

Modular analysis of numerical properties by abstract interpretation

Rémy Boutonnet

▶ **To cite this version:**

Rémy Boutonnet. Modular analysis of numerical properties by abstract interpretation. Data Structures and Algorithms [cs.DS]. Université Grenoble Alpes [2020-..], 2020. English. NNT : 2020GRALM004 . tel-02554125v2

HAL Id: tel-02554125
https://theses.hal.science/tel-02554125v2

Submitted on 27 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Rémy BOUTONNET

Thèse dirigée par **Nicolas HALBWACHS**, Directeur de Recherche émérite, Université Grenoble Alpes

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Analyse modulaire de propriétés numériques par interprétation abstraite

Modular analysis of numerical properties by abstract interpretation

Thèse soutenue publiquement le **9 mars 2020**,
devant le jury composé de :

Monsieur NICOLAS HALBWACHS
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur ANDREAS PODELSKI
PROFESSEUR, UNIVERSITE DE FRIBOURG - ALLEMAGNE, Rapporteur
Monsieur ANTOINE MINE
PROFESSEUR, SORBONNE UNIVERSITE - PARIS, Rapporteur
Monsieur CESARE TINELLI
PROFESSEUR, UNIVERSITE DE L'IOWA - ETATS-UNIS, Examinateur
Madame CORINNE ANCOURT
MAITRE DE RECHERCHE, MINES PARISTECH, Examinateur
Monsieur OLIVIER BOUISSOU
DOCTEUR-INGENIEUR, SOCIETE THE MATHWORKS A MEUDON, Examinateur
Madame SUZANNE GRAF
DIRECTRICE DE RECHERCHE, CNRS DELEGATION ALPES, Président

How can one check a routine in the sense of making sure that it is right ?

ALAN M. TURING
Checking a large routine
Report of a Conference on High Speed Automatic Calculating Machines, Cambridge,
24th June 1949.

Abstract

Any software bug or device malfunction in safety-critical systems can have catastrophic consequences. The validation and analysis of programs in critical systems is of paramount importance to guarantee that the software satisfies its specification and that it is devoided of runtime errors.

Static program analysis by abstract interpretation computes a sound approximation of the set of reachable states of a program. It discovers invariant properties of programs which are represented by elements of an abstract domain. Most industrial static analysis tools do not use expressive relational numerical abstract domains like convex polyhedra due to their computational cost.

We propose a new modular analysis for the automatic discovery of numerical properties based on the computation of disjunctive relational summaries of procedures. Procedure summaries are computed once and for all, and used to compute the effect of procedure calls, in a bottom-up fashion. Our approach is especially applied to improve the scalability of Linear Relation Analysis, or abstract interpretation with convex polyhedra, although it is based on a more general framework usable with any relational abstract domain.

Disjunctive relational summaries are finite sets of abstract input-output relations represented by elements of a relational abstract domain. They are computed based on a partitioning of procedure preconditions. We give heuristics to compute an abstract partition of a procedure precondition. We also give ways to improve the precision of summary computation, notably through a careful treatment of preconditions during analysis. Our approach also applies to recursive procedures where summaries are computed recursively in terms of themselves.

We implemented our approach in a new static analysis platform for C programs called MARS. We conducted experiments on programs from the Mälardalen benchmark showing that our approach can significantly reduce the analysis time for Linear Relation Analysis compared to a full context-sensitive analysis where procedures are analyzed completely in each call context. Analysis precision is not significantly damaged and can even be improved due to the use of disjunction.

In a second part, we present an approach for the modular analysis of reactive systems for numerical properties. We propose a flexible representation of the behavior of reactive components called *Relational Mode Automata* (RMA), which allows the analysis of reactive systems behavior at various levels of abstraction. RMA can be constructed automatically from disjunctive summaries of the procedures implementing component reactions. The analysis results of individual components using RMA can be reused to analyze larger reactive systems with multiple component instantiations in a modular way. We give an application of this approach to the analysis of a simplified automated subway control system.

Keywords: static analysis, abstract interpretation, interprocedural analysis, procedure summaries, convex polyhedra, reactive systems, synchronous programming.

Résumé

La présence de bugs ou de dysfonctionnements dans les systèmes critiques peut avoir des conséquences terribles. La validation et l'analyse des programmes embarqués dans les systèmes critiques est d'une importance majeure pour garantir que les logiciels présents soient conformes à leur spécification et dépourvus d'erreurs à l'exécution.

L'analyse statique par interprétation abstraite calcule une approximation sûre de l'ensemble des états accessibles d'un programme. Elle permet de découvrir des propriétés invariantes des programmes en les représentant par des éléments d'un domaine abstrait. Les domaines abstraits numériques, comme le domaine des octogones ou des polyèdres convexes, ont des niveaux de précision différents. Les outils industriels d'analyse statique n'utilisent généralement pas les domaines abstraits numériques les plus expressifs, comme le domaine des polyèdres convexes, à cause de la complexité de leurs opérations.

Nous présentons dans cette thèse une analyse modulaire des programmes pour la découverte de propriétés numériques, basée sur le calcul de résumés disjonctifs et relationnels de procédures. Les résumés de chaque procédure sont calculés de manière ascendante (*bottom-up*) et utilisés dans l'analyse de l'effet des appels de procédure. Bien que notre approche soit appliquée à l'analyse de relations linéaires, ou interprétation abstraite polyédrique, pour améliorer son passage à l'échelle, elle est définie dans un cadre plus large applicable à n'importe quel domaine abstrait relationnel.

Les résumés disjonctifs de procédures sont des ensembles finis de relations d'entrée-sortie représentées par des éléments d'un domaine abstrait relationnel. Ils sont calculés en utilisant un partitionnement de la précondition d'une procédure. Nous proposons des heuristiques pour calculer des partitions de préconditions. Nous présentons aussi des améliorations concernant la précision du calcul des résumés, notamment grâce à un traitement particulier des préconditions. Notre approche s'applique également aux procédures récursives, où les résumés sont calculés en fonction d'eux-mêmes.

Notre analyse modulaire à été implémentée dans un nouvel outil d'analyse statique pour les programmes C, appelé MARS. Nos expérimentations montrent que notre approche peut réduire significativement le temps d'analyse pour l'analyse des relations linéaires, par comparaison avec une analyse classique qui analyse les procédures dans chaque contexte d'appel. La précision des résultats n'est pas considérablement diminuée et peut même être améliorée grâce à l'usage de la disjonction dans les résumés.

Dans une seconde partie, nous présentons une approche pour l'analyse modulaire des systèmes réactifs. Nous proposons une représentation flexible des composants réactifs appelée *Automates Relationnels de Mode*, ou *Relational Mode Automata* (RMA), permettant une analyse des systèmes réactifs à différents niveaux d'abstraction. Les automates relationnels de mode peuvent être construits automatiquement à partir des résumés disjonctifs des procédures implémentant la réaction de chaque composant. Les résultats de l'analyse de chaque composant peuvent être réutilisés dans l'analyse de systèmes réactifs de plus grande taille où chaque composant peut être instancié plusieurs fois. Cette approche est appliquée à l'analyse d'un système simplifié de contrôle d'un réseau de métro.

Mots clés : analyse statique, interprétation abstraite, analyse interprocédurale, résumés de procédures, polyèdres convexes, systèmes réactifs, programmation synchrone.

Acknowledgements

I thank Antoine Miné and Andreas Podelski for reviewing this manuscript and for their remarks. I thank Cesare Tinelli, Corinne Ancourt, Olivier Bouissou and Susanne Graf for taking part in this thesis jury.

I thank my advisor Nicolas Halbwachs for having given me the amazing opportunity to work with him, since my TER research internship five years ago. This work would not have been possible without him. I am grateful and indebted to his immense wisdom, vast scientific knowledge, his constant support and great availability.

I thank Claire Maiza and Fabienne Carrier for their support and for giving me the chance to discover research in computer science during my L3 magistère internship.

I thank my fellow colleagues and students from Verimag and the IMAG building, most notably Valentin, Yanis, Clément, Cyril, Vincent, Matheus, Bai and Maëva for their friendship and all the meals and discussions that we shared during these three years. I thank Valentin for all the interesting and passionate discussions that we had about abstract interpretation, the most arcane areas of computer science and various other subjects during *la pause*.

I thank my friends from university Raphaël and Timothée for their companionship.

I thank Adam, Damien, Romain and Théotime for being the most amazing friends since all this time.

I thank Jérémy for being my friend and being in my life since our first days at university, and also as one of the most talented persons I have the honor to know.

I would like to thank my grandfather, who gave me a taste for science since an early age. He transmitted me his love for the subtle art and exact science of tinkering, finding creative technical solutions, and encouraged me to listen to the amazing stories that old books and objects are waiting to tell.

I would like to thank my grandmother Jeanne for her wisdom, for teaching me the benefits of hard work and various rare skills.

I thank my parents and my brother for their constant and unwavering support, especially in times of doubt and adversity. My parents transmitted me the values of gentleness, openness and striving for what is right. They showed me how to be curious about the world around us. I am especially grateful for the Sunday family tradition of forest walks and exploration of all sorts of ancient, mysterious or simply long-forgotten buildings and places.

Contents

1 Introduction **15**
1.1 Context . . . 15
1.2 Contributions . . . 16
1.3 Outline . . . 17

I State of the Art **19**

2 Program Analysis by Abstract Interpretation **21**
2.1 Programs and Semantics . . . 21
2.1.1 Transition Systems . . . 21
2.1.2 Programs as Control Flow Graphs . . . 22
2.1.3 Collecting Semantics . . . 23
2.2 Fixpoint Theorems . . . 24
2.2.1 Complete Lattices . . . 24
2.2.2 Monotonic Functions and Tarski's Fixpoint Theorem . . . 25
2.2.3 Continuous Functions and Kleene's Theorem . . . 25
2.3 Concrete and Abstract Domains . . . 26
2.3.1 Abstract Domains . . . 26
2.3.2 Fixpoint Computation . . . 27
2.3.3 Convergence . . . 28
2.4 Numerical Abstract Domains . . . 31
2.4.1 Affine Equalities . . . 31
2.4.2 Zones . . . 31
2.4.3 Octagons . . . 32
2.4.4 Convex Polyhedra . . . 33
2.5 Advanced techniques . . . 36
2.5.1 Guided Static Analysis . . . 36
2.5.2 Trace Partitioning . . . 37

3 Interprocedural Analysis: A State of the Art **39**
3.1 The Origins of Interprocedural Analysis . . . 39
3.1.1 The Functional Approach . . . 40
3.1.2 The Call Strings Approach . . . 42
3.2 Interprocedural Data Flow Analysis via Graph Reachability . . . 44
3.3 Stack Abstractions For Interprocedural Analysis . . . 46
3.4 Statement-level Summaries . . . 47

3.5 Procedure Summaries Using Generic Assertions . . . 48
3.6 Relational Interprocedural Analyses . . . 48
3.6.1 Modular Static Analysis . . . 48
3.6.2 Relational Abstractions of Functions . . . 49
3.6.3 Interprocedural Analyses based on Linear Algebra . . . 49
3.6.4 Dual Interprocedural Analysis using Linear Arithmetic . . . 50

II Relational Summaries for Interprocedural Analysis 51

4 Relational Abstract Interpretation 53
4.1 Relations on States and the Transitive Closure . . . 53
4.2 Concrete Relational Semantics . . . 56
4.2.1 Concrete Relational Summaries . . . 56
4.2.2 Concrete Semantics of Procedure Calls . . . 56
4.2.3 From State to Relational Collecting Semantics . . . 57
4.2.4 A very simple example . . . 58
4.3 Relational Abstract Interpretation . . . 59
4.3.1 General Framework . . . 59
4.3.2 Building Summaries Using LRA . . . 61
4.3.3 Example . . . 62
4.4 Preconditions . . . 63
4.4.1 Widening Under a Precondition . . . 64
4.5 Conclusion . . . 65

5 Disjunctive Relational Procedure Summaries 67
5.1 Motivating Example . . . 68
5.2 Disjunctive Refinement of Abstract Relations . . . 69
5.2.1 General Framework . . . 69
5.2.2 Refinement by Precondition Partitioning . . . 71
5.2.3 Abstract Effect of a Call . . . 74
5.2.4 Application to Linear Relation Analysis . . . 74
5.3 Partition Refinement . . . 76
5.3.1 Refinement Heuristics . . . 76
5.3.2 Refinement According to Local Reachability . . . 77
5.3.3 Refinement According to the Summary of a Called Procedure . . . 79
5.3.4 Iterative Refinement . . . 81
5.4 A Last Improvement: Postponing Loop Feedback . . . 86
5.5 Summaries of Recursive Procedures . . . 87
5.6 Conclusion . . . 89

6 Implementation and Experiments 91
6.1 The MARS Static Analyzer . . . 91
6.1.1 Basic Usage . . . 92
6.2 Design and Implementation . . . 92
6.2.1 Abstract Domains . . . 94

6.2.2 Scheduler . . . 94
6.2.3 Transfer Function Module . . . 95
6.3 The MARS Intermediate Representation . . . 96
6.3.1 Abstract Syntax . . . 96
6.3.2 Translation of Tests . . . 98
6.4 Non-duplication of Procedure Parameters . . . 99
6.4.1 Boolean attributes . . . 99
6.4.2 Semantic equations . . . 100
6.5 Experiments . . . 101
6.6 Conclusion . . . 104

III Modular Analysis of Reactive Systems 107

7 Analysis and Verification of Reactive Systems 109
7.1 Introduction to Reactive Systems . . . 109
7.2 Synchronous Languages . . . 110
7.3 The LUSTRE Programming Language . . . 111
7.4 Compilation of Synchronous Languages . . . 112
7.4.1 Modular Compilation . . . 113
7.5 Structured Reactive Programs . . . 114
7.6 Analysis and Verification . . . 116
7.6.1 Verification of Synchronous Programs and Observers . . . 116
7.6.2 Analysis of Synchronous Programs . . . 116
7.7 Conclusion . . . 119

8 Towards a Modular Analysis of Reactive Systems using Relational Mode Automata 121
8.1 The Bounded Event Counter . . . 122
8.2 Disjunctive Summaries of Step Procedures . . . 123
8.2.1 General Principle . . . 123
8.2.2 Example . . . 125
8.2.3 From Disjunctive Summaries To Modes . . . 128
8.3 Relational Mode Automata . . . 129
8.3.1 Definitions . . . 129
8.3.2 Semantics . . . 131
8.3.3 Parallel Composition . . . 132
8.4 Reachability Analysis of Relational Mode Automata . . . 136
8.4.1 Computation Strategy . . . 136
8.4.2 Reachability Analysis of the Bounded Event Counter . . . 138
8.4.3 Iterative Construction of the Control Transition Relation . . . 139
8.5 Analysis of a Parallel Composition of Relational Mode Automata . . . 140
8.5.1 Computation Strategy . . . 141
8.5.2 Construction of the Reduced Product Automaton . . . 142
8.5.3 Example . . . 143
8.6 Reduction of Relational Mode Automata . . . 145
8.6.1 Reduction by Merging Modes . . . 145

8.6.2 Internal Reduction . . . 146
8.7 Invariants . . . 146
8.7.1 Invariant of a Relational Mode Automaton . . . 146
8.7.2 Weaker invariants . . . 147
8.7.3 Disjunctive Invariant . . . 147
8.7.4 Relational Augmentation . . . 148
8.8 Conclusion . . . 148

9 Example: A Subway Control System **149**
9.1 Detailed Summary of the `train_step` Procedure . . . 150
9.1.1 Refinement on Memory Variables . . . 152
9.1.2 Refinement on Input Variables . . . 154
9.1.3 Summary . . . 157
9.2 Relational Mode Automaton for a Train . . . 159
9.3 Analysis of a Single Train . . . 160
9.4 Modular Analysis of a Pair of Trains . . . 163
9.5 Conclusion . . . 165

10 Conclusion **167**

Bibliography **171**

Chapter 1

Introduction

Many things have changed since the first paper on program verification by Alan Turing in 1949. Abstract interpretation and interprocedural analysis were born more than four decades ago. This thesis stands on the shoulders of giants.

1.1 Context

Today, embedded systems are everywhere. They are computer systems which are part of a larger device. The examples are numerous, from consumer electronics with smartphones or digital watches, and to very large and complex systems in avionics, transportation or nuclear power plants. Many of them are produced in large volumes and most of them are difficult to update, such as the firmware deeply embedded in communication subsystems of smartphones or the control software of nuclear reactors.

Many embedded systems have real-time constraints and some of them are deemed *safety-critical* in which any bug of the controlling software or malfunction of the device can have catastrophic consequences, with a major impact on the environment and may cause a large number of injuries or death. The examples of incidents involving a software bug are well-known. The Ariane 5 crash [85] in 1996 was due to an integer overflow. The Therac-25 radiation therapy machine [84] gave patients massive overdoses of radiation in at least six accidents between 1985 and 1987 due to various software defects and wider software engineering problems. More recently, the Boeing Dreamliner 787 [2] had to be rebooted every 248 days due to a counter overflow in the firmware of the AC power management system.

This motivates the need for a reliable and systematic validation process of safety-critical systems providing trustworthy guarantees that the software satisfies its specification and that it is devoided of runtime errors. There are several families of approaches:

- Testing or dynamic analysis consists in observing the behavior of the software at runtime on some well-chosen inputs. Although this approach can detect bugs, it is not exhaustive, since all inputs and scenarios can not be tested in general. Testing can not prove the absence of bugs.
- Static analysis is a class of techniques performed on a static representation of a program without executing it. Some of these techniques mathematically prove that a program does not have bugs whereas others are able to find bugs.

Program analysis by abstract interpretation computes a sound approximation of the

set of reachable states of a program. It discovers invariant properties of programs which are represented by elements of an abstract domain. These invariants can be used to prove the absence of errors at runtime, such as arithmetic overflows, or to prove a given assertion. Industrial tools such as Polyspace [94] or Astrée [19] can prove the absence of runtime errors in large embedded software written in C. Some tools such as PIPS [69] Clousot [46] or Infer [29] are applied to a more general variety of software.

1.2 Contributions

Astrée made the achievement to scale to large embedded programs written in C, at the expense of avoiding more precise abstract domains like convex polyhedra due to their computational complexity. Less expressive abstract domains such as intervals of integers or octagons are used instead. Despite large programs being organized as collections of functions or procedures, Astrée performs a top-down analysis of programs without really considering their interprocedural structure. Procedures are either inlined or reanalyzed each time they are called.

Modular Analysis with Relational Procedure Summaries

In this thesis, we are interested in improving the scalability of Linear Relation Analysis, or abstract interpretation based on convex polyhedra, by computing precise relational summaries of procedures. We make use of the procedural structure of programs to propose a new modular analysis for the automatic discovery of numerical properties. Although specifically applied to convex polyhedra, it is based on a more general framework usable with any relational abstract domain.

Our analysis framework comes with a formalization of relational abstract interpretation that we did not find elsewhere. Based on relational collecting semantics, we present the construction of relational procedure summaries as abstract input-relations between the initial values of procedure parameters and their final value at the exit of a procedure.

Procedures can exhibit very different behaviors which can not be expressed precisely by a single element of a classical abstract domain such as convex polyhedra. In order to have precise procedure summaries, we compute disjunctive relational summaries, where each disjunct is an abstract input-output relation represented by a separate element of a relational abstract domain.

Taken together, the sources of each individual relation in a disjunctive summary form an abstract partition of the procedure precondition to account for all the possible values of parameters. The precision of a disjunctive summary is determined by the quality of the precondition partitioning. We give heuristics to compute an abstract partition of a procedure precondition. We also give ways to improve the precision of summary computation. As we show that preconditions are at the heart of precise summaries, they are entitled to a specific and careful treatment during summary computation. Summaries are computed once and for all in a bottom-up fashion and used to analyze the effect of procedure calls.

We also show that our approach applies to recursive procedures. In some way, the summary of a recursive procedure must be computed in terms of itself.

We implemented this approach in a new static analyzer called MARS (Mars Abstract Interpretation Research System) based on CLANG which consists of 20000 lines of C++ and 4000 lines of OCAML. It was designed from the ground-up for the experimentation of new static analyses. The MARS intermediate representation has been designed specifically to ease the development of static analyses by abstract interpretation and to provide high-quality source traceability information. This work has been published in [27].

Modular Analysis of Reactive Programs

Many embedded programs must react continuously to inputs at a speed determined by their environment. They are called *reactive programs*. Reactive programs can be written in languages like C or in special-purpose languages like synchronous languages such as LUSTRE, ESTEREL or SIGNAL. The flight control software of the Airbus A380 was written in SCADE which is an industrial dataflow synchronous language based on LUSTRE.

Reactive programs are usually organized as collections of components computing the system reaction to inputs and producing outputs. Reactive components can also have memory which is updated at each reaction step. They are implemented by step procedures which are called repeatedly inside a global infinite loop.

Although Astrée scales to large C programs generated from SCADE, it does not take advantage of the specific nature of reactive programs and the structuring into components is completely lost during the analysis.

In a second part, we pave the way toward a new modular analysis of reactive systems, based on the computation of disjunctive relational summaries of step procedures. We propose an abstraction of reactive programs designed specifically for analysis called *Relational Mode Automata* (RMA). It provides a flexible representation allowing the expression of reactive systems behavior at various levels of detail. RMA are constructed automatically from the disjunctive relational summary of a step procedure. We propose a reachability analysis of RMA and their parallel composition which does not require a costly explicit construction of the parallel product prior to the analysis.

Various heuristics are given to reduce the level of precision of a given relational mode automaton, in order to achieve different tradeoffs with respect to precision and analysis performance. The analysis results of a component can be used for the modular analysis of a larger system containing it, by computing its abstract effect in other step procedures.

This approach is currently implemented on small examples using the PYTHON library PYAPRON which provides a high-level binding to the APRON abstract domain library.

Other Works

During this thesis, the continuation of earlier works has been published, regarding some improvements of abstract interpretation beyond the classical increasing and decreasing sequences [26] and an application of Linear Relation Analysis to the estimation of the Worst-Case Execution Time of programs [115].

1.3 Outline

This thesis is organized into three parts:

Part I gives an introduction to static analysis by abstract interpretation and gives an overview of the state of the art regarding the interprocedural analysis of programs. It first presents the program representation used throughout this thesis and recalls some basic notions of semantics, along with the standard framework of abstract interpretation. It also mentions some improvements to classical abstract interpretation for precision. Then, it gives an overview of interprocedural analysis and its main approaches, from foundational works to the state of the art, giving a modest glimpse to its abounding literature ranging over more than four decades.

Part II presents our modular analysis based on the computation of disjunctive relational procedure summaries. It first describes our general framework with a formalization of relational abstract interpretation. We give a particular application of this framework to the convex polyhedra abstract domain. A special light is shed on the treatment of preconditions and their role in summary computation. Then, we shows how disjunctive procedure summaries can be obtained based on precondition partitioning. We give several partitioning heuristics and improvements to summary computation. Finally, we present the implementation of the MARS static analyzer, as well as some experimental results.

Part III is dedicated to our approach for a modular analysis of reactive systems. It starts with an introduction to reactive programs and synchronous languages, along with the main approaches for analysis and verification. Then, we propose a representation of reactive programs called *Relational Mode Automata* which are constructed automatically from the summary of a step procedure. We present the analysis of RMA and ways to tune their precision. Finally, we illustrate this approach on the example of a simplified subway control system.

Part I

State of the Art

Chapter 2

Program Analysis by Abstract Interpretation

Abstract interpretation [34, 37] is a general theory of the sound approximation of the behavior of dynamic discrete systems. Its elegance resides in its ability to be applied to a wide range of systems, from biological systems [43] to programs. It is especially applied in program analysis to the automatic discovery of program properties and to prove that a given program satisfies some important properties such as "*the program never does an arithmetic overflow*", "*the program never dereferences a null pointer*", or "*the program never accesses an array out of bounds*" and more generally "*no runtime error can happen*". Abstract interpretation is used by static analysis tools such as Polyspace [94], Astrée [19], Clousot [46], TVLA [83] or Infer [29], notably for the analysis of safety-critical software, such as in avionics or railway transportation systems.

Most interesting properties in program analysis, including the aforementioned safety properties, are undecidable in general due to the expressivity of program semantics. The sets of reachable states of programs are generally not computable by machine.

Abstract interpretation gives a general framework for the sound approximation of program semantics, through a decidable abstraction of semantics, which can be used to compute an over-approximation of the set of reachable states in a program.

We will recall in what follows the basic principles of abstract interpretation which are necessary for the subsequent chapters.

2.1 Programs and Semantics

2.1.1 Transition Systems

The behavior of a program can be described by a transition system $\mathcal{T} = (S, I, \tau)$ where S is a set of states, $I \subseteq S$ is a set of initial states and $\tau \subseteq S \times S$ is a transition relation over the set S of states. We denote as $s_1 \to s_2$ that there exists a transition $(s_1, s_2) \in \tau$ from a state $s_1 \in S$ to a state $s_2 \in S$. If $U \subseteq S$, we denote as $\tau(U) = \{s' \in S \mid \exists s \in U, (s, s') \in \tau\}$ the set of successors of U.

A state $s \in S$ is said to be reachable if it can be produced from an initial state in a finite number of computation steps. Thus the set *reach* of reachable states is defined

formally as follows:

$$reach = \bigcup_{k \in \mathbb{N}} \tau^k(I)$$

A state $s \in S$ is reachable if it is connected to an initial state by the transitive closure $\tau^\star$ of the transition relation τ.

Stated differently, a state is reachable if it is either an initial state or the successor of a reachable state by the transition relation τ. We can also define *reach* as the least solution of the fixpoint equation:

$$reach = I \cup \tau(reach)$$

In Section 2.2, we recall some basic notions of fixpoint theory which ensures that such a recursive definition makes sense.

2.1.2 Programs as Control Flow Graphs

We adopt the classical representation of programs by Control Flow Graphs (CFG), in contrast with other works using Abstract Syntax Trees, such as Astrée [19] or Mopsa [101], or directly expressing program semantics as Horn clauses such as SeaHorn [56].

Definition 2.1.1 (Control-Flow Graph)**.** A control-flow graph (or CFG) is a directed graph $G = (N, E)$ such that:

- $N = \{\nu_0, ..., \nu_n\}$ is a finite set of nodes being of one of the following types: start node, junction node or statement node. A partial transfer function $f_\nu : D \to D_\bot$ is associated to each statement node ν, where $D = Vars \to Values$ is the set of variable valuations.
- $E \subseteq N \times N$ is a set of control-flow edges. Start nodes have no incoming edge whereas statement nodes have only one incoming edge and junction nodes can have several incoming edges.

For simplicity, each node has a single output possibly leading to several successor nodes. This means that classical test nodes are transformed into pairs of conditions appearing at the beginning of statement nodes, where the associated transfer functions intersect their argument with the condition of the test (the *then* part) or its negation (the *else* part). Since conditions guard statement nodes, the transfer functions f_ν are partial, returning $\bot$ when their argument does not satisfy the condition.

From CFGs to transition systems

The semantics of a CFG can be expressed as a transition system (S, I, τ) where the set of states is $S = N \times D$ and each state is a pair (ν, d) made of a node and a variable valuation. The set of initial states is $I = N_0 \times D$ where N_0 is the set of start nodes of the CFG. The transition relation τ is defined as:

$$((\nu, d), (\nu', d')) \in \tau \Leftrightarrow (\nu, \nu') \in E \wedge d' = \begin{cases} f_{\nu'}(d) \neq \bot & \text{if } \nu' \text{ is a statement node} \\ d & \text{if } \nu' \text{ is a junction node} \end{cases}$$

```
i := 0;

while i < 100
    i := i + 1;
end;
```

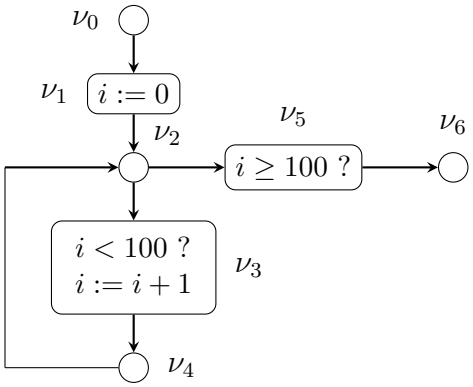


Figure 2.1: Simple program with a while loop incrementing a variable i at each iteration. It is represented by the control flow graph on its right.

2.1.3 Collecting Semantics

Classically, program states are partitioned according to the nodes of the CFG. It is a particular instance of what is called *trace partitioning* [117], see Section 2.5.2. Instead of computing the reachable states of the transition system, we compute the set X_ν of reachable variable valuations at each CFG node ν, defined as:

$$X_\nu = \{d \mid (\nu, d) \in \mathit{reach}\}$$

The sets X_ν satisfy the following system of equations called the *collecting semantics* of the program:

$$X_\nu = \begin{cases} D & \text{if } \nu \text{ is a start node} \\ f_\nu(X_{\nu'}) & \text{if } \nu \text{ is a statement node and } (\nu', \nu) \in E \\ \bigcup\limits_{(\nu', \nu) \in E} X_{\nu'} & \text{if } \nu \text{ is a junction node} \end{cases}$$

We can also write this system of equations as a vectorial fixpoint equation $X = F(X)$ where $X \in D^k$.

Example 2.1.1. We give the collecting semantics of the simple program of Figure 2.1. There is a unique variable i of integer type, the set of valuations is $D = \mathbb{Z}$. The semantic equations are:

$$\begin{array}{lcl} X_0 & = & \mathbb{Z} \\ X_1 & = & X_0[i := 0] \\ X_2 & = & X_1 \cup X_4 \\ X_3 & = & (X_2 \cap (i < 100))[i := i + 1] \\ X_4 & = & X_3 \\ X_5 & = & X_2 \cap (i \geq 100) \\ X_6 & = & X_5 \end{array}$$

From this system of equations, we define a semantic function $F : D^k \to D^k$ as follows:

$$F\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{pmatrix} = \begin{pmatrix} \mathbb{Z} \\ f_{\nu_1}(X_0) \\ X_1 \cup X_4 \\ f_{\nu_3}(X_2) \\ X_3 \\ f_{\nu_5}(X_2) \\ X_5 \end{pmatrix} = \begin{pmatrix} \mathbb{Z} \\ X_0[i := 0] \\ X_1 \cup X_4 \\ (X_2 \cap (i < 100))[i := i + 1] \\ X_3 \\ X_2 \cap (i \geq 100) \\ X_5 \end{pmatrix}$$

The functions $f_{\nu_1} : D \to D$, $f_{\nu_3} : D \to D$ and $f_{\nu_5} : D \to D$ are the transfer functions representing the effect of statement nodes ν_1, ν_3, ν_5. The system of semantic equations is equivalent to the vectorial fixpoint equation $X = F(X)$ with $X \in D^k$.

2.2 Fixpoint Theorems

As most interesting objects in computer science, program semantics are defined by systems of fixpoint equations. The existence of solutions to such equations results from fixpoint theory and order theory. We introduce in this section some basic definitions and notations regarding partially ordered sets and lattices.

2.2.1 Complete Lattices

Definition 2.2.1 (Partially ordered set)**.** A partially ordered set $(D, \sqsubseteq)$ is a non-empty set D and a binary relation $\sqsubseteq$ over D satisfying the following properties:

$$\begin{array}{ll} \forall a, b \in D,\ a \sqsubseteq b & \text{(reflexivity)} \\ \forall a, b, c \in D,\ a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c & \text{(transitivity)} \\ \forall a, b \in D,\ a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b & \text{(antisymmetry)} \end{array}$$

An element $m \in D$ is an upper bound of $S \subseteq D$ if $\forall x \in S, x \sqsubseteq m$. An element $M \in D$ is the *least upper bound* of S, denoted as $\sqcup S$, if $M \sqsubseteq m$ for every upper bound m of S. An element $l \in D$ is a lower bound of $S \subseteq D$ if $\forall x \in S, l \sqsubseteq x$. An element $L \in D$ is the *greatest lower bound* of S, denoted as $\sqcap S$, if $l \sqsubseteq L$ for every lower bound l of S.

