dbg.value that relates the variable 'x' to an LLVM value is encountered.

- If we reach a control point that has already been visited, the data structure is updated by adding a new predecessor to the visited block (in our example from Figure 7.9, it happens for B3 and B5). After having traversed the entire graph, the data structure is the one depicted in Figure 7.10 Step 1.

- Then, the datastructure is simplified using two rules:

    1. Φ-*merging*: if a block has several successors in the data structure (the successors in the data structure are actually predecessors in the CFG), each of them with a different assignment, and the block contains a Φ-instructions whose definition perfectly matches the different cases, replace the successors by a direct assignment to the Φ-instruction. Figure 7.10 Step 2 illustrates this simplification for block B3.

2. *simplify*: If a block B1 dominates another block B2 in the data structure, and every path from B1 goes to B2, replace everything in between by a direct link. Figure 7.10 Step 3 and Figure 7.10 Step 4 apply this simplification twice.

These rules are applied successively until the data structure has one single element, and its corresponding assignment. The assignment is the correct LLVM value that equals the C variable at this position. Such computation should be done for any C variable the user wants to track.

Note that Pagai computes invariants that only involve *live-by-linearity* variables. Then, it may happen that at a particular point where an invariant has been computed, the LLVM value that actually equals some C variable is not a dimension of the abstract value — e.g. if B5 contains llvm.dbg.value('%v ',...,' x') where %v = x.0 + 1, the abstract value in B7 will contain x.0 but not %v —. This is not a problem since it is sufficient to intersect the abstract value with the constraint %v = x.0 + 1, and project the variable x.0, for having an invariant for the C variable 'x'.

In the current implementation, this "back-to-source" transformation does not use the algorithm we detailed here, and is based on the syntactic name of the LLVM Φ-values and their possible assignments compared to the llvm.dbg.value. Consequently, it sometimes happen that the C output of Pagai gives incorrect results, even though the LLVM output is correct.

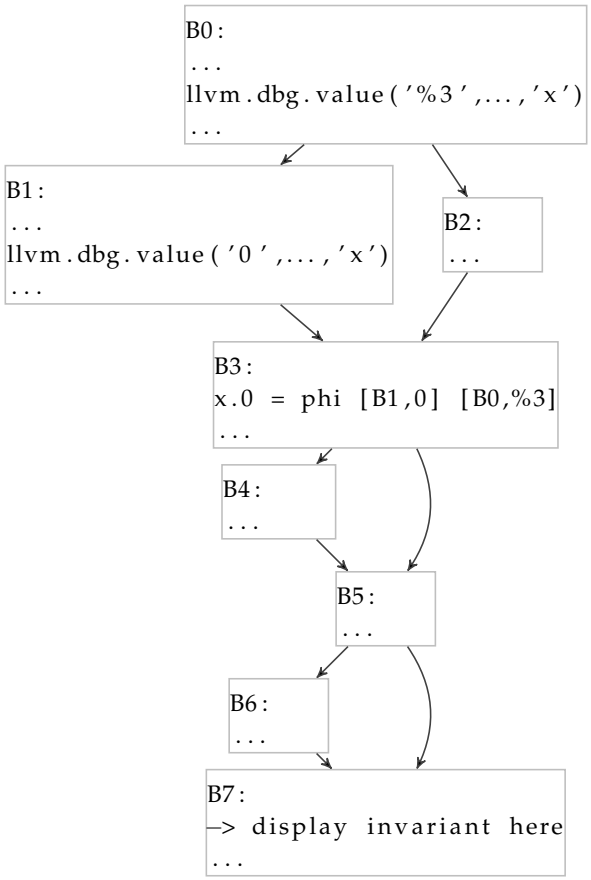


Figure 7.9

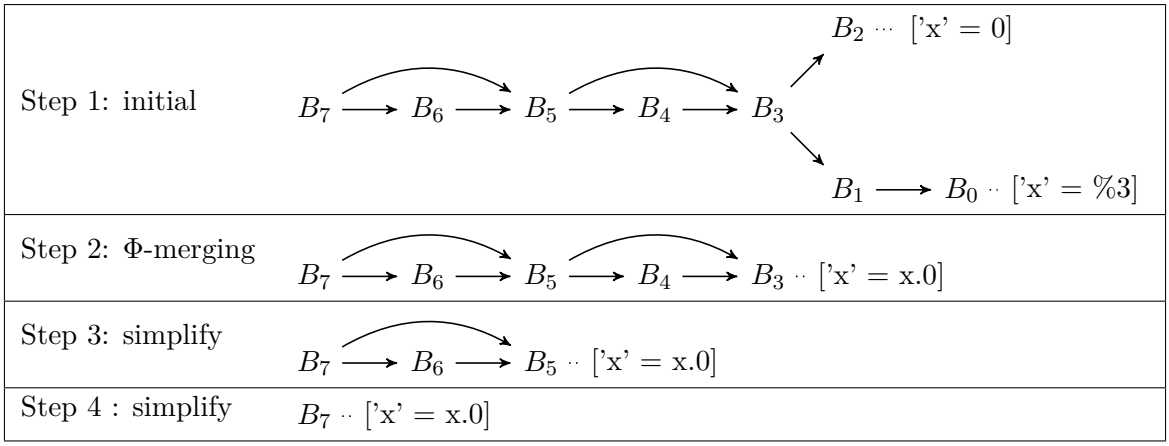


Figure 7.10

7.2.4 From LLVM Bitcode to SMT Formula

We briefly detail our encoding of LLVM instructions into SMT formulas in QF_LIRA. Integer values (e.g. i32, i64) or characters (i8) are considered to be mathematical integers in $\mathbb{Z}$ and floating points (e.g. double, float) are encoded using real numbers. As a consequence, our encoding is not sound. Future work includes using the theory of bitvectors for encoding integers. However, since many arithmetic operations over integers are followed by overflow checks, as explained in subsection 7.2.2, PAGAI can still detect possible overflows, and the checks restrict the result of the operation to the correct machine bounds. We provide in Table 7.1 our encoding of the main LLVM instructions.

7.3 Experiments

7.3.1 Comparison of the Different Techniques

In this section, we provide an experimental evaluation of the various techniques implemented in PAGAI. We compare them in terms of precision and cost, with two sets of benchmarks:

1. benchmarks from the Malärdalen research group on WCET, available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. These benchmarks provide an interesting panel of challenging programs, such as matrix operations, sorts, nested loops programs, programs with many paths, etc.

2. a panel of GNU programs: grep, libsuperlu, tar, libgmp, libjpeg, sed, gzip, wget and libpng16. The purpose of these benchmarks is to give a idea of the benefits of the various techniques on real-life programs.

We experimented with the following techniques:

- **S**: standard abstract interpretation from [CC77, CH78].
- **C**: the combination of Path Focusing and Guided Static Analysis, also called Guided Path Analysis, and described in subsection 4.2.2.
- **PF**: Path Focusing
- **IN**: Incremental Path Focusing: first run **S**, followed by **PF**, intersecting each result from **S** during **PF**.

LLVM Instruction	**SMT encoding in LIRA**
br i1 <cond>, label <iftrue>, label <iffalse>	$(t\_\text{<bb>}\_\text{<iftrue>} = (var(\text{<cond>}) \wedge b\_\text{<bb>}))$ $\wedge \quad (t\_\text{<bb>}\_\text{<iffalse>} = (\neg var(\text{<cond>}) \wedge b\_\text{<bb>}))$
br label <dest>	$t\_\text{<bb>}\_\text{<dest>} = b\_\text{<bb>}$
<result> = invoke ... to label <label> unwind label <exception label>	$(t\_\text{<bb>}\_\text{<label>} = (\text{undet} \wedge b\_\text{<bb>}))$ $\wedge \quad (t\_\text{<bb>}\_\text{<exception label>} = (\neg\text{undet} \wedge b\_\text{<bb>}))$
switch <intty> <value>, label <defaultdest> [...]	removed by -lowerswitch optimization pass
<result> = select i1 <cond>, <ty> <val1>, <ty> <val2>	(if-then-else $var(\text{<cond>})var(\text{<val1>})var(\text{<val2>})$
indirectbr <somety>* <address>, [label <dest1>, ..., label <destN>]	$1 \leq \text{index} \leq N$ $\wedge \quad t\_\text{<bb>}\_\text{<dest1>} = (\text{index} = 1)$ $\wedge \quad ...$ $\wedge \quad t\_\text{<bb>}\_\text{<destN>} = (\text{index} = N)$
<result> = (f)add <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) + var(\text{<op2>})$
<result> = (f)sub <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) - var(\text{<op2>})$
<result> = (f)mul <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) * var(\text{<op2>})$
<result> = udiv <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>})/var(\text{<op2>})$
<result> = sdiv <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>})/var(\text{<op2>})$
<result> = fdiv <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>})/var(\text{<op2>})$
<result> = shl <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) * 2^{var(\text{<op2>})}$
<result> = and i1 <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) \wedge var(\text{<op2>})$
<result> = or i1 <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>}) \vee var(\text{<op2>})$
<result> = xor i1 <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>})$ xor $var(\text{<op2>})$
<result> = zext <ty> <value> to <ty2>	$var(\text{<result>}) = var(\text{<value>})$
<result> = icmp <cond> <ty> <op1>, <op2>	$var(\text{<result>}) = var(\text{<op1>})\text{<ty>}var(\text{<op2>})$ where <ty> $\in \{<, \leq, >, \geq, =\}$
every other instructions	true (i.e. large overapproximation)

Table 7.1: SMT encoding of most LLVM instructions. var(<op1>) is either the value of the constant if the operand is a constant, or a fresh SMT variable otherwise.