Definition 2.2.2 (Lattice)**.** A lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set in which every pair $\{a, b\} \in L$ of elements of L has a least upper bound $a \sqcup b = \sqcup\{a, b\}$ and a greatest lower bound $a \sqcap b = \sqcap\{a, b\}$ in L. The least upper bound $\sqcup : L \times L \to L$ is called the *join* operator of L and $\sqcap : L \times L \to L$ is called the *meet* operator of L.

Definition 2.2.3 (Complete lattice)**.** A lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice if every subset $S \subseteq L$ has a least upper bound $\sqcup S$ and a greatest lower bound $\sqcap S$ in L. The least element of L, called *bottom*, is denoted as $\bot = \sqcap L$ and the greatest element of L, called *top*, is denoted as $\top = \sqcup L$.

Example 2.2.1. For any set X, the set 2^X of all subsets of X, also called the powerset of X, is a complete lattice $(2^X, \subseteq, \cup, \cap, \emptyset, X)$ for inclusion. For example, the powerset $2^{\mathbb{Z}}$ of $\mathbb{Z}$ is a complete lattice $(2^{\mathbb{Z}}, \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$.

2.2.2 Monotonic Functions and Tarski's Fixpoint Theorem

Definition 2.2.4 (Monotonic function)**.** Let $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ be two partially ordered sets. A function $F : D_1 \to D_2$ from D_1 to D_2 is monotonic if it satisfies the following property:

$$\forall a, b \in D_1, a \sqsubseteq_1 b \Rightarrow F(a) \sqsubseteq_2 F(b)$$

Definition 2.2.5 (Fixpoint)**.** Let $(D, \sqsubseteq)$ be a partially ordered set. Let $F : D \to D$ be a function on D. An element $x \in D$ is a *pre-fixpoint* of F if $x \sqsubseteq F(x)$. An element $x \in D$ is a *post-fixpoint* of F if $F(x) \sqsubseteq x$. An element $x \in D$ is a *fixpoint* of F if $F(x) = x$. We denote as $\mathrm{lfp}(F)$ the least fixpoint of F and as $\mathrm{gfp}(F)$ the greatest fixpoint of F if they exist.

Tarski's fixpoint theorem ensures the existence of fixpoints for a monotonic function on a complete lattice, such that the set $\mathrm{fix}(F)$ of such fixpoints is itself a complete lattice. The set $\mathrm{fix}(F)$ being a complete lattice, it has a least element and a greatest element. Thus a monotonic function F on a complete lattice L has always a least fixpoint $\mathrm{lfp}(F)$ and a greatest fixpoint $\mathrm{gfp}(F)$ in L.

Theorem 2.2.1 (Tarski's fixpoint theorem)**.** Let $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete non-empty lattice and $F : L \to L$ be a monotonic function on L. The set $\mathrm{fix}(F)$ of fixpoints of F is a non-empty complete lattice:

$$\mathrm{lfp}(F) = \sqcap\{x \in L \mid F(x) \sqsubseteq x\} \qquad \mathrm{gfp}(F) = \sqcup\{x \in L \mid x \sqsubseteq F(x)\}$$

2.2.3 Continuous Functions and Kleene's Theorem

Tarski's fixpoint theorem ensures the existence of fixpoints for monotonic functions on complete lattices. In practice, we use Kleene's fixpoint theorem for continuous functions which gives a method to compute fixpoints using Kleene iterations.

Definition 2.2.6 (Continuous function)**.** Let $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. A function $f : D \to D$ is Scott-continuous if it preserves least upper bounds:

$$\forall X \subseteq D, f(\sqcup X) = \sqcup f(X)$$

Remark: a continuous function on a complete lattice is necessarily monotonic.

Theorem 2.2.2 (Kleene's fixpoint theorem)**.** Let $F : D \to D$ be a continuous function on a complete lattice $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The extreme fixpoints of F exist:

$$\mathrm{lfp}(F) = \bigsqcup_{n \geq 0} F^n(\bot) \quad \mathrm{gfp}(F) = \bigsqcap_{n \geq 0} F^n(\top)$$

Example 2.1.1 (continued) We apply Kleene's fixpoint theorem to our example by trying to compute the least solution of the system of semantic equations:

$$\begin{array}{lcl} X_0 & = & \mathbb{Z} \\ X_1 & = & X_0[i := 0] \\ X_2 & = & X_1 \cup X_4 \\ X_3 & = & (X_2 \cap (i < 100))[i := i + 1] \\ X_4 & = & X_3 \\ X_5 & = & X_2 \cap (i \geq 100) \\ X_6 & = & X_5 \end{array}$$

The solution is be computed in the complete lattice $(2^{\mathbb{Z}}, \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$ of possible valuations for i. According to Kleene's theorem, we start from $(X_k = \emptyset)_{k=0..6}$. At the first iteration, we get:

$$X_0 = \mathbb{Z} \quad X_1 = \{0\} \quad X_2 = \{0\} \quad X_3 = \{1\} \quad X_4 = \{1\} \quad X_5 = \emptyset \quad X_6 = \emptyset$$

We can compute some more steps:

$$\begin{array}{lllllll} X_0 = \mathbb{Z} & X_1 = \{0\} & X_2 = \{0,1\} & X_3 = \{1,2\} & X_4 = \{1,2\} & X_5 = \emptyset & X_6 = \emptyset \\ X_0 = \mathbb{Z} & X_1 = \{0\} & X_2 = \{0,1,2\} & X_3 = \{1,2,3\} & X_4 = \{1,2,3\} & X_5 = \emptyset & X_6 = \emptyset \\ X_0 = \mathbb{Z} & X_1 = \{0\} & X_2 = \{0,1,2,3\} & X_3 = \{1,2,3,4\} & X_4 = \{1,2,3,4\} & X_5 = \emptyset & X_6 = \emptyset \end{array}$$

We can foresee that this kind of computation does not terminate in general for arbitrary sets of integers. Also, it is well-known that arbitrary sets of integers are not machine-representable. Thus it entails two main necessities: the use of a decidable, machine-representable abstraction of sets of states and a way to enforce the convergence of fixpoint computations. This is the main goal of abstract interpretation.

2.3 Concrete and Abstract Domains

For the reachable states of a program, the most precise property of interest is the least fixpoint of collecting semantics, which is not computable in general or would result in computations over a very complex domain which would track precisely all the aspects of a program implementation, such as the state of registers, the stack and the heap. Such a domain is called a *concrete domain*. The concrete semantics can be approximated by an abstract semantics with values belonging to an *abstract domain*. The elements of the concrete domain are said to be approximated by the elements of the abstract domain.

2.3.1 Abstract Domains

Definition 2.3.1 (Galois Connection)**.** Let $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ be two partially ordered sets. A Galois connection from D_1 to D_2 is a pair (α, γ) of functions such that $\alpha : D_1 \to D_2$ is called the *abstraction* function and $\gamma : D_1 \to D_1$ is called the *concretization* function satisfying the following property:

$$\alpha(d_1) \sqsubseteq_2 d_2 \Leftrightarrow d_1 \sqsubseteq_1 \gamma(d_2)$$

Definition 2.3.2 (Abstract Domain)**.** Let $(C, \sqsubseteq_C, \sqcup_C, \sqcap_C)$ be a lattice of elements to be abstracted considered as the concrete domain. An abstract domain is a lattice $(D, \sqsubseteq, \sqcup, \sqcap)$ related to C by a Galois connection (α, γ) with $\alpha : C \to D$ and $\gamma : D \to C$.

Example 2.3.1 (Interval abstract domain)**.** In Example 2.1.1, the concrete domain is the complete lattice $(2^{\mathbb{Z}}, \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$. A possible abstract domain is the complete lattice of intervals of integers $(\mathcal{I}, \sqsubseteq, \sqcup, \sqcap, \bot, (-\infty, +\infty))$ defined as:

$$\begin{array}{l} \mathcal{I} = (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\}) \\ \forall (a,b), (c,d) \in \mathcal{I}, \\ \qquad (a,b) \sqsubseteq (c,d) \;\Leftrightarrow\; c \leq a \wedge b \leq d \\ \qquad (a,b) \sqcup (c,d) \;=\; (\mathit{inf}(a,c), \mathit{sup}(b,d)) \\ \qquad (a,b) \sqcap (c,d) \;=\; (\mathit{sup}(a,c), \mathit{inf}(b,d)) \end{array}$$

We assume that $(a, b) = \emptyset$ for $a > b$. The Galois connection (α, γ) between $2^{\mathbb{Z}}$ and $\mathcal{I}$ is:

$$\forall X \subseteq \mathbb{Z},\ \alpha(X) = \begin{cases} (\mathit{inf}(X), \mathit{sup}(X)) & \text{if } X \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

$$\forall (a,b) \in \mathcal{I}, \gamma((a,b)) = \begin{cases} [a,b] & \text{if } a \leq b \\ \emptyset & \text{otherwise} \end{cases}$$

2.3.2 Fixpoint Computation

The set of reachable states is the least fixpoint of a function F over a concrete domain D. The elements of the concrete domain may not be representable by machine and the concrete function F may not be computable. Instead, we compute the least fixpoint of a sound approximation $F^\sharp : D^\sharp \to D^\sharp$ of F over an abstract domain $D^\sharp$.

Definition 2.3.3 (Sound approximation)**.** Let $(D, \sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D)$ and $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be complete lattices related by a Galois connection (α, γ). A function $F^\sharp : D^\sharp \to D^\sharp$ on $D^\sharp$ is a sound approximation of a concrete function $F : D \to D$ if it satisfies the following property:

$$\forall d^\sharp \in D^\sharp, \quad F(\gamma(d^\sharp)) \sqsubseteq_D \gamma(F^\sharp(d^\sharp))$$

The fixpoint transfer theorem guarantees that the least fixpoint of $F^\sharp$ over the abstract domain $D^\sharp$ is a sound approximation of the least fixpoint of F over the concrete domain D.

Theorem 2.3.1 (Fixpoint Transfer Theorem)**.** Let $(D, \sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D)$ and $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be complete lattices related by a Galois connection (α, γ). Let $F : D \to D$ be a monotonic function on D and let $F^\sharp : D^\sharp \to D^\sharp$ be a monotonic function on $D^\sharp$ such that $F^\sharp$ is a sound approximation of F. The least fixpoint $\mathrm{lfp}(F)$ of F and the least fixpoint $\mathrm{lfp}(F^\sharp)$ of $F^\sharp$ are related as follows:

$$\mathrm{lfp}(F) \sqsubseteq_D \gamma(\mathrm{lfp}(F^\sharp))$$

If $F^\sharp$ is continuous, it follows from Kleene's Fixpoint Theorem that the least fixpoint $\mathrm{lfp}(F^\sharp)$ of $F^\sharp$ is the limit of the Kleene's sequence $(x_i)^\sharp_{i \geq 0}$ defined as:

$$\begin{array}{lcl} x^\sharp_0 & = & \bot \\ x^\sharp_{i+1} & = & F^\sharp(x^\sharp_i) \end{array}$$

Example 2.1.1 (continued) We want to compute a solution of the system of equations of our program, in the complete lattice of intervals of integers. The system of abstract semantic equations over $\mathcal{I}$ is defined as:

$$\begin{array}{lcl} X_0 & = & \top \\ X_1 & = & X_0[i := 0] \\ X_2 & = & X_1 \sqcup X_4 \\ X_3 & = & (X_2 \sqcap (i < 100))[i := i + 1] \\ X_4 & = & X_3 \\ X_5 & = & X_2 \sqcap (i \geq 100) \\ X_6 & = & X_5 \end{array}$$

We compute a Kleene's sequence starting with $(X_k = \bot)_{k=0..6}$. We get:

$$X_0 = (-\infty, +\infty)\ X_1 = (0,0)\ X_2 = (0,0)\ X_3 = (1,1)\ X_4 = (1,1)\ X_5 = \bot\ X_6 = \bot$$

We can compute some more steps:

$$\begin{array}{lllllll} X_0 = (-\infty, +\infty) & X_1 = (0,0) & X_2 = (0,1) & X_3 = (1,2) & X_4 = (1,2) & X_5 = \bot & X_6 = \bot \\ X_0 = (-\infty, +\infty) & X_1 = (0,0) & X_2 = (0,2) & X_3 = (1,3) & X_4 = (1,3) & X_5 = \bot & X_6 = \bot \\ X_0 = (-\infty, +\infty) & X_1 = (0,0) & X_2 = (0,3) & X_3 = (1,4) & X_4 = (1,4) & X_5 = \bot & X_6 = \bot \end{array}$$

A Kleene's sequence in the lattice of intervals of integers does not terminate in general. The Kleene's fixpoint theorem gives a method to compute least fixpoints of a continuous function over an abstract domain, by recasting them as limits of possibly infinite iterations. However, we have no way to compute limits of infinite iterations in general.

2.3.3 Convergence

A Kleene's sequence is trivially converging when the abstract domain is a finite lattice, such as the lattice of signs or the lattice $\mathbb{Z}_k$ of integers representable on k bits, or if it does not contain any infinite strictly increasing sequence. This is not the case for many interesting abstract domains such as the lattice $\mathcal{I}$ of intervals of integers. The convergence of a Kleene's sequence can not be guaranteed in general. A solution is to extrapolate the limit of a Kleene's sequence using a *widening operator*, which guarantees a finite computation converging to an upper approximation of the exact limit of the sequence.

Definition 2.3.4 (Widening operator)**.** Let $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. A widening operator is a binary operator $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ satisfying the following properties:

1. $\forall x, y \in D^\sharp, x \sqcup y \sqsubseteq x \nabla y$
2. For every increasing sequence $(x_k)_{k\geq 0}$, the increasing sequence $(y_k)_{k\geq 0}$ defined as follows:

   $$y_0 = x_0 \quad y_{k+1} = y_k \nabla x_k$$

   is ultimately stationary: $\exists \ell \in \mathbb{N}, \forall k \geq \ell, y_{k+1} = y_k$.

For a discussion on the design of widening operators, see [38] and [33].

Theorem 2.3.2 (Fixpoint extrapolation)**.** Let $F^\sharp : D^\sharp \to D^\sharp$ be a continuous function on an abstract domain $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. Let $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ be a widening operator on $D^\sharp$. The increasing sequence $(X_k)_{i\geq 0}$ defined as:

$$\begin{cases} X_0 &= \bot \\ X_{k+1} &= X_k \nabla F^\sharp(X_k) \end{cases}$$

is ultimately stationary: $\exists \ell \in \mathbb{N}, \forall k \geq \ell, X_{k+1} = X_k$. The limit X^∇ of the increasing sequence $(X_k)_{k\geq 0}$ is a post-fixpoint of $F^\sharp$ and thus a sound approximation of the least fixpoint $\mathrm{lfp}(F^\sharp)$ with $\mathrm{lfp}(F^\sharp) \sqsubseteq X^\nabla$.

When X^∇ is not a fixpoint of $F^\sharp$, the solution given by the limit X^∇ of the increasing sequence can be improved by computing a *decreasing sequence* using a narrowing operator.

Definition 2.3.5 (Narrowing Operator)**.** Let $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. A narrowing operator is a binary operator $\triangle : D^\sharp \times D^\sharp \to D^\sharp$ satisfying the following properties:

1. $\forall x, y \in D^\sharp, y \sqsubseteq x \triangle y \sqsubseteq x$
2. For every decreasing sequence $(t_k)_{k\geq 0}$, the decreasing sequence $(z_k)_{k\geq 0}$ defined as:

$$z_0 = t_0 \qquad z_{k+1} = z_k \triangle t_k$$

is ultimately stationary: $\exists \ell \in \mathbb{N}, \forall k \geq \ell, z_{k+1} = z_k$.

The decreasing sequence $(Y_k)_{k\geq 0}$ starting with the limit X^∇ of the increasing sequence is defined as:

$$\begin{cases} Y_0 &= X^\nabla \\ Y_{k+1} &= Y_k \triangle F^\sharp(Y_k) \end{cases}$$

Every term of the decreasing sequence is a post-fixpoint of $F^\sharp$ and a sound approximation of the least fixpoint. We can stop the decreasing sequence arbitrarily after any number of steps, without waiting for convergence, and still get a correct approximation of $\mathrm{lfp}(F^\sharp)$. In practice, we can compute a decreasing sequence without using any narrowing operator, by just applying $F^\sharp$ for a finite number of terms as follows:

$$Y_0 = X^\nabla \qquad Y_{k+1} = F^\sharp(Y_k)$$

We denote as $X^{\nabla\triangle}$ the limit of the decreasing sequence. It is decreasing since $F^\sharp$ is monotonic and its terms are post-fixpoints of $F^\sharp$.

Example 2.3.2 (Widening and narrowing on intervals)**.** The following operator $\nabla : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ on intervals of integers is a widening operator:

$$\forall (a,b),(c,d) \in \mathcal{I}, (a,b)\nabla(c,d) = \left(\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right)$$

The following operator $\triangle : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ on intervals of integers is a narrowing operator:

$$\forall (a,b),(c,d) \in \mathcal{I}, (a,b)\triangle(c,d) = \left(\begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & \text{if } b = +\infty \\ b & \text{otherwise} \end{cases} \right)$$

Example 2.1.1 (continued) We compute an approximation of the sets of reachable valuations of i in the interval abstract domain $\mathcal{I}$. We define a system of abstract fixpoint equations where the widening operator is applied on X_2:

$$\begin{array}{lcl} X_0 & = & \top \\ X_1 & = & X_0[i := 0] \\ X_2 & = & X_2 \nabla (X_1 \sqcup X_4) \\ X_3 & = & (X_2 \sqcap (i < 100))[i := i + 1] \\ X_4 & = & X_3 \\ X_5 & = & X_2 \sqcap (i \geq 100) \\ X_6 & = & X_5 \end{array}$$

We compute an increasing sequence as follows:

X_0	X_1	X_2	X_3	X_4	X_5	X_6
$[-\infty, +\infty]$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$[-\infty, +\infty]$	$[0, 0]$	$[0, 0]$	$[1, 1]$	$[1, 1]$	$\bot$	$\bot$
$[-\infty, +\infty]$	$[0, 0]$	$[0, +\infty]$	$[1, 100]$	$[1, 100]$	$[100, +\infty]$	$[100, +\infty]$

The increasing sequence has converged. It gives the approximation $[0, +\infty]$ for the set of reachable valuations of i at the head of the while loop at node 2. This post-fixpoint can be improved by computing a decreasing sequence:

X_0	X_1	X_2	X_3	X_4	X_5	X_6
$[-\infty, +\infty]$	$[0, 0]$	$[0, +\infty]$	$[1, 100]$	$[1, 100]$	$[100, +\infty]$	$[100, +\infty]$
$[-\infty, +\infty]$	$[0, 0]$	$[0, 100]$	$[1, 100]$	$[1, 100]$	$[100, 100]$	$[100, 100]$

The decreasing sequence converges in a single iteration and gives the invariant $i \in [0, 100]$ at the head of the while loop.

Choice of Widening Nodes

The increasing and the decreasing sequence are computed in a chaotic way, by propagating abstract values along the paths of the CFG. Since the widening operator loses information, it is only applied to a selected set $\mathcal{W}$ of widening nodes cutting each loop of the CFG. In example 2.1.1, widening was applied on X_2 which is associated to the head ν_2 of the while loop.

Since finding a minimal cutting set $\mathcal{W}$ is an NP-complete problem, the set of widening nodes is classically chosen heuristically using the method of *strongly connected subcomponents* proposed by Bourdoncle [23, 24]. The method uses recursively Tarjan's algorithm [123] to find the strongly connected components (SCC) of a directed graph and their entry nodes. The entry nodes are the targets of back edges, thus they are all junction nodes. Bourdoncle's method adds the entry node of each SCC to $\mathcal{W}$, then removes them from the graph and recursively applies Tarjan's algorithm to the rest of each SCC. The result is a hierarchy of strongly connected subcomponents, each of which is cut by a junction node in $\mathcal{W}$.

2.4 Numerical Abstract Domains

We present in this section some numerical abstract domains abstracting subsets of $\mathbb{K}^n$ for $\mathbb{K} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$.

2.4.1 Affine Equalities

An affine relationship is an equality of the form $\sum a_j x_j = b_i$ between a linear combination of numerical variables x_j and a constant b_i. M. Karr [76] introduced the abstract domain of affine equalities to discover all valid affine equalities among the variables of a program. The domain representation and its operations are based on linear algebra.

A set of affine equalities over n variables determines an affine subspace and can be represented by an $m \times n$ matrix M, where $m \leq n$, along with an m-dimensional column vector C. Each row of M and C represents an affine equality. The pair (M, C) is called the constraint representation of the set of affine equalities. The domain of affine equalities is a relational domain of finite height, which does not contain infinite strictly increasing sequences.

2.4.2 Zones

Zones are subsets of $\mathbb{K}^n$ represented by sets of constraints of the form $x_i - x_j \leq c$ or $\pm x_i \leq c$ where $c \in \mathbb{K}$ is a constant and x_i, x_j are numerical variables. Zones were proposed in [44, 127] for the model-checking of timed automata and were later adapted to abstract interpretation [96].

Representation

A zone is represented by a *potential graph* with nodes labeled by numerical variables. There is an edge from a node labeled by a variable x_i to a node labeled by a variable x_j with weight $c \in \mathbb{K}$ if there exists a constraint $x_i - x_j \leq c$. Unary constraints of the form $x_i \leq c$ are rewritten as binary constraints $x_i - x_0 \leq c$ with a phantom variable x_0 which is always equal to the constant zero. A potential graph is represented classically by its adjacency matrix called a *Difference Bound Matrix* (DBM).

Operations on Zones

A DBM is normalized by computing its transitive closure by the Floyd-Warshall algorithm. The equality or inclusion of two zones is tested by first normalizing their respective DBMs and then comparing the values of the normalized DBMs.

Zones are closed under union and the least upper bound of two DBMs is obtained by merging the two graphs and normalizing the resulting adjacency matrix. The greatest lower bound of two DBMs m_1 and m_2 is computed by taking, for each pair (x_i, x_j) of variables, the minimum of the weights associated to the edge (x_i, x_j) in m_1 and in m_2. It gives the smallest constant c such that $x_i - x_j \leq c$.

The widening $m \nabla n$ of two DBMs m and n is defined as:

$$(m \nabla n)_{i,j} = \begin{cases} m_{i,j} & \text{if } n_{i,j} \leq m_{i,j} \\ +\infty & \text{otherwise} \end{cases}$$

The narrowing $m \triangle n$ of two DBMs m and n is defined as:

$$(m \triangle n)_{i,j} = \begin{cases} n_{i,j} & \text{if } n_{i,j} = +\infty \\ m_{i,j} & \text{otherwise} \end{cases}$$

Operations on DBMs have a $\mathcal{O}(n^3)$ complexity in the number of numerical variables due to the transitive closure algorithm used for normalization.

2.4.3 Octagons

The octagons abstract domain presented in [97, 100] is an extension of zones to sets of constraints of the form $\pm x_i \pm x_j \leq c$ where x_i, x_j are numerical variables and $c \in \mathbb{K}$ is a constant.

Representation

We consider that variables come in two flavors, each variable x_i comes in a positive form x_i^+ and in a negative form x_i^-. A constraint of the form $x_i + x_j \leq c$ can be represented by the pair of potential constraints $x_i^+ - x_j^- \leq c$ and $x_j^+ - x_i^- \leq c$ involving the positive and the negative forms of the original variables x_i and x_j. More generally, any set of constraints of the form $\pm x_i \pm x_j \leq c$ can be represented by a DBM using the following translation scheme:

Octagon constraint	Potential constraints
$x_i - x_j \leq c$	$x_i^+ - x_j^+ \leq c, \quad x_j^- - x_i^- \leq c$
$-x_i - x_j \leq c$	$x_i^- - x_j^+ \leq c, \quad x_j^- - x_i^+ \leq c$
$x_i + x_j \leq c$	$x_i^+ - x_j^- \leq c, \quad x_j^+ - x_i^- \leq c$
$x_i \leq c$	$x_i^+ - x_i^- \leq 2c$
$x_i \geq c$	$x_i^- - x_i^+ \leq -2c$

Operations

The DBM representation of octagons is normalized using a variant of the Floyd-Warshall algorithm called the *strong closure algorithm* [98].

In a similar way to zones, the emptiness of an octagon is tested as a side-product of the strong closure algorithm. An octagon represented by a normalized DBM m is not empty if there exists variables x_i, x_j such that $m_{i,j} \neq 0$. The inclusion of two octagons is tested by comparing the values in their respective normalized DBMs.

The intersection of two octagons is computed using the intersection of their DBMs. The least upper bound of two octagons is the least upper bound of their normalized DBMs.

The matrix-based representation of zones and octagons using DBMs is amenable to fast parallel implementations. An implementation of the octagon abstract domain on GPUs (Graphics Processing Unit) is described in [11].

Complexity and Expressivity

Zones and octagons are able to represent sets of constraints of the form $x_i - x_j \leq c$ for zones and $\pm x_i - \pm x_j \leq c$ for octagons. Operations have a $\mathcal{O}(n^3)$ time complexity in the

worst-case as they are based on the Floyd-Warshall algorithm. These domains are not able to represent arbitrary linear relations of the form $\sum a_j x_j \geq b_i$ between numerical variables, whereas linear relations can be expressed in their full generality by the more costly abstract domain of convex polyhedra.

2.4.4 Convex Polyhedra

The convex polyhedra abstract domain was proposed in [41, 60] to discover sets of linear relations of the form $\sum a_j x_j \geq b_i$ between program variables where the x_j are numerical variables and the a_j and b_i are constants. The set of solutions of a system of linear relations is a convex polyhedron.

Representations of Convex Polyhedra

A convex polyhedron P can be represented either as a set of linear constraints:

$$P = \{x = (x_1, ..., x_n) \mid \bigwedge_{1 \leq i \leq m} \sum_{1 \leq j \leq n} a_j x_j \geq b_i\} = \{x \in \mathbb{Z}^n \mid Ax \geq B\}$$

or as a system of generators:

$$P = \{\sum_{s_i \in V} \lambda_i s_i + \sum_{r_j \in R} \mu_j r_j \mid \lambda_i \geq 0 \wedge \sum \lambda_i = 1 \wedge \mu_j \geq 0\}$$

where V is a finite set of vertices and R is a finite set of rays. The pair (A, B) is called a *system of constraints* of P and (V, R) is called a *system of generators* of P.

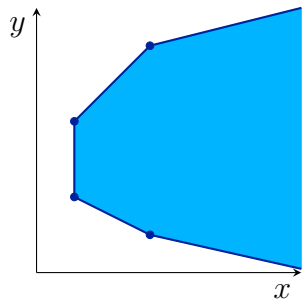


Figure 2.2: Unbounded convex polyhedron representing the set of constraints $x + 4y \geq 7$, $x + 2y \geq 5$, $x \geq 1$, $y - x \leq 3$, $4y - x \leq 21$ and the system of generators with the set of vertices $V = \{(3, 1), (1, 2), (1, 4), (3, 6)\}$ and the set of rays $R = \{(4, 1), (4, -1)\}$.

The system of constraints (A, B) represents the polyhedron P by giving the constraints satisfied by the points included in P. The system of constraints is equivalently given in matrix form as $AX \geq B$ where X is an n-row vector of variables $x_1, .., x_n$.

The system of generators (V, R) represents a convex polyhedron P as the set of all points which are sums of a convex combination $\sum_{s_i \in V} \lambda_i s_i$ of vertices in S and of a positive combination $\sum_{r_j \in R} \mu_j r_j$ of rays in R.

A generator G *satisfies* a linear constraint $A_i X \geq B_i$ if $A_i \cdot G \geq B_i$. A system of generators (V, R) satisfies a system of constraints (A, B) if all vertices in V and rays in R

satisfy the constraints $AX \geq B$. A generator G *saturates* a linear constraint $A_i X \geq B_i$ if $A_i \cdot G = B_i$.

The lattice of convex polyhedra is not complete since the convex hull of an infinite set of polyhedra is not necessarily a polyhedron, as it can be, for example, a disk. However, it is a sublattice of the complete lattice of convex sets. An analysis using convex polyhedra involves only a finite number of operations, due to the use of widening, thus it is guaranteed to stay within the sublattice of convex polyhedra.

Operations on Convex Polyhedra

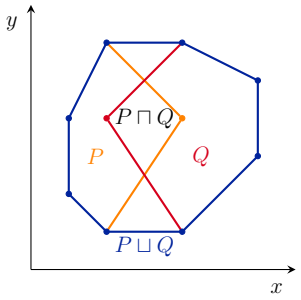


Figure 2.3: Intersection and convex hull of convex polyhedra.

Conversion of Representations

A system of constraints or a system of generators is said to be minimal if no constraint or generator can be removed without changing the represented set of points. Chernikova's algorithm [31], later improved by Le Verge [81], constructs the minimal system of generators equivalent to a given system of constraints and conversely.

Intersection

The intersection of two convex polyhedra P and Q is the convex polyhedron $P \sqcap Q$ computed simply as the conjunction of the sets of constraints of P and Q.

Convex Hull

The union of two convex polyhedra is not convex in general. Instead, we compute the convex hull of convex polyhedra, which is the smallest convex polyhedron containing both polyhedra. The convex hull of two convex polyhedra P and Q is denoted as $P \sqcup Q$. Every point in the convex hull $P \sqcup Q$ is a convex combination of the generators of P and Q. The convex hull is computed by joining together the generators of P and Q. This can generate redundant generators and a pass of Chernikova's algorithm should be applied to obtain a minimal representation of the result.

Inclusion and Equality

The inclusion of two convex polyhedra P and Q is tested by checking if the system of generators (S, R) of P satisfies the system of constraints of Q. The equality of two convex

polyhedra P and Q is checked by testing the double inclusion of P in Q and symmetrically of Q in P.

Emptiness Test

An empty convex polyhedron has an empty system of generators $(\emptyset, \emptyset)$ and infinitely many constraint representations. The emptiness of a convex polyhedron P can be tested by checking the emptiness of a minimal generator representation of P.

Widening

The classical widening operator on convex polyhedra was introduced in [60]. The widening $P \nabla Q$ of a convex polyhedron P by a convex polyhedron Q is defined *roughly* as removing the constraints of P which are not satisfied by Q. The result is an extrapolation of both P and Q as it contains both polyhedra.

More precisely, the widening $P \nabla Q$ is defined as the set of constraints of P which are satisfied by Q and the constraints of Q which are mutually redundant with constraints of P. Two constraints are *mutually redundant* if they saturate the same set of generators, either vertices or rays.

When the dimension of P is less than the dimension n of the space, the constraints of P are rewritten to maximize the number of constraints of Q which are mutually redundant with constraints of P. The aim is to maximize the number of constraints of Q which are kept in $P \nabla Q$.

Example 2.4.1. We give a simple example of widening over convex polyhedra. Let $P = (i \geq 0 \wedge i \leq 0 \wedge n \geq 0)$ and $Q = (i \geq 0 \wedge i \leq 1 \wedge i \leq n)$. The widening of P by Q is $P \nabla Q = (i \geq 0 \wedge i \leq n)$.

Many proposals concerning the improvement of widening operators have been made, especially for the convex polyhedra abstract domain [9].

Widening with thresholds [61, 67, 18] consists in choosing constraints, from conditions appearing in the program or some propagation of them [79], as candidate limits to the widening, which has the effect of relaxing bounds of variables more gradually instead of moving them to infinity. Widening with landmarks follows a similar idea where the selection of constraints is made dynamically [120]. Widening with a care set [125] and interpolated widening [54] make use of a proof objective.

Another improvement of widening is limited widening, defined as the intersection $(P \nabla Q) \sqcap A$ of the classical widening $P \nabla Q$ by a limiting abstract value A. For limited widening to be sound, the limiting value A must be an invariant at the node where the limited widening is applied, such as, for example, a precondition of the procedure. An increasing sequence using limited widening is not guaranteed to be converging in general. For the convex polyhedra abstract domain, we can make the same argument as for the termination of the classical widening, that either the number of constraints in the result decreases or the dimension of the resulting convex polyhedron increases.

Complexity

Each operation on convex polyhedra have a preferred representation. Some operations need either the system of constraints, such as intersection, or the system of generators, for

the convex hull, or both for the inclusion test. Converting one representation to the other can be costly, producing a representation which is $\mathcal{O}(2^n)$ in size with respect to the original representation. This is particularly the case for an axis-aligned n-dimensional hypercube, where n variables have bounds with no relations between the variables, represented by $2n$ constraints and a system of generators with 2^n vertices. In order to alleviate the costs due to the double representation of convex polyhedra, a *constraint-only* representation was proposed in [14, 48, 3, 92, 93].

2.5 Advanced techniques

In this section, we present some advanced techniques to improve the precision of properties computed using abstract interpretation.

2.5.1 Guided Static Analysis

Guided static analysis [52, 53] was proposed to improve the precision of properties of loops with multiple phases having very different behaviors which can depend on non-deterministic choices.

We remember the set of program points which are reachable before the first application of the widening operator during the increasing sequence. The widening operator is applied and the increasing sequence is then computed only on the restricted program consisting of the reachable program points before widening. When the increasing sequence has converged, the decreasing sequence is computed on the program restriction to improve the result of the increasing sequence.

A new program restriction is obtained at the end of the decreasing sequence, by taking into account the program locations which are now reachable and which were previously ignored. An increasing sequence, followed by a decreasing sequence, is restarted on the newly derived program restriction.

More and more program points are added to the program restriction in a succession of increasing and decreasing sequences, until the entire program is considered. A guided static analysis is converging since there is only a finite number of program locations.

Intuitively, the guided static analysis approach discovers distinct loop phases and analyzes them successively, avoiding the loss of precision occurring in a standard analysis due to the simultaneous consideration of possibly very different behaviors.

Example 2.5.1. We consider the program shown in Figure 2.4. A standard Linear Relation Analysis applies widening at node ν_1 and gives the following abstract value for ν_1 after one iteration of the increasing sequence:

$$x + y = 50 \wedge x \geq 0$$

The node ν_3 becomes reachable and is taken into account by the analysis, causing a tremendous loss in precision at the join node ν_4. When the analysis has converged, the loop invariant obtained at ν_1 is:

$$x + y \leq 50 \wedge y \geq 1$$

```
x := 0;
y := 50;

while true
    if x < 50 then
        x := x + 1;
        y := y - 1;
    else
        x := x - 1;
        y := y - 1;
    end;

    if y <= 0 then
        break;
    end;
end;
```

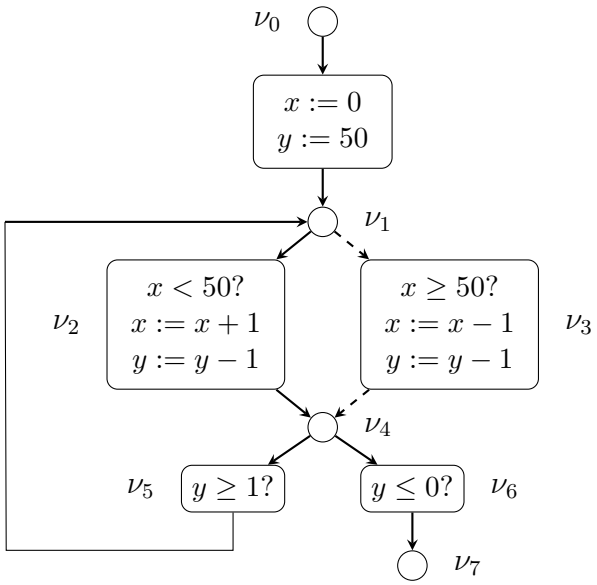


Figure 2.4: A loop with two phases.

The abstract value obtained after the while loop at ν_7 is:

$$y = 0 \wedge x \leq 50$$

A guided static analysis first computes an increasing sequence followed by a decreasing sequence in which the dashed program edges are ignored. A more precise loop invariant is obtained at node ν_1:

$$x + y = 50 \wedge 0 \leq x \leq 49$$

The node ν_3 remains unreachable when analyzing the entire program. We obtain the following property at ν_7 after the while loop:

$$x = 50 \wedge y = 0$$

We discovered the equality constraint $x = 50$ at node ν_7 with the guided static analysis whereas the classical Linear Relation Analysis only found the inequality constraint $x \leq 50$.

2.5.2 Trace Partitioning

A way to avoid the loss of precision occurring at join nodes is to keep separate the abstract values obtained at conditional branches. Trace partitioning [95, 117] was proposed to abstract sets of program traces.

The idea is to duplicate abstract states and to keep them separate by associating an abstract value to the set of program edges that a trace goes through to obtain that value. Each time such a disjunction is introduced, the size of the abstract graph is doubled.

Heuristics are given to choose dynamically the nodes where the splitting of abstract states is applied and when nodes in the graph of abstract states can be merged.

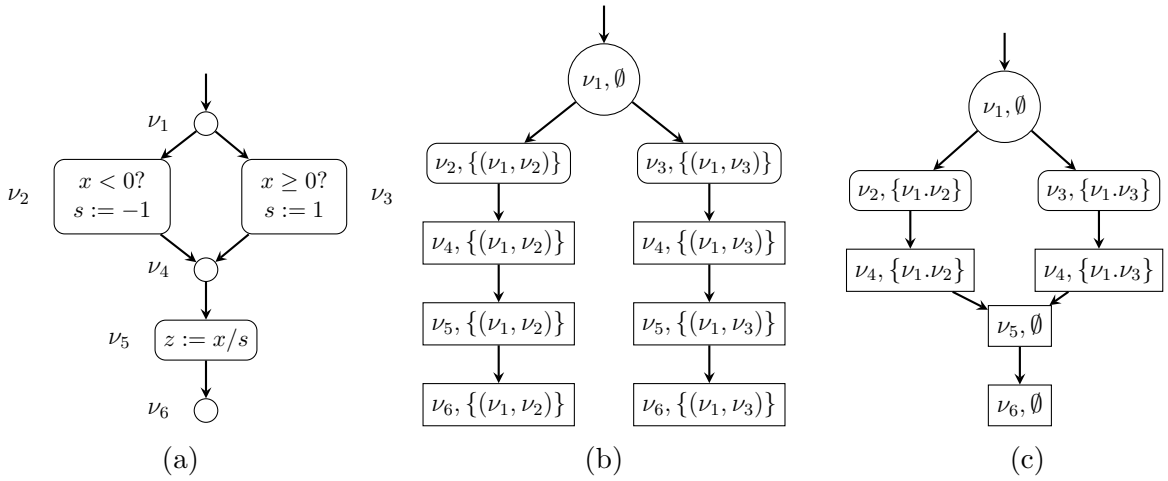


Figure 2.5: An example of trace partitioning to prove the absence of division by zero at node ν_5.

Example 2.5.2. We consider in this example the program fragment presented in Figure 2.5a which computes the absolute value of a variable x. We are interested in proving the absence of division by zero at node ν_5. We can prove the property by distinguishing all paths such as in Figure 2.5b, since we have $s = -1$ at node $\nu_5, \{(v_1, v_2)\}$ and $s = 1$ at node $\nu_5, \{(v_1, v_3)\}$. The property can still be proven in Figure 2.5c if we merge the paths at nodes ν_5 and ν_6.

Chapter 3

Interprocedural Analysis: A State of the Art

3.1 The Origins of Interprocedural Analysis

Interprocedural analysis originated in side-effects analysis for compiler optimization from works of Spillman [122], Allen [4] and Barth [12]. Frances E. Allen coined the term of *interprocedural analysis* in 1974 and introduced some of the foundational concepts of interprocedural analysis in [4]. At that time, early works on global dataflow analysis [78] were considering whole programs, represented by a single global control-flow graph, without any particular focus on the procedural structure of programs.

Programs as Collections of Procedures

Programs can be seen as collections of procedures $K = \{p_1, .., p_n\}$ possibly referencing each other. The references between the procedures can be expressed as a directed graph $\mathcal{C} = (N, E)$ of nodes $n_i \in N$ and edges $(e_i, e_j) \in E$ as follows:

1. Each node n_i represents a procedure p_i and the set of nodes N is in a one-to-one correspondence with K.
2. Each edge $(e_i, e_j) \in E$ represents one or more references from the procedure p_i to the procedure p_j.

Such a graph $\mathcal{C}$ is termed the *call graph* of the program.

Cycles in the call graph indicate the presence of recursive procedures. A self-loop in the call graph denote a simple recursive procedure referencing itself, whereas a non-trivial cycle denote a set of mutually recursive procedures.

A procedure p_i can reference a procedure p_j by a call statement to p_j. Other kinds of procedure referencing are possible in languages allowing the explicit manipulation of references or pointers to procedures, such as function pointers in C-based languages.

We should note that a call graph is not a control flow graph and does not represent returns from procedure calls.

Frances Allen gives in [4] a classical algorithm for the construction of the call graph of a collection of procedures.

Completeness of a Collection of Procedures

Incomplete programs are programs where the implementation is only partially known statically, prior to their concrete execution. It encompasses programs with dynamically-linked libraries, which can be potentially different for each concrete execution platform, and programs with reflective features, capable of modifying their own structure at runtime, such as dynamic code generation in Lisp dialects or Javascript.

Definition 3.1.1 (Completeness of a set of procedures)**.** A set K of procedures is complete if there is a unique initial entry procedure p_1 in K and if $p_i \in K$ references a procedure p_j then $p_j \in K$.

We consider implicitly complete sets of procedures. The adaptation to incomplete programs is considered to be outside the scope of this work.

Top-down and Bottom-up Analyses

We define a partial order relation $<_K$ called an *invocation order* from the call graph $\mathcal{C}$ where $n_i <_K n_j$ if n_i is a direct predecessor of n_j in the call graph $\mathcal{C}$. The invocation order embodies the notion of interprocedural dependency between procedures. We define conversely the *inverse invocation order* $<_{\overline{K}}$ on procedures where $n_i <_{\overline{K}} n_j$ if $n_j <_K n_i$, which can be seen as a sort of bottom-up view of the call graph.

Interprocedural analyses can be categorized according to the order in which procedures are traversed. In a top-down analysis, procedures are analyzed according to the invocation order $<_K$ from callers to callees, whereas in a bottom-up analysis, procedures are analyzed according to the inverse invocation order $<_{\overline{K}}$ from the callees up to the callers, usually by computing summaries of procedures. Recently, hybrid analyses [128] have been devised, in an attempt to combine the strengths of bottom-up analyses and top-down analyses.

In this thesis, we are interested in bottom-up approaches since procedures are analyzed only once, regardless of the number of calling contexts and in possibly much smaller abstract environments, allowing a modular analysis with potential scalability improvements for numerical analyses by abstract interpretation such as Linear Relation Analysis.

The next contribution of historical significance with regards to its influence on many posterior approaches is the work of M. Sharir and A. Pnueli in [118] which introduced the functional approach and the call strings approach for distributive data flow frameworks.

3.1.1 The Functional Approach

The functional approach computes data flow properties of programs using summaries of procedures, either from the bottom-up composition of individual propagation functions or by propagating data flow properties in a top-down fashion and by tabulating the properties obtained at the exit node of a procedure with the associated property at entry.

Data Flow Frameworks

The functional approach of Sharir and Pnueli considers finite data flow properties in a distributive data flow framework.

Definition 3.1.2 (Data flow framework)**.** A data flow framework (L, F) is made of a semilattice of data flow properties $(L, \sqsubseteq, \sqcap, \bot, \top)$ and a set $F \subseteq L \to L$ of monotonic propagation functions on L.

The operation of the semilattice L is denoted by the meet operator $\sqcap$ also called the data flow confluence operator.

The set F of propagation functions contains the identity function on L and is closed under the meet operation of L and functional composition.

Definition 3.1.3 (Distributive data flow framework)**.** A distributive data flow framework (L, F) is a data flow framework (L, F) where the propagation functions $f : L \to L$ in the set F are distributive for the operation $\sqcap$ of the semilattice L as follows:

$$\forall f \in F, \forall x, y \in L,\ f(x \sqcap y) = f(x) \sqcap f(y)$$

We use the control flow graph representation of procedures and we consider that each procedure has a control-flow graph $G_p = (V_p, E_p)$ where V_p is the set of the basic blocks of procedure p and E_p is the set of edges. The root block of the procedure is denoted by r_p.

In the data flow framework (L, F), a propagation function $f_{(n_i,n_j)} : L \to L$ is associated to each edge $(n_i, n_j) \in E_p$ in the CFG of each procedure p. It represents the transformation of the data flow properties when control passes from the beginning of basic block n_i to the end of n_i and then to the beginning of basic block n_j.

Example 3.1.1 (Reaching definitions)**.** Reaching definitions is a classical intraprocedural data flow analysis computing the set of the variable definitions $v := e$ reachable at a given basic block $n_i \in G_p$ of a procedure p with no assignment to v between n_i and a definition of v. It is represented by a distributive data flow framework (L, F) as follows:

1. The semilattice of data flow properties is $L = \mathcal{P}(\mathrm{Def}_p)$ where Def_p is the set of the variable definitions in p. The semilattice is partially ordered by set inclusion $\subseteq$ and the semilattice operation is set union.
2. The set $F \subseteq L \to L$ contains propagation functions $f_{(n_i,n_j)} : L \to L$ for each edge $(n_i, n_j) \in E_p$ in the CFG of p which are defined as:

   $$f_{(n_i,n_j)}(d) = \mathrm{GEN}(n_i) \cup (d - \mathrm{KILL}(n_i))$$

   where $\mathrm{GEN}(n_i)$ is the set of variable definitions in a basic block n_i:

   $$\mathrm{GEN}(n_i) = \{\ (v := e) \in \mathrm{Def}_p \mid (v := e) \in n_i\ \}$$

   and $\mathrm{KILL}(n_i)$ is the set of definitions where the variable v is assigned in n_i:

   $$\mathrm{KILL}(n_i) = \{(v := e_1) \in \mathrm{Def}_p \mid \exists (v := e_2) \in n_i\}$$

The set R_{n_j} of reaching definitions at the beginning of a basic block n_j is the least solution of the following system of equations:

$$\forall n_j \in N_p, \quad R_{r_p} = \emptyset \qquad R_{n_j} = \bigcup_{(n_i,n_j)\in E_p} f_{(n_i,n_j)}(R_{n_i})$$

Constant propagation can be extracted from the results of reaching definitions. A variable v is always equal to a constant c at the beginning of a basic block n_i if a unique definition $v := c$ of v is reachable at n_i, with $(v := c) \in R_{n_i}$.

It can be confusing that the operation of the semilattice of data flow properties is denoted by the meet operator but it is common practice in data flow analysis. The operator is instantiated in each particular analysis, such as set union for reaching definitions in Example 3.1.1.

Interprocedural Analysis over a Distributive Data Flow Framework

The functional approach considers data flow frameworks in the interprocedural case.

Let (L, F) be a distributive data flow framework. For each entry block or root block r_p of a procedure p and each basic block n_i of p, we add propagation functions $\varphi_{(r_p,n_i)} : L \to L$ to the function space F which represent the propagation of data flow properties in p from r_p to n_i. They are defined by the following set of equations:

$$\begin{array}{l} \forall x \in L,\ \varphi_{(r_p,r_p)}(x) = x \\ \forall n_i \in N_p - \{r_p\},\ \varphi_{(r_p,n_i)}(x) = \sqcap_{(n_i,n_j)\in E_p}\ h_{(n_i,n_j)}(\varphi_{(r_p,n_i)}(x)) \\ h_{(n_i,n_j)}(x) = \begin{cases} f_{(n_i,n_j)}(x) & \text{if } n_i \text{ is not a call block} \\ \sqcap_{e_q \in Exit_q}\ \varphi_{(r_q,e_q)}(x) & \text{if } n_i \text{ ends with a call to procedure } q \end{cases} \end{array}$$

where $h(n_i, n_j)$ denotes the propagation of properties from a basic block n_i to its successor n_j. If n_i ends with a call to a procedure q, properties are propagated in q and the returned property is the meet of the properties obtained at the exit blocks of q. We compute the maximal fixed point solution of the following set of equations:

$$\begin{array}{lcl} X_{(main,r_{main})} & = & \bot \\ X_{(q,r_q)} & = & \sqcap X_{(p,c)} \text{ for each procedure } p \text{ calling } q \text{ in a basic block } c \\ X_{(p,n_i)} & = & \varphi_{(r_p,n_i)}(X_{(p,r_p)}) \end{array}$$

3.1.2 The Call Strings Approach

The call strings approach uses the *interprocedural control flow graph* of a collection of procedures where the individual control flow graphs of each procedure are merged into a single graph.

Definition 3.1.4 (Interprocedural control flow graph)**.** Let K be a finite set of procedures. Let E^0 be the set of all intraprocedural edges (n_i, n_j) of procedures in K, such that (n_i, n_j) represents a direct jump from basic block n_i to basic block n_j.

Let E^1 be the set of all interprocedural edges (n_i, n_j) of procedures in K, such that (n_i, n_j) represents either a procedure call in basic block n_i to a procedure q (call edge), with n_j the root block of q, or a procedure return from the exit block n_i of a procedure q to the basic block n_j in procedure p (return edge).

The interprocedural control flow graph of K is $G^* = (N^*, E^*, r_{main})$ where the set of nodes is $N^* = \bigcup_{p\in K} N_p$, the set of edges is $E^* = E^0 \cup E^1$ and r_{main} is the root block of the initial procedure of K.

Not all paths in G^* are valid from an interprocedural point of view since G^* ignores the specificity of procedure calls and returns, which is that a given procedure call must be eventually matched by a return to that same procedure in a well-parenthesized fashion.

Definition 3.1.5 (Interprocedurally valid paths)**.** A path is valid if it respects the fact that when a procedure returns, it returns to the most recent call site. More precisely, a path is valid if it is represented by a string in the language of the following context-free grammar:

$$\begin{array}{lcl} \mathit{valid} & \rightarrow & \mathit{matched}\ (_p\ \mathit{valid}\ \mid\ \mathit{matched} \\ \mathit{matched} & \rightarrow & n_i\ \mid\ (_p\ \mathit{matched}\ )_p\ \mid\ \mathit{matched}\ \mathit{matched} \end{array}$$

where $(_p$ is representing a call to a procedure p and $)_p$ a return from p.

We denote as $\mathrm{IVP}(n_i, n_j)$ the set of all interprocedurally valid paths from a basic block n_i to a basic block n_j in G^*.

Definition 3.1.6 (Call string)**.** A call string $c = n_1...n_k \in N^*$ is a tuple of basic blocks such that there exists an interprocedurally valid path $q \in \mathrm{IVP}(r_{main}, n)$ starting at the root block r_{main} of the initial procedure *main* and ending in a basic block $n \in N^*$ which can be decomposed as

$$q = q_0.n_1.r_1.q_1\ ....\ n_k.r_k.q_k$$

where q_0 is an intraprocedural path in the initial procedure starting at block r_{main}, q_1 is an intraprocedural path in procedure p_1 starting at its root block r_1 and q_k is an intraprocedural path in procedure p_k starting at r_k and ending in block n.

The call strings approach is still a top-down functional approach, although it avoids the propagation of data flow properties along non-interprocedurally valid paths. Each propagated property is tagged by a call string, which encodes all the procedure calls through which it has been propagated. Data flow properties associated to different call strings are kept separated.

During propagation, data flow properties are tagged by call strings which are managed as stacks and updated when propagating though a call edge or a return edge.

Definition 3.1.7 (Semilattice of data flow properties with call strings)**.** Let Γ be the set of all call strings corresponding to interprocedurally valid paths in G^*. The call strings approach uses the data flow framework (L^*, F^*) where $L^* = \Gamma \rightarrow L$ is the semilattice of functions associating data flow properties to call strings. The meet operation on L^* is defined as a pointwise meet operator:

$$\forall f, g \in L^*,\ \forall c \in \Gamma,\ (f \sqcap g)(c) = f(c) \sqcap g(c)$$

We describe the set F^* of propagation functions. We define the *update* function, which updates call strings when procedure calls and returns are traversed during propagation.

Definition 3.1.8 (Call strings *update* function)**.** Let c be a call string and $(n_i, n_j) \in E^*$ be an edge in G^*. Let $last(c)$ be the last block in the call string c and $c - b$ be the call string c in which the basic block b has been removed. The *update* function is defined as:

$$\mathit{update}(c, (n_i, n_j)) = \begin{cases} c & \text{if } (n_i, n_j) \in E^0 \text{is an intraprocedural edge} \\ c.n_i & \text{if } (n_i, n_j) \in E^1 \text{is a call edge and } n_i \text{ is a call block} \\ c - \mathit{last}(c) & \text{if } (n_i, n_j) \in E^1 \text{is a return edge} \end{cases}$$

The *update* function appends the current block n_i to the call string c when propagating c through a procedure call and removes the last block of c if we are propagating through a return edge. Thus call strings are managed as stacks, storing interprocedural events encountered during propagation.

Definition 3.1.9 (Set $F^\star$ of propagation functions)**.** For each edge$(n_i, n_j) \in E^*$ of the interprocedural control flow graph, $F^\star$ contains a propagation function $f^*_{(n_i,n_j)} : L^* \to L^*$ defined as:

$$\forall g \in L^*,\ f^*_{(n_i,n_j)}(g) = \lambda c. \begin{cases} f_{(n_i,n_j)}(g(c')) & \text{if } \exists c', update(c', (n_i, n_j)) = c \\ \top & \text{otherwise} \end{cases}$$

The functions $f^*_{(n_i,n_j)}$ are monotonic and distributive due to the functions $f_{(n_i,n_j)}$ being themselves monotonic and distributive. We compute the maximal fixed point solution of the following system of equations:

$$\begin{array}{lcl} X_{r_{main}} & = & \bot \\ X_{n_j} & = & \sqcap_{(n_i,n_j)\in E^\star}\ f^*_{(n_i,n_j)}(X_{n_i}) \end{array}$$

Unbounded Call Strings and Termination

We can have unbounded call strings in the presence of unbounded recursion. In that case, the lattice L^* is infinite and the classic functional approach does not converge without using any widening on call strings.

We can obtain a converging analysis if we bound the length of call strings. In that case, the set Γ of call strings is finite and the lattice L^* has a finite height. This solution is not sound in general and we can loose information because we ignore some possible execution paths.

However, according to theorem 5.4 in [118], if the lattice L of data flow properties is finite, there is a bound M on the length of call strings for the bounded call strings approach to give a sound result. This bound is $M = K(|L| \mid +1)^2$ where K is the number of call edges and $|L|$ is the cardinality of L.

3.2 Interprocedural Data Flow Analysis via Graph Reachability

T. Reps, S. Horwitz, and M. Sagiv proposed an algorithm in [116] to solve data flow problems in polynomial time over finite lattices of data flow properties and distributive propagation functions, in a top-down fashion, similarly to the functional approach of M. Sharir and A. Pnueli. This algorithm solves all data flow problems belonging to this class by converting them to a graph reachability problem. Truly live variables, copy constant propagation and possibly non-initialized variables are examples of such problems. The considered class of data flow problems is called the IFDS class.

We consider interprocedurally valid paths, just as in the call strings approach. We denote as $IVP(n_i, n_j)$ the set of interprocedurally valid paths from a node n_i to a node n_j in the interprocedural control flow graph of the set of procedures.

Definition 3.2.1 (Interprocedural Finite data flow Subset (IFDS) problem)**.** An instance of an Interprocedural Finite Data flow Subset (IFDS) problem is a 5-tuple $(G^*, L, F, M, \sqcap)$ such that

1. $G^* = (N^*, E^*)$ is the interprocedural control graph of a finite set K of procedures.
2. L is a finite lattice of data flow properties of cardinality $|L|$.
3. $F \subseteq \mathcal{P}(L) \to \mathcal{P}(L)$ is a set of distributive monotonic propagation functions on the powerset of L.
4. $M : E^* \to F$ is a function associating a data flow propagation function to each edge of the interprocedural control flow graph G^*.
5. The meet operator on L is denoted as $\sqcap$ which can be either the set union operator or the set intersection operator.

Distributive propagation functions $f : \mathcal{P}(L) \to \mathcal{P}(L) \in F$ can be represented in a compact way as a graph with at most $2(|L|+1)$ nodes and $(|L|+1)^2$ edges. We describe below this representation.

Definition 3.2.2 (Representation relation)**.** The representation relation $R_f \subseteq (L \cup \{\mathbf{0}\}) \times (L \cup \{\mathbf{0}\})$ of a propagation function $f : \mathcal{P}(L) \to \mathcal{P}(L) \in F$ is:

$$\begin{array}{rcl} R_f & = & \{(\mathbf{0}, \mathbf{0})\} \\ & \cup & \{(\mathbf{0}, y) \mid y \in f(\emptyset)\} \\ & \cup & \{(x, y) \mid y \in f(\{x\}) \wedge y \notin f(\emptyset)\} \end{array}$$

The element $\mathbf{0}$ represents the empty set $\emptyset$. The relation R_f is the union of the singleton set $\{(\mathbf{0}, \mathbf{0})\}$ representing the empty set, the set $\{(\mathbf{0}, y) \mid y \in f(\emptyset)\}$ representing the images of the empty set by the function f and the set $\{(x, y) \mid y \in f(\{x\}) \wedge y \notin f(\emptyset)\}$ of pairs made of elements of L and their images by the function f.

The set $(L \cup \{\mathbf{0}\})$ has $|L|+1$ elements, the lattice L having a finite cardinality $|L|$. Thus the representation relation R_f of a propagation function f can be thought as a graph with $2(|L|+1)$ nodes and $(|L|+1)^2$ edges.

Example 3.2.1. The representation relation R_f of the function $f : \mathcal{P}(\{a, b, c\}) \to \mathcal{P}(\{a, b, c\})$ such that $\forall S \in \mathcal{P}(\{a, b, c\}), f(S) = \{a\}$ is defined as follows:

$$R_f = \{(0, 0), (0, a)\}$$

We should remark that there is no pair (b, a) because $a \in f(\{b\})$ but $a \in f(\emptyset)$.

Because the IFDS problem class is specialized for finite lattices L and propagation functions on the powerset $\mathcal{P}(L)$ of L, a finite representation of propagation functions can be computed, and represented in a compact manner, using BDDs (Boolean Decision Diagrams) for boolean lattices.

Converting a problem of the IFDS class into a graph reachability problem

Definition 3.2.3. Let IP $= (G^*, L, F, M, \sqcap)$ be an IFDS problem instance. We define an exploded interprocedural control flow graph for IP denoted by $G^\sharp_{IP} = (N^\sharp, E^\sharp)$ as follows:

$$\begin{array}{rcl} N^\sharp & = & N^\star \times (L \cup \{\mathbf{0}\}) \\ E^\sharp & = & \{((n_i, d_1), (n_j, d_2)) \mid (n_i, n_j) \in E^\star \wedge (d_1, d_2) \in R_{M(n_i, n_j)}\} \end{array}$$

Each node $n_i \in N^\star$ in the interprocedural control flow graph has been exploded into $|L|+1$ nodes. There is a pair $(n_i, d) \in N^\sharp$ for each node n_i and each data flow property $d \in (L \cup \{\mathbf{0}\})$. There is an edge in $E^\sharp$ between two nodes (n_i, d_1) and (n_j, d_2) if n_j is a successor of n_i in the interprocedural control flow graph and if (d_1, d_2) is in the relation R_f, representing the propagation function $M(n_i, n_j)$ from basic block n_i to basic block n_j.

The IP problem corresponds to the interprocedurally valid paths reachability problem in the graph $G^\sharp_{IP}$ from the source node $(r_{main}, \mathbf{0})$.

Let $X = (X_1, ..., X_k)$ be the maximal fixed point solution of an IFDS problem instance IP and $X_{n_i} \subseteq L$ be the component of this solution associated to a basic block n_i. Let $d \in L$ be a data flow property in lattice L.

According to theorem 3.8 in [116], we have $d \in X_{n_i}$ if there is an interprocedurally valid path in $G^\sharp_{IP}$ from $(r_{main}, \mathbf{0})$ to (n_i, d). Thus we can obtain the solution of an IFDS problem instance by solving a graph reachability problem in $G^\sharp_{IP}$ and by considering only interprocedurally valid paths.

3.3 Stack Abstractions For Interprocedural Analysis

We present in this section some approaches to interprocedural analysis which are using an abstraction of the stack.

In [22], F. Bourdoncle addresses the interprocedural abstract interpretation of a Pascal-like imperative language. The author describes an analysis to partition the variables of procedures into sets of variables sharing the same location in the stack. Concrete stacks which can be encountered during a concrete execution are abstracted by abstract stacks. The call contexts of a procedure corresponding to the same abstract stack are merged. Procedures are then analyzed for each call context associated to a distinct abstract stack. We must observe that in this approach, the construction of abstract stacks is quite heavy and their definition is tedious.

Jeannet et al. [75, 121] proposed a method reminiscent of the call strings approach for the relational numerical analysis of programs with recursive procedures and pointers to the stack. It is a top-down approach based on an abstraction of the stack. An implementation is available in the Interproc tool [71].

Several abstractions of procedures are used. First the standard semantics is abstracted by a local semantics, in which the effect of a procedure is described only on the top activation record. In the local semantics, procedures work on local copies of memory locations.

A stack abstraction is used to obtain a reachability analysis of sets of activation records from the local semantics. The abstract stacks which are reachable at a program point in a procedure are collapsed and abstracted into sets of activation records.

We compute the set of the reachable activation records at each program point of a procedure, using a relational interprocedural abstract interpretation. Finally, we obtain an invariant which is a set of reachable activation records.

The hierarchy of abstractions used in this approach is as follows:

1. Standard semantics.
2. Local semantics: the effect of a procedure is restricted to the top activation record.

3. Instrumented semantics: procedures are defined as transition systems on a set of states (σ_0, σ) where σ_0 is the initial state at the beginning of a call and σ is the current state.
4. Stack abstraction: sets of stacks are abstracted by sets of activation records.
5. Activation record abstraction: activation records are represented by functions $Id \to \mathcal{B} \cup D^\sharp$. The computed invariant are sets of such functions and are represented symbolically by MTBDDs (Multi-Terminal Binary Decision Diagrams) with numerical abstract values as terminal nodes.

We can make here nearly the same remark as for the previous approach. Several abstractions of the stack and activation records are performed. Thus we can question the necessity of such stack abstractions and the possible alternatives for interprocedural abstract interpretation especially regarding the complexity of their definition and their construction.

A stack abstraction is also performed in [117] for trace partitioning. Interprocedural traces are abstracted here by functions $Stack \times Loc \to \mathcal{P}(S)$ from pairs $(k, l) \in Stack \times Loc$ made of a stack and a control state, to a set S of memory states. Thus we obtain an analysis which is sensitive to the calling context, at the expense of keeping potentially large abstractions of the stack.

The call strings approach, which is separating data flow properties according to the encountered call contexts, is an early example of stack abstraction since call strings are updated with representations of the procedure calls and procedure returns encountered during propagation.

3.4 Statement-level Summaries

An approach based on statement-level summaries implemented in the PIPS tool [5, 69] was proposed to discover linear invariants in a collection of procedures by abstract interpretation. Statements in procedures are abstracted by affine transformers [87] which are input-output relations represented by convex polyhedra. The summary of a whole procedure is obtained from the composition of statement transformers, in a bottom-up fashion.

This analysis is divided into two phases, a transformer computation phase and an invariant generation phase.

Each statement is first abstracted by a convex polyhedron representing a linear relation between the state of the variables before the statement and their state after the execution of that statement. Transformers for control structures are computed from the transformers of individual statements. An algorithm is given to compute an abstraction of the transformer of a loop from the transformer of its body, which is termed the Affine Derivative Closure algorithm.

Invariants are obtained by forward propagation of the procedure precondition using the transformers computed in the previous phase.

However, precision can be lost during the composition of individual statement transformers. For each transformer, the number of variables can be doubled in the worst-case to express the initial states of numerical variables. The execution time can increase exponentially during the computation of statement transformers due to the complexity of convex polyhedra operations.

Statement-level transformers are also used in [126], to solve interprocedural data flow problems belonging to the IFDS class (see Definition 3.2.1). Procedure summaries are derived from the composition of statement-level transformers. Transformers of loop statements are constructed by iterated composition of the loop body transformer, until a fixed point is reached. Convergence is guaranteed because the IFDS problem class is concerned only by finite lattices.