- **LW**: Lookahead Widening
- **G**: Guided Static Analysis

Figure 7.11 and Figure 7.13 give the results for the two benchmarks sets when comparing by pair the different techniques, instantiated with the abstract domain of convex polyhedra. The histograms should be understood as follows: for a pair $\begin{bmatrix} T1 \\ T2 \end{bmatrix}$ of techniques, $\subsetneq$, in dashed blue (resp. $\supsetneq$, in dashed red) gives the percentage of control points where $T1$ gives a strictly better (resp. worse) result than $T2$. The *uncomparable* bar shows the control points for which none abstract values is included in the other.

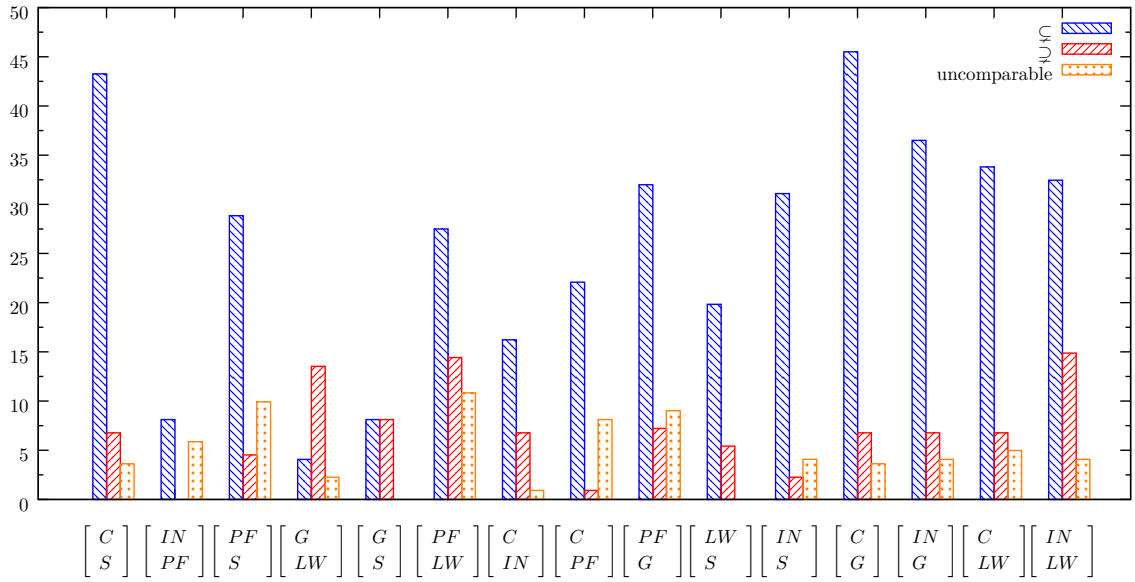


Figure 7.11: Comparison of the various analysis techniques over the Malärdalen benchmarks

Intuitively, the bar in blue should be high, while the bar in red should be low, since we compare pairs $T1$ and $T2$ in an order such that $T1$ is supposedly better than $T2$. Experiments show that improved techniques give much better invariants: for instance, **C** improves the results over **S** on the Malärdalen benchmarks in 43% of the control points.

Our Guided Path Analysis algorithm (C) takes the best of Path Focusing (PF) and Guided Static Analysis (G): it improves G in 46% and PF in 23% of the abstract values (for the Malärdalen benchmarks).

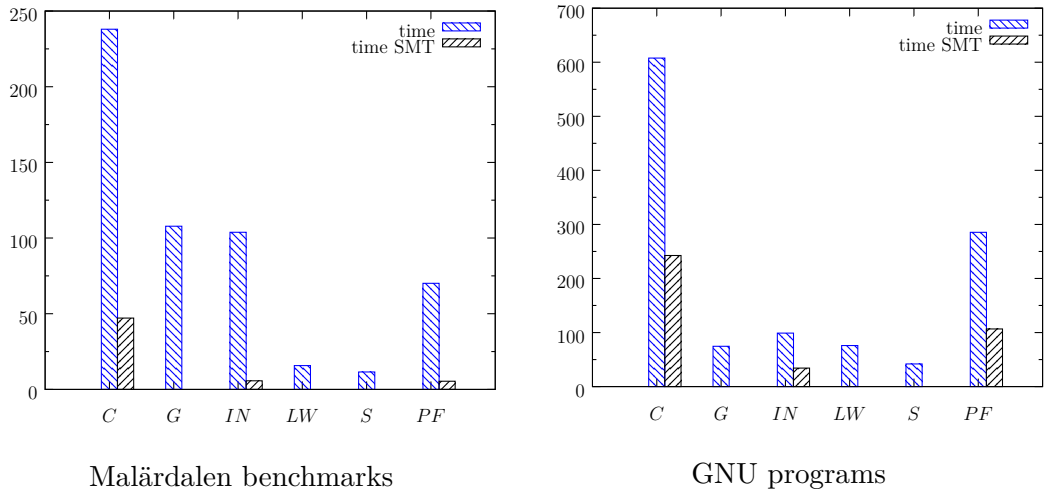
The second set of benchmarks, made of GNU programs, also illustrates the precision of Guided Path Analysis: while we cannot say which one of **PF** and **G** is better (they both are better than the other in around 6% of the control points), **C** improves them in more than 10%. One should also notice that it very rarely provides a less precise invariant than other techniques.

Most probably because of the non-monotonicity of the widening operator, it happens regularly that a supposedly more precise technique gives worse result in the end. For instance, **G** often gives disappointing results compared to a classical abstract interpretation.

One should notice a surprising result: **IN** should theoretically never be less precise than **S**, since the abstract values are always intersected with the fixpoint of **S** during the second ascending sequence. However, the histograms show that it is sometimes worse: it seems that

the descending sequence of the **PF** technique sometimes leads to a greater invariant, which is unexpected and possibly a bug, being investigated.

Figure 7.12 gives the execution time of the different techniques. For the techniques involving an SMT solver (Z3), we detail the time spend only for solving the SMT problems. Experiments show that Guided Path Analysis takes twice longer than Path Focusing, which is an important overhead. An interesting result is that on the GNU programs, **IN** is faster than **PF**: we can deduce that the initial cheap analysis by standard interpretation helps the Path Focusing algorithm to converge faster. The intersections with the first obtained invariants help to reach a fixpoint faster (in particular, less narrowing iterations are required).



Malärdalen benchmarks GNU programs

Figure 7.12: Execution time of the analysis (in seconds)

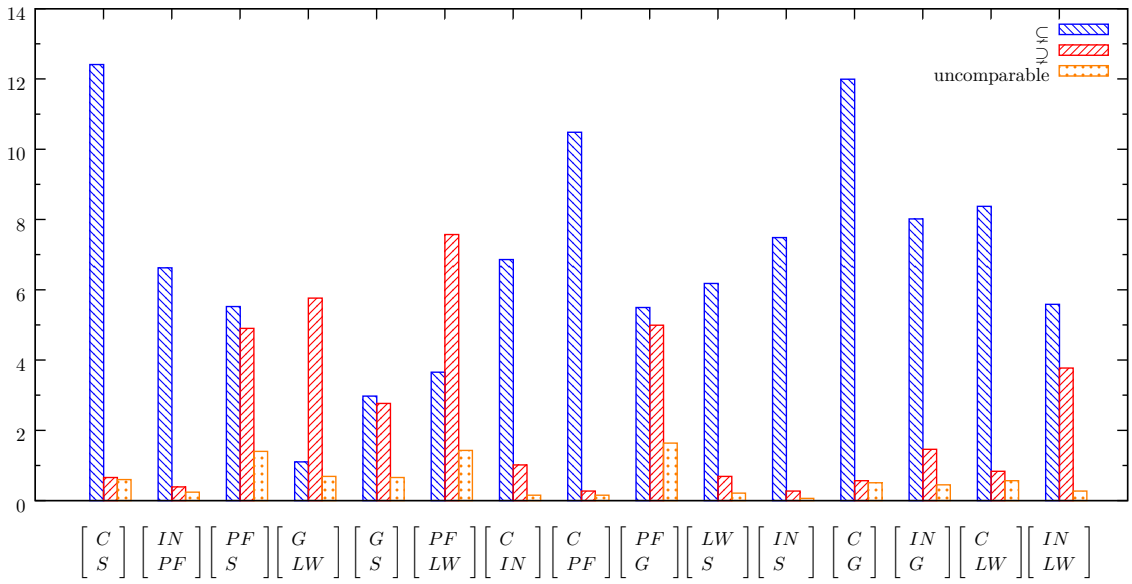


Figure 7.13: Comparison of the various analysis techniques on GNU softwares

Benchmark	#Func	C/S				C/PF				C/G				PF/G			
		$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$	$\sqsubset$	$\sqsupset$	$=$	$\neq$
grep	372	24	6	272	3	16	2	286	1	21	4	277	3	13	10	275	7
libsuperlu	187	74	0	607	0	87	0	594	0	79	0	602	0	15	39	616	11
tar	1038	43	0	513	1	26	1	530	0	28	0	528	1	19	13	521	4
libgmp	299	40	10	162	4	40	2	174	0	34	11	168	3	13	32	169	2
libjpeg	335	57	1	369	7	51	3	377	3	52	1	374	7	24	23	385	2
sed	196	2	0	75	1	1	0	77	0	1	0	76	1	0	0	77	1
gzip	212	60	2	160	1	49	0	174	0	58	0	165	0	24	18	167	14
wget	707	56	1	436	0	31	0	462	0	45	1	447	0	30	12	448	3
libpng16	498	62	2	315	3	52	1	328	1	86	2	292	2	47	21	303	11

Table 7.2: Analysis results for the GNU programs benchmarks. #Func gives the number of analyzed functions.