Aggregate domains, abstract domains which can be expressed as aggregates of simpler domains, allow transformers to be broken into micro-transformers, each micro-transformer operating on a sub-domain. Each micro-transformer can separate cases corresponding to distinct classes of values in a sub-domain, each having a precondition that defines a distinct class of values and a postcondition describing how values are transformed. The aim is to have micro-transformers consisting of cases defined on values behaving uniformly with simpler postconditions.

3.5 Procedure Summaries Using Generic Assertions

The approach proposed in [55] is a bottom-up approach computing summaries of procedures by backward propagation of generic assertions. A generic assertion is an assertion which must be instantiated by symbols of an underlying abstraction.

For example, a generic assertion in the theory of linear arithmetic is $\alpha x+\beta y = \gamma$ where α, β and γ are unknown variables. The assertion which has to hold after a procedure call must be given manually. It is matched using a unification algorithm to the generic assertion computed for the called procedure.

Example 3.5.1. We assume that the generic assertion summarizing a procedure p is $\alpha x + \beta y = \gamma$. We want to check that the assertion $y = 2x + 1$ holds after a call to p. We match $y = 2x + 1$ with the generic assertion $\alpha x + \beta y = \gamma$ and we obtain the substitution $\alpha \mapsto -2$, $\beta \mapsto 1$ and $\gamma \mapsto 1$. This substitution can be used to compute the summary of the calling procedure, in a bottom-up fashion.

Procedure summaries are obtained by computing the weakest precondition of each generic assertion at each point of a procedure. A fixed point computation is performed to handle loops. The authors describe a generic assertion simplification technique because a naive weakest precondition computation can be exponential in the number of operations performed.

3.6 Relational Interprocedural Analyses

3.6.1 Modular Static Analysis

Cousot et al. [39, 40] describes the symbolic relational separate analysis for abstract interpretation, which uses relational abstract domains, relational semantics and symbolic names to represent initial values of parameters modified by a procedure. When used with the polyhedra abstract domain, the approach computes procedure summaries which are input-output relations represented by a single convex polyhedron, with no ability to capture disjunctive behaviors in procedures. Recursive procedures are supported, although the technique is quite classical and was already presented earlier in [35, 36, 60].

3.6.2 Relational Abstractions of Functions

A relational abstraction of sets of functions for shape analysis is proposed in [72]. It considers functions of signature $D_1 \to D_2$, provided that there exists abstractions A_1 of $\mathcal{P}(D_1)$ and A_2 of $\mathcal{P}(D_2)$ and that A_1 has a finite cardinality. This abstraction is relational since it is able to express relations between elements mapped by a set of functions. However the abstraction A_1 is required to be of finite cardinality, thus excluding usual numerical abstract domains such as convex polyhedra.

3.6.3 Interprocedural Analyses based on Linear Algebra

Müller-Olm et al. [103] proposed an interprocedural bottom-up analysis discovering all Herbrand equalities between program variables at each program point in polynomial time. It considers abstracted programs with only affine assignments, ignoring conditions on branches, with other assignments handled soundly as non-deterministic assignments. An algorithm for precise interprocedural value numbering is derived from this approach.

An extension was proposed in [47] to obtain an interprocedural analysis to discover linear two-variable equalities. It has a worst-case complexity of $\mathcal{O}(nk^4)$ where n is the program size and k is the number of program variables. It infers all valid equalities between program variables of the form $x_i = c$ or $x_i = x_j + c$ where $c \in \mathbb{Z}$. No widening operator is necessary.

Each procedure p is represented by a control flow graph G_p with control locations as nodes and edges labeled with program statements. Let X be the set of all program variables. A finite satisfiable set of equalities E can be represented by a weighted directed graph, where the set of nodes is $X \cup \{0\}$ and there is an edge from x_i to x_j with weight b if there the equality $x_i = x_j + b$ is valid. The graph is assumed to be symmetric and transitive. An equality $x_i = c$ with $c \in \mathbb{Z}$ is represented by an edge from x_i to 0 with weight c.

For every maximal connected component C of the graph, we choose a reference node, which is either 0 or the variable x_i with the least index i. For every other variable x_j in C, we only record the equality $x_j = b$ or $x_j = x_i + b$. The conjunction of all those equalities is equivalent to E. A conjunction of such properties is said to be normalized. A normalized conjunction consists of at most k equalities and can be represented by an array of size k.

The length of every strictly increasing chain $false \sqsubseteq E_1 \sqsubseteq ... \sqsubseteq E_n$ is bounded by $k+1$, because in every pair $E_i \sqsubseteq E_{i+1}$, E_{i+1} has strictly more connected components than E_i and the number of connected components in a graph of $k+1$ nodes is bounded by $k+1$.

Summaries of procedures are obtained by computing weakest preconditions of generic postconditions. For example, consider the procedure p below.

```
void p()
{
    x1 = x1 + 1;
    x2 = x2 - 1;
}
```

The procedure p increments the global variable `x1` and decrements the global variable `x2`. We are interested in the postconditions at the end of the procedure which are of the form:

$$x_1 = a \wedge x_2 = b \wedge x_2 = x_1 + c$$

with $a, b, c \in \mathbb{Z}$. The postcondition $x_1 = a \wedge x_2 = b \wedge x_2 = x_1 + c$, where a, b, c are left as placeholder variables, is called a generic postcondition. The summary of procedure p is obtained by computing the weakest precondition of the generic postcondition:

Precondition	$x_1 = a - 1 \wedge x_2 = b + 1 \wedge x_2 = x_1 + c + 2$
Postcondition	$x_1 = a \wedge x_2 = b \wedge x_2 = x_1 + c$

This approach is not able to express disjunctions in the procedure summary. It is extended to discover linear two-variable equalities, which are equalities of the form $x_i = c$ or $x_i = ax_j + b$. Such a set of linear equalities can be put into a normal form where each variable appears in only one equality. Therefore, when in normal form, a set E of linear equalities has at most k equalities. The summary of a procedure is obtained by computing weakest preconditions of linear generic postconditions of the form $a_1 x_i = a_2$ or $a_1 x_i = a_2 x_j + a_3$.

3.6.4 Dual Interprocedural Analysis using Linear Arithmetic

Popeea et al. [106, 107, 108] presented an analysis to both prove user-supplied safety properties and to find bugs by deriving a condition *ok* leading to success and a condition *err* leading to failure for each procedure of a program. The summary of a procedure is a pair (ok, err) of these two conditions, with *ok* and *err* being formulas in the first-order theory of linear arithmetic.

The conditions *ok* and *err* of the procedure summary are inferred in a bottom-up fashion, from callees to callers. The analysis is done in two steps:

1. A constraint abstraction of the procedure body is generated according to Hoare-style rules for success and failure outcomes.
2. A fixpoint computation is used to derive closed-form formulas for both *ok* and *err*.

Disjunctive numerical properties can be represented in the *ok* and *err* formulas, with linear equalities and inequalities. Simplification of formulas and checking for satisfiability is handled by a complete decision procedure for linear arithmetic provided by the Omega Test [77]. The Omega Test is known to have a doubly-exponential worst-case complexity. This approach is implemented in the DUALYZER tool.

Part II

Relational Summaries for Interprocedural Analysis

Chapter 4

Relational Abstract Interpretation

Relational abstract interpretation is concerned by the automatic discovery of relations between the variables of a program at each program point. Linear Relation Analysis (LRA) proposed in [41, 60, 69] is the archetypal numerical relational analysis. Although we are particularly interested in its application to Linear Relation Analysis, relational abstract interpretation is not specific to numerical analyses, it also encompasses any analysis inferring relations among states of a program, such as for example in shape analysis, where relations may be inferred between memory elements or parts of data structures.

By inferring abstract relations between states in a program, particularly between the initial values of variables at the entry of a procedure and their values at exit, relational abstract interpretation can serve as a foundation to compute relational summaries of procedures.

We present in this chapter a formalization of relational abstract interpretation that we did not find elsewhere. We start by giving a general characterization of relational abstract domains, as abstractions of binary relations over sets of program states, since they are the key ingredient to represent relations between states. Then, we take a particular attention to procedure preconditions and we show that preconditions must be treated carefully in order to obtain precise relational procedure summaries.

4.1 Relations on States and the Transitive Closure

We introduce some definitions and notations on states and relations as a prelude to our presentation of relational abstract interpretation.

States and Relations

Let S be a set of states. We denote as $\rho \subseteq S \times S$ a relation over states. We denote as $\mathcal{R} = \mathcal{P}(S \times S)$ the set of binary relations over S.

A relation ρ over S is a set of pairs (s_1, s_2) of states $s_1, s_2 \in S$, where s_1 is termed the *source* component of the pair and s_2 is termed the *target* component of the pair. We define classically the source projection $src : \mathcal{R} \to S$ and the target projection $tgt : \mathcal{R} \to S$

of binary relations as:

$$\forall \rho \in \mathcal{R}, \quad src(\rho) = \{s_1 \in S \mid \exists s_2 \in S, (s_1, s_2) \in \rho\}$$
$$\forall \rho \in \mathcal{R}, \quad tgt(\rho) = \{s_2 \in S \mid \exists s_1 \in S, (s_1, s_2) \in \rho\}$$

If $U \subseteq S$ is a subset of S, then we denote the identity relation over U as $\mathrm{Id}_U = \{(s, s) \mid s \in U\}$.

Definition 4.1.1 (Composition of binary relations)**.** Let $\rho_1, \rho_2 \in \mathcal{R}$ be binary relations over a set S. The composition $\rho_1 \circ \rho_2$ of ρ_1, ρ_2 is defined as:

$$\forall \rho_1, \rho_2 \in \mathcal{R}, \quad \rho_1 \circ \rho_2 = \{(s_1, s_3) \mid \exists s_2 \in S, (s_1, s_2) \in \rho_1 \wedge (s_2, s_3) \in \rho_2\}$$

Transitive Closure

Definition 4.1.2 (Transitive closure)**.** Let $\rho \subseteq S \times S$ is a binary relation over S. The transitive closure $\rho^\star$ of ρ is defined as:

$$\rho^\star = \bigcup_{k \geq 0} \rho^k$$

If we consider the transition relation $\rho \subseteq S \times S$ of a transition system, the transitive closure $\rho^\star$ relates states which are reachable from one another in a finite number of computation steps of the system described by the transition relation. The system can be, for example, a program or a procedure. The image $tgt(\rho^\star(I))$ of a set I of initial states by the transitive closure $\rho^\star$ is the set of reachable states of the transition system under consideration.

As an endpoint to these introductory matters on relations, we point the interested reader to some classical references on the theory of binary relations, such as Tarski's classical revival paper [124] on the calculus of relations and [111].

Forward and Backward Relational Semantic Equations

The transitive closure $\rho^\star$ of ρ can be expressed as the least solution of a forward fixpoint equation and equally as the least solution of a backward fixpoint equation:

$$\rho^\star = \mu r.\, \mathrm{Id}_S \cup (r \circ \rho) \quad \text{(forward equation)}$$
$$\rho^\star = \mu r.\, \mathrm{Id}_S \cup (\rho \circ r) \quad \text{(backward equation)}$$

Trace Partitioning

Following the classical trace partitioning technique, we assume that the set S of states is finitely partitioned according to a partition $\delta = \{S_1, S_2, ..., S_n\}$ as follows:

$$\forall S_i, S_j \in \delta, i \neq j \Rightarrow S_i \cap S_j = \emptyset$$
$$\bigcup_{S_i \in \delta} S_i = S$$

We can also denote S as the disjoint sum $S = S_1 \oplus S_2 \oplus ... \oplus S_n$ of the members of the partition δ,

Example 4.1.1. Trace partitioning can be done according to the control locations of a program. We assume that $L = \{l_1, l_2, ..., l_n\}$ is a finite set of control locations, $\mathcal{V}$ is a set of values and the set S of program states is the cartesian product $S = L \times \mathcal{V}^n$. We construct classically the partition δ_L of S according to control locations in L as follows:

$$\delta_L = \bigoplus_{i=1}^{n} \{(l_i, V) \in L \mid V \in \mathcal{V}^n\}$$

A partitioning can also be defined in a more semantic fashion and can express state properties, like preconditions.

Definition 4.1.3 (Restriction of a relation)**.** For a binary relation $\rho \in \mathcal{R}$ and $S_i, S_j \in \delta$, we define the restricted relation $\rho(S_i, S_j)$ as follows:

$$\forall S_i, S_j \in \delta, \quad \rho(S_i, S_j) = \rho \cap (S_i \times S_j)$$

Property 4.1.1. The restriction $\rho^\star(S_i, S_j)$ of the transitive closure $\rho^\star$ to $S_i, S_j \in \delta$ can be expressed as the least fixpoint of the following system of forward fixpoint equations:

$$\forall i \neq j, \quad \rho^\star(S_i, S_j) = \bigcup_{k=1}^{n} \rho^\star(S_i, S_k) \circ \rho(S_k, S_j)$$
$$\rho^\star(S_i, S_i) = \mathrm{Id}_{S_i} \cup \bigcup_{k=1}^{n} \rho^\star(S_i, S_k) \circ \rho(S_k, S_i)$$

Intuitively, for any states $s_i \in S_i$ and $s_j \in S_j$, the membership $(s_i, s_j) \in \rho^\star(S_i, S_j)$ of the pair (s_i, s_j) in $\rho^\star(S_i, S_j)$ means that s_j can be reached from s_i by iterated composition of the binary relation ρ with itself, and since S_i and S_j are disjoint, the binary relation ρ must be applied at least once. If we consider ρ as the relation of a transition system on S, this means that the state $s_j \in S_j$ is reachable from the state $s_i \in S_i$ in a finite, but non-zero, number of computation steps. Additionally, if $(s_i, s_j) \in \rho^\star(S_i, S_i)$, the state s_j is reachable from s_i in zero computation steps if $s_i = s_j$.

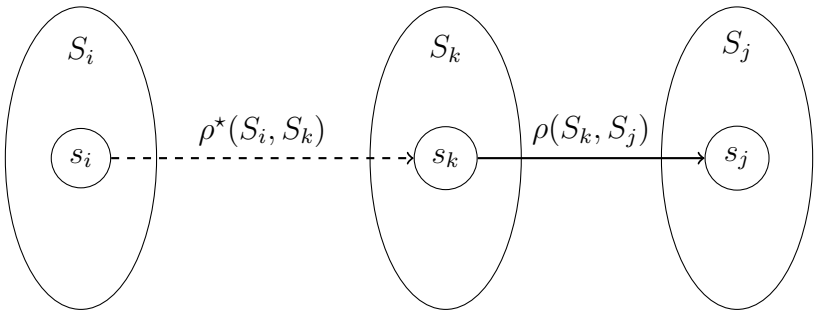


Figure 4.1: Graphical illustration of the forward fixpoint characterization of $\rho^\star(S_i, S_j)$ for $i \neq j$.

As illustrated in Figure 4.1, we can split the path from a state $s_i \in S_i$ to a state $s_j \in S_j$ into a finite path from s_i to a state $s_k \in S_k$ with $S_k \in \delta$, denoted as $(s_i, s_k) \in \rho^\star(S_i, S_k)$, and an atomic computation step from s_k to s_j, denoted as $(s_k, s_j) \in \rho(S_k, S_j)$. Then, the transitive closure $\rho^\star(S_i, S_j)$ is the union of all such splittings for each $S_k \in \delta$.

4.2 Concrete Relational Semantics

We denote as $(S, \rho, I, \mathcal{E})$ the transition system describing a procedure p where S is a set of states, $\rho \subseteq S \times S$ is the transition relation of p, $I \subseteq S$ is the set of initial states and $\mathcal{E} \subseteq S$ is the set of exit states.

4.2.1 Concrete Relational Summaries

Definition 4.2.1 (Concrete Relational Summary)**.** The concrete relational summary of a procedure p is $\sigma_p = \rho^\star(I, \mathcal{E})$.

The concrete summary σ_p of a procedure p is the transitive closure of the transition relation ρ restricted to the set I of initial states and to the set $\mathcal{E}$ of exit states. It is the set of pairs $(s_I, s_\mathcal{E})$ made of an initial state $s_I \in I$ and of an exit state $s_\mathcal{E} \in \mathcal{E}$ reachable from s_I in a finite number of computation steps in the procedure p.

For the forward computation of a summary σ_p we are interested in the computation of $\rho^\star(I, S_j)$ for each $S_j \in \delta$ by instantiating the fixpoint characterization of $\rho^\star(S_i, S_j)$ given in Property 4.1.1 using the following equations:

$$\rho^\star(I, S_j) = \left(\bigcup_{k=1}^{n} \rho^\star(I, S_k) \circ (S_k, S_j) \right) \cup \left\{ \begin{array}{ll} \mathrm{Id}_I & \text{if } S_j = I \\ \emptyset & \text{otherwise} \end{array} \right\}$$

4.2.2 Concrete Semantics of Procedure Calls

The concrete summary of a procedure p can be used to express the set of states reachable after a call to p given the states reachable immediately before the call. We can use the summary of a called procedure to obtain its effect in a caller.

We make first some assumptions on the programs that we are considering in this presentation for the sake of clarity, and we will consider thereafter that the necessary transformations are available in practice.

We assume that all procedure parameters are given and that there are no parameters with default values, neither partially-applied procedures.

We consider that all procedure parameters in a program are passed by reference, we are not concerned with pointer manipulation in this presentation. We entrust existing pointer analyses to detect aliasing problems and to provide us an abstraction of numerical procedures consistent with our assumptions, on top of which our analysis should be built in practice.

Instead of an explicit program transformation prior to our analysis, our approach could also be used on top of another analysis providing us only a suitable view of procedures, in which we would only work with abstract numerical variables and expressions generated by a memory abstract domain, such as in [99].

We assume that global variables are dealt with as additional procedure parameters. All variables are assumed to be parameters, since local variables don't raise any problem from the point of view of relational collecting semantics.

Let p be a procedure with a set S of states and let T be the set of states of a program calling p. We denote as $\pi : \mathcal{P}(S \times S) \to \mathcal{P}(T \times T)$ the parameter passing mechanism for the summary of p, which consists in renaming formal parameters into actual ones.

Definition 4.2.2. Let $T_i \subseteq T$ be the set of states reachable just before the call to p and let $T_j \subseteq T$ be the set of states reachable just after the call to p. The effect of the call to p is described by the elementary relation $\rho(T_i, T_j) = \pi(\rho_p)$.

4.2.3 From State to Relational Collecting Semantics

We are particularly interested in the relational analysis of numerical procedures and we describe how a relational collecting semantics can be obtained from the usual collecting semantics of a procedure.

States of Numerical Procedures

A state of a numerical procedure with n variables is a pair $(\ell, V) \in \mathit{Loc} \times \mathcal{N}^n$ where $\ell \in \mathit{Loc}$ is a control location such as a line, a statement or a block in a control-flow graph, and $V = (v_1, ..., v_n) \in \mathcal{N}^n$ is a vector of numerical values in $\mathcal{N}$.

Partitioning

Control locations $\ell \in \mathit{Loc}$ provide a natural partitioning of sets of states. We denote as $S_\ell = \{(\ell, V) \mid V \in \mathcal{N}^n\}$ the set of states at a control location ℓ.

The set of initial states of a procedure at an entry control location ℓ_I is denoted as S_{ℓ_I} and can be further restricted by a precondition $A_I \in \mathcal{N}^n$ such as:

$$I = \{(\ell_I, V) \mid V \in A_I\}$$

State Collecting Semantics

The usual collecting semantics defines the set A_ℓ of reachable variable valuations at a control location ℓ in a procedure with $A_\ell = \{V \mid (\ell, V) \text{ is a reachable state from } I\}$ as the least fixpoint of a system of fixpoint equations:

$$A_\ell = F_\ell(\{A_{\ell'} \mid \ell' \in \mathit{Loc}\}) \cup \left\{ \begin{array}{ll} A_I & \text{if } \ell = \ell_I \\ \emptyset & \text{otherwise} \end{array} \right\}$$

where the transfer function F_ℓ expresses how the states in S_ℓ depends on the states at other control points. The sets A_ℓ of reachable states at each control location ℓ are termed the collecting semantics of the procedure.

Relational Collecting Semantics

Collecting semantics on states can be extended to relational semantics as follows: for each variable v_i, a new variable v_i^0 is introduced to record the initial value of the variable v_i. The new set of states is $S^R = \mathit{Loc} \times \mathcal{N}^{2n}$ and the new set of initial states is defined as follows:

$$A_I^R = \{(v_1^0, ..., v_n^0, v_1, ..., v_n) \mid (v_1^0, ..., v_n^0) \in A_I \wedge v_i = v_i^0, i \in \{1, ..., n\}\}$$

$$I^R = \{(l_I, V) \mid V \in A_I^R\}$$

The relational collecting semantics of the procedure is equivalent to the usual state collecting semantics of the procedure initialized with the assignments $v_i^0 = v_i$ for each $i \in \{1, ..., n\}$.

The relational collecting semantics A_ℓ^R at a control location $\ell \in Loc$ are defined as:

$$A_\ell^R = F_\ell^R(\{A_{\ell'}^R \mid \ell' \in Loc\}) \cup \left\{ \begin{array}{ll} A_I^R & \text{if } \ell = \ell_I \\ \emptyset & \text{otherwise} \end{array} \right\}$$

The introduction of v_i^0 variables to record initial values of variables is the only change made with respect to the usual collecting semantics. Extended states are propagated and transformed in the classical way by the transfer functions F_ℓ^R representing the effect of statements at each control location $\ell \in Loc$. We assume that the propagation functions F_ℓ^R do not change the values of the v_i^0 variables.

Let $\mathcal{E} \in Loc$ be the set of exit control locations of a numerical procedure p. The concrete summary σ_p of the numerical procedure p can be defined as follows:

$$\sigma_p = \bigcup_{\ell \in \mathcal{E}} A_\ell^R$$

If we consider the presence of local variables, they should be eliminated from this expression by existential quantification.

Thus the relational collecting semantics of a numerical procedure can be obtained in a rather simple fashion from the usual state collecting semantics.

4.2.4 A very simple example

We give thereafter a very simple example of relational concrete summary. We consider the example program in Figure 4.2, which implements the classical Euclidean division, by the method of successive subtractions, as described originally in [45].

```
  void div(int a, int b, int * q, int * r)
  {
      assume(a >= 0 && b >= 1);
1:    *q = 0;
      *r = a;
2:    while
3:        (*r >= b)
      {
          *r = *r-b;
4:        *q = *q + 1;
5:    }
6:}
```

$$\begin{aligned}
A_1 &= \{(a_0, b_0, q_0, r_0, a, b, q, r) \mid \\
&\quad a_0 \geq 0 \wedge b_0 \geq 1 \wedge a = a_0 \\
&\quad \wedge\ b = b_0 \wedge q = q_0 \wedge r = r_0\} \\
A_2 &= A_1[q := 0][r := a] \\
A_3 &= A_2 \cup A_5 \\
A_4 &= A_3 \cap (r \geq b) \\
A_5 &= A_4[r := r - b][q := q + 1] \\
A_6 &= A_3 \cap (r \leq b - 1)
\end{aligned}$$

Figure 4.2: Example program implementing the classical Euclidean division by the method of successive subtractions.

The system of concrete fixpoint equations representing the `div` procedure is given on the right-hand side of Figure 4.2. The set of reachable variable valuations at a control location i is given by the set A_i.

The concrete relational summary of the `div` procedure is given by the least fixpoint solution for A_6, at the unique exit point of the procedure:

$$A_6 = (a = a_0 \wedge b = b_0 \wedge a = bq + r \wedge q \geq 0 \wedge r \leq b - 1 \wedge r \geq 0 \wedge a_0 \geq 0 \wedge b_0 \geq 1)$$

It describes the effect of the `div` procedure on its parameters, by relating their final value at the exit point of the procedure back to their initial value at the entry of the `div` procedure. Relations between final values of parameters are recorded as well.

We should notice that the concrete summary of the `div` procedure contains a non-linear relation, $a = bq + r$, which is the well-known property of Euclidean division, and thus will not be obtainable by Linear Relation Analysis, as non-linear relations can not be represented by convex polyhedra.

4.3 Relational Abstract Interpretation

4.3.1 General Framework

Relational Abstract Domain

A relational abstract domain is an abstract domain providing an abstraction of sets of binary relations over program states.

Definition 4.3.1 (Relational abstract domain)**.** A relational abstract domain is a complete lattice $(\mathcal{R}^\sharp, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$ related to $\mathcal{R}$ by a Galois connection $(\alpha_\mathcal{R}, \gamma_\mathcal{R})$.

Example 4.3.1. We consider the set $\{x, y, z\}$ of numerical variables and the set $S = \mathbb{Z}^3$ of variables valuations. We give some abstract relations over S expressed in the convex polyhedra abstract domain:

$$\begin{array}{rcl} r_1^\sharp & = & (x = x_0 + 1 \wedge y = y_0 \wedge z_0 \geq z \wedge y \geq 23 \wedge x \geq 1) \\ r_2^\sharp & = & (y = 2y_0 + 1 \wedge x_0 \leq x \leq x_0 + 2y \wedge 1 \leq y \leq z) \end{array}$$

It is well-known that we can express abstract relations between numerical states in the convex polyhedra abstract domain by introducing a doubled vocabulary of variables X_0, X where variables in X_0 represent the source state and variables in X represent the target state. In example 4.3.1, we duplicated variables x, y, z and introduced variables x_0, y_0, z_0 to denote the initial value of variables x, y, z.

For $U \subseteq S$, we denote as $\mathrm{Id}^\sharp(U)$ the abstract relation $\alpha_\mathcal{R}(\mathrm{Id}_U)$ which is the abstraction in $\mathcal{R}^\sharp$ of the identity relation over U.

Definition 4.3.2 (Composition of abstract relations)**.** The composition $r_1^\sharp \circ r_2^\sharp$ of two abstract relations $r_1^\sharp, r_2^\sharp \in \mathcal{R}^\sharp$ is defined as:

$$r_1^\sharp \circ r_2^\sharp = \alpha_\mathcal{R}(\gamma_\mathcal{R}(r_1^\sharp) \circ \gamma_\mathcal{R}(r_2^\sharp))$$

A relational abstract domain $\mathcal{R}^\sharp$ induces two abstract domains $S^\sharp_\rightarrow$ and $S^\sharp_\leftarrow$ on the set S of states as follows:

$$\forall U \subseteq S, \quad \alpha_{S_\rightarrow}(U) = \alpha_\mathcal{R}(U \times S) \quad \alpha_{S_\leftarrow}(U) = \alpha_\mathcal{R}(S \times U)$$

We should note that both abstract domains $S^\sharp_\rightarrow$ and $S^\sharp_\leftarrow$ are included in $\mathcal{R}^\sharp$. Abstract domains $S^\sharp_\rightarrow$ and $S^\sharp_\leftarrow$ represent a set U of states as an abstract relation where U can be either abstracted as the source or as the target of the abstract relation respectively.

Definition 4.3.3 (Abstract projections)**.** The abstract source projection $src^\sharp : \mathcal{R}^\sharp \to S^\sharp_\rightarrow$ of abstract relations in a relational abstract domain $\mathcal{R}^\sharp$ is defined as:

$$src^\sharp(r^\sharp) = \alpha_{S_\rightarrow}(src(\gamma_\mathcal{R}(r^\sharp)))$$

Similarly, we define the abstract target projection $dst^\sharp : \mathcal{R}^\sharp \to S^\sharp_\leftarrow$ as follows:

$$tgt^\sharp(r^\sharp) = \alpha_{S_\leftarrow}(tgt(\gamma_\mathcal{R}(r^\sharp)))$$

Relational Abstract Analysis

Let $\rho \subseteq S \times S$ be a transition relation over a set S of states. Let $\rho^\sharp \in \mathcal{R}^\sharp$ be an upper bound of its abstraction in the relational abstract domain $\mathcal{R}^\sharp$ with $\alpha_\mathcal{R}(\rho) \sqsubseteq \rho^\sharp$. In order to ensure convergence, we assume that the relational abstract domain $\mathcal{R}^\sharp$ provides both a widening operator $\nabla : \mathcal{R}^\sharp \times \mathcal{R}^\sharp \to \mathcal{R}^\sharp$ and a narrowing operator $\triangle : \mathcal{R}^\sharp \times \mathcal{R}^\sharp \to \mathcal{R}^\sharp$.

An upper approximation of the transitive closure $\rho^{\sharp\star}$ of an abstract relation $\rho^\sharp$ can be obtained by computing the limit $r^{\sharp\nabla}$ of an increasing sequence $(r^\sharp_n)_{n\geq 0}$ defined as:

$$\begin{cases} r^\sharp_0 & = & \bot \\ r^\sharp_{n+1} & = & r^\sharp_n \nabla (r^\sharp_n \circ \rho) \end{cases}$$

and by computing the limit $r^{\sharp\nabla\triangle}$ of a decreasing sequence $(r'^\sharp_n)_{n\geq 0}$ defined as:

$$\begin{cases} r'^\sharp_0 & = & r^\sharp_\nabla \\ r'^\sharp_{n+1} & = & r'^\sharp_n \triangle (r'^\sharp_n \circ \rho) \end{cases}$$

The limit $r^{\sharp\nabla\triangle}$ is a sound approximation of the transitive closure $\rho^\star$ of the concrete transition relation ρ where $\rho^\star \subseteq \gamma_\mathcal{R}(r^{\sharp\nabla\triangle})$.

Abstract Partition

A relational abstract domain $\mathcal{R}^\sharp$ induces two abstract domains $S^\sharp_\leftarrow$ and $S^\sharp_\rightarrow$ both providing abstractions of the set S of states. We can define an abstract partition of the abstract domain $S^\sharp_\leftrightarrow$ for each direction $\leftrightarrow \in \{\leftarrow, \rightarrow\}$.

Definition 4.3.4 (Abstract partition)**.** Let $S^\sharp_\leftrightarrow$ be an abstract domain obtained from a relational abstract domain $\mathcal{R}^\sharp$. A finite abstract partition $\delta^\sharp \subseteq S^\sharp_\leftrightarrow$ of $S^\sharp_\leftrightarrow$ is a finite set $\delta^\sharp = \{S^\sharp_1, ..., S^\sharp_n\}$, such that $\{S_i = \gamma_{S_\leftrightarrow}(S^\sharp_i) \mid S^\sharp_i \in \delta^\sharp\}$ is a partition of S.

It is interesting to consider an abstract partition $\delta^\sharp = \{S^\sharp_1, ..., S^\sharp_n\}$ of $S^\sharp_\leftrightarrow$ if the behavior of a transition system can be split into several distinct behaviors expressed by abstract elements $S^\sharp_1, ..., S^\sharp_n$.

More generally, for any subset $U \subseteq S$ of states, we can define an abstract partition $\delta^\sharp$ of $U^\sharp_\leftarrow = \alpha_{S_\leftrightarrow}(U)$ as a finite set $\delta^\sharp = \{U^\sharp_1, ..., U^\sharp_n\}$ where $\{U_i = \gamma_{S_\leftrightarrow}(U^\sharp_i) \mid U^\sharp_i \in \delta^\sharp\}$ is a partition of U. This is especially useful when we consider a precondition $A \subseteq S$ and we are interested in an abstract partition relative to the precondition A, which is an abstract partition of $\alpha_{S_\leftrightarrow}(A)$.

Partitioned Relational Analysis

Let $\rho \subseteq S \times S$ be a transition relation over S. Let $\delta^\sharp = \{S_1^\sharp, ..., S_n^\sharp\}$ be an abstract partition of $S^\sharp = \alpha(S)$.

For each $S_i^\sharp, S_j^\sharp \in \delta^\sharp$, we assume that we have an upper bound $\rho^\sharp(S_i^\sharp, S_j^\sharp) \in \mathcal{R}^\sharp$ of the abstraction $\alpha_{\mathcal{R}}(\rho(S_i, S_j))$ of $\rho(S_i, S_j)$ with $\alpha_{\mathcal{R}}(\rho(S_i, S_j)) \sqsubseteq \rho^\sharp(S_i^\sharp, S_j^\sharp)$.

An upper approximation of the vector $\{\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp) \mid S_i^\sharp, S_j^\sharp \in \delta^\sharp\}$ can be obtained as the limit of the increasing-decreasing sequences computed from the following system of fixpoint equations:

$$\forall i \in \{1, ..., n\}, \forall i \neq j,\ \rho^{\sharp\star}(S_i^\sharp, S_j^\sharp) = \bigsqcup_{k=1}^{n} \rho^{\sharp\star}(S_i^\sharp, S_k^\sharp) \circ \rho^\sharp(S_k^\sharp, S_j^\sharp)$$

$$\rho^{\sharp\star}(S_i^\sharp, S_i^\sharp) = \mathrm{Id}_{S_i^\sharp}^\sharp \sqcup \bigsqcup_{k=1}^{n} \rho^{\sharp\star}(S_i^\sharp, S_k^\sharp) \circ \rho^\sharp(S_k^\sharp, S_i^\sharp)$$

Abstract Summary of a Procedure

Let p be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$. We denote as $I^\sharp = \alpha(I)$ the abstraction of initial states and as $\mathcal{E}^\sharp = \alpha(\mathcal{E})$ the abstraction of exit states.

Definition 4.3.5 (Abstract summary)**.** The abstract summary of a procedure p described by a transition system $(S, \rho, I, \mathcal{E})$ is $\sigma_p^\sharp = \rho^{\sharp\star}(I^\sharp, \mathcal{E}^\sharp)$.

The abstract effect of a call to a procedure p with an abstract parameter passing mechanism $\pi^\sharp : \mathcal{R}^\sharp \to \mathcal{R}^\sharp$, located between $T_i^\sharp$ and $T_j^\sharp$ is $\rho^\sharp(T_i^\sharp, T_j^\sharp) = \pi^\sharp(\sigma_p^\sharp)$.

4.3.2 Building Summaries Using LRA

The convex polyhedra abstract domain is a relational abstract domain abstracting a set of numerical vectors by its convex hull which is its least convex superset. A detailed presentation of the convex polyhedra abstract domain is given in Chapter 2.

Classical abstract domain operations are available, such as intersection ($P_1 \sqcap P_2$), convex hull ($P_1 \sqcup P_2$), effect of variable assignment ($P[x := expr]$), widening ($P_1 \nabla P_2$), test for inclusion ($P_1 \sqsubseteq P_2$) and emptiness ($P = \emptyset$). Instead of using a narrowing operator to ensure the convergence of the decreasing sequence, only a limited number of iterations of the decreasing sequence is computed. It is sound to do so since all terms of the decreasing sequence are post-fixpoints of the abstract transfer function. A projection operation ($\exists X, P$) also termed existential quantification is provided.

As shown in Example 4.3.1, convex polyhedra can represent abstract input-output relations. We introduce additional variables to denote initial values and the source of the relation. We denote as X_0 the finite set of variables representing initial values of variables. We denote as $P(X_0, X)$ the convex polyhedron involving variables in X_0 denoting initial values and variables in X denoting current values.

The source and the target projection of an abstract relation $r^\sharp$ represented by a convex polyhedron can be defined straightforwardly using the projection operator.

Definition 4.3.6. The abstract source projection $src^\sharp : \mathcal{N}^n \to \mathcal{N}^n$ of abstract relations represented by convex polyhedra is defined as:

$$src^\sharp(P(X_0, X)) = \exists X, P(X_0, X)$$

Similarly, we define the abstract target projection $tgt^\sharp : \mathcal{N}^n \to \mathcal{N}^n$ as follows:

$$tgt^\sharp(P(X_0, X)) = \exists X_0, P(X_0, X)$$

Example 4.3.2. The source and target projections of the abstract relations defined in Example 4.3.1 are:

$$\begin{array}{lcl} r_1^\sharp & = & (x = x_0 + 1 \wedge y = y_0 \wedge z_0 \geq z \wedge y \geq 23 \wedge x \geq 1) \\ src^\sharp(r_1^\sharp) & = & (x_0 \geq 0 \wedge y_0 \geq 23) \\ tgt^\sharp(r_1^\sharp) & = & (x \geq 1 \wedge y \geq 23) \end{array}$$

$$\begin{array}{lcl} r_2^\sharp & = & (y = 2y_0 + 1 \wedge x_0 \leq x \leq x_0 + 2y \wedge 1 \leq y \leq z) \\ src^\sharp(r_2^\sharp) & = & (y_0 \geq 0) \\ tgt^\sharp(r_2^\sharp) & = & (y \geq 1 \wedge z \geq y) \end{array}$$

The composition of abstract relations expressed as convex polyhedra can be defined naturally using classical operators.

Definition 4.3.7 (Composition of convex polyhedra)**.** Let $r_1^\sharp$ and $r_2^\sharp$ be abstract relations represented by convex polyhedra $P_1(X_0, X_1)$ and $P_2(X_1, X_2)$ over finite sets of variables X_0, X_1, X_2, X_3. The abstract composition of $r_1^\sharp$ and $r_2^\sharp$ is defined as:

$$r_1^\sharp \circ r_2^\sharp = \exists X_1, (P_1(X_0, X_1) \sqcap P_2(X_1, X_2))$$

We can compute relational procedure summaries using Linear Relation Analysis where procedure summaries are input-output relations represented by convex polyhedra.

We introduce X_0 variables to denote initial values of procedure parameters. For the sake of clarity, we duplicate all procedure parameters for the time being, at least in principle. In practice, we do not have to duplicate a procedure parameter if it is a pure input parameter or a pure output parameter. Following relational collecting semantics, we introduce equalities at procedure entry between X_0 variables denoting initial values and X variables denoting current values of procedure parameters.

The relational summary of a procedure is given by the convex polyhedra obtained at the procedure exit.

4.3.3 Example

We compute an abstract relational summary of the `div` procedure defined in our Euclidean division example. We give the system of abstract fixpoint equations over the convex polyhedra abstract domain associated to the `div` procedure.

```
  void div(int a, int b, int * q, int * r)
  {
      assume(a >= 0 && b >= 1);
1:    *q = 0;
      *r = a;
2:    while
3:        (*r >= b)
      {
4:        *r = *r-b;
          *q = *q + 1;
5:    }
6:}
```

$$
\begin{aligned}
P_1 &= (a_0 \geq 0 \wedge b_0 \geq 1 \wedge a = a_0 \\
&\quad \wedge b = b_0 \wedge q = q_0 \wedge r = r_0) \\
P_2 &= P_1[q := 0][r := a] \\
P_3 &= P_2 \sqcup P_5 \\
P_4 &= P_3 \sqcap (r \geq b) \\
P_5 &= P_4[r := r - b][q := q + 1] \\
P_6 &= P_3 \sqcap (r \leq b - 1)
\end{aligned}
$$

Figure 4.3: System of abstract fixpoint equations associated to the Euclidean division example using LRA.

A standard Linear Relation Analysis, where the widening operator is applied on P_3 during the increasing sequence and the decreasing sequence is limited to 2 iterations, gives the following fixpoint solution for the `div` procedure:

$$
\begin{aligned}
P_1 &= (a_0 \geq 0 \wedge b_0 \geq 1 \wedge a = a_0 \wedge b = b_0 \wedge q = q_0 \wedge r = r_0) \\
P_2 &= (a_0 \geq 0 \wedge b_0 \geq 1 \wedge a = a_0 \wedge b = b_0 \wedge q = 0 \wedge r = a) \\
P_3 &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\
P_4 &= (a = a_0 \wedge b = b_0 \wedge r \geq b \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\
P_5 &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 1 \wedge b \geq 1 \wedge a + 1 \geq b + q + r) \\
P_6 &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq r + 1 \wedge a \geq q + r)
\end{aligned}
$$

The program point 6 is the unique exit point of the `div` procedure. In our system of abstract fixpoint equations, P_6 is the convex polyhedron associated to the exit point of the `div` procedure. Although the non-linear property $a = bq + r$ of Euclidean division can not be expressed in the convex polyhedra abstract domain, we still obtain a rather weak summary for the `div` procedure given by P_6. We shall also remark in P_6 that the precondition $a_0 \geq 0$ has been lost. This suggests that preconditions should be considered more carefully.

4.4 Preconditions

For closed programs, the initial state is generally not relevant, since normally, variables are explicitly assigned an initial value before being used. Even global variables can be considered to be assigned by a global initialization procedure executed right before the main procedure. Global variables may also have a default initial value according to their type in programming languages such as C. From the point of view of a static analysis tool, we can consider that even if these default values are implicit from the programmer side, they can still be seen as explicit variable assignments.

When looking at procedures during a concrete program execution, the initial values of formal parameters comes from the values passed as actual parameters by a calling

procedure. Conversely, when considering procedures individually in themselves, the initial values of parameters are left undetermined and parameters can possibly take initially any value in their semantic domain with respect to their type.

The correct behavior of a procedure often depends on a precondition, which is a property constraining the initial values that parameters are allowed to take. We denote as I_p the concrete precondition of a procedure p. We term as *global precondition* the abstraction $I_p^\sharp$ of the set of legal initial states of a procedure p where $I_p^\sharp = \alpha_{S_\rightarrow}(I_p)$.

We took into account the global precondition $a_0 \geq 0 \wedge b \geq 1$ of the `div` procedure in our Euclidean division example. Such a global precondition can be user-supplied, as a manually given annotation in the program source, or discovered by another analysis from the calling context, or in a weaker form from the types of parameters. The global precondition can also be simply $\top$ representing the total absence of information about initial values of parameters.

4.4.1 Widening Under a Precondition

In a relational analysis, a precondition is an obvious invariant, since it is a property on initial values of procedure parameters and a procedure can not change its initial state. Thus any concrete summary σ_p of a procedure p and any concrete relation $\rho^\star(I_p, S_i)$ has its source within I_p. This is less obvious for an abstract analysis because of the use of widening.

It may happen that the result $r^{\sharp\nabla\triangle}$ of a relational abstract analysis does not satisfy the invariant given by a precondition, when $\gamma(r^{\sharp\nabla\triangle})$ is not included in $I_p \times S$ due to classical widening operators being defined independently of any procedure precondition. This is what happened in our Euclidean division example during the computation of an abstract relational summary of the `div` procedure.

As a consequence, it is both sound and interesting to use a *limited widening* operator, taking into account the procedure precondition during the increasing sequence of the relational abstract analysis to compute $r^{\sharp\nabla}$.

Definition 4.4.1 (Widening limited by precondition)**.** The widening operator $\nabla_{I_p^\sharp} : \mathcal{R}^\sharp \to \mathcal{R}^\sharp$ limited by an abstract precondition $I_p^\sharp \in \mathcal{R}^\sharp$ is defined as:

$$\forall r_1^\sharp, r_2^\sharp \in \mathcal{R}^\sharp, \quad r_1^\sharp \nabla_{I_p^\sharp} r_2^\sharp = (r_1^\sharp \nabla r_2^\sharp) \sqcap I_p^\sharp$$

Example 4.4.1. In our Euclidean division example, when computing a relational abstract summary of the `div` procedure, the classical widening operator was applied on P_3 with $P_3 = P_3 \nabla (P_2 \sqcup P_5)$. Instead, we limit the widening on P_3 by the global precondition of the `div` procedure:

$$P_3 = (P_3 \nabla (P_2 \sqcup P_5)) \sqcap (a_0 \geq 0 \wedge b_0 \geq 1)$$

We compute a new relational summary of the `div` procedure according to the following

system of abstract fixpoint equations:

$$\begin{array}{lcl} I^\sharp & = & (a_0 \geq 0 \wedge b_0 \geq 1) \\ P_1 & = & I^\sharp \sqcap (a = a_0 \wedge b = b_0 \wedge q = q_0 \wedge r = r_0) \\ P_2 & = & P_1[q := 0][r := a] \\ P_3 & = & (P_3 \nabla (P_2 \sqcup P_5)) \sqcap I^\sharp \\ P_4 & = & P_3 \sqcap (r \geq b) \\ P_5 & = & P_4[r := r - b][q := q + 1] \\ P_6 & = & P_3 \sqcap (r \leq b - 1) \end{array}$$

Using Linear Relation Analysis, we obtain the following abstract values for $I^\sharp, P_1, P_2, P_3, P_4, P_5, P_6$:

$$\begin{array}{lcl} I^\sharp & = & (a_0 \geq 0 \wedge b_0 \geq 1) \\ P_1 & = & (a = a_0 \wedge b = b_0 \wedge q = q_0 \wedge r = r_0 \wedge b \geq 1 \wedge a \geq 0) \\ P_2 & = & (r = a \wedge a = a_0 \wedge b = b_0 \wedge q = 0 \wedge b \geq 1 \wedge a \geq 0) \\ P_3 & = & (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\ P_4 & = & (a = a_0 \wedge b = b_0 \wedge r \geq b \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\ P_5 & = & (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 1 \wedge b \geq 1 \wedge a + 1 \geq b + q + r) \\ P_6 & = & (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq r + 1 \wedge a \geq q + r) \end{array}$$

The relational summary σ of the `div` procedure given by P_6 at the unique exit point of the procedure is:

$$\sigma_{div} = (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq r + 1 \wedge a \geq q + r)$$

We have the precondition $a_0 \geq 0$ in σ_{div}. Instead of just recovering the precondition $a_0 \geq 0$, we obtained the stronger constraint $a \geq q + r$. This suggests that using a widening limited by the procedure precondition is not only a technique to preserve and recover the precondition in the procedure summary, and that beyond its initial intended objective it is an effective general precision improvement technique.

4.5 Conclusion

We saw that the relational semantics of a procedure can be derived from the usual state collecting semantics, using additional variables to denote initial values of variables. It is not really new [69] in principle but a new general formalization that we did not find elsewhere.

The relational semantics of a procedure can be used in a forward analysis to discover abstract input-output relations between the possible values of procedure parameters at a given control location and their initial values at procedure entry. The abstract relation discovered at the exit node of a procedure is an approximate relational summary of the procedure behavior.

The approach described in this chapter forms the basis of a general framework to compute relational procedure summaries, which can be instantiated with any relational abstract domain.

We presented the application of this approach to Linear Relation Analysis and the convex polyhedra abstract domain to obtain relational procedure summaries which are

sets of linear constraints. Finally, we discussed the importance of preconditions for precise procedure summaries. Preconditions should be the object of a special attention, and be handled carefully, notably through the use of a limited widening.

A single convex polyhedron is generally not precise enough due to convex approximation. It can not distinguish very different behaviors of procedures. In the next chapter, we propose to compute disjunctive relational procedure summaries driven by preconditions. Disjunctive relational summaries will be finite sets of abstract relations, representing different procedure behaviors, represented by elements of a relational abstract domain.

Chapter 5

Disjunctive Relational Procedure Summaries

We presented the computation of relational summaries based on the relational collecting semantics of procedures. We shown that relational collecting semantics can be defined in terms of the usual collecting semantics of procedures by introducing variables denoting initial values of parameters. As it can be expected, concrete relational summaries are not computable by machine in general. We departed from them and described how abstract relational summaries can be computed effectively, using a classical analysis by abstract interpretation over a relational abstract domain, which is able to discover approximations of relations over states.

We had a particular focus on the use of trace partitioning to compute approximations of reachable relations over states and taking care of preconditions. In this chapter, we propose to refine the partitioning by distinguishing different calling contexts, represented by preconditions. We can see calling contexts as sets of possible initial states of procedures, giving initial values to procedure parameters. Since preconditions are subsets of initial states, they provide a simple way to represent calling contexts.

Usual abstract domains are generally not closed under disjunction, like octagons and convex polyhedra. In a sense, not being able to express disjunctions, joining together disjuncts into a single, possibly greater abstract value, is the essence of abstraction. It is thus quite natural in order to have more precise procedure summaries to consider explicit disjunctions of abstract relations.

Restrictions must be applied to be able to compute on such disjunctions. Moreover, to be able to use a disjunctive procedure summary to analyze a procedure call, the values of actual parameters must determine which disjunct of the summary applies. Thus distinct disjuncts should have disjoint sources.

We present a method to compute disjunctive relational summaries of procedures based on a recursive refinement of preconditions enabling each disjunct to express the behavior of a procedure on a distinct set of calling contexts. We give several heuristics for automatic precondition partitioning. We also show how disjunctive summaries are used to get the abstract effect of a call in the calling procedure.

Finally, we apply our approach to summaries of recursive procedures. As recursive procedures depend on themselves from an interprocedural point of view, the summary of a recursive procedure is computed in terms of itself.

5.1 Motivating Example

We consider the very simple `abs` procedure given in Figure 5.1, which computes the absolute value of its parameter x and stores the result in r. Its control-flow graph is given on the right-hand side.

```
void abs(int * r, int x)
{
    if(x >= 0){
        *r = x;
    } else {
        *r = -x;
    }
}
```

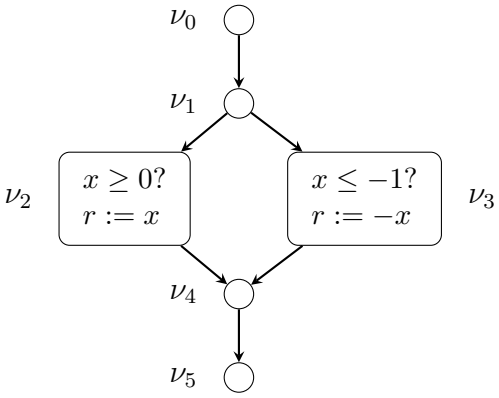


Figure 5.1: The simple `abs` procedure computing the absolute value of the parameter x and storing the result in r.

We compute a relational summary of the `abs` procedure using Linear Relation Analysis. Following the method presented in Chapter 4, we consider this system of abstract fixpoint equations in which a convex polyhedra P_i is associated to each program node ν_i:

$$\begin{aligned} P_0 &= (r = r_0 \wedge x = x_0) \\ P_1 &= P_0 \\ P_2 &= (P_1 \sqcap (x \geq 0))[r := x] \\ P_3 &= (P_1 \sqcap (x \leq -1))[r := -x] \\ P_4 &= P_2 \sqcup P_3 \\ P_5 &= P_4 \end{aligned}$$

We introduced additional variables x_0 and r_0 to denote the initial value of the parameters x and r respectively, and we added equality constraints $x = x_0$ and $r = r_0$ in P_0 to record the initial value of x and r. A classical Linear Relation Analysis gives the following results for P_2, P_3, and P_5:

$$\begin{aligned} P_2 &= (r = x_0 \wedge r = x \wedge r \geq 0) \\ P_3 &= (r = -x_0 \wedge r = -x \wedge r \geq 1) \\ P_5 &= (x = x_0 \wedge r \geq x \wedge r \geq -x) \end{aligned}$$

Since ν_5 is the unique exit node of the `abs` procedure, its relational summary $\sigma_{abs} = P_5$ is as follows:

$$\sigma_{abs} = (x = x_0 \wedge r \geq x \wedge r \geq -x)$$

This is a rather weak summary for such a simple procedure, more so considering the precise relations $r = x$ and $r = -x$ discovered respectively in P_2 and P_3. This loss of precision is due to the join operator applied for P_4, which computes the convex hull of P_2 and P_3. The convex polyhedra abstract domain, as most numerical abstract domains, is not able to represent exactly the disjunction $P_2 \vee P_3$, as it is not a convex set.

Although very simple, the `abs` procedure has two distinct behaviors, represented by P_2 and P_3, which are triggered depending on the initial value of the parameter x respectively when $x \geq 0$ and when $x \leq -1$ initially. A precise procedure summary should express properly these two behaviors in a separate way.

Thus we can see on this simple procedure that we need to keep separate behaviors of procedures which are distinct in some sense, to obtain precise procedure summaries. We propose a method to compute disjunctive relational summaries of procedures, each element of the summary being an abstract input-output relation, denoting a distinct procedure behavior.

5.2 Disjunctive Refinement of Abstract Relations

We described in Chapter 4 how to compute procedure summaries following the relational collecting semantics of procedures. These relational summaries were made of a single abstract input-output relation, represented by an element of a relational abstract domain. We use this approach as a basis for the computation of disjunctive relational summaries.

5.2.1 General Framework

We present the refinement of abstract input-output relations into finite disjunctions of abstract relations.

Definition 5.2.1 (Disjunctive Refinement of an Abstract Relation)**.** Let p be a procedure described by a transition system $(S, \rho, I, \mathcal{E})$ with $I \subseteq S$ being the global precondition of p and $I^\sharp = \alpha(I)$ being the abstraction of the global precondition I. Let $\delta^\sharp = \{S_1^\sharp, ..., S_n^\sharp\}$ be an abstract partition of the set S of states.

A disjunctive refinement of the abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$, for $S_i^\sharp, S_j^\sharp \in \delta^\sharp$, is a finite set $R = \{r_1^{\sharp\star}, ..., r_m^{\sharp\star}\}$ of abstract relations such that:

1. $\forall k \in \{1, ..., m\}, r_k^{\sharp\star} \sqsubseteq \rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$
2. $\forall k_1, k_2 \in \{1, ..., m\}, k_1 \neq k_2 \Rightarrow \gamma(src^\sharp(r_{k_1}^{\sharp\star})) \cap \gamma(src^\sharp(r_{k_2}^{\sharp\star})) = \emptyset$
3. $\bigcup_{k=1}^{m} \gamma(src^\sharp(r_k^{\sharp\star})) = \gamma(S_i^\sharp)$

We assume that, for a given procedure p, we have an abstract partition $\delta^\sharp = \{S_1^\sharp, ..., S_n^\sharp\}$ of the set S of states of the procedure p. This means that each member $S_i^\sharp \in \delta^\sharp$ of the abstract partition $\delta^\sharp$ denotes a part of the set S of states, and that the set $\{\gamma(S_i^\sharp) \mid S_i^\sharp \in \delta^\sharp\}$ of the concretizations $\gamma(S_i^\sharp)$ of each member $S_i^\sharp \in \delta^\sharp$ is a partition of S.

The abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$, for some $S_i^\sharp, S_j^\sharp \in \delta^\sharp$, is the abstract transitive closure between $S_i^\sharp$ and $S_j^\sharp$, denoting all procedure behaviors starting from a state described by $S_i^\sharp$ and ending in a state described by $S_j^\sharp$. Each disjunct $r_k^{\sharp\star}$ of the refinement $R = \{r_1^{\sharp\star}, ..., r_m^{\sharp\star}\}$ of the abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$ is itself an abstract relation, representing a part of these behaviors.

Soundness of the Refinement

We first give a soundness condition for each disjunct $r_k^{\sharp\star} \in R$. We require that each disjunct $r_k^{\sharp\star}$ must be included in the original abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$ where:

$$r_k^{\sharp\star} \sqsubseteq \rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$$

This ensures that each disjunct $r_k^{\sharp\star} \in R$ denotes a part of the procedure behaviors in the original abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$.

Partition of the Source

As we intend to use disjunctions of abstract relations to represent distinct procedure behaviors directed by the initial values of parameters triggering that behavior, the sources of each disjunct $r_k^{\sharp\star} \in R$ must be distinct in some way.

The concrete sources $\gamma(src^\sharp(r_k^{\sharp\star}))$ of each disjunct $r_k^{\sharp\star} \in R$ of the refinement R must be disjoint. This is expressed in Definition 5.2.1 by the following requirement:

$$\forall k_1, k_2 \in \{1, ..., m\}, k_1 \neq k_2 \Rightarrow \gamma(src^\sharp(r_{k_1}^{\sharp\star})) \cap \gamma(src^\sharp(r_{k_2}^{\sharp\star})) = \emptyset$$

The concretization function γ of a Galois connection is known to be a morphism for the greatest lower-bound operator:

$$\forall d_1^\sharp, d_2^\sharp \in \mathcal{R}^\sharp, \gamma(d_1^\sharp \sqcap d_2^\sharp) = \gamma(d_1^\sharp) \cap \gamma(d_2^\sharp)$$

Thus we have fortunately that:

$$\forall r_1^\sharp, r_2^\sharp \in \mathcal{R}^\sharp, \quad src^\sharp(r_1^\sharp) \sqcap src^\sharp(r_2^\sharp) = \bot \Leftrightarrow \gamma(src^\sharp(r_1^\sharp)) \cap \gamma(src^\sharp(r_2^\sharp)) = \emptyset$$

Therefore in practice we can check that the abstract sources of the disjuncts are disjoint in the abstract domain:

$$\forall k_1, k_2 \in \{1, ..., m\}, k_1 \neq k_2 \Rightarrow src^\sharp(r_{k_1}^{\sharp\star}) \sqcap src^\sharp(r_{k_2}^{\sharp\star}) = \bot$$

Finally, the set $\{src^\sharp(r_k^{\sharp\star}) \mid r_k^{\sharp\star} \in R\}$ of the abstract sources of the disjuncts $r_k^{\sharp\star} \in R$ of a refinement R must cover the abstract source $S_i^\sharp$ of the original abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$, from the point of view of the concrete domain in order to encompass all possible initial states of the procedure. This is why the set $\{src^\sharp(r_k^{\sharp\star}) \mid r_k^{\sharp\star} \in R\}$ of abstract sources of the disjuncts must be an abstract partition of $S_i^\sharp$.

Example 5.2.1. We give an example of disjunctive refinement of an abstract relation represented by convex polyhedra. We consider the abstract relation $r_1^\sharp$ given in Example 4.3.1, which is defined as follows:

$$r_1^\sharp = (x = x_0 + 1 \wedge y = y_0 \wedge z_0 \geq z \wedge y \geq 23 \wedge x \geq 1)$$

The abstract source $src^\sharp(r_1^\sharp)$ of $r_1^\sharp$ is:

$$src^\sharp(r_1^\sharp) = (x_0 \geq 0 \wedge y_0 \geq 23)$$

We may want to refine $r_1^\sharp$ according to the initial value of x compared to the initial value of y, leading to two cases for the initial values of parameters x and y, which can be expressed by the inequality constraints $x_0 < y_0$ and $x_0 \geq y_0$. Thus we can refine $r_1^\sharp$ into $r_{1,1}^\sharp$ and $r_{1,2}^\sharp$, according to the constraints $x_0 < y_0$ and $x_0 \geq y_0$ as follows:

$$\begin{array}{lcl} r_{1,1}^\sharp & = & r_1^\sharp \sqcap (x_0 < y_0) = (x = x_0 + 1 \wedge y = y_0 \wedge y \geq x \wedge z_0 \geq z \wedge y \geq 23 \wedge x \geq 1) \\ r_{1,2}^\sharp & = & r_1^\sharp \sqcap (x_0 \geq y_0) = (x = x_0 + 1 \wedge y = y_0 \wedge z_0 \geq z \wedge y \geq 23 \wedge x \geq y + 1) \end{array}$$

We denote this refinement as $R = \{r_{1,1}^\sharp, r_{2,2}^\sharp\}$. The disjuncts $r_{1,1}^\sharp$ and $r_{1,2}^\sharp$ have disjoint abstract sources $src^\sharp(r_{1,1}^\sharp)$ and $src^\sharp(r_{1,2}^\sharp)$:

$$src^\sharp(r_{1,1}^\sharp) \sqcap src^\sharp(r_{1,2}^\sharp) = (x_0 < y_0 \wedge y_0 \geq 23 \wedge x_0 \geq 0) \sqcap (x_0 \geq y_0 \wedge y_0 \geq 23 \wedge x_0 \geq 0) = \bot$$

We should also see that the concretizations $\gamma(src^\sharp(r_{1,1}^\sharp))$ and $\gamma(src^\sharp(r_{1,2}^\sharp))$ of the abstract sources of $r_{1,1}^\sharp$ and $r_{1,2}^\sharp$ are covering $\gamma(src^\sharp(r_1^\sharp))$ as follows:

$$\begin{array}{lcl} \gamma(src^\sharp(r_{1,1}^\sharp)) \cup \gamma(src^\sharp(r_{1,2}^\sharp)) & = & \{(x_0, y_0, z_0) \in \mathbb{Z}^3 \mid x_0 \geq 0 \wedge y_0 \geq 23\} \\ & = & \gamma(src^\sharp(r_1^\sharp)) \end{array}$$

and naturally, as stated earlier, the abstract sources $src^\sharp(r_{1,1}^\sharp)$ and $src^\sharp(r_{1,2}^\sharp)$ are disjoint in the abstract domain:

$$src^\sharp(r_{1,1}^\sharp) \sqcap src^\sharp(r_{1,2}^\sharp) = \bot$$

Disjunctive Refinements as Conjunctions of Implications

A disjunctive refinement $R = \{r_1^\sharp, ..., r_m^\sharp\}$ of an abstract relation $\rho^{\sharp\star}(S_i^\sharp, S_j^\sharp)$ is a finite set of possibly smaller abstract relations $r_1^\sharp, ..., r_m^\sharp$ which can be considered as formulas over variables denoting initial values and variables denoting current values. The disjunctive refinement R can be seen as a disjunctive formula constraining the initial and current values of variables:

$$R = \bigvee_{k=1}^{m} r_k^\sharp$$

Since a disjunctive refinement is directed by the abstract sources of its members which are disjoint, it can be also seen as a conjunction of implications:

$$R = \bigwedge_{k=1}^{m} (src^\sharp(r_k^\sharp) \Rightarrow r_k^\sharp)$$

5.2.2 Refinement by Precondition Partitioning

We are interested in computing disjunctive relational summaries of procedures directed by the initial values of parameters leading to that behavior. Each of these sets of initial values should be a distinct subset $I_k \subseteq I$ of the global precondition I of the procedure. They should form a partition $\delta_I = \{I_1, ..., I_m\}$ of the global precondition I of the procedure.

We should thus construct disjunctive relational procedure summaries from an abstract partition $\delta_I^\sharp = \{I_k^\sharp = \alpha(I_k) \mid I_k \in \delta\}$ of the global precondition I of the procedure.

We can construct a disjunctive refinement $R = \{r_1^{\sharp\star}, ..., r_m^{\sharp\star}\}$ of the abstract transitive closure $\rho^{\sharp\star}(I^\sharp, S_j^\sharp)$ from an abstract partition $\delta_I^\sharp = \{I_1^\sharp, ..., I_m^\sharp\}$ of the global precondition I by computing:

$$r_k^{\sharp\star} = \rho^{\sharp\star}(I_k^\sharp, S_j^\sharp)$$

for each $I_k^\sharp \in \delta_I^\sharp$ of the abstract partition $\delta_I^\sharp$.

Definition 5.2.2 (Precondition partitioning)**.** The disjunctive refinement $R = \{r_1^{\sharp\star}, ..., r_m^{\sharp\star}\}$ of $\rho^{\sharp\star}(I^\sharp, S_j^\sharp)$ according to the abstract partition $\delta_I^\sharp$ of the global precondition I is defined as:

$$\forall k \in \{1, ..., m\}, r_k^{\sharp\star} = \rho^{\sharp\star}(I_k^\sharp, S_j^\sharp)$$

The partitioning of preconditions is used to define disjunctive relational procedure summaries consisting of a finite set of abstract input-output relations. Each abstract relation, as a member of the procedure summary, describes a distinct part of the procedure behavior. It is separated from the others by the particular precondition leading to that behavior. They are covering collectively the global precondition of the procedure. By using an abstract partition of the global precondition, we ensure that all possible initial states of the procedure are accounted for.

We give below a definition of disjunctive relational summaries based on a partitioning of the global precondition.