7.3.2 Software-Verification Competition (SV-COMP)

PAGAI has recently been extended for being able to analyze benchmarks from the Software Verification Competition (SV-COMP) [SVC]. SV-COMP consists in a set of benchmarks grouped in different categories. The objective is to prove the unreachability of a certain label ERROR (the analyzer then returns TRUE), or provide an error trace leading to this error (return FALSE). The main advantage of these benchmarks is the possibility of comparing PAGAI with other state-of-the-art program verification tools. We experimented with PAGAI on the categories ProductLines, DeviceDrivers64 and ControlFlowInteger, using the Path Focusing technique, the Z3 SMT solver, and the abstract domains of intervals and convex polyhedra. While the Path Focusing (**PF**) technique, based on SMT, is intuitively more expensive than simple abstract interpretation (**S**) from [CC77, CH78], our experiments show that it performs much better on the SV-COMP benchmarks. Most of the benchmarks consist in one big loop: **PF** computes an abstract value at one single program location (the loop header), while **S** computes abstraction at each program point, leading to many costly operations in the abstract domain (convex hulls, least upper bounds, etc.). This is a noticeable result that argues in favor of SMT-based techniques.

Since PAGAI is a static analyzer and thus only computes invariants, it is not able to provide error traces and thus can not deal with the benchmarks that should return FALSE. In our experiments, we restrict ourselves to the benchmarks that are TRUE, and try to prove the unreachability of the error state. Experimental results including the FALSE benchmarks are given in section A.2.

For ranking the different tools in the competition, every correctly analyzed benchmarks gives 2 points to the total score of the tool. We add a penalty of -8 for each benchmark for which an analyzer incorrectly returns TRUE, while is should return FALSE (i.e. false negatives). As in the competition, we also add a penalty of -4 for FALSE results in case the benchmark is TRUE (erroneous error path). We provide the plots for the three categories we experimented on. These plots can be understood as follows: the curves give the score of the tool in the x-axis, when we fix a time limit for analyzing each benchmark. This timeout is represented in the y-axis. If the analyzer returned false-negatives, its corresponding curve will start with negative points.

- At a given position in the y-axis, the best tool is the one with the rightmost curve, since it got the maximal score within the given timeout bound.

- At a given position in the x-axis, the best tool is the one with the smaller timeout limit, since it was the fastest to reach this score.

The results for the other tools are taken from the SV-COMP 2014, available at `http://sv-comp.sosy-lab.org/2014/results/index.php`. The comparison between PAGAI and the others should be taken with precaution: the experiments with the various tools have been conducted on machines with Intel Core i7-2600 CPU @ 3.4GHz, while PAGAI has been run on a Intel Xeon CPU E5-2650 @ 2.00GHz. We assume that this difference does not advantage PAGAI.

ProductLines PAGAI scores 618 while the maximum is 664 (there are 332 TRUE benchmarks, see), so it is competitive with the other participating tools. One should note that PAGAI returns **no** false negatives. Among the 23 benchmarks that are not solved:

- PAGAI timeouts for 16 of them while using the abstract domain of polyhedra, and intervals are not sufficient.

- The unsolved benchmarks actually encode automaton, for which one has to prove that the error state is unreachable. The reachability of this error state depends on some Boolean variables, that PAGAI does not precisely handle: it considers them as integers in the abstract values, which is not satisfactory here. This motivates in a future work a better handling of Boolean: we are confident that the use of logico-numerical abstract domains [Jea] would behave nicely on these examples, but are unfortunately unavailable in the Apron library used in our implementation.

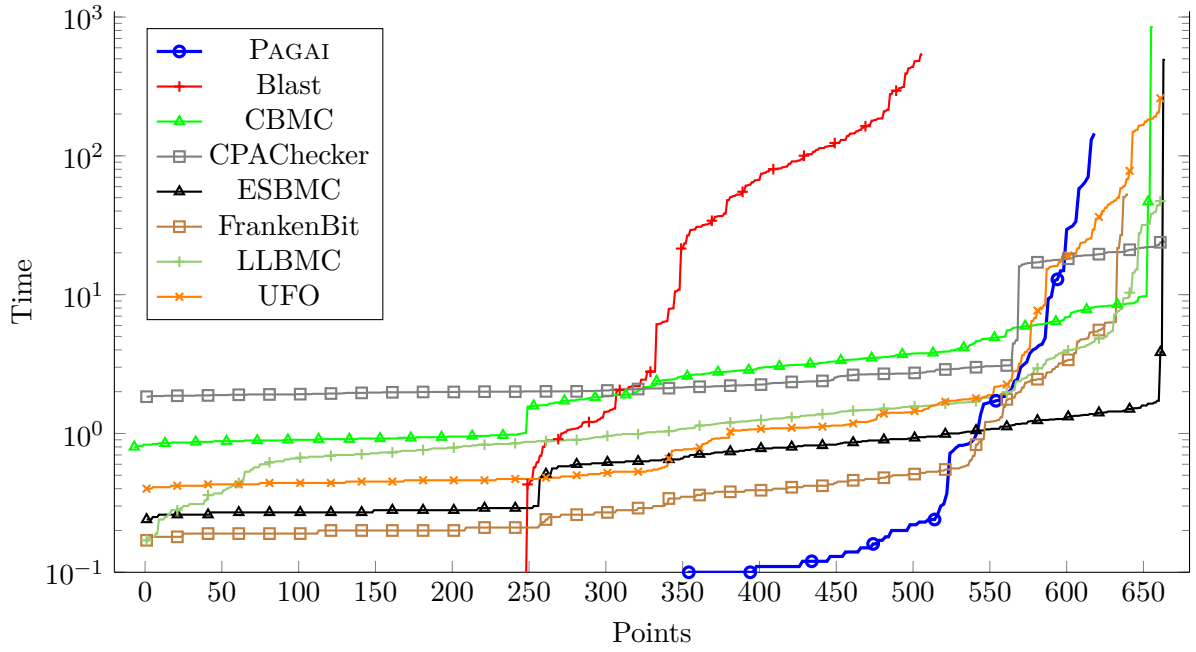


Figure 7.14: Category ProductLines, TRUE benchmarks

DeviceDrivers64 PAGAI gives interesting results in the DeviceDrivers64 category (Figure 7.15): it proves 1193 out of the 1336 TRUE benchmarks with a timeout of 500 second.

It has also 3 false negatives (thus, a penalty of -24): at that time, it is hard to understand the reason why; note that PAGAI has some known sources of unsoundness, e.g. in the presence of overflows. One should however notice that every other competing tools also have important penalties due to incorrect results (ranged between -20 for CPAChecker and -88 for CBMC).

While winning tools obtain up to 2600 points, PAGAI only scores 2362 with a maximum of 2672. Benchmarks that are not proved by PAGAI can range into 2 categories:

- The LLVM IR after inlining every functions exceeds the memory limit (7Gb).
- The invariants are not strong enough. In particular, in the same way as the ProductLines category, certain benchmarks use variables from `enum` types (or more generally integers that can take a very limited number of values), for which one should apply a special treatment instead of considering them as "normal" integers. Again, logico-numerical abstract domains would probably behave efficiently on these benchmarks.