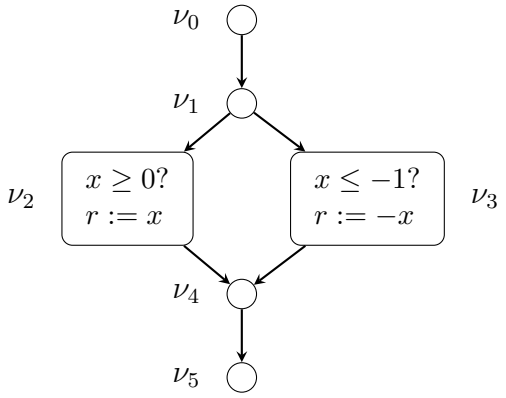
Definition 5.2.3 (Disjunctive relational summary)**.** Let p be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$. Let $\delta_I^\sharp = \{I_1^\sharp, ..., I_m^\sharp\}$ be a finite abstract partition of the global precondition I of the procedure p. The disjunctive relational summary $R_p = \{r_1^\sharp, ..., r_m^\sharp\}$ of the procedure p with respect to the abstract partition $\delta_I^\sharp$ of the precondition I is defined as:

$$\forall r_k^\sharp \in R, \quad r_k^\sharp = \rho^{\sharp\star}(I_k^\sharp, \mathcal{E}^\sharp)$$

Example 5.2.2. We computed previously the following relational summary for the `abs` procedure:

$$\sigma_{abs} = (x = x_0 \wedge r \geq x \wedge r \geq -x)$$

The `abs` procedure has the implicit precondition $I = \mathbb{Z}^2$ which is abstracted by $I^\sharp = \top$ in the convex polyhedra abstract domain.



$$\begin{aligned} P_2 &= (r = x_0 \wedge r = x \wedge r \geq 0) \\ P_3 &= (r = -x_0 \wedge r = -x \wedge r \geq 1) \end{aligned}$$

We can distinguish the two procedure behaviors expressed by P_2 and P_3 according to the sign of the x parameter, respectively when $x \geq 0$ and when $x \leq -1$. We construct an abstract partition $\delta_I = \{I_1^\sharp, I_2^\sharp\}$ of the global precondition I of the `abs` procedure:

$$\delta_I = \{I_1^\sharp, I_2^\sharp\}$$

$$\begin{array}{rcl} I_1^\sharp & = & (x_0 \geq 0) \\ I_2^\sharp & = & (x_0 \leq -1) \end{array}$$

The summary σ_{abs} of the `abs` procedure is an approximation of the transitive closure $\rho^{\sharp\star}(I^\sharp, \mathcal{E}^\sharp)$ with $\rho^{\sharp\star}(I^\sharp, \mathcal{E}^\sharp) \sqsubseteq \sigma_{abs}$. The summary σ_{abs} can be refined into $R = \{r_1^\sharp, r_2^\sharp\}$ where abstract relations $r_1^\sharp$ and $r_2^\sharp$ are obtained by computing approximations of $\rho^{\sharp\star}(I_1^\sharp, \mathcal{E}^\sharp)$ and $\rho^{\sharp\star}(I_2^\sharp, \mathcal{E}^\sharp)$.

Computing summary member $r_1^\sharp$ with precondition $I_1^\sharp$

We compute the summary member $r_1^\sharp$ as a sound approximation of $\rho^{\sharp\star}(I_1^\sharp, \mathcal{E}^\sharp)$. A Linear Relation Analysis of the `abs` procedure under the precondition $I_1^\sharp = (x_0 \geq 0)$ gives the following results:

$$\begin{array}{rcl} P_0 & = & (r = r_0 \wedge x = x_0 \wedge x \geq 0) \\ P_1 & = & (r = r_0 \wedge x = x_0 \wedge x \geq 0) \\ P_2 & = & (r = x_0 \wedge x = x_0 \wedge r \geq 0) \\ P_3 & = & \bot \\ P_4 & = & (r = x_0 \wedge r = x \wedge r \geq 0) \\ P_5 & = & (r = x_0 \wedge r = x \wedge r \geq 0) \end{array}$$

We have $P_3 = \bot$, thus the program point ν_3 is unreachable under $I_1^\sharp$. It is worth to observe and to keep in mind that cleverly chosen preconditions can make unreachable a given program location. The summary member $r_1^\sharp$ under precondition $I_1^\sharp$ is given by P_5 as follows:

$$r_1^\sharp = (r = x_0 \wedge r = x \wedge r \geq 0)$$

Computing summary member $r_2^\sharp$ under precondition $I_2^\sharp$

Similarly, we compute $r_2^\sharp$ as an approximation of $\rho^{\sharp\star}(I_2^\sharp, \mathcal{E}^\sharp)$ using Linear Relation Analysis. The summary member $r_2^\sharp$ under precondition $I_2^\sharp$ is as follows:

$$r_2^\sharp = (r = -x_0 \wedge r = -x \wedge r \geq 1)$$

Disjunctive Relational Summary of the `abs` Procedure

We computed a disjunctive relational summary $R = \{r_1^\sharp, r_2^\sharp\}$ of the `abs` procedure directed by an abstract partition $\delta_I^\sharp = \{I_1^\sharp, I_2^\sharp\}$ of its global precondition $I = \mathbb{Z}^2$ with $I_1^\sharp = (x_0 \geq 0)$ and $I_2^\sharp = (x_0 \leq -1)$. The disjunctive relational summary R of the `abs` procedure is:

$$R = \{r_1^\sharp, r_2^\sharp\}$$

$$\begin{array}{rcl} r_1^\sharp & = & (r = x_0 \wedge r = x \wedge r \geq 0) \\ r_2^\sharp & = & (r = -x_0 \wedge r = -x \wedge r \geq 1) \end{array}$$

5.2.3 Abstract Effect of a Call

We show how to use the disjunctive summary R of a procedure p to get the abstract effect of a call to p in the calling procedure. We denote as $\pi^\sharp : \mathcal{R}^\sharp \to \mathcal{R}^\sharp$ a parameter passing mechanism which renames, in abstract relations, formal parameters into actual parameters.

Definition 5.2.4 (Abstract Effect of a Call)**.** Let p be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$. Let $R = \{r_1^\sharp, ..., r_m^\sharp\}$ be a disjunctive relational summary of the procedure p. The abstract effect of a call to the procedure p situated between $T_i^\sharp$ and $T_j^\sharp$ with a parameter-passing mechanism $\pi^\sharp : \mathcal{R}^\sharp \to \mathcal{R}^\sharp$ is defined as follows:

$$\rho(T_i^\sharp, T_j^\sharp) = \bigsqcup_{r_k^\sharp \in R} \pi(r_k^\sharp)$$

Example 5.2.3. The effect Q' of the call statement `abs(&r,&x)` on the convex polyhedron $Q = (x \geq 10)$ is computed as follows:

$$\begin{array}{rcl} Q' & = & (\exists x_0, Q[x/x_0] \sqcap r_1^\sharp) \sqcup (\exists x_0, Q[x/x_0] \sqcap r_2^\sharp) \\ & = & (\exists x_0, (x_0 \geq 10) \sqcap (r = x_0 \wedge r = x \wedge r \geq 0)) \\ & & \sqcup (\exists x_0, (x_0 \geq 10) \sqcap (r = -x_0 \wedge r = -x \wedge r \geq 1)) \\ & = & (r = x \wedge r \geq 10) \sqcup \bot \\ & = & (r = x \wedge r \geq 10) \end{array}$$

5.2.4 Application to Linear Relation Analysis

We are now interested in discussing some specific considerations for the construction and manipulation of disjunctive relational summaries, particularly their application to Linear Relation Analysis with abstract relations being represented by convex polyhedra.

Disjunctive Polyhedral Summaries

Let p be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$. Let X be the finite set of the formal parameters of p. Let $I^\sharp$ be its polyhedral abstract global precondition with $I = \alpha(I^\sharp)$.

A disjunctive relational summary of the procedure p is a finite set of convex polyhedra $R = \{R_1, ..., R_m\}$ where each member $R_k \in R$ of the summary is a convex polyhedron representing an abstract input-output relation. The finite set $\{I_k\}_{k \in \{1,...,m\}}$ of convex polyhedra defined as:

$$\forall k \in \{1, ..., m\}, \quad I_k^\sharp = src^\sharp(R_k) = \exists X, R_k$$

constitutes a finite abstract partition of the global precondition I of the procedure p. This is analogous to what we introduced for general abstract relations, but now stated specifically for convex polyhedra. We should note that the abstract source operation $src^\sharp$ is expressed through the variable elimination operation on convex polyhedra.

Polyhedron Transformer of a Procedure Call

Let q be a procedure calling the procedure p. We assume that a call to the procedure p in the procedure q is denoted as a special edge (n_i, n_j) between two program nodes n_i and n_j in the control-flow graph of the procedure q. This configuration is illustrated in Figure 5.2. The procedure p has a finite sequence $X = (x_1, .., x_n)$ of formal parameters and the calling procedure q supplies the finite sequence $A = (a_1, .., a_n)$ of actual parameters in the call to procedure p, which is labeled on the edge (n_i, n_j) as follows:

$$call\ p(a_1, ..., a_n)$$

For a discussion of our assumptions regarding procedure parameters, see Section 4.2.2. We assume that the length of the sequence A of actual parameters supplied by the calling procedure q is equal to the length of the sequence X of formal parameters expected by the called procedure p. We also assume that actual parameters are distinct variables.

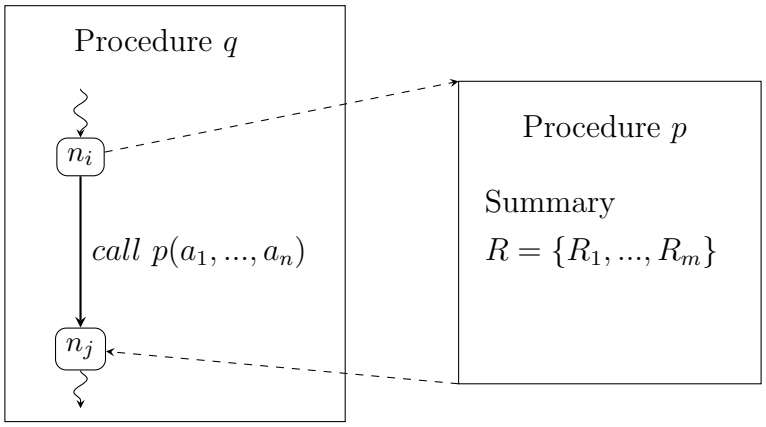


Figure 5.2: Graphical illustration of a procedure q calling a procedure p with actual parameters $A = (a_1, ..., a_n)$.

We denote as Q_i the convex polyhedron associated to n_i in the calling procedure q right before the call to p and as Q_j the convex polyhedron associated to n_j just after the call. In order to avoid a top-down analysis of the procedure p each time the control-flow edge (n_i, n_j) is examined, we compute the convex polyhedron Q_j after the call in terms of the summary R of p and the convex polyhedron Q_i discovered before the call.

The application of a disjunctive summary involves a substantial amount of variable renaming to encode the parameter passing mechanism.

Given the sequence $A = (a_1, ..., a_n)$ of actual parameters, we introduce an auxiliary sequence $A^0 = (a_1^0, ..., a_n^0)$ of variables denoting the values of actual parameters just before the call to the procedure p.

We transform the calling context Q_i into a property $Q_i[A/A^0]$ on variables denoting actual parameters before the call.

Then, we match A^0 variables denoting initial values of actual parameters in the input property $Q_i[A/A^0]$ with the X^0 variables denoting initial values of formal parameters in each summary member $R_k \in R$. We denote as $R_k[X^0/A^0]$ the renaming, in a summary member $R_k \in R$, of each variable $x_0 \in X^0$ into $a_i^0 \in A^0$.

Finally, we also rename the X variables denoting final values of procedure parameters into variables denoting the values of actual parameters after the call to p. For each summary member $R_k \in R$, we compute:

$$R_k[X^0/A^0][X/A]$$

Definition 5.2.5 (Polyhedron Transformer of a Procedure Call)**.** The convex polyhedron Q_j associated to n_j after the call to the procedure p in the procedure q is defined as:

$$Q_j = \bigsqcup_{k=1}^{m} (\exists A^0, Q_i[A/A^0] \sqcap R_k[X^0/A^0][X/A])$$

It collects the contributions of each summary member R_k of the called procedure p, which are constrained by the calling context Q_i. A renaming of variables ties actual parameters in the caller with the formal parameters of the called procedure.

5.3 Partition Refinement

We saw how to compute disjunctive relational summaries of procedures given an abstract partition of the global precondition of a procedure. This abstract partition of the global precondition will be sometimes simply termed the *partition* of the summary.

The precondition of a procedure is the set of admissible initial values of parameters. We get this global precondition either in a very simple way, from the the types of parameters, or right from above, in the form of a user-given annotation. In the worst case, or in the absence of any specific property of initial values of parameters, the global precondition of a procedure can be chosen soundly to be the universal condition.

When we defined disjunctive relational summaries, we assumed that we had, given before our hands, an abstract partition of the global precondition of a procedure. We discuss how the abstract partition of the global precondition can be computed effectively.

Different objectives can be pursued regarding the computation of partitions of preconditions. They are used to separate procedure behaviors which would involve a great loss of precision if they would have been represented by a single element of a relational abstract domain.

Thus we may be interested in choosing an abstract partition which tries to maximize the separation of highly different procedure behaviors, in order to avoid that loss of precision. We also want to limit the size of disjunctive relational summaries, with a regard to the number of analyses which are computed for a given procedure. So, we have here, as in some other occasions, a tradeoff between expressivity and cost. We propose several heuristics corresponding to various intents and tradeoffs.

5.3.1 Refinement Heuristics

We can split recursively the global precondition of a procedure, into new preconditions, which when taken back together, cover all the behaviors which were previously described by the global precondition alone.

We say that these new preconditions *refine* the global precondition of the procedure. We call *refinement* the process of splitting a precondition and *refinement heuristic* the algorithm or scheme describing this process.

Definition 5.3.1 (Refinement of a Precondition)**.** Let p be a procedure described by a transition system $(S, \rho, I, \mathcal{E})$. Let $I^\sharp$ be the global abstract precondition of the procedure p, such that $I \subseteq \gamma(I^\sharp)$. Let $A^\sharp$ be an abstract precondition of p such that $A^\sharp \sqsubseteq I^\sharp$.

Two preconditions $A_1^\sharp$ and $A_2^\sharp$ are said to refine the precondition $A^\sharp$ if they are covering $A^\sharp$ as follows:

$$A_1^\sharp \sqcup A_2^\sharp = A^\sharp$$

We do not require in the previous definition that the two preconditions $A_1^\sharp$ and $A_2^\sharp$ don't intersect since we are only interested in their covering of the abstract precondition $A^\sharp$ in order for the admissible initial states denoted by $A^\sharp$ to be preserved by the refinement.

5.3.2 Refinement According to Local Reachability

We present a first refinement heuristic called refinement according to *local reachability*. Its overall principle is to refine a precondition according to the abstract relation discovered at a given control point of a procedure, and particularly according to the information relative to the conditions under which that control point may be possibly reachable or not.

Intuitively, for a given abstract relation $r_k^\sharp$ discovered at a program point ν_k in a procedure p, the abstract source $src^\sharp(r_k^\sharp)$ of $r_k^\sharp$ is a property on initial values of parameters, which have been propagated along paths in the procedure during a relational analysis. Since the abstract source $src(r_k^\sharp)$ is a property on the initial states of the procedure p, it is thus a precondition of p. More precisely, the abstract source $src(r_k^\sharp)$ is a *necessary abstract precondition* for the program point ν_k to be reachable in the procedure p.

Complementable Abstract Values

We want to characterize procedure behaviors where a given control point is not reachable. We believe that, for some well-chosen control points in the procedure, the separation of procedure executions according to whether that particular program point is reachable or not can lead to great insights on the procedure behavior. We present the principles enabling that separation.

As the abstract precondition $I_k^\sharp = src^\sharp(r_k^\sharp)$ is a necessary precondition for the control point ν_k to be reachable, we would like to compute a precondition under which ν_k is unreachable as some sort of negation of $I_k^\sharp$. This negation operation is the *complementation* of an abstract value, and the values for which such a complement value exists are said to be complementable.

Definition 5.3.2 (Complementable abstract value)**.** Let $(D^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be an abstract domain related to a concrete domain $(D, \subseteq, \cup, \cap, \emptyset)$ by a Galois connection (α, γ). An abstract value $d^\sharp \in D^\sharp$ is said to be complementable if there exists an abstract value $\overline{d^\sharp} \in D^\sharp$ termed its complement such that:

1. $d^\sharp \sqcap \overline{d^\sharp} = \bot$
2. $\gamma(d^\sharp) \cup \gamma(\overline{d^\sharp}) = D$

We can note that $d^\sharp$ being disjoint from its complement $\overline{d^\sharp}$ implies that this is also the case in the concrete domain, such that $\gamma(d^\sharp) \cap \gamma(\overline{d^\sharp}) = \emptyset$.

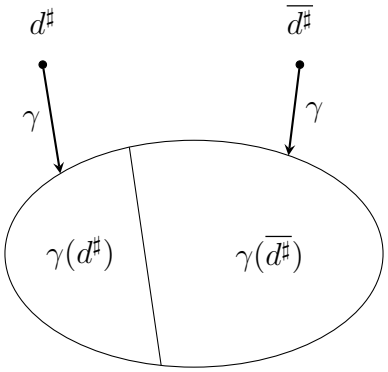


Figure 5.3: Illustration of the complement $\overline{d^\sharp}$ of an abstract value $d^\sharp$.

Example 5.3.1. In the convex polyhedra abstract domain, only half-spaces are complementable, which are the abstract values defined by a single linear inequality. We give below some examples of complementable abstract values, along with their complement, in the convex polyhedra abstract domain. All involved variables are integers.

$$\begin{array}{lcll} d_1^\sharp & = & (x < y) & \overline{d_1^\sharp} = (x \geq y) \\ d_2^\sharp & = & (x < C) & \overline{d_2^\sharp} = (x \geq C) \qquad \text{where } C \text{ is a constant} \\ d_3^\sharp & = & (2x + 3y \leq z + t) & \overline{d_3^\sharp} = (2x + 3y \geq z + t + 1) \end{array}$$

Refinement of the Global Precondition

As before, we denote as $I_k^\sharp = src^\sharp(r_k^\sharp)$ the necessary abstract precondition for a control point ν_k in the procedure p to be reachable. A sufficient precondition $I'^\sharp$ for ν_k to be unreachable in the procedure p can be obtained if there exists a complementable abstract value $s^\sharp \in D^\sharp$ containing $I_k^\sharp$ with $I_k^\sharp \sqsubseteq s^\sharp$ defined as:

$$I'^\sharp = I^\sharp \sqcap \overline{s^\sharp}$$

For the convex polyhedra abstract domain, it means that the complementable abstract value $s^\sharp$ is an half-space, in which $I_k^\sharp$ is included, since only half-spaces are complementable for convex polyhedra. Therefore $s^\sharp$ can be a constraint of the convex polyhedron $I_k^\sharp$.

Definition 5.3.3 (Refinement According to Local Reachability)**.** Let p be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$. Let $I^\sharp$ be the abstract global precondition of the procedure p, such that $I = \gamma(I^\sharp)$. Let $\delta^\sharp = \{S_1^\sharp, ..., S_n^\sharp\}$ be a finite abstract partition of the set S of states.

Let $\{r_k^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp)\}_{k=1...n}$ be the result of a relational analysis of the procedure p from the abstract precondition $I^\sharp$, such that each abstract relation $r_k^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp)$ of the result is an approximation of $\rho^{\sharp\star}(I^\sharp, S_k^\sharp)$.

For a given member $S_k^\sharp \in \delta^\sharp$ of the abstract partition $\delta^\sharp$ of S, the abstract source $I_k^\sharp = src^\sharp(r_k^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp))$ of the abstract relation $r_k^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp)$ discovered for $S_k^\sharp$ is a necessary precondition for $S_k^\sharp$ to be reachable. If $s^\sharp$ is a complementable abstract value such that:

1. $I_k^\sharp \sqsubseteq s^\sharp$

2. $I'^\sharp = I^\sharp \sqcap \overline{s^\sharp} \neq \bot$ and $I''^\sharp = I^\sharp \sqcap s^\sharp \neq \bot$

then $I'^\sharp$ and $I''^\sharp$ are refining the global abstract precondition $I^\sharp$. The condition $I'^\sharp$ is a sufficient precondition for $S_k^\sharp$ to be unreachable.

We denote as $\mathrm{RLR}_k(I^\sharp) = \{I'^\sharp, I''^\sharp\}$ a refinement according to local reachability of the global abstract precondition $I^\sharp$ into the two new abstract preconditions $I'^\sharp$ and $I''^\sharp$, with respect to the abstract relation $r_k^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp)$ discovered for S_k by a relational analysis starting with $I^\sharp$.

The member $S_k^\sharp$ of the abstract partition $\delta^\sharp$ of the set S of states of the procedure p can be the set of all states associated to a given control location ν_k in p. In that case $r^{\sharp\nabla\triangle}(I^\sharp, S_k^\sharp)$ is the abstract relation discovered by a classical relational analysis at ν_k. We only use the abstract relation $r_k^\sharp$ discovered at a given control location ν_k by a relational analysis to refine the precondition $I^\sharp$, hence the term of refinement according to local reachability.

Algorithm

The refinement according to local reachability can be seen as consisting of the following steps:

1. Analyze the procedure p with a classical relational analysis, starting with an abstract precondition $I^\sharp$. We denote as $\{r_k^\sharp\}_{k=1,..,n}$ the results discovered for each control point $\nu_1, ..., \nu_n$ of the procedure p.
2. Choose a control point ν_k in the procedure p such that $r_k^\sharp \neq \bot$.
3. Choose a separating abstract value $s^\sharp \in D^\sharp$, which is complementable, such that $src(r_k^\sharp) \sqsubseteq s^\sharp$, for which the following properties hold:

$$I'^\sharp = I^\sharp \sqcap s^\sharp \neq \bot$$
$$I''^\sharp = I^\sharp \sqcap \overline{s^\sharp} \neq \bot$$

We hold the intuition that the reachability of some well chosen control points in a procedure, and the conditions leading to it, denote interesting cases of the procedure behavior. With respect to callers of the procedure, we would like that calling contexts have an empty intersection with as much summary members as possible, to lessen the loss of precision during the computation of the effect of a call.

5.3.3 Refinement According to the Summary of a Called Procedure

In the previous heuristic, we compute a refinement of the precondition of a procedure p from the results of an analysis of p, with the intent to separate, by an explicit disjunction of calling contexts, some interesting behaviors of p, with the hope that they may represent typical cases of the procedure usage by its potential callers.

The effect of a call to a procedure with a disjunctive summary $R = \{\sigma_k^\sharp\}_{k=1,...,m}$ defined in 5.2.5 involves a least upper-bound in the calling procedure which is likely to lose precision. Thus it can be interesting to refine the partitioning of the caller to split this least upper-bound according to the disjunctive summary of the called procedure.

Applicability of Summary Members

We are interested in deriving conditions under which a given summary member $\sigma_k^\sharp$ is applicable at a call site in a caller, with respect to the possible values of actual parameters.

A summary member $\sigma_k^\sharp \in R$ of a procedure p is an abstract input-output relation, respectively between initial and final values of formal parameters of the called procedure p. The abstract source $src^\sharp(\sigma_k^\sharp)$ is thus a condition on the formal parameters of the procedure p for $\sigma_k^\sharp$ to be applicable. More properly, the formal parameters satisfy the property $tgt^\sharp(\sigma_k^\sharp)$ at the end of an execution of the procedure p, if their initial values satisfy the condition $src^\sharp(\sigma_k^\sharp)$.

From the point of view of the caller, the actual parameters satisfy the property $\pi^\sharp(tgt^\sharp(\sigma_k^\sharp))$ under the condition $\pi^\sharp(src^\sharp(\sigma_k^\sharp))$ where $\pi^\sharp$ is a parameter-passing mechanism. We can say that the effect on actual parameters in the caller, described by $\pi^\sharp(\sigma_k^\sharp)$, only happen under the condition $\pi^\sharp(src^\sharp(\sigma_k^\sharp))$, on the values of actual parameters just before the call. Let us denote as $\mathcal{J}_k^\sharp = \pi^\sharp(src^\sharp(\sigma_k^\sharp))$ the condition on actual parameters for $\sigma_k^\sharp$ to be applicable at a given call site in the calling procedure.

Refinement of the Global Precondition

We want to refine the abstract precondition $I^\sharp \in D^\sharp$ of the caller according to whether the summary case $\sigma_k^\sharp$ is applicable or not.

Let n_i be a control point in a procedure calling the procedure p just before the call to p. The control point n_i will also be termed a call site to the procedure p. This situation is illustrated in Figure 5.2.

Using a classical relational analysis, $r_i^\sharp$ is the abstract relation discovered at n_i before the call p under the abstract precondition $I^\sharp$. Then $I_k^\sharp = src^\sharp(r_i^\sharp \sqcap \mathcal{J}_k)$ is a necessary precondition of the caller for the condition $\mathcal{J}_k$ to be satisfiable, and thus for the summary case $\sigma_k^\sharp$ to be applicable at the call site n_i.

We thus have a way to express the applicability of a summary case $\sigma_k^\sharp$ of a called procedure in terms of a precondition $I_k^\sharp$ of the caller.

As for the refinement according to local reachability, we can separate the executions of the caller in which the summary case $\sigma_k^\sharp$ is applicable, from the executions in which it is not, by splitting the abstract precondition $I^\sharp$ of the caller according to a complementable abstract value $s^\sharp \in D^\sharp$ such that $I_k^\sharp \sqsubseteq s^\sharp$. The condition $I_k^\sharp$ is entirely contained in $s^\sharp$ and thus the summary case $\sigma_k^\sharp$ is applicable, $\mathcal{J}_k$ is satisfied, when $s^\sharp$ holds. As a consequence, conversely, the summary case $\sigma_k^\sharp$ do not apply when the complement $\overline{s^\sharp}$ is satisfied.

We can split the precondition $I^\sharp$ of q into two new preconditions $I'^\sharp$ and $I''^\sharp$ according to the separating value $s^\sharp$ and its complement $\overline{s^\sharp}$. The separation of $I^\sharp$ into $I'^\sharp$ and $I''^\sharp$ is defined as:

$$I'^\sharp = I^\sharp \sqcap \overline{s^\sharp}$$
$$I''^\sharp = I^\sharp \sqcap s^\sharp$$

Definition 5.3.4 (Refinement According to the Summary of Called Procedures)**.** Let p be a procedure with a disjunctive relational summary $R_p = \{\sigma_k^\sharp\}_{k=1,\ldots,m}$. Let q be a procedure represented by a transition system $(S, \rho, I, \mathcal{E})$ calling the procedure p. Let $I^\sharp$ be the abstract global precondition of the procedure q where $\gamma(I^\sharp) = I$. Let $\pi^\sharp : D^\sharp \to D^\sharp$ be the parameter-passing mechanism corresponding to the call of p.

Let $\sigma_k^\sharp \in R_p$ be a member of the summary of p and let J_k denote the precondition of $\sigma_k^\sharp$ with $J_k = \pi^\sharp(src^\sharp(\sigma_k^\sharp))$. In the calling procedure q, $I_k^\sharp = src^\sharp(r^{\sharp\nabla\triangle}(I^\sharp, J_k))$ is a necessary precondition of q for J_k to be satisfiable and the condition for $\sigma_k^\sharp$ to be applicable in the calling procedure q. If $s^\sharp \in D^\sharp$ is a complementable abstract value such that:

$$I_k^\sharp \sqsubseteq s^\sharp$$
$$I_k^{'\sharp} = I^\sharp \sqcap \overline{s^\sharp} \neq \bot \wedge I_k^{''\sharp} = I^\sharp \sqcap s^\sharp \neq \bot$$

then $I^{'\sharp}$ and $I^{''\sharp}$ are refining the global precondition $I^\sharp$ of the procedure q. The condition $I^{'\sharp}$ is a sufficient precondition of q for $\mathcal{J}_k$ to be unsatisfiable.

We denote as $\text{RSCP}_k^\sharp(R_p, I^\sharp) = \{I^{'\sharp}, I^{''\sharp}\}$ a refinement of the abstract precondition $I^\sharp$ of a calling procedure according to the summary case $\sigma_k^\sharp \in R_p$ of the summary R_p of the called procedure p.

Algorithm

The refinement of the abstract precondition $I^\sharp$ of a procedure q according to the summary of a called procedure p can be seen as consisting of the following steps:

1. Analyze the calling procedure q with a classical relational analysis, starting with an abstract precondition $I^\sharp$. We denote as $\{r_c^\sharp\}_{c=1,\ldots,n}$ the results obtained for each control point of the procedure q.
2. Choose a control point c in the procedure q located before a call to the procedure p, such that $r_c^\sharp \neq \bot$.
3. Choose a member $\sigma_k^\sharp \in R_p$ of the disjunctive summary R_p of the called procedure p.
4. Compute $\mathcal{J}_k = \pi^\sharp(src^\sharp(\sigma_k^\sharp))$ and $I_k^\sharp = src^\sharp(r^{\sharp\nabla\triangle}(I^\sharp, \mathcal{J}_k))$.
5. Choose a separating abstract value $s^\sharp \in D^\sharp$, which is complementable, such that $I_k^\sharp \sqsubseteq s^\sharp$, for which the following properties hold:

$$I^{'\sharp} = I^\sharp \sqcap s^\sharp \neq \bot$$
$$I^{''\sharp} = I^\sharp \sqcap \overline{s^\sharp} \neq \bot$$

An example of refinement according to summaries of called procedures is given in 5.5.

5.3.4 Iterative Refinement

We presented two heuristics in the previous sections to refine the global precondition of a procedure. The refinement according to local reachability and the refinement according to the summaries of called procedures are embodying different ways to compute an abstract partition of a precondition. These partitioning heuristics can be used iteratively, in a repeated fashion, to obtain more and more refined abstract partitions of the global precondition.

Our proposal is to build the summary of a procedure as a sequence of analyses, working on more and more refined partitions of the global precondition. Each analysis starts on a refined abstract precondition, obtained by one of our partitioning heuristics, and its results are used in turn by the heuristics, to split further the precondition. Once the procedure has been analyzed according to each member of the current abstract partition,

the partitioning heuristics are used to derive a new abstract partition, by refining the preconditions, using the analysis results obtained at the current step.

The disjunctive relational summary of the procedure is constructed by collecting the abstract relations discovered at the exit point of the procedure, respectively under each precondition in the current abstract partition. If the procedure has several distinct exit points, the abstract relations discovered at these exit points may be collapsed into a single relation by the join operator of the relational abstract domain. As the abstract partition of the global precondition of the procedure is refined further, the disjunctive summary computed for the procedure becomes itself more and more refined and precise.

The iterative construction of procedure summaries can be seen as a sequence of steps. We can denote as $\mathcal{P}^{(\ell)} = \{I_k^{\sharp(\ell)}\}_{k=1,\ldots,m_\ell}$ the abstract partition of the global precondition of a procedure at the ℓ-th step. We start initially with $\mathcal{P}^{(0)} = \{I^\sharp\}$ containing only the global abstract precondition $I^\sharp$. At subsequent steps, for each ℓ, we analyze the procedure with a relational analysis respectively for each precondition $I_k^{\sharp(\ell)}$ in the current abstract partition $\mathcal{P}^{(\ell)}$, computing the abstract relations $\{r_k^{\sharp(\ell)}\}_{k=1,\ldots,m_\ell}$ as a result, which are then used to refine $\mathcal{P}^{(\ell)}$ into $\mathcal{P}^{(\ell+1)}$ through the refinement techniques presented before.

Algorithm

We present the iterative construction of procedure summaries in a more rigorous way, in the form of the pseudocode given in Algorithm 1.

Algorithm 1 Construction of the disjunctive summary of a procedure p by iterative refinement of the global abstract precondition $I^\sharp$.

procedure BUILD_SUMMARY($I^\sharp, \theta$)
  $\mathcal{P}^{(0)} \leftarrow \{I^\sharp\}$
  $\ell \leftarrow 0$
  repeat
    $\mathcal{P}^{(\ell+1)} \leftarrow \emptyset$
    for all $I_k^\sharp \in \mathcal{P}^{(\ell)}$ **do**
      $r_k^{\sharp(\ell)} \leftarrow$ Analyze the procedure p with a relational analysis starting with $I_k^\sharp$
      $\mathcal{S}^{(\ell)} \leftarrow \mathcal{S}^{(\ell)} \cup \{r_k^{\sharp(\ell)}(I_k^\sharp, \mathcal{E}^\sharp)\}$
      Ref $\leftarrow$ REFINE($I_k^\sharp, r_k^{\sharp(\ell)}$)
      $\mathcal{P}^{(\ell+1)} \leftarrow \mathcal{P}^{(\ell+1)} \cup$ Ref
    end
    $\ell \leftarrow \ell + 1$
  until $(\mathcal{P}^{(\ell)} = \mathcal{P}^{(\ell-1)}) \vee (|\mathcal{S}^{(\ell)}| = \theta)$
  return $\mathcal{S}^{(\ell)}$
end

The BUILD_SUMMARY procedure computes a disjunctive relational summary of a procedure p with a global abstract precondition $I^\sharp \in D^\sharp$.