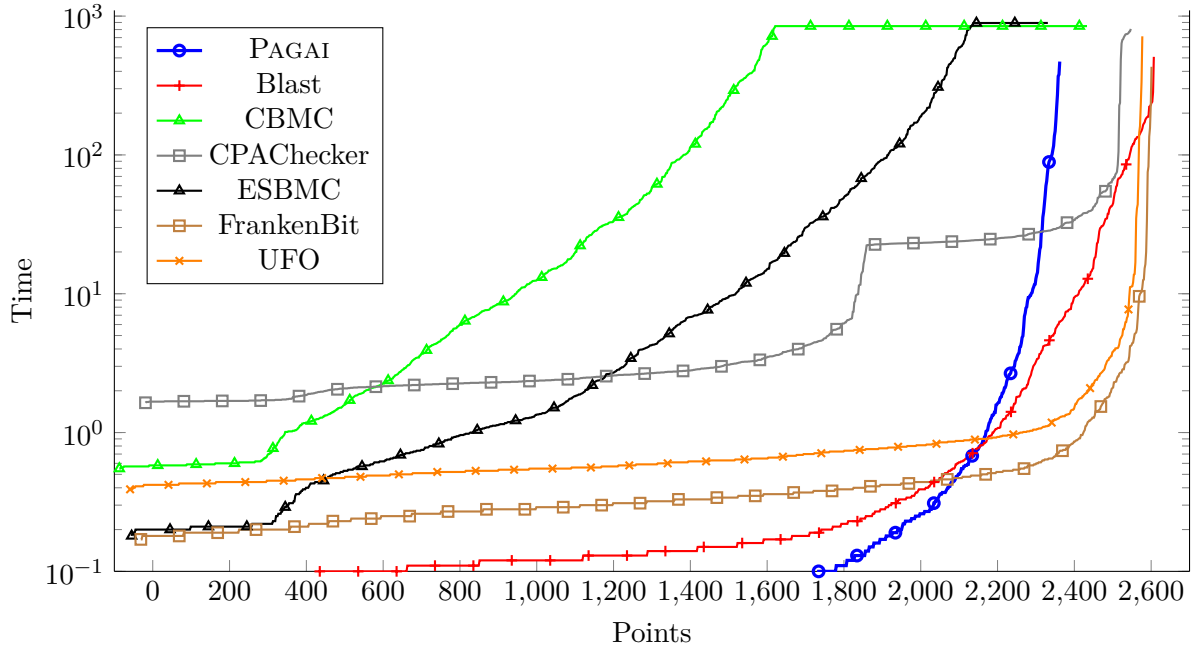


Figure 7.15: Category DeviceDrivers64, TRUE benchmarks

ControlFlowInteger The last category we tried was ControlFlowInteger (Figure 7.16).

PAGAI is still competitive compared to the other tools, but shows some limits of scalability: the timing grows exponentially in the number of points obtained. The main reasons of this blowup are:

- the size of the SMT formula turns out to be very big for some benchmarks.
- the benchmarks make intensive use of complicated arithmetic operations, e.g. multiplications and modulos. These operations are encoded as is in our SMT encoding, but should probably be handled with better abstractions.
- the fixpoint computation over the abstract domain of polyhedra takes long to converge, either because of an important number of dimensions in the abstract values, or an important number of constraints with possibly huge coefficients.

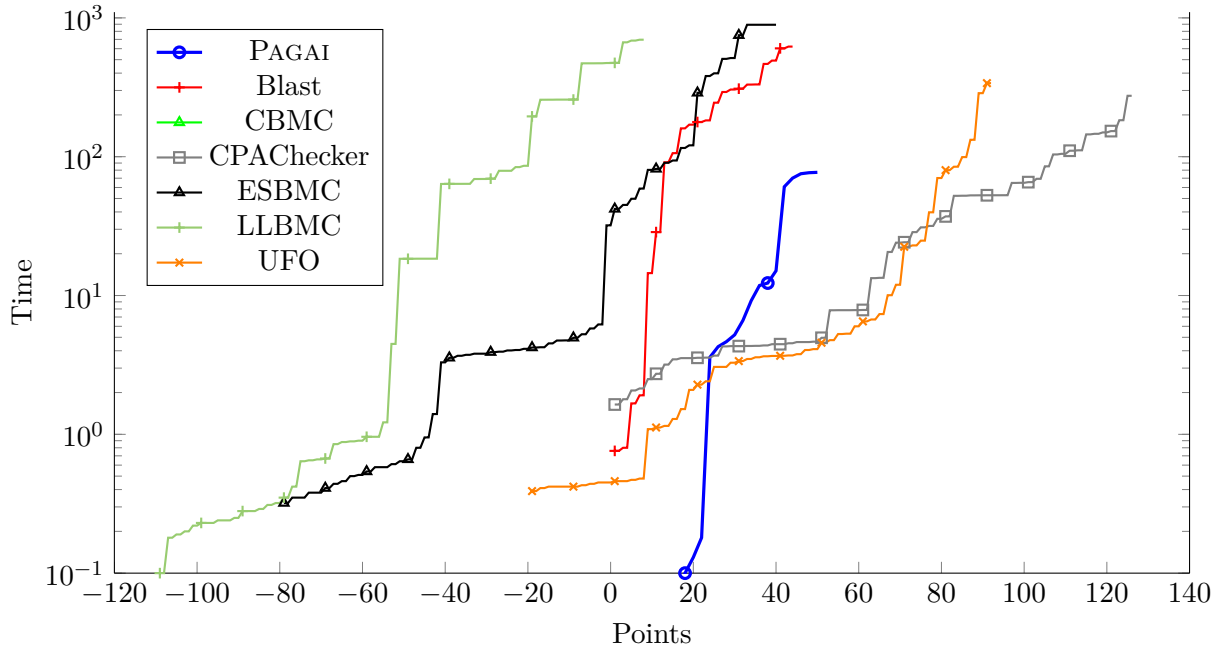


Figure 7.16: Category ControlFlowInteger, TRUE benchmarks

Loops We finally experimented with the Loops category and get 40 points while the maximal score is 68 (Figure 7.17). The score seems low, but this category actually includes many benchmarks that requires precise analysis of arrays; Pagai is not yet able to derive array properties. On every benchmark that does not require an array analysis, Pagai performs well and is faster than the other tools. One should notice though that all the benchmarks in this category are very small (less than 50 lines), and extremely simple to analyze by abstract interpretation.

Prospects Future work for being able to really enter the competition include the implementation of an LLVM IR slicing algorithm for removing every program instructions that does not affect the reachability of the error state, in order to improve scalability. We would also like to combine Pagai with other existing tools for being able to handle FALSE benchmarks and return error traces. Combinations with UFO[ALGC12] are investigated.

7.4 Conclusion

In this chapter, we presented Pagai, a static analyzer initiated and entirely developed during this thesis. It implements many abstract interpretation-based and SMT-based techniques for computing invariants for LLVM bitcode. These invariants can be used to show the absence of arithmetic overflows, or to prove user-provided program assertions. We claim that Pagai is robust implementation, given that it is able to run experiments on real code as well as standard benchmarks in the software verification community. Pagai could of course be improved and extended in many ways, in order to be able to compete with other production-grade program verifiers.

Pagai has already been used for various reasons by external people. We mention here some known users:

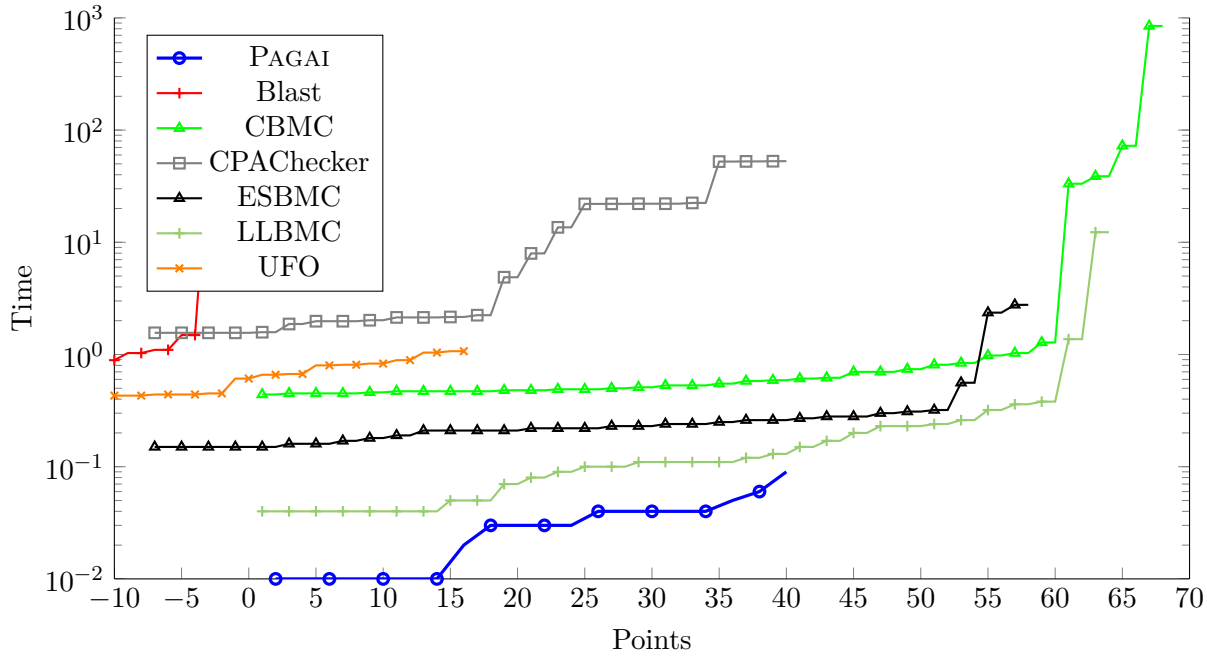


Figure 7.17: Category Loops, TRUE benchmarks. Pagai's score is low since most of the benchmarks requires invariants involving array properties.