The computation of a disjunctive summary of the procedure p works iteratively as a sequence of steps. Each step ℓ produces a disjunctive summary $\mathcal{S}^{(\ell)}$ of p, according to a current abstract partition $\mathcal{P}^{(\ell)}$ of the global precondition of p. The current abstract

partition $\mathcal{P}^{(\ell)}$ is then possibly refined by one of the partition refinement heuristics presented earlier, which is denoted as REFINE in the above pseudocode, and which should be instantiated by the actually used heuristic, such that refinement according to local reachability (denoted as RLR) or refinement according to summaries of called procedures (denoted as RSCP). A new abstract partition $\mathcal{P}^{(\ell+1)}$ is constructed, which will be used in the next step.

During an ℓ-th step of the BUILD_SUMMARY procedure, the next abstract partition $\mathcal{P}^{(\ell+1)}$ is set to be empty initially, at line 6, and then, in a pointwise way, for each $I_k^\sharp$ in the current partition $\mathcal{P}^{(\ell)}$, the procedure p is analyzed under the precondition $I_k^\sharp$ at line 8 by some classical relational analysis, such as LRA, which provides $r_k^{\sharp(\ell)}$ as a result. If we assume that there is an abstract partition $\delta^\sharp = \{S_1^\sharp, ..., S_m^\sharp\}$ of the set S of states of the procedure p and that each $S_i \in \delta$ is the set of states associated to some control point n_i in the procedure p, then the abstract relation discovered by the relational analysis at point n_i can be expressed as $r_k^{\sharp(\ell)}(I_k^\sharp, S_i^\sharp)$.

The analysis results $r_k^{\sharp(\ell)}$ are used by the refinement heuristic denoted by REFINE at line 9, possibly producing a set of new abstract preconditions refining the current abstract precondition $I_k^\sharp$. There are two possible outcomes for the result Ref of the refinement heuristic:

1. The refinement criteria of the heuristic used are satisfied by the current precondition $I_k^\sharp$ and the abstract relations $r_k^{\sharp(\ell)}$ discovered for the procedure p and the refinement heuristic produces a set $\text{Ref} = \{I'^\sharp, I''^\sharp\}$ of new abstract preconditions refining the precondition $I_k^\sharp$.
2. In the case where the heuristic can not refine the precondition $I_k^\sharp$, when its refinement criteria can not be satisfied in any way, then it returns a singleton set containing only the precondition $I_k^\sharp$ itself, with $\text{Ref} = \{I_k^\sharp\}$, indicating that no refinement of $I^\sharp$ is possible by the chosen heuristic.

The result of the refinement heuristics denoted by Ref is then added to the next abstract partition $\mathcal{P}^{(\ell+1)}$ at line 10.

Termination of the Refinement

This iterative refinement process can continue for a while, as it is not guaranteed to terminate in general. However, since the disjunctive summary $\mathcal{S}^{(\ell)}$ of the procedure p computed at the end of each ℓ-th step is sound, this refinement process can be stopped arbitrarily at any step. From a correctness standpoint, the termination of the BUILD_SUMMARY procedure is therefore not an issue. It is thus only a matter of desired precision and expressivity of the resulting disjunctive summary, for the refinement process to be continued and for the refinement steps to be further pursued.

Moreover, for any practical purposes, we would want to have reasonably-sized summaries, particularly regarding the number of necessary analyses for a given procedure. Each summary member adds an additional abstract value that will potentially become an operand of the join operator, which is costly for convex polyhedra, when computing the effect of call statements in callers.

There is a tradeoff that must be made in practice regarding the maximum cardinality of a disjunctive summary that we allow to be constructed by the BUILD_SUMMARY procedure. Since the refinement process can be stopped after any step and the computed

summary still being sound, we can just stop the refinement process whenever the current computed summary $\mathcal{S}^{(\ell)}$ exceeds this fixed maximum cardinality. The maximum cardinality of a disjunctive summary is considered to be a parameter of our approach in practice, that will be denoted as θ. We will term as *θ-limiting* the technique consisting of stopping the refinement process whenever the size of the summary becomes equal to the θ parameter.

The refinement heuristics that we proposed earlier attempt to refine an abstract precondition using the results of a relational analysis of the procedure, by searching for a control point n_i at which some refinement criterion is satisfied. Usually, many control points of the procedure can satisfy the refinement criterion, producing potentially different refinements of the precondition, with various levels of interest and usefulness.

Because we wanted our refinement heuristics to be as generic as possible, we did not place any formal requirement on the choice of such a control point in the case where several distinct points could satisfy the heuristic criterion. One may choose for example, to refine repeatedly a precondition according to the same control point in the procedure. This is especially why, in the presence of loops, we can not guarantee in general that the iterative refinement process terminates.

We should note however, that the refinement process terminates if we restrict ourselves to choose at most once a given control point along a chain of successive refinements of an abstract precondition $I^\sharp$. This is especially the case, as we do so in practice, if we examine once only control points which are direct successors of conditions in the procedure.

Ensuring the Monotonicity of the Refinement

By splitting a precondition, we intend to express more precisely the behavior of a procedure specialized to these two new preconditions. We hoped intuitively that a relational analysis would provide more precise results given more precise preconditions. However, this is not guaranteed because of the non-monotonicity of the widening operator.

Thus at a step ℓ of the iterative refinement in the BUILD_SUMMARY procedure, it is both sound and interesting to use, in the relational analysis of the procedure p at line 8, a widening limited by the result $r^{\sharp(\ell-1)}$ obtained at the previous step. The limited widening ∇_ℓ ensuring the monotonicity of the refinement process is defined as:

$$\forall r_1^\sharp, r_2^\sharp \in D^\sharp, \quad r_1^\sharp \ \nabla_\ell \ r_2^\sharp = \begin{cases} r_1^\sharp \nabla r_2^\sharp & \text{if } \ell = 0 \\ (r_1^\sharp \nabla r_2^\sharp) \sqcap r^{\sharp(\ell-1)} & \text{if } \ell \geq 1 \end{cases}$$

where ∇ is the widening operator used by the underlying relational analysis, which can be itself a limited widening.

Using such a limited widening is especially interesting for the construction of summaries of recursive procedures, and avoids the difficulties addressed in [8] regarding non-monotonic systems of fixpoint equations.

Example 5.3.2. We compute a disjunctive relational summary of the `div` procedure, which was presented in Section 4.3.3.

The relational analysis of the `div` procedure, starting from the abstract precondition $I^\sharp = (a_0 \geq 0 \wedge b_0 \geq 1)$, provides for the branches of the loop condition at program points

```
  void div(int a, int b, int * q, int * r)
  {
      assume(a >= 0 && b >= 1);
1:    *q = 0;
      *r = a;
2:    while
3:        (*r >= b)
      {
4:
          *r = *r-b;
          *q = *q + 1;
5:
      }
6:}
```

$$
\begin{aligned}
P_1 &= (a_0 \geq 0 \wedge b_0 \geq 1 \wedge a = a_0 \\
&\quad \wedge b = b_0 \wedge q = q_0 \wedge r = r_0) \\
P_2 &= P_1[q := 0][r := a] \\
P_3 &= P_2 \sqcup P_5 \\
P_4 &= P_3 \sqcap (r \geq b) \\
P_5 &= P_4[r := r - b][q := q + 1] \\
P_6 &= P_3 \sqcap (r \leq b - 1)
\end{aligned}
$$

4 and 6, the following solutions:

$$
\begin{aligned}
P_4(I^\sharp) &= (a = a_0 \wedge b = b_0 \wedge r \geq b \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\
P_6(I^\sharp) &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge b \geq r + 1 \wedge a \geq q + r)
\end{aligned}
$$

The program point 4 is right at the entry of the while loop in the `div` procedure and the program point 6 is at the exit of the while loop. Since the while loop is the main control structure of interest in the `div` procedure, we believe that the entry and the exit of the while loop, at program points 4 and 6 respectively, can be relevant control points to refine the abstract precondition $I^\sharp$, following the spirit of the refinement according to local reachability. We are interested in the conditions, on the initial values of the procedure parameters, which can make these control points potentially reachable or conversely, unreachable. These conditions can be derived from the sources of the abstract relations discovered at program points 4 and 6, represented respectively by the convex polyhedra $P_4(I^\sharp)$ and $P_6(I^\sharp)$.

The projection of these solutions on the variables denoting initial values of parameters are as follows:

$$
\begin{aligned}
src^\sharp(P_4(I^\sharp)) &= (a_0 \geq b_0 \wedge b_0 \geq 1) \\
src^\sharp(P_6(I^\sharp)) &= (a_0 \geq 0 \wedge b_0 \geq 1)
\end{aligned}
$$

We have at program point 6 that $src^\sharp(P_6(I^\sharp)) = I^\sharp$, thus it can not induce any refinement of $I^\sharp$. However, we have the interesting constraint $a_0 \geq b_0$ in $src^\sharp(P_4(I^\sharp))$ at program point 4. More importantly, we have $src^\sharp(P_4(I^\sharp)) \neq I^\sharp$, thus we can try to apply the criterion of refinement according to local reachability at program point 4.

We can choose the constraint $s^\sharp = (a_0 \geq b_0)$ as a complementable abstract value to refine $I^\sharp$. As $s^\sharp$ is a constraint of $src^\sharp(P_4(I^\sharp))$, then $src^\sharp(P_4(I^\sharp))$ is included in the half-space represented by $s^\sharp$, and thus $src^\sharp(P_4(I^\sharp)) \sqsubseteq s^\sharp$. We can refine the abstract precondition $I^\sharp$ into $I_1^{\sharp(1)}$ and $I_2^{\sharp(1)}$ which are defined as follows:

$$
\begin{aligned}
I_1^{\sharp(1)} &= I^\sharp \sqcap s^\sharp \\
&= (a_0 \geq b_0 \wedge b_0 \geq 1)
\end{aligned}
$$

$$
\overline{s^\sharp} = (a_0 < b_0 \wedge a_0 \geq 0)
$$

$$\begin{aligned} I_2^{\sharp(1)} &= I^\sharp \sqcap \overline{s^\sharp} \\ &= (a_0 < b_0 \wedge a_0 \geq 0) \end{aligned}$$

We can obviously check that the criterion of the RLR heuristic holds as we have both $I_1^{\sharp(1)} \neq \bot$ and $I_2^{\sharp(1)} \neq \bot$. $I_1^{\sharp(1)}$ corresponds to the case where the loop is entered at least once and $I_2^{\sharp(1)}$ corresponds to the case where the loop is never entered.

Then, we analyze the `div` procedure using Linear Relation Analysis under the new preconditions $I_1^{\sharp(1)}$ and $I_2^{\sharp(1)}$ respectively. We get the following results for program points 4 and 6:

$$\begin{aligned} P_4(I_1^{\sharp(1)}) &= (a = a_0 \wedge b = b_0 \wedge r \geq b \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\ P_6(I_1^{\sharp(1)}) &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge q + r \geq 1 \wedge b \geq r + 1 \\ &\quad \wedge a + 1 \geq b + q \wedge a \geq b) \\ P_4(I_2^{\sharp(1)}) &= \bot \\ P_6(I_2^{\sharp(1)}) &= (a = a_0 \wedge b = b_0 \wedge b - 1 \geq a \wedge a \geq 0 \wedge q = 0 \wedge r = a) \end{aligned}$$

We can use these results to further refine the partitioning of the global abstract precondition $I^\sharp$. We can apply again the RLR heuristic. The projection of these solutions on the variables denoting initial values of parameters are:

$$\begin{aligned} src^\sharp(P_4(I_1^{\sharp(1)})) &= (a_0 \geq b_0 \wedge b_0 \geq 1) = I_1^{\sharp(1)} \\ src^\sharp(P_6(I_1^{\sharp(1)})) &= (a_0 \geq b_0 \wedge b_0 \geq 1) = I_1^{\sharp(1)} \\ src^\sharp(P_4(I_2^{\sharp(1)})) &= \bot \\ src^\sharp(P_6(I_2^{\sharp(1)})) &= (a_0 < b_0 \wedge a_0 \geq 0) = I_2^{\sharp(1)} \end{aligned}$$

The abstract sources $src^\sharp(P_4(I_1^{\sharp(1)}))$ and $src^\sharp(P_6(I_1^{\sharp(1)}))$ are equal to the precondition $I_1^{\sharp(1)}$ itself. Similarly, $src^\sharp(P_6(I_2^{\sharp(1)}))$ is equal to the precondition $I_2^{\sharp(1)}$. Thus according to criterion of the refinement according to local reachability, the abstract preconditions $I_1^{\sharp(1)}$ and $I_2^{\sharp(1)}$ can not be further refined. Finally, the partition of the global abstract precondition $I^\sharp$ of the `div` procedure is $\mathcal{P} = \{I_1^{\sharp(1)}, I_2^{\sharp(1)}\}$. The disjunctive summary $\mathcal{S} = \{R_1, R_2\}$ of the `div` procedure associated to the partition $\mathcal{P}$ has thus two members R_1 and R_2 computed as follows:

$$R_1 = P_6(I_1^{\sharp(1)}) \quad R_2 = P_6(I_2^{\sharp(1)})$$

The disjunctive relational summary $\mathcal{S} = \{R_1, R_2\}$ of the `div` procedure is:

$$\begin{aligned} R_1 &= (a = a_0 \wedge b = b_0 \wedge r \geq b \wedge q \geq 0 \wedge b \geq 1 \wedge a \geq q + r) \\ R_2 &= (a = a_0 \wedge b = b_0 \wedge r \geq 0 \wedge q \geq 0 \wedge q + r \geq 1 \wedge b \geq r + 1 \\ &\quad \wedge a + 1 \geq b + q \wedge a \geq b) \end{aligned}$$

5.4 A Last Improvement: Postponing Loop Feedback

The summary $\mathcal{S}$ of the `div` procedure was partitioned according to whether the loop is entered at least once, which is the situation represented by R_1, or never entered, which is represented by R_2. However, in the R_1 case, if the loop is entered at least once, then q should be incremented at least once and we should have at least $q \geq 1$. As we can see in

R_1, this is a fact missed by the analysis. More specifically, the convex polyhedron R_1 is not included in the half-space denoted by the constraint $q \geq 1$.

We could recover this fact by systematically unrolling once each loop that gives rise to such a partitioning. So, in our example, we could first unroll once the while loop and get the constraint $q \geq 1$ by a Linear Relation Analysis performed afterward. We propose however another, cheaper solution, based on trace partitioning. The problem comes from the least upper-bound computed at loop entry, as denoted in the abstract fixpoint equation:

$$P_3 = P_2 \sqcup P_5$$

which is then followed in the abstract equation for P_6 by the test on the loop condition:

$$P_6 = P_3 \sqcap (r \leq b - 1)$$

The loss of precision caused by the least upper-bound operator happens before the test $r \leq b - 1$ has any chance to improve the result. The solution consists in permuting the least upper-bound and the test, computing instead P_6 as follows:

$$P_6 = (P_2 \sqcap (r \leq b - 1)) \sqcup (P_5 \sqcap (r \leq b - 1))$$

Using this new abstract equation for P_6, the summary member R_1 becomes:

$$R_1 = (a = a_0 \wedge b = b_0 \wedge a_0 \geq b_0 \wedge b_0 \geq 1 \wedge r \geq 0 \wedge q \geq 1 \wedge b \geq r + 1 \wedge a + 1 \geq b + q + r)$$

We now have in R_1 the constraint $q \geq 1$ which was lost by the previous analysis. Furthermore, once again, we recovered more precision than expected, since in addition to finding $q \geq 1$, the constraint $a + 1 \geq b + q$ has been strengthened into $a + 1 \geq b + q + r$, which ties nicely together all parameters of the `div` procedure.

We should note that this rewriting of the system of abstract fixpoint equations can be also obtained by a program transformation applied on the procedure, transforming each loop of the form `while c do S` into a do-while loop `if c {do S while c}` guarded by the condition c. This transformation, often applied by compilers, is called *loop inversion*.

Therefore, beyond the particular example, our postponing of loop feedback can be a general precision improvement technique for loops.

5.5 Summaries of Recursive Procedures

The relational abstract interpretation of recursive procedures was proposed a long time ago in [35, 39, 60]. It involves the use of widening, since the summary of a recursive procedure depends on itself. Moreover, a collection of mutually recursive procedures must be analyzed simultaneously, with widening applied to a cut-set of their call graph.

In this section, we show on a simple example how our technique can be applied to the construction of the disjunctive relational summary of a recursive procedure. Since the procedure is recursive, its summary will have to be used to analyze procedure calls during its own construction, and thus this will present an opportunity to refine the procedure summary according to itself. In its own recursive way, it will illustrate the refinement according to the summary of a called procedure.

Example 5.5.1. The `f91` procedure is an implementation of the well-known *91 function* defined by John McCarthy in [89, 88], which is a classical test case for the analysis of recursive functions. For simplicity in this example, we do not duplicate parameters, knowing that x is a value parameter, whose value is thus never changed, and that y is a pure result parameter, whose initial value is never accessed.

```
void f91(int x, int * y) {
    int z, t;
    if(x > 100){ *y = x-10;
1:
    } else {
        z = x + 11;
        f91(z, &t);
        f91(t, y);
2:
    }
}
```

Figure 5.4: Definition of the `f91` procedure.

The polyhedral summary $R(x, y)$ of the `f91` procedure can be defined by the following system of abstract fixpoint equations:

$$\begin{array}{lcl} R(x,y) & = & P_1 \sqcup P_2 \\ P_1 & = & (x \geq 101 \wedge y = x - 10) \\ P_2 & = & (x \leq 100 \sqcap (\exists t, R(x + 11, t) \sqcap R(t, y))) \end{array}$$

The summary R is used in its own definition in the abstract equation defining P_2 since there are two recursive calls to the `f91` procedure.

We can use a classical Linear Relation Analysis to compute approximate solutions of this system of abstract fixpoint equations. The global abstract precondition $I^\sharp$ of the `f91` procedure is $I^\sharp = \top$. Without partitioning and after one widening step, we get:

$$\begin{array}{lcl} P_1 & = & (x \geq 101 \wedge y = x - 10) \\ P_2 & = & (x \leq 100 \wedge y + 9 \geq x \wedge y \geq 91) \\ R^{(0)} & = & (x \leq y + 10 \wedge y \geq 91) \end{array}$$

We can use the refinement according to local reachability to refine the global abstract precondition $I^\sharp$. We compute the abstract sources of P_1 and P_2 and we get:

$$\begin{array}{lcl} src^\sharp(P_1) & = & (x \geq 101) \\ src^\sharp(P_2) & = & (x \leq 100) \end{array}$$

Since the abstract source $src^\sharp(P_1) = (x \geq 101)$ of P_1 is a single linear constraint, it is a complementable abstract value that we can use to split the precondition $I^\sharp$. We refine the global abstract precondition $I^\sharp$ into two new preconditions $I_1^{\sharp(1)}$ and $I_2^{\sharp(1)}$ defined as follows:

$$\begin{array}{lcl} I_1^{\sharp(1)} & = & (x \geq 101) \\ I_2^{\sharp(1)} & = & (x \leq 100) \end{array}$$

We analyze the `f91` procedure with respect to this obvious partitioning, under each new precondition $I_1^{\sharp(1)}$ and $I_2^{\sharp(1)}$ successively. Under the precondition $I_1^{\sharp(1)}$, we get:

$$\begin{array}{lcl} P_1(I_1^{\sharp(1)}) & = & (x \geq 101 \wedge y = x - 10) \\ P_2(I_1^{\sharp(1)}) & = & \bot \\ R^{(1)}(I_1^{\sharp(1)}) & = & (x \geq 101 \wedge y = x - 10) \end{array}$$

and likewise under the precondition $I_2^{\sharp(1)}$:

$$\begin{array}{lcl} P_1(I_2^{\sharp(1)}) & = & \bot \\ P_2(I_2^{\sharp(1)}) & = & (x \leq 100 \wedge y \geq 91) \\ R^{(1)}(I_2^{\sharp(1)}) & = & (x \leq 100 \wedge y \geq 91) \end{array}$$

The results are not much better, we still have only the inequality $y \geq 91$ for the value of the parameter y in $R^{(1)}(I_2^{\sharp(1)})$. We refined the summary of the `f91` procedure according to local reachability and our source of interesting control points for refinement has dried out. No control point satisfies the refinement criterion of the RLR heuristic anymore.

The summary of the `f91` procedure can be refined further according to the summary of the called procedure, which is here the `f91` procedure itself. We can refine the precondition $I_2^{\sharp(1)}$ according to the condition $s^\sharp = (x + 11 \geq 101)$ at the first recursive call. We obtain two new abstract preconditions $I_1^{\sharp(2)}$ and $I_2^{\sharp(2)}$ which are defined as follows:

$$\begin{array}{lcl} I_1^{\sharp(2)} & = & (90 \leq x \leq 100) \\ I_2^{\sharp(2)} & = & (x \leq 89) \end{array}$$

We obtain the following disjunctive summary for the `f91` procedure:

$$\begin{array}{lcl} R^{(1)}(I_1^{\sharp(1)}) & = & (x \geq 101 \wedge y = x - 10) \\ R^{(2)}(I_1^{\sharp(2)}) & = & (90 \leq x \leq 100 \wedge y = 91) \\ R^{(2)}(I_2^{\sharp(2)}) & = & (x \leq 89 \wedge y = 91) \end{array}$$

This our most precise summary for the `f91` procedure. We now have precise constraints on the value of each parameter in every summary member. We have particularly that $y = 91$ in both $R^{(2)}(I_1^{\sharp(2)})$ and $R^{(2)}(I_2^{\sharp(2)})$.

Even if there are seemingly two cases in the behavior of the `f91` procedure, as denoted explicitly by the two parts of the `if` statement, our disjunctive summary of the `f91` procedure features three cases. Thus our approach was able to uncover non-trivial procedure behavior which was not explicitly written in the procedure definition.

5.6 Conclusion

Procedures can commonly have very different behaviors, which are not precisely expressed by a single element of a relational abstract domain like convex polyhedra. We extended the approach presented in Chapter 4 to compute disjunctive relational summaries of procedures based on a partitioning of a procedure precondition. Disjunctive summaries are finite sets of abstract relations represented by elements of a relational abstract domain.

Several partitioning heuristics are given to compute automatically an abstract partition of a precondition by iterative refinement. We also proposed a last improvement to summary computation itself to improve the precision of summaries. Since recursive procedures are often not supported by existing interprocedural analysis, we shown that our approach can discover interesting disjunctive summaries for recursive procedures. Our approach computes an analysis of a procedure for each precondition in the summary partition. Thus we may wonder how it compares in practice with respect to other interprocedural analysis. We give some experimental results in Chapter 6 using the MARS static analyzer.

Chapter 6

Implementation and Experiments

We describe in this chapter the design and implementation of our static analysis tool, called MARS. The MARS (Mars Abstract interpretation Research System) system is a static analysis framework for the analysis of C programs by abstract interpretation, with numerical relational abstract domains like convex polyhedra. With a strong emphasis on modularity and separation of concerns, enabling the change of multiple aspects of an analysis, the MARS static analysis system has been designed to foster the experimentation of new interprocedural analyses, new iteration strategies and new abstract domains. We present also some experiments of our approach, using the MARS static analyzer on a set of benchmark programs.

6.1 The MARS Static Analyzer

The MARS static analyzer is organized as a collection of tools computing numerical invariants of programs written in a significant subset of C. A frontend tool based on CLANG [80] and LIBTOOLING translates the abstract syntax tree of each C source file passed as input into the MARS intermediate representation. The analyzer tool of MARS, which is called `mars`, computes numerical invariants at each program point of interest in the intermediate representation. The implementation of numerical abstract domains, is provided by the Apron library [74].

The MARS intermediate representation is the common exchange format between the different tools. It is generated by the `marsc` frontend tool from a C source file given as input, which can be then processed by the `mars` analyzer tool. The results computed by the analyzer tool are associated to intermediate representation objects, such as program nodes and procedures.

The MARS intermediate representation is designed specifically to ease the implementation of new analyses by abstract interpretation. It is tailored to the needs of numerical analyses such as Linear Relation Analysis. At the same time, it provides also a general representation of programs, capable of representation many programming constructs commonly found in imperative languages like C. With numerical abstract interpretation in mind, the MARS intermediate representation eliminates the unnecessary features usually found in compiler frameworks, such as LLVM IR, and provides a clean idealized view of programs to numerical static analyses.

More specifically, programs are collections of procedures represented by a control-flow

graph which is not based on an SSA-form, in order to offer high-precision source traceback information associated to analysis results.

Another key objective of the design of MARS is to provide very precise source locations, tracing every entity in the MARS intermediate representation back to the original C source code as given by the user, including across complex preprocessor usage, benefiting from the rich location information available in the Clang AST (Abstract Syntax Tree). It contrasts with other open-source static analysis frontends such as CIL [104], which takes as input C programs which have already been processed by the C preprocessor. The source locations given by CIL are only relative to the preprocessed C source. A more detailed presentation of the MARS intermediate representation can be found in 6.3.

C programs given as inputs are significantly processed by the `marsc` frontend to generate the intermediate representation. The `mars` analyzer tool is implemented in the OCAML programming language and consists of approximately 4000 lines of code. The `marsc` frontend tool, which is based on Clang, is implemented in C++ and contains approximately 20000 lines of code on its own, excluding Clang itself.

6.1.1 Basic Usage

We describe some basic usage of the MARS tools to provide a brief practical overview of our static analysis system.

A C file named `program.c` is translated into the MARS intermediate representation by the `marsc` frontend using the command:

```
marsc program.c
```

The `marsc` tool creates an output directory, named by default `out-program.c`, in the current working directory. The output directory holds the intermediate representation of the input program encoded in the JSON (Javascript Common Object Notation) format, along with all the results computed for that program and the files produced by any analysis of the program. The output directory serves as a common storage location in the user filesystem for the MARS tools and as the single source of truth for a given program.

The `mars` analyzer tool can be used to analyze the program using the intermediate representation generated previously and possibly previous analysis results stored in the output directory. It is run using the following command:

```
mars -o out-dir -t analysis_name -m main
```

where the output directory `out-dir` for the program is specified by the `-o out-dir` option, the name of the particular analysis to be run is given by the `-t analysis_name` option and the name of the program entry procedure is given by the `-m main` option.

Analysis results can be found in the output directory encoded in JSON, or depending on the analysis implementation, displayed as an annotated C source file on the standard output.

6.2 Design and Implementation

The MARS system is structured as a collection of tools based on a common set of OCaml modules. We give an overview of how these tools work together in Figure 6.1.

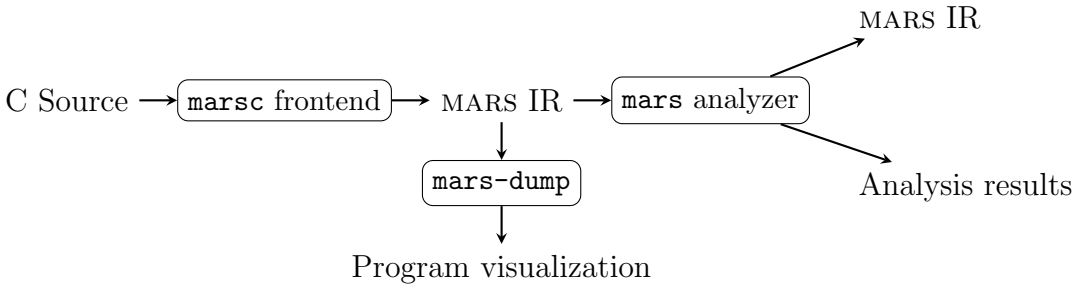


Figure 6.1: Overview of the MARS tools.

The `mars` tool is a modular static analyzer which runs analyses by abstract interpretation on a program expressed in the MARS intermediate representation. It is based on a generic OCaml functor module named `AbstractInterpreter`, which takes as parameters a module implementing an abstract domain, along with a module implementing transfer functions on that domain, and a module describing an iteration strategy. The abstract domain module must satisfy the `Domain` signature, while the transfer functions module must satisfy the `Transfer` signature and the module giving the iteration strategy must satisfy the `Scheduler` signature. This architecture is similar to other modular static analyzers such as INFER [29].

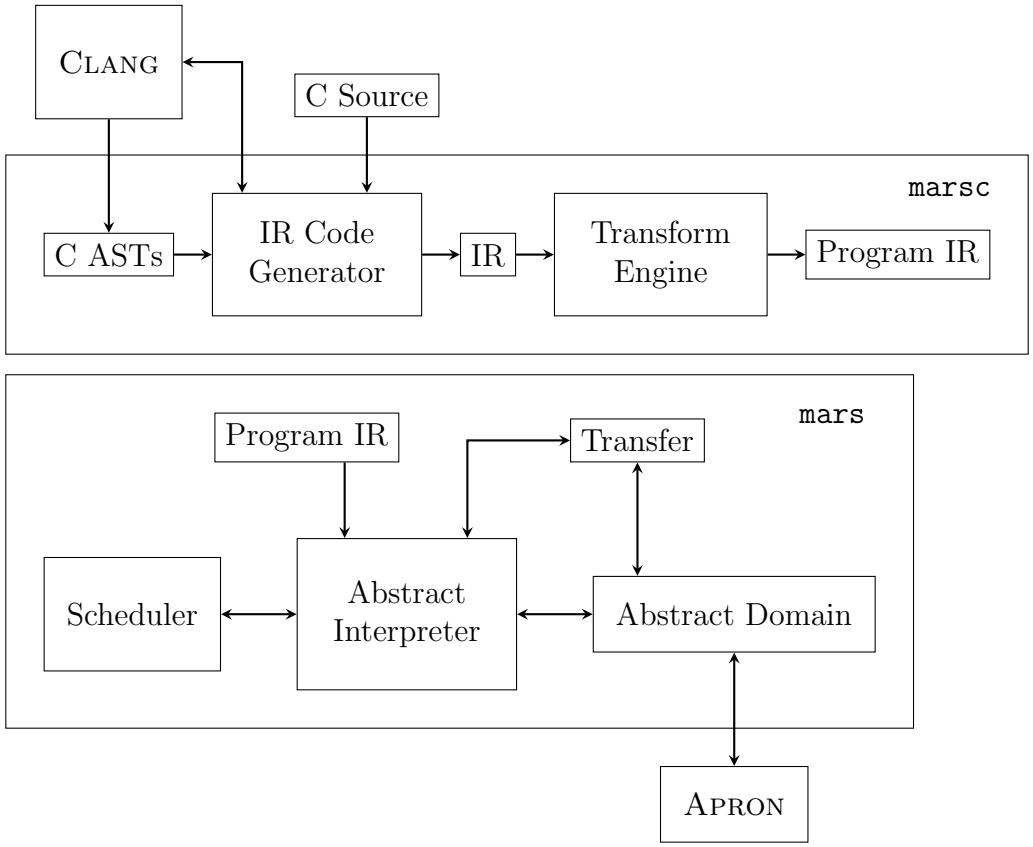


Figure 6.2: Architecture of the `marsc` frontend and the `mars` analyzer.

6.2.1 Abstract Domains

We assume that the abstract domain is a lattice. The operators of the abstract domain are given classically as functions in the abstract domain module, as specified by the `Domain` signature given in Figure 6.3.

```
module type Domain =
sig
    (* Type of abstract domain elements *)
    type t

    (* Type of the domain manager *)
    type man

    (* Create the domain manager *)
    val create : unit -> man

    (* Operators *)
    val bottom : man -> t
    val top : man -> t
    val join : man -> t list -> t
    val meet : man -> t -> t -> t
    val widening : man -> t -> t -> t
    val is_bottom : man -> t -> bool
    val is_top : man -> t -> bool
    val is_leq : man -> t -> t -> bool
end
```

Figure 6.3: Signature for abstract domain modules.

Abstract domain implementations can have a *manager* data structure, of type `man` in the `Domain` signature, to hold the context usually required by abstract domain libraries such as Apron.