- the Embedded Systems Research Group, IIT Kharagpur[9], India, for computing invariants on embedded control software;
- people from the ANR W-SEPT project[10], who derive ILP constraints from Pagai invariants obtained after analyzing code instrumented with counters;
- people from the CAPP group[11], LIG, Grenoble, work on Satisfiability Modulo Theories an use Pagai for producing "real-life" SMT benchmarks;
- people at Verimag and ENS Lyon, are using Pagai invariants for proving program termination;
- Raphael Ernani Rodrigues, from ENS Lyon, implemented backwards analysis in Pagai;
- Gianluca Amato and Francesca Scozzari implemented their "Localizing Widening and Narrowing"[AS13] technique in Pagai and experimented with it.

A number of improvements, orthogonal to the approaches developped in this thesis, could be implemented in the future:

- Implement better abstractions of certain LLVM instructions: one could for instance mention *load* and *store* instructions that are currently brutally abstracted, while a minimalistic memory model would already avoid many imprecisions.
- Further improve the scalability of the analysis. For instance, instead of computing abstract values with sometimes unreasonably many dimensions, one should trade precision for efficiency by "packing" the variables into different sets, chosen heuristically, as it is done by e.g. the Astrée analyzer [BCC+03]. Then, the analysis would compute relational invariants over variables that are in the same packet only, and could even use different abstract domains for each packet. This idea would certainly combine well with our modular property-guided static analysis.

[9] `http://iitkgp.ac.in/`
[10] `http://wsept.inria.fr/`
[11] `http://capp.imag.fr`

Chapter 8

Conclusion

In this thesis, we proposed several new analysis techniques that improve upon state-of-the-art in abstract interpretation, by using decision procedures, that benefit from major advances in the field of *Satisfiability Modulo Theories* in the last decade. These methods have proved themselves efficient when instanciated with usual numerical abstract domains such as intervals, octagons, or convex polyhedra. Besides these theoretical contributions, we also propose a strong practical contribution with the implementation of a new static analyzer. This thesis could be considered as part of the ERC STATOR[1] project, which focuses on developing new methods for static analysis of softwares. In this conclusion, we first recall our main contribution. Then, we present some possibilities of future work and interesting research directions. We conclude with general remarks.

Main Contributions

In a first part, we presented a theoretical contribution with the design of Guided Path Analysis, that reuses the concepts of Path Focusing and apply them to smarter iteration strategies for the fixpoint computation. The final algorithm behaves experimentally better with respect to the precision usually lost by widenings and least upper bounds operators, and thus yields more precise results than existing techniques. We presented several possibilities of extensions; for instance, for enabling the possibility of computing *disjunctive* invariants, or for avoiding a possible exponential enumeration of focused paths. In collaboration with Nicolas Halbwachs, we also presented an original method for improving the precision of a fixpoint in the case a classical descending sequence fails. By restarting a new fixpoint computation with the knowledge of an already discovered but imprecise fixpoint, we were able to converge to a strictly smaller fixpoint in many benchmarks. This idea applies for any abstract domain and iteration strategy, which makes it always applicable. Since the two methods we described are orthogonal, a natural extension would be to combine them together to see whether we can further improve precision.

In a second part, we presented another theoretical contribution: the design of modular analysis algorithms called *Modular Path Focusing*, and its "property-guided" derivative. These algorithms improve the scalability of the previously described techniques by abstracting program portions into summaries, i.e. relations between their input and output variables, that are correct under some assumptions over the input variables called *context*. We presented original ideas for generalizing the input context in which a summary holds while not sacrificing precision. We showed how this technique can be used for inferring useful preconditions, e.g. for functions, and prove user-given properties at the same time. We think that this generation of precondition is very useful in practice, e.g. when embedded in an integrated development environment (IDE). From the perspective of scalability, we described a framework for iteratively

[1] `http://stator.imag.fr`

refining precision of the summaries using abstract interpretation, in contrast with other existing techniques based on interpolation, in order to prove the unreachability of a given error state. In this work, decision procedures are used both for finding which summaries are required, and also for computing precise invariants as in our Guided Path Analysis. This works blends with en vogue techniques based on Horn Clause encoding of programs [GLPR12, HBdM11, BMR13], while proposing a different approach for computing summaries.

A working implementation and experimental evaluation will be part of our work in the near future. We expect the experiments to show that plugging in summaries for complicated program portions into the SMT encoding will improve performance of the SMT solvers.

In a third part, we proposed a novel method for estimating *Worst-Case Execution Time* (WCET): while current approaches compute semantic-insensitive WCET approximations by encoding only the program control-flow into an ILP problem, we showed how to encode both the control-flow and the instruction semantics into a problem of *Optimization Modulo Theory*. SMT-based approaches for estimating WCET has never been proposed, for the reason that it does not scale to the size of standard benchmarks. We detailed the theoretical reason why WCET problems are hard to solve by production-grade SMT solvers based on DPLL($\mathcal{T}$), and how the results of a high level static analysis dramatically improves solving time. We claim that our results is a first attempt to show that SMT is a workable and more precise alternative to classical approaches for estimating WCET.

A large amount of time during this thesis has been dedicated to the development of a new analyzer called Pagai, that currently implements 6 different static analysis techniques. The tool is open source, and has already been used by other researchers for conducting experiments, e.g. [SNA14], or implementing new techniques [AS13]. It is a major contribution to the abstract interpretation community from both the technical and theoretical point of view: there exist very few static analyzers capable of analyzing large programs written in a usual language like C; we claim that our implementation is robust and extensible, and provides usable results, e.g. annotated C code and/or instrumented LLVM IR. We presented in this report the most interesting design and implementation aspects of Pagai, together with experimental results on real-life code bases and benchmarks. We showed that this tool is competitive against other academic program verifiers, and that it can be used for computing precise numerical invariants for real C programs thanks to the use of the industrial-strength LLVM compiler infrastructure. Pagai's invariants can already be fed as input into other LLVM-based analysis tools, and are already used by various projects at Verimag and other research teams.

Future Work & Research Directions

Our principal theoretical contributions are the design of several original static analysis algorithms. In this thesis, we focused only on numerical abstract domains. We would be interested in future work to see how well these techniques behave when instantiated with other abstract domains and theories, in particular those dealing with the memory, arrays, etc. We should note that all of these new analysis techniques described here are not inherently restricted to numerical abstract domains, and only require appropriate decision procedures for the corresponding theories. Since current SMT solvers work with theories like arrays or uninterpreted functions, this research direction seems quite accessible in the near future.

The algorithms described in this thesis all share the property of producing invariants. Currently, we prove user-given properties by checking whether the intersection of their negation with the invariants is empty. Drawbacks of this approach are that it will produce false

alarms, but also cannot extract error traces from the invariants. Consequently, modern academic program verifiers (e.g. CPAChecker [BHT07] or UFO [ALGC12]) mix different techniques together: we can cite among others bounded model checking, predicate abstraction with CEGAR[CGJ+00] and interpolation-based refinement [McM03, McM06]. It is well-known that invariants given by external tools can help certain verification techniques. We would like to experiment with these verification algorithms that can be speed up by externally provided invariants, such as k-induction [GKT13], in order to see to which extent the precision of the invariant is important.

Our work on WCET estimation with SMT also opens many interesting future research topics, in particular, how to improve SMT solvers with techniques derived from the static analysis community, such as abstract interpretation. While it was quite intuitive to see which static analysis results are useful in the particular case of WCET problems, it is much more challenging to try to apply similar ideas to any SMT formula and improve the overall efficiency of DPLL($\mathcal{T}$)-based solvers. The topic of inferring good summaries is "hot": a tightly related work has been conducted by McMillan in parallel with us and has been published very recently in [McM14]. We would like to extend our idea of "cuts" to arbitrary formulas in future work, and also enhance the way of deriving cuts in the case of WCET to further improve scalability. Another research direction in the field of WCET estimation is the usage of our invariant generation algorithms as is on programs instrumented with counters, as explained in section 6.8. This last research area is currently tackled by the French ANR W-SEPT project, and powered by our tool Pagai.

From a implementation point of view, Pagai can be extended in many directions: current work consists in the implementation of our modular static analysis framework. There is also big rooms for improvement in some parts of the implementation, for instance in the abstract domain side; an example would be to pack the variables into different sets and use products of different abstract domains to improve configurability and scalability. We would also like to enable the use of logico-numerical abstract domains [Jea], which would perfectly combine with our techniques. Another improvement for long term is to implement a memory model. We would also be interested in combining Pagai with other program verifiers to compete in the SV-Competition, for instance bounded model checkers for providing error traces. Another interesting future development would be to integrate it into the official clang static analyzer[2].

Concluding Remarks

This thesis bridges the gap between at least three different active research communities: abstract interpretation, SMT, and WCET. As a consequence, it is a starting point for many improvements and future work in these three areas: The algorithms described in this thesis, mixing abstract interpretation with decision procedures, demonstrate that the use of satisfiability modulo theories is a good research direction for improving static analysis based on abstract interpretation. Conversely, our work on WCET estimation showed that decision procedures can be improved with the help of ideas from the static analysis community, in particular abstract interpretation. This last point, which has also been pointed out by several other recent works [DHK14, DHK13, TR12a], opens many interesting research problems for the near future in order to further improve SMT solvers. Finally, this thesis shows that SMT is a workable approach to precise WCET estimation and motivates new research in this direction.

To conclude, this thesis offers a good balance between strong practical contributions with

[2] `http://clang-analyzer.llvm.org/`

the implementation of a new and competitive static analyzer, and theoretical contributions with the design of new abstract interpretation and SMT-based static analysis algorithms, with already real-life applications to WCET estimation.

Bibliography

[ALGC12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *Computer Aided Verification (CAV)*, pages 672–678, 2012.

[AM13] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *Computer Aided Verification (CAV)*, pages 313–329, 2013.

[AMR13] Mihail Asavoae, Claire Maiza, and Pascal Raymond. Program semantics in model-based WCET analysis: A state of the art perspective. In Claire Maiza, editor, *International Workshop on WCET Analysis (WCET)*, volume 30 of *OASICS*, pages 32–41. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[Ari96] Ariane 5 Flight 501 Failure: Report by the Inquiry Board. Technical report, July 1996.

[AS13] Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In *Static Analysis Symposium (SAS)*, pages 25–42, 2013.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification (CAV)*, pages 171–177, 2011.

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered smt($\mathcal{BV}$) solver for hard industrial verification problems. In *Computer Aided Verification (CAV)*, pages 547–560, 2007.

[BCR13] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Precise micro-architectural modeling for WCET analysis via AI+SAT. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–96, 2013.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 35–46, 2010.

[BDdM08] Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura. Accelerating lemma learning using joins - DPLL(⊔), 2008. Appeared as short paper in LPAR 2008, outside of proceedings.

[BFG+97] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design (FMSD)*, 10(2/3):171–206, 1997.

[BHJM07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[BHRZ03] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *Static Analysis Symposium (SAS)*, pages 337–354, 2003.

[BHRZ05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, October 2005.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification (CAV)*, pages 504–518, 2007.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, 2009.

[BHZ] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. *The Parma Polyhedra Library, version 0.9*.

[BKKZ13] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. The Auspicious Couple: Symbolic Execution and WCET Analysis. In *International Workshop on WCET Analysis (WCET)*, volume 30 of *OASIcs*, pages 53–63. dagstuhl, 2013.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[BMR13] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Static Analysis Symposium (SAS)*, pages 105–125, 2013.

[Boi10] Benoit Boissinot. *Towards an SSA based Compiler Back-End: some Interesting Properties of SSA and its Extensions*. PhD thesis, École Normale Supérieure de Lyon, 2010.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, 1992.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 70–87, 2011.

[Bra12] Aaron R. Bradley. Understanding ic3. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 1–14, 2012.

[BSIG09] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Refining the control structure of loops using static analysis. In *International Conference on Embedded Software (EMSOFT)*, pages 49–58, 2009.

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, pages 511–547, August 1992.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming (ESOP)*, 2005.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGG+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification (CAV)*, pages 462–475, 2005.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.

[CGMT14] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 46–61, 2014.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*. Springer, 2013.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978.

[CJ11] Duc-Hiep Chu and Joxan Jaffar. Symbolic simulation on complicated loops for WCET path analysis. In *International Conference on Embedded Software (EMSOFT)*, pages 319–328, 2011.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[CKSY04] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design (FMSD)*, 25(2-3):105–127, 2004.

[Cla77] Edmund M. Clarke. Program invariants as fixed points (preliminary reports). In *Foundations of Computer Science (FOCS)*, pages 18–29, 1977.

[Cot09] Scott Cotton. *On Some Problems in Satisfiability Solving*. PhD thesis, Université Joseph Fourier, Grenoble, 2009.

[Cot10] Scott Cotton. Natural domain SMT: A preliminary assessment. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 77–91, 2010.

[CR13] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 49(4):517–562, 2013.

[CRT08] Paul Caspi, Pascal Raymond, and Stavros Tripakis. Handbook of real-time and embedded systems. In *Handbook of Real-Time and Embedded Systems*, chapter 14, Synchronous Programming. Chapman & Hall / CRC, 2008.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commununication of the ACM*, 56(2):82–90, February 2013.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV)*, pages 81–94, 2006.

[DHK13] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *Principle of Programming Language (POPL)*, pages 143–154, 2013.

[DHK14] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract satisfaction. In *Principle of Programming Language (POPL)*, pages 139–150, 2014.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[dMJ13] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–12, 2013.

[DOT+10] Andreas Dalsgaard, Mads Olesen, Martin Toft, Rene Hansen, and Kim Larsen. METAMOC: Modular execution time analysis using model checking. In *International Workshop on WCET Analysis (WCET)*, pages 113–123, 2010.

[EJ02] Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static wcet analysis. In *International Conference on Embedded Software (EMSOFT)*, pages 334–348, 2002.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *Formal Methods in System Design (FMCAD)*, pages 125–134, 2011.

[FG10] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electronic Notes in Theoretical Computer Science*, 267(2):3–13, 2010.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *Workshop on Worst-Case Execution TimeAnalysis (WCET)*, volume 15, pages 136–146. dagstuhl, 2010.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *European Symposium on Programming (ESOP)*, pages 237–252, 2007.

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *Static Analysis Symposium (SAS)*, pages 144–160, 2006.

[GKT13] Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In G. Brat, N. Rungta, and A. Venet, editors, *Proceedings of the 5th NASA Formal Methods Symposium (Moffett Field, CA, USA)*, volume 7871 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2013.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Programming Language Design and Implementation (PLDI)*, pages 405–416, 2012.

[Gon07] Laure Gonnord. *Accélération Abstraite pour l'Amélioration de la Précision en Analyse des Relations Linéaires.* Thèse de doctorat, Université Joseph Fourier, Grenoble, October 2007.

[GR06] Denis Gopan and Thomas W. Reps. Lookahead widening. In *Computer Aided Verification (CAV)*, pages 452–466, 2006.

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In *Static Analysis Symposium (SAS)*, pages 349–365, 2007.

[GS07a] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *European Symposium on Programming (ESOP)*, pages 300–315, 2007.

[GS07b] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *Conference on Computer Science Logic (CSL)*, pages 23–40, 2007.

[GS11] Thomas Martin Gawlitza and Helmut Seidl. Solving systems of rational equations through strategy iteration. *ACM Trans. Program. Lang. Syst.*, 33(3):11, 2011.

[GT07] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *European Symposium on Programming (ESOP)*, pages 253–267, 2007.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Programming Language Design and Implementation (PLDI)*, pages 292–304, 2010.

[Hal79] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Grenoble University, 1979.

[Hal93] Nicolas Halbwachs. Delay analysis in synchronous programs. In *Computer Aided Verification (CAV)*, pages 333–346, 1993.

[HAMM14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2014.

[HB12] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 157–171, 2012.

[HBdM11] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. μz - an efficient engine for fixed points with constraints. In *Computer Aided Verification (CAV)*, pages 457–462, 2011.

[HH12] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *Static Analysis Symposium (SAS)*, pages 198–213, 2012.

[HMM12a] Julien Henry, David Monniaux, and Matthieu Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.

[HMM12b] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science. Springer Verlag, 2012.

[Hol97] Gerard J Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hol08] Niklas Holsti. Computing time as a program variable: a way around infeasible paths. In *International Workshop on WCET Analysis (WCET)*, volume 08003 of *Dagstuhl Seminar Proceedings*. dagstuhl, 2008.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design (FMSD)*, 11(2):157–185, 1997.

[HS09] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In *International Workshop on WCET Analysis (WCET)*, 2009.

[HSIG10] William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In *Principle of Programming Language (POPL)*, pages 71–82, 2010.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal Methods in System Design (FMCAD)*, pages 1–9, 2008.

[HW02] Christopher Healy and David Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), August 2002.

[ISO99] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[Jea] Bertrand Jeannet. Bddapron: A logico-numerical abstract domain library.

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design (FMSD)*, 23(1):5–37, 2003.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification (CAV)*, pages 661–667, 2009.

[JPC14] Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. Springer, 2014.

[JSS14] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *Principle of Programming Language (POPL)*, pages 529–540, 2014.

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Computer Aided Verification (CAV)*, pages 17–34, 2014.

[Kin76] James C. King. Symbolic execution and program testing. *Communication of the ACM*, 19(7):385–394, 1976.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Programming Language Design and Implementation (PLDI)*, pages 193–204, 2012.

[KKZ13] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Real-Time and Network Systems (RTNS)*, pages 161–170, 2013.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Washington, DC, USA, August 2004. IEEE Computer Society.

[LAK+14] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. In *Principle of Programming Language (POPL)*, pages 607–618, 2014.

[LCJG11] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *Automated Technology for Verification and Analysis (ATVA)*, pages 492–502, 2011.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1–3):56–67, 2007.

[LM97] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *Computer-Aided Design of Integrated Circuits and Systems (CADICS)*, 16(12):1477–1487, 1997.

[Log11] Francesco Logozzo. Practical verification for the working programmer with code-contracts and abstract interpretation - (invited talk). In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 19–22, 2011.

[MB11] David Monniaux and Martin Bodin. Modular abstractions of reactive nodes using disjunctive invariants. In *Programming Languages and Systems (APLAS)*, pages 19–33, 2011.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification (CAV)*, volume 2725, pages 1–13. Springer, 2003.

[McM05] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–12, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, pages 123–136, 2006.

[McM14] Kenneth L. McMillan. Lazy annotation revisited. In *Computer Aided Verification (CAV)*, 2014.

[Met04] Alexander Metzner. Why model checking can improve WCET analysis. In *Computer Aided Verification (CAV)*, pages 334–347, 2004.

[MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Static Analysis Symposium (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 369–385. Springer, 2011.

[MG12] David Monniaux and Julien Le Guen. Stratified static analysis based on variable dependencies. *Electric Notes in Theoretical Computer Science*, 288:61–74, 2012.

[Min04] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Dec. 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[Min06] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 348–363, 2006.

[MKS09] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In *Computer Aided Verification (CAV)*, pages 462–476, 2009.

[Mon10] David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer-aided verification (CAV)*, number 6174 in Lecture Notes in Computer Science, pages 585–599. Springer Verlag, 2010.

[Moy08] Yannick Moy. Sufficient preconditions for modular assertion checking. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 188–202, 2008.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzer. In *European Symposium on Programming (ESOP)*, number 3444 in Lecture Notes in Computer Science, pages 5–20. Springer, 2005.

[NCS+06] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *International Workshop on WCET Analysis (WCET)*, 2006.

[NRM04] Hemendra Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying WCET analysis by code transformations. In *International Workshop on WCET Analysis (WCET)*, 2004.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems (RTS)*, 5(1):31–62, 1993.

[Pol] The mathworks polyspace verifier.

[Rei09] Jan Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, University of Saarland, 2009.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[RMPVC13] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, and Fabienne Carrier. Timing analysis enhancement for synchronous program. In *Real-Time and Network Systems (RTNS)*, pages 141–150, 2013.

[RW10] Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[SDDA11] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification (CAV)*, pages 703–719, 2011.

[Sha79] Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal of Computing*, 8(4):645–655, 1979.

[SJ11] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis Symposium (SAS)*, pages 233–248, 2011.

[SNA14] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Bias-variance tradeoffs in program analysis. In *Principle of Programming Language (POPL)*, pages 127–138, 2014.

[SP81] M Sharir and M Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125, 2000.

[ST12] Roberto Sebastiani and Silvia Tomasi. Optimization in smt with $\mathcal{LA}(\mathbb{Q})$ cost functions. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 484–498, 2012.

[STDG12] Mohamed Amin Ben Sassi, Romain Testylier, Thao Dang, and Antoine Girard. Reachability analysis of polynomial systems using linear programming relaxations. In *Automated Technology for Verification and Analysis (ATVA)*, pages 137–151, 2012.

[SVC] Software verification competition (sv-comp). Organized every year, and colocated with the TACAS conference.

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *Formal Methods (FM)*, volume 5850 of *LNCS*, pages 532–546. Springer, 2009.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *International Journal of Time-Critical Computing Systems*, 18:157–179, 2000.

[Tin02] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 308–319, 2002.

[TR12a] Aditya V. Thakur and Thomas W. Reps. A generalization of stålmarck's method. In *Static Analysis Symposium (SAS)*, pages 334–351, 2012.

[TR12b] Aditya V. Thakur and Thomas W. Reps. A method for symbolic computation of abstract operations. In *Computer Aided Verification (CAV)*, pages 174–192, 2012.

[W+08] Reinhard Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Computer Systems*, 7(3), 2008.

[Wil06] Reinhard Wilhelm. Determining bounds on execution times. In *Handbook on Embedded Systems*, chapter 14. CRC Press, 2006.

[WYGI07] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Computer Aided Verification (CAV)*, pages 352–365, 2007.

[ZKW+06] Wankang Zhao, William C. Kreahling, David B. Whalley, Christopher A. Healy, and Frank Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Systems*, 34(2):129–152, 2006.

Chapter A

Experimental Results

A.1 Extra experiments for section 4.1

We provide other experiments with our new decreasing sequence, on the benchmarks from the Mälardalen WCET benchmarks set [GBEL10] available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. The results are depicted in Table A.1

A.2 SV-Comp Experiments

In this section, we plot the results of the SV-Comp in the exact same way as in subsection 7.3.2, but we also give points for correct FALSE results (1 point), while they were ignored in the plots of subsection 7.3.2. Pagai always returns TRUE or UNKNOWN, so it gets the exact same number of points with this rule.

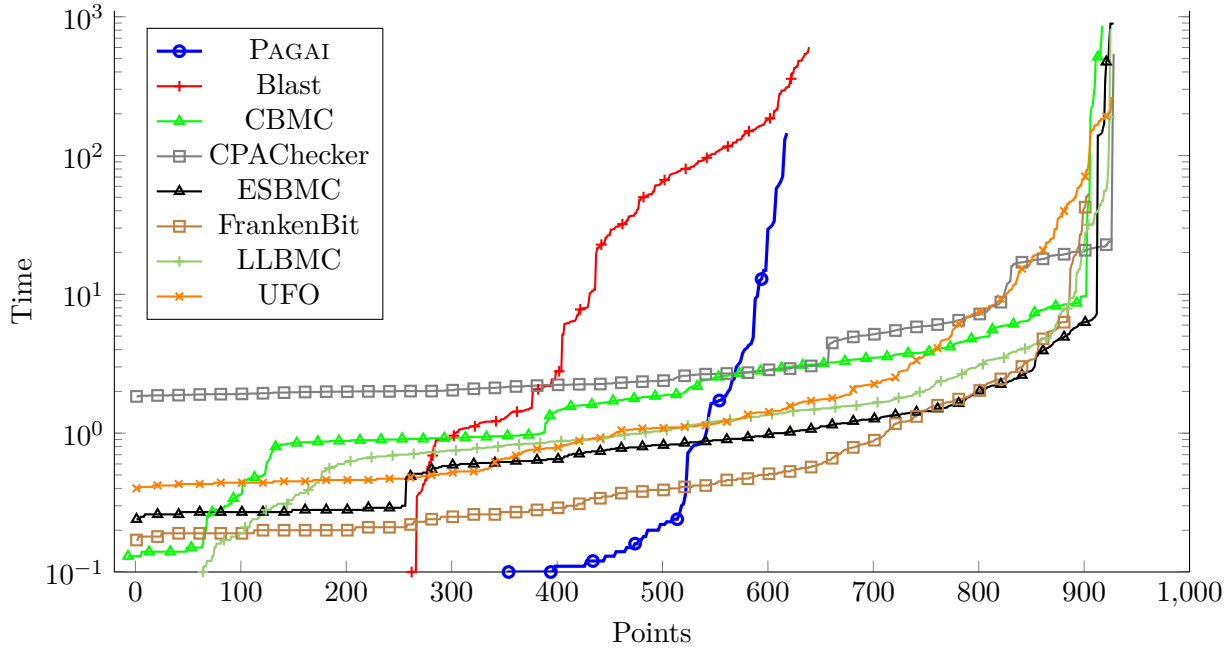


Figure A.1: Category ProductLines, TRUE and FALSE benchmarks

Benchmark	Comparison		Time		Functions				Time eq	
	$\sqsubset$	=	N	S	#total	#seeds	$\sqsubset$	=	N	S
bs_annotated	0	1	0.88	0.76	1	0	0	1	0.88	0.76
insertsort	2	0	0.97	0.37	1	1	1	0	0.00	0.00
adpcm	6	21	0.54	0.39	1	1	1	0	0.00	0.00
lcdnum	1	0	0.39	0.29	1	1	1	0	0.00	0.00
fdct	0	2	0.12	0.96	1	0	0	1	0.12	0.96
select	3	1	0.16	0.11	1	1	1	0	0.00	0.00
fir	2	0	0.69	0.46	1	1	1	0	0.00	0.00
cnt	4	0	0.60	0.27	1	1	1	0	0.00	0.00
duff	0	2	0.30	0.27	1	0	0	1	0.30	0.27
prog9000	1	38	5.59	2.40	1	1	1	0	0.00	0.00
recursion	0	0	0.51	0.43	3	0	0	3	0.51	0.43
jfdctint	0	3	0.20	0.16	1	0	0	1	0.20	0.16
fft1	8	21	1.79	1.20	1	1	1	0	0.00	0.00
sqrt	0	1	0.73	0.70	1	0	0	1	0.73	0.70
matmult	6	1	0.39	0.17	1	1	1	0	0.00	0.00
qurt	3	0	0.52	0.36	1	1	1	0	0.00	0.00
statemate	1	0	0.56	0.21	1	1	1	0	0.00	0.00
minver	7	10	0.34	0.25	1	1	1	0	0.00	0.00
bsort100	1	2	0.92	0.66	1	1	1	0	0.00	0.00
ns	2	2	0.37	0.19	1	1	1	0	0.00	0.00
qsort-exam	0	6	0.15	0.11	1	1	0	1	0.15	0.11
nsichneu	0	1	2.29	2.77	1	0	0	1	2.29	2.77
lms	6	6	0.66	0.47	1	1	1	0	0.00	0.00
edn	5	7	0.11	0.58	1	1	1	0	0.00	0.00
ludcmp	9	2	0.11	0.56	1	1	1	0	0.00	0.00
compress	1	9	0.32	0.25	1	1	1	0	0.00	0.00
expint	0	2	0.25	0.13	1	1	0	1	0.25	0.13
bs	0	1	0.83	0.70	1	0	0	1	0.83	0.70
fac	0	1	0.35	0.29	2	0	0	2	0.35	0.29
prime	0	2	0.30	0.26	1	0	0	1	0.30	0.26
janne_complex	0	2	0.42	0.36	1	1	0	1	0.42	0.36
cover	0	3	35.71	36.11	1	0	0	1	35.71	36.11
st	0	7	0.36	0.30	1	0	0	1	0.36	0.30
crc	1	5	0.52	0.28	1	1	1	0	0.00	0.00
ud	9	2	0.93	0.50	1	1	1	0	0.00	0.00
ndes	5	7	0.49	0.33	1	1	1	0	0.00	0.00
fibcall	0	1	0.97	0.94	1	0	0	1	0.97	0.94
loop	0	1	0.38	0.41	1	0	0	1	0.38	0.41

Table A.1: Experiments with the new decreasing sequence, on the

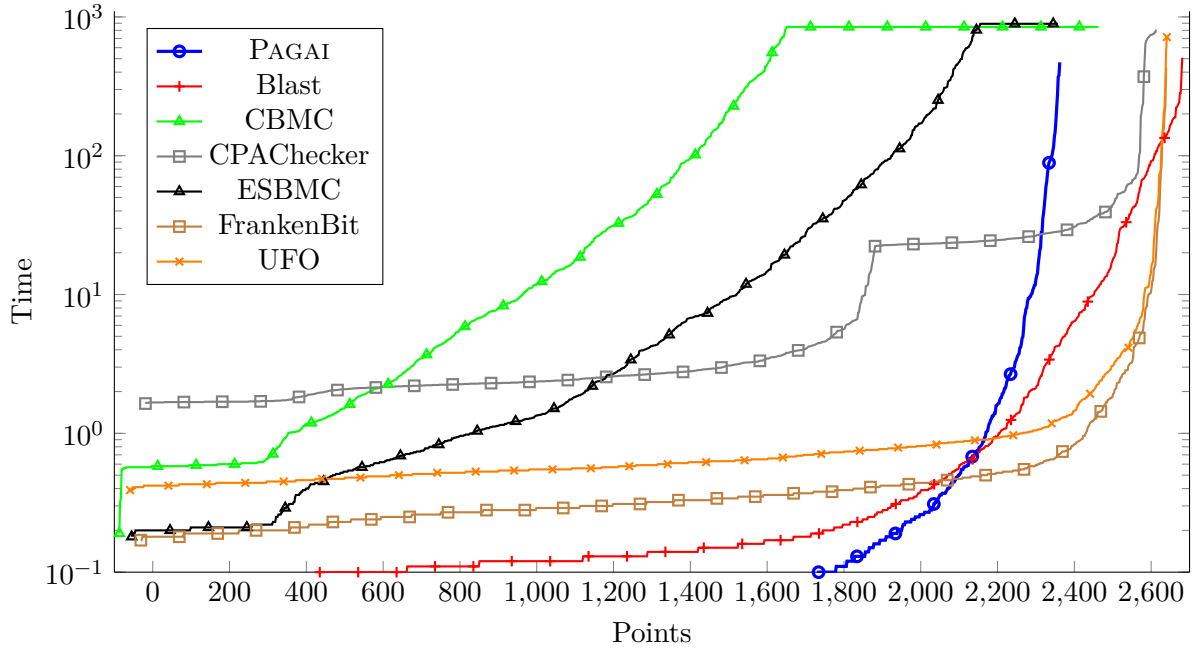


Figure A.2: Category DeviceDrivers64, TRUE and FALSE benchmarks

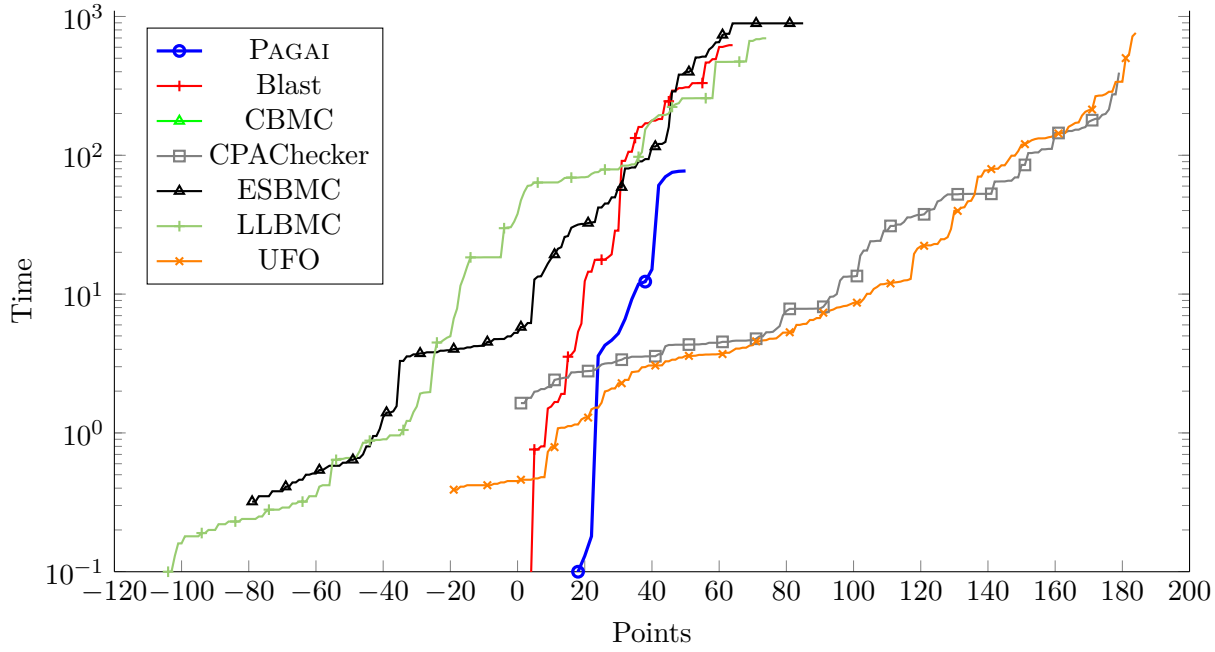


Figure A.3: Category ControlFlowInteger, TRUE and FALSE benchmarks

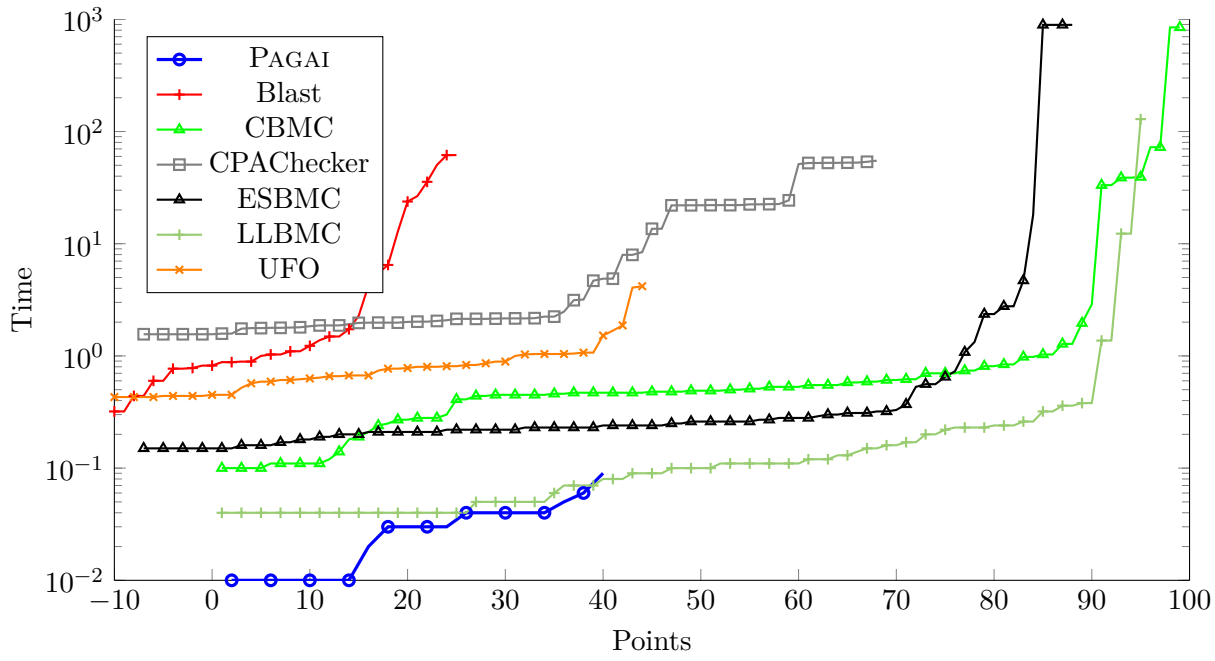


Figure A.4: Category Loops, TRUE and FALSE benchmarks.