6.2.2 Scheduler

A module providing an iteration strategy for the `AbstractInterpreter` functor is termed a *Scheduler*. Scheduler modules must be conforming to the `Scheduler` signature, which is given in Figure 6.4.

A scheduler module provides an `extract` function which is called by the abstract interpreter to get the next program node to be examined during an analysis, deciding of the order in which program nodes are visited. The Scheduler has access to the map associating the current abstract value to a given program node, as computed so far by the abstract interpreter. It can thus base its decision on abstract values associated to nodes and express non-trivial iteration strategies such as the guided static analysis technique [53]. After the update of a node, the abstract interpreter calls the `schedule` function,

with a boolean parameter indicating whether the abstract value of the current node has changed, which triggers the update of the successors of the current node.

The default scheduler module of MARS uses the information given by Bourdoncle's algorithm [23, 24] on strongly-connected components of a procedure, with a priority queue, to decide of the exploration order of program nodes.

We made the iteration strategy an explicit parameter of the `AbstractInterpreter` functor as we wanted to experiment with various ways to explore new paths and nodes in a program, to improve analysis precision, which was the subject of our previous work [26]. The iteration strategy is usually not a changeable parameter in modular denotational static analyzers such as [101], which are based on a recursive exploration of the abstract syntax tree of procedures.

```
module type Scheduler =
sig
    module Domain : Domain.S

    type t

    val create : IR.t
               -> Proc.t
               -> Domain.t StateMap.t
               -> Domain.t
               -> t
    val is_empty : t -> bool
    val extract : t -> Node.t
    val do_widening : t-> Node.t -> bool
    val schedule : t -> Node.t -> bool -> unit
end
```

Figure 6.4: Signature for the scheduler modules.

6.2.3 Transfer Function Module

A `Transfer` module defines a transfer function `eval_node` for the evaluation of nodes in a procedure over some abstract domain. The signature which must be satisfied by a `Transfer` module is given in Figure 6.5.

Along with the incoming abstract value to the current node, the `eval_node` function must be given a `proc_data` data structure which can be used to store procedure summaries. An interprocedural analysis, such as our approach, will store the computed procedure summaries in `proc_data`, which are implemented by the `Summary` module, and that the matching `Transfer` module will be able to use for the evaluation of call statements. A strictly intraprocedural analysis on the other side can use a dummy `proc_data` structure, parameterized by the `NoSummary.t` type (which is defined to be `unit`).

```
module type Transfer =
sig
    module Domain : Domain.S
    module Summary : Summary.S
        with module Domain = Domain

    type proc_data = Summary.t ProcData.t

    val eval_node : IR.t
                  -> Node.t
                  -> proc_data
                  -> Domain.t
                  -> Domain.t
end
```

Figure 6.5: Signature for transfer function modules.

6.3 The MARS Intermediate Representation

We want analyses to be free from concerns about some features of real-world languages such as C and particularly memory manipulation constructs like pointer dereferencing and pointer arithmetics.

We consider that memory manipulation constructs, particularly operation on pointers, should be handled by specialized pointer and alias analyses, transforming a procedure accessing memory through pointers into a procedure with abstract memory locations, manipulated through scalar variables and references. It enables numerical analyses to focus on their main concern, which is the analysis of numerical statements in a procedure. We implement such an analysis, in a simple way for the time being, in the `marsc` frontend.

6.3.1 Abstract Syntax

A program in the MARS intermediate representation is a collection of procedures.

Procedures $p \in Procs$

A procedure p can be seen formally as a tuple of the form:

$$p = (id, form, d_v, g)$$

where $id \in$ Id is the name of the procedure, $form \in Decl_p$ is a sequence of formal parameter declarations, $d_v \in Decl_v$ is a sequence of local variable declarations and $g \in$ CFG is the control-flow graph of the procedure.

Formal Parameter Declaration $form \in Decl_p$

Formal parameters declarations follow the following rule:

$$form ::= \epsilon \mid id : pmode\ t; form$$

A declaration of a formal parameter id is of the form $id : pmode\ t$ where $pmode$ is the parameter-passing mode of the parameter id and t is its type. For simplicity, parameters are passed by reference.

Parameter Mode $pmode \in PModes$

The mode of a formal parameter indicates whether, a given parameter is either a pure-value parameter (**in** mode), a pure-result parameter (**out** mode) or a value-result parameter (**in-out** mode).

$$PModes = \{\textbf{in}, \textbf{out}, \textbf{in-out}\}$$

The value of a pure-value parameter can be only used but not modified, the value of a pure-result parameter can not be used in the procedure before being set and the value of value-result parameter can be both used and modified in the procedure.

Local Variable Declaration $d_v \in Decl_v$

A sequence of declarations of local variables in a procedure is of the following form:

$$d_v ::= \epsilon \mid id : t; d_v$$

Control-Flow Graphs $g \in \text{CFG}$

The control-flow graph of a procedure is a graph (N, E) where:

- The finite set N of nodes is made of three types of nodes: entry nodes, junction nodes and statement nodes.
- $E \in N \times N$ is the set of control-flow edges.

Each node has a single output, possibly leading to several successor nodes. There are no explicit control-transfer instructions. Classical test nodes are split into several conditions appearing in statement nodes. Their associated transfer functions intersect their argument with the condition of the test.

We define the function $nsuccs : Nodes \to \mathcal{P}(Nodes)$ associating to a node the set of its successor nodes and the function $npreds : Nodes \to \mathcal{P}(Nodes)$ associating to a node the set of its predecessors.

$$\forall n \in Nodes, nsuccs(n) = \{s \in N \mid (n, s) \in E\} \quad npreds(n) = \{p \in N \mid (p, n) \in E\}$$

Entry Nodes $Entries$

An entry node $n \in Entries$ has no predecessor node and a single successor node.

$$\forall n \in Entries, npreds(n) = \emptyset \wedge |nsuccs(n)| = 1$$

Statement nodes $Stmts$

A statement node $n \in Stmt$ has a single predecessor node and possibly many successor nodes.

$$\forall n \in Stmts, |npreds(n)| = 1$$

Statement nodes contain a sequence of statements interpreted sequentially.

Statements can be of one of the following kind:

- Deterministic assignment: $x := e$ where e is an expression.
- Non-deterministic assignment: $x := \mathit{undet}$. The transfer function associated to a non-deterministic assignment discards all information available on the variable x in its argument.
- Assume statement: **assume** *cond*. The transfer function associated to an assume statement intersects its argument with the condition *cond*.

The non-deterministic assignment $x := \mathit{undet}$ can be used to handle conservatively in the intermediate representation the unsupported constructs of the source language.

Junction nodes *Junctions*

A junction node can have several predecessor nodes.

$$\forall n \in \mathit{Stmts}, |\mathit{npreds}(n)| \geq 1$$

6.3.2 Translation of Tests

Test statements in the source language, such as *if* statements in C, are encoded by a splitting of the test condition, through **assume** *cond* statements in statement nodes denoting the several possible outcomes of a test condition. We give examples below.

Example 6.3.1 (Simple *if* statement)**.** This simple *if* statement is encoded by the fragment of the MARS intermediate representation given in the right-hand side.

```
if(x >= y)
{
    x = x - c;
}
else
{
    x = x + c;
}
```

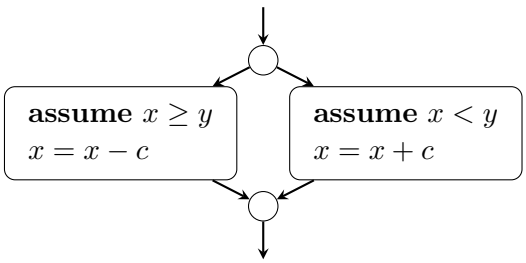


Figure 6.6: Translation of a simple if statement.

Example 6.3.2 (Ternary split of conditions)**.** Equality tests are encoded by a ternary split of the test condition, such that each condition in the generated assume statements has a linear comparison operator. This transformation is aimed at improving the precision of numerical analyses like Linear Relation Analysis.

```
if(x != y)
{
    z = x+c;
}
else
{
    z = 42;
}
```

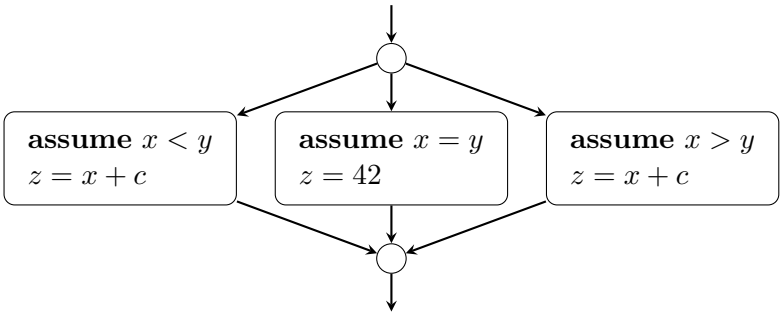


Figure 6.7: Translation of an if statement by a ternary split of the condition.

More complex conditions are translated, in a classical fashion, into a DAG of assume statements, to respect the short-circuit evaluation semantics of boolean operators in C.

Example 6.3.3. We give an example of the translation of an *if* statement with a condition containing boolean operators, respecting the semantics of short-circuit evaluation in C.

```
if((x <= y) || (x <= z))
{
    z = 42;
}
else
{
    z = 0;
}
```

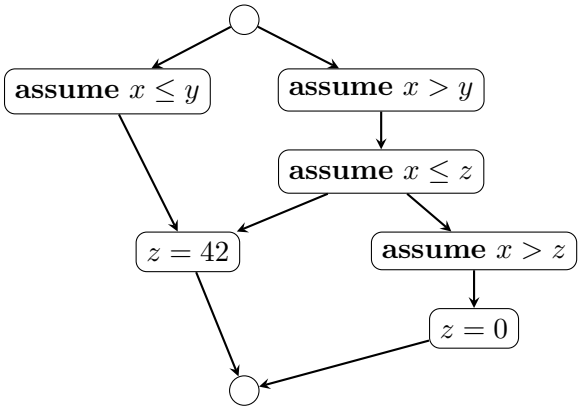


Figure 6.8: Translation of an if statement with C boolean operators in the condition.

6.4 Non-duplication of Procedure Parameters

We assumed in our formal setting, for clarity, that procedure parameters are duplicated during the computation of a relational summary. We proposed to introduce for each formal parameter x an additional variable x_0 denoting the initial value of x. This duplication can become costly as it increases the number of variables in abstract values, and in particular for Linear Relation Analysis, the dimension of convex polyhedra.

We should note that this duplication is not necessary for pure-value or pure-result parameters, which are parameters whose values are respectively not modified or not used before being modified. The initial value of a pure-value parameter is indistinguishable from its final value and the initial value of a pure-result parameter can not be referred to. We do therefore in practice a pre-analysis to determine the status of procedure parameters, that is whether they are pure-value, pure-result or value-result parameters. We describe the design of this specific data-flow analysis.

6.4.1 Boolean attributes

We associate with each formal parameter X of a procedure, three boolean attributes X_i, X_c, X_u which are defined as follows:

- X_i is true when X is certainly equal to its initial value X_0
- X_c is true when the value of X has certainly been changed
- X_u is true when the value of X may have been used

Quite clearly, we have the following properties for the attributes:

- X is a pure value-parameter if X_i is true at each exit point of the procedure.
- X is a pure result-parameter if X_c is true whenever X_u is true.

We then define how these attributes are initialized and propagated along the nodes of the control-flow graph of a procedure.

6.4.2 Semantic equations

With each node n_k of the CFG, we want to associate a transfer function f_k mapping the set of formal parameters of a procedure to a boolean vector of $\mathbb{B}^3$, such that for each parameter X, we have $f_k(X) = (X_i, X_c, X_u)$. We define f_k from the individual transfer functions f_k^1, f_k^2 and f_k^3, updating respectively the attributes X_i, X_c and X_u as follows:

$$f_k(X) = (f_k^1(X), f_k^2(X), f_k^3(X))$$

Entry Node

Let n_k is an entry node. We define the transfer function f_k as follows:

$$\forall X, f_k(X) = (tt, ff, ff)$$

It indicates that each parameter is equal to its initial value, and has neither been changed nor used.

Junction Nodes

Let n_k is a junction node. We define the transfer function f_k as follows:

$$\forall X, f_k(X) = \left(\bigwedge_{(n_\ell, n_k) \in E} f_\ell^1(X), \bigwedge_{(n_\ell, n_k) \in E} f_\ell^2(X), \bigvee_{(n_\ell, n_k) \in E} f_\ell^3(X) \right)$$

The attributes X_i and X_c are true respectively, if they were also true at each predecessor node n_ℓ. The attribute X_u is true if it was true for at least once predecessor node.

Statement Nodes

Let n_k be a statement node, with a condition C_k and assignment statements:

$$(X_k^1 := E_k^1, ..., X_k^k := E_k^j)$$

where $(X_k^1, ..., X_k^j)$ are formal parameters of the procedure. We define the predicates $\mathcal{U}(X)$ and $\mathcal{C}(X)$ for each parameter X such that:

- $\mathcal{U}(X)$ is true if and only if X appears either in the condition C_k or in some expression E_k^i, which is if and only if the value of X is used in the node n_k.
- $\mathcal{C}(X)$ is true if and only if X is one of the assigned variables X_k^i, which is if X is assigned at node n_k.

Let n_p be the unique predecessor node of the statement node n_k. Then, the transfer function f_k is defined as follows:

$$\forall X, f_k(X) = \big(f_p^1(X) \wedge \neg\mathcal{C}(X)\ ,\ f_p^2(X) \vee \mathcal{C}(X)\ ,\ f_p^3(X) \vee \mathcal{U}(X)\big)$$

6.5 Experiments

We presented only tiny examples to illustrate our approach for which the partitioning of the precondition is obvious. However, in presence of more complex control structures, such as nested or successive loops, and when preconditions result from more involved invariants, the interest of our method for discovering relevant preconditions is more convincing.

More thorough experiments are necessary to validate our approach and in particular to answer the following questions.

1. Since we analyze a procedure several times to construct its summary, as required by the iterative refinement of its precondition, it is likely to be time-consuming. Thus it is interesting to measure to cost of summary construction with respect to the time hopefully saved by using the summary.
2. Precondition partitioning is a heuristic process, thus it is important to evaluate the precision lost or gained by using a disjunctive summary instead of analyzing the procedure for each calling context.

Therefore our experiments consists in comparing our bottom-up approach computing disjunctive relational summaries of procedures with an analysis of inlined programs, both with respect to the analysis time and the precision of results. Several difficulties must be addressed first.

Most public benchmarks are not usable, since they contain very few numerical programs with procedures. For instance, in the SV-COMP benchmark[1], most numerical examples are inlined and the ALICe benchmark[2] also contains only monolithic programs. For our assessment, we used the benchmark of the Mälardalen[3] WCET research group, which contains various small and middle-sized programs, such as sorts, matrix computations and fft. Moreover, some programs of this benchmark were sometimes extended with auxiliary variables counting the number of executions of each block to help the evaluation of the execution time [25]. These extensions, the name of which are prefixed with `cnt_` are particularly interesting for us, since they contains more numeric variables and should stress the performance of our approach.

The comparison of results which are convex polyhedra is not straightforward. On one hand, we must decide which polyhedra to compare. The correspondence of control points between the inlined program and the structured program with procedures is not easy to preserve. In our experiments, we only compared the results at the end of the main program. Of course, for the comparison to be meaningful, the results on the inlined program must be first projected on the variables of the original program. On the other hand, while a qualitative comparison of two convex polyhedra is easy, by checking their inclusion in both directions, a quantitative comparison is more difficult. It could be achieved precisely by comparing their volumes, dedicated algorithms are available for that [13, 32], but it is only possible for bounded polyhedra. In our assessment, besides a qualitative comparison, we only compare the number of constraints.

All our experiments are done using the convex polyhedra abstract domain. Widening is never delayed and decreasing sequences are limited to 7 terms. The analysis times are those obtained on an Intel Xeon E5-2630 v3 2.40Ghz machine with 32GB of RAM and

[1] sv-comp.sosy-lab.org/2018/benchmarks.php
[2] alice.cri.mines-paristech.fr/models.html
[3] www.mrtc.mdh.se/projects/wcet/benchmarks.html

20MB of L3 cache running Linux.

Table 6.1 compares our method with a standard LRA on inlined programs, in terms of analysis time, qualitative precision and number of constraints of results found at the exit points of the main procedures. The "# procs" column gives the number of procedures in each program and the "max. # calls" column gives the maximum number of call sites per procedure in a program. We denote as:

- t_{IL} the time in seconds for analyzing the inlined program.
- t_{IP} the time in seconds for the interprocedural analysis of the original structured program.
- P_{IL} the polyhedron result of the inlined analysis at the exit points of the procedure.
- P_{IP} the polyhedron result of the interprocedural analysis at the exit points of the main procedure.
- C_{IL} the number of constraints of P_{IL}.
- C_{IP} the number of constraints of the polyhedron P_{IP}.

The qualitative results comparison is shown by column "cmp. res." which indicates whether the result P_{IP} is better ($\sqsupset$), worse ($\sqsubset$), equal ($=$) or incomparable ($<>$) with respect to P_{IL}. The $\mathcal{S}$ column gives for each program the speedup of our method compared to standard LRA with inlining, defined as:

$\mathcal{S}$ = Time for standard LRA with inlining/Time for disjunctive relational summaries

where:

$$\mathcal{S} = t_{\text{IL}}/t_{\text{IP}}$$

Our method is significantly faster than standard LRA using inlining for 13 over 19 programs ($\approx$ 68% of programs), with an average speedup of 2.9. The loss of precision is very moderate since only 1 over 19 programs, namely **minver**, has a less precise convex polyhedra at the exit node of the main procedure.

Interestingly, our method also leads to precision improvements for some programs, such as **janne_complex**, **my_sin** and **cnt_minver**, due to the use of disjunction, enabling a more accurate analysis of procedure behaviors. Moreover, those precision improvements are not necessarily obtained at the expense of analysis time, since the **janne_complex** program has a more precise convex polyhedra at the exit of the main procedure, with a 60% increase in the number of constraints and has also the highest speedup with $\mathcal{S} = 15.34$.

Table 6.2 reports the computation times of the summary of each procedure in each program. The τ_c column gives the fraction of the analysis time using our method spent during the computation of each procedure summary, defined as follows:

τ_c = Procedure summary comp. time/Program analysis time using rel. summ.

We can also note that the **janne_complex** program, which has the highest speedup in Table 6.1, is consisting only of a single other procedure besides the main procedure. Thus the speedup given by our method is already noticeable on small programs, even with few procedures and few procedure calls. This is due to procedures being analyzed separately in smaller variable environments compared to the inlined program. Overall, our method is mostly faster than inlining when procedures have a much larger number of local variables compared to the number of parameters.

The summary construction time for small utility procedures, such as the **my_fabs**, **my_sin**, **my_cos** and **my_log** procedures, in the **fft1** and **cnt_fft1** programs, are very

Program	# procs	max. # calls	Inlining t_{IL}	Inlining C_{IL}	Interprocedural t_{IP}	Interprocedural C_{IP}	cmp. res.	$\mathcal{S}$
fabs	2	1	0.013	4	0.015	4	$=$	0.87
fdct	2	1	0.084	0	0.069	0	$=$	1.22
fft1	6	3	0.742	4	0.465	3	$<>$	1.59
fir	2	1	0.040	1	0.072	1	$=$	0.55
janne_complex	2	1	0.948	5	0.062	8	$\sqsupset$	15.34
minver	4	2	0.155	1	0.686	2	$\sqsubset$	0.23
my_sin	2	1	0.032	1	0.028	5	$\sqsupset$	1.14
jfdctint	2	1	0.082	3	0.060	3	$=$	1.38
ludcmp	3	1	0.074	3	0.102	3	$=$	0.73
ns	2	1	0.057	0	0.051	0	$=$	1.13
qurt	4	1	0.057	1	0.028	1	$=$	2.06
select	2	1	0.097	0	0.057	0	$=$	1.69
ud	2	1	0.093	3	0.118	3	$=$	0.79
cnt_fdct	2	1	0.098	1	0.075	1	$=$	1.31
cnt_fft1	6	3	33.417	5	2.646	3	$<>$	12.63
cnt_jfdctint	2	1	0.102	5	0.070	5	$=$	1.46
cnt_ns	2	1	0.085	0	0.067	0	$=$	1.25
cnt_qurt	4	1	0.601	2	0.063	2	$=$	9.54
cnt_minver	4	2	1.008	1	3.424	6	$\sqsupset$	0.29

Table 6.1: Experimental results.

small (lower than 4ms) and often individually negligible with respect to the analysis time of the entire program (with τ_c often lower than 1%). This suggests that our method could be particularly beneficial, in terms of analysis performance, for programs built on top of a collection of utility procedures or a library of such procedures, each procedure summary being computed only once and possibly used in many call contexts.

Our last experiment concerns the speedup of our interprocedural analysis with respect to the number of calls. Notice that the Mälardalen benchmark is not very favorable in this respect, since most procedures are called only once. Our analysis on the **cnt_ns** program has a moderate speedup of 1.25. In order to observe the evolution of the speedup with the number of calls, we increase the number of calls to the *foo* procedure in the main procedure of the **cnt_ns** program. The graph of Figure 6.9 shows the evolution of the analysis times of these successive versions, comparing our analysis with respect to standard LRA with inlining.

The analysis of the **cnt_ns** program using our disjunctive relational summaries analysis becomes significantly faster than standard LRA with inlining when there are more than 2 calls to the *foo* procedure in the main procedure. Additionally, Fig. 6.9 shows that the analysis time of the **cnt_ns** program using standard LRA with inlining grows exponentially in the number of calls to the *foo* procedure, while the analysis time using our method grows much slower when the number of calls is increasing.

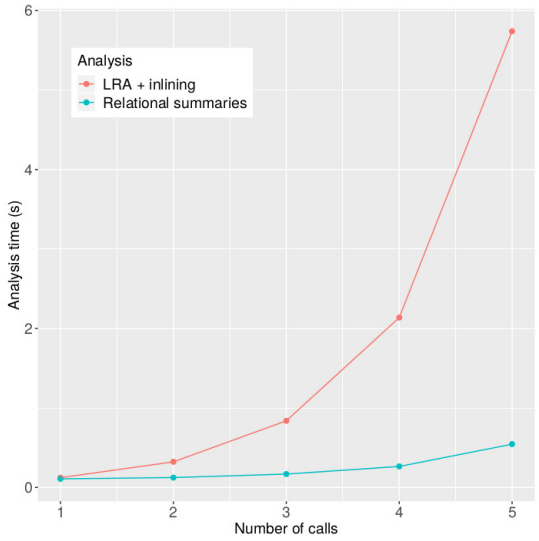


Figure 6.9: Analysis times of the **cnt_ns** program when increasing the number of calls in the main procedure to the *foo* procedure.

6.6 Conclusion

We presented MARS, a static analysis system designed and developed during this thesis with two main objectives: facilitate the development of new static analyses and providing high-quality source traceability information. The MARS static analyzer uses its own intermediate representation designed specifically for numerical abstract interpretation. A frontend tool translates C programs into the MARS intermediate representation from the abstract syntax tree provided by CLANG. It also extracts traceability information given by the compiler frontend which locates in the source each individual part of intermediate representation. The MARS static analyzer is based on a generic abstract interpreter allowing the decoupling of the abstract domain, the transfer functions used on this domain and the iteration strategy. These elements can be given independently as OCAML modules.

We conducted experiments of our modular analysis on a standard benchmark in the WCET community. First, we shown that the construction time of procedure summaries is often negligible with respect to the analysis time of the entire program. Then, we compared our approach with respect to an analysis of the entire program using inlining. We could be wondering whether the interest of using procedure summaries would be showing only on programs with a sufficiently large number of procedure calls. Even on small programs with few procedure calls, our method often provides a significant speedup compared to a classical analysis using inlining. When the number of calls increases, the analysis time of our method grows much slower than the duration of the analysis using inlining which grows exponentially.

A number of improvements could be implemented in the future:

- When computing disjunctive procedure summaries using the refinement according to local reachability, we examine the successors of test nodes to refine preconditions. Advanced heuristics could be designed to examine only the successors of tests which are deemed interesting in some sense to get precise summaries.
- Similarly, when searching for a constraint in a convex polyhedron to refine a precon-

dition, we currently examine all possible constraints and select one that satisfies the refinement criterion in no particular order. Heuristics could be proposed to select the most interesting constraints.

- We compute a relational analysis for each refined precondition when computing a disjunctive summary to take into account the effects of loops and widening. In practice, an analysis optimized for performance could avoid to reanalyze completely a procedure for each precondition by just splitting recursively the preconditions according to tests, at the expense of less precise summaries.
- Disjunctive procedure summaries could be refined according to properties given by an alias analysis and different aliasing situations involving procedure parameters. A proper alias analysis should be implemented to improve the support of parameters of pointer type.
- Procedure summaries only depend on the summaries of called procedures and could be computed in parallel according to the dependencies given in the call graph of the program.

Program	Function	Time (s)	τ_c
fabs	fabs	0.001	0.067
fdct	fdct	0.050	0.588
fft1	my_fabs	< 0.001	0.001
	my_sin	0.002	0.004
	my_cos	< 0.001	0.001
	my_log	< 0.001	< 0.001
	fft1	0.350	0.753
fir	fir	0.019	0.267
janne	janne	0.037	0.602
minver	mmul	0.047	0.069
	minver_fabs	< 0.001	< 0.001
	minver	0.616	0.897
my_sin	my_sin	0.003	0.098
jfdctint	jpeg_fdct_islow	0.031	0.528
ludcmp	fabs	< 0.001	0.002
	ludcmp	0.055	0.540
ns	foo	0.011	0.215
qurt	qurt_fabs	< 0.001	0.007
	qurt_sqrt	0.004	0.138
	qurt	0.002	0.066
select	select	0.042	0.730
ud	ludcmp	0.050	0.425
cnt_fdct	fdct	0.070	0.941
cnt_fft1	my_fabs	0.001	< 0.001
	my_sin	0.004	0.001
	my_cos	< 0.001	< 0.001
	my_log	< 0.001	< 0.001
	fft1	1.750	0.661
cnt_jfdctint	jpeg_fdct_islow	0.035	0.500
cnt_ns	foo	0.026	0.382
cnt_qurt	qurt_fabs	0.001	0.010
	qurt_sqrt	0.019	0.308
	qurt	0.003	0.047
cnt_minver	mmul	0.126	0.037
	minver_fabs	< 0.001	< 0.001
	minver	2.925	0.854

Table 6.2: Summaries computation times.

Part III

Modular Analysis of Reactive Systems

Chapter 7

Analysis and Verification of Reactive Systems

In this part, we want to design a modular analysis of reactive systems for numerical properties. We give a brief overview of reactive systems in 7.1. We are especially interested in synchronous reactive programs. We present synchronous languages in 7.2 and a particular synchronous language called Lustre in 7.3. The compilation of synchronous languages to sequential code is discussed in 7.4. Under certain conditions, reactive components can be compiled modularly, where each component is implemented by a step procedure invoked inside a global infinite loop. We give in 7.5 a definition of the class of structured reactive programs that will be considered in the following chapters. Finally, we describe in 7.6 the state of the art regarding the analysis and verification of reactive programs.

7.1 Introduction to Reactive Systems

In the light of many foundational works [15, 62], reactive systems can be defined as systems continuously reacting to their environment, at a speed determined by their environment. They may receive inputs from the environment, such as sensor values or signals, and produce outputs in reaction to these inputs. They are called reactive by contrast with classical systems which are termed *transformational systems*, receiving their inputs at the beginning of their execution and delivering their results upon termination. Most industrial systems are reactive, such as control systems and process supervision systems. Communication and network protocols, human-machine interfaces are other examples of reactive systems.

The reliability of reactive systems is often highly desirable or even critical, as it is well-known that errors in the operation of reactive systems can have terrible consequences, such as the loss of human life or of massive amounts of money. The analysis and verification of reactive systems is thus of paramount importance. Therefore, we may want to either check that the system satisfies some given property, to ensure the safety of its operation, or to compute an approximation of the set of its reachable states.

A reactive system can be seen as computing a sequence of reaction steps, where in each step, current inputs are read, the system memory is updated, current outputs are computed and finally delivered to the environment of the system, which can be for example input-output devices. As such, a reactive program can be seen as a single, infinite, non-

terminating loop, which implements the sequence of reaction steps. Each loop iteration consists in reading current inputs, calling a *step function* on the current inputs and the current memory, which returns the current outputs with a new memory. We give in Figure 7.1 an example of a loop representing a reactive program.

We denote as X the set of input variables, as Y the set of output variables and as M the set of memory variables. We denote as *read* the operation which consists in reading the current inputs and similarly, we denote as *write* the operation delivering the current outputs to the program environment. The step function is denoted by $\mathcal{S}$.

$$
\begin{array}{l}
M = M_0; \\
\text{loop } \{ \\
\quad read(X); \\
\quad (Y, M) = \mathcal{S}(X, M); \\
\quad write(Y); \\
\}
\end{array}
$$

Figure 7.1: Representation of a reactive program as a single infinite loop.

7.2 Synchronous Languages

Synchronous languages [62, 15] have been designed to express deterministic, structured reactive programs. Synchronous programs are organized as collections of parallel reactive components. However, we should note that the parallelism between components of a synchronous program is only *logical*, being more of a logical concurrency, than an effectively parallel or distributed implementation, as synchronous programs are classically compiled into sequential programs.

The *synchrony hypothesis* considers that components react instantaneously to their inputs in a logical instant. Physical time disappears and time is seen as a sequence of discrete logical instants. Although such a reaction may involve many computations for its implementation, they are seen as atomic at a logical level. Synchronous programs have a uniquely defined behavior for every possible sequence of inputs, hence being deterministic, with no possibility of deadlock, thus being reactive, as the absence of deadlocks is guaranteed by rules enforced by compilers.

Synchronous languages come in different styles, such as the imperative languages Argos [90], Esterel [16], Mode Automata [91], SyncCharts [6], Statecharts [42], and the functional languages Lustre [63] and Signal [82]. Esterel is based on the parallel composition of processes with interrupts, communicating through the instantaneous broadcast of signals. Lustre and Signal are data-flow languages, based on data streams carrying boolean, integer or arbitrarily typed data at each instant at which they are defined. Streams are defined according to clocks which are themselves boolean streams.

Although our work can be applied to reactive programs in any synchronous language, we have a particular interest in the analysis of Lustre programs.

7.3 The Lustre Programming Language

A Lustre program defines its output variables and its memory variables as functions of its input variables and the previous value of its memory variables. An expression e_t can be seen semantically as a function of discrete time, such that $(e_t)_{t\geq 0}$ is a sequence of values for each instant $t \geq 0$. Variables are defined by equations, such that the equation $X = e$ means that the variable X is always equal to the expression e.

$$\begin{aligned} Y_1 &= 2 \times X_1 + X_2 \\ Y_2 &= Y_1 + 1 \end{aligned}$$

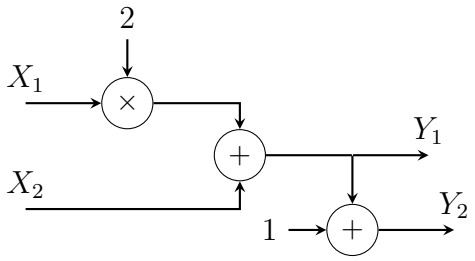


Figure 7.2: Equational and graphical descriptions of a simple data-flow program.

In the data flow program represented in Figure 7.2, the variables X_1, X_2, Y_1, Y_2 can be interpreted as functions of time:

$$\begin{aligned} &\forall t \in \mathbb{N}, \quad Y_1(t) = 2 * X_1(t) + X_2(t) \\ &\forall t \in \mathbb{N}, \quad Y_2(t) = Y_1(t) + 1 \end{aligned}$$

In Lustre, expressions are built from identifiers, constants, arithmetical and logical operations, along with two specific temporal operators:

- A delay operator `pre`, defined as follows:

$$\forall t \geq 1, (\texttt{pre}(E))_t = E_{t-1}$$

  At the initial reaction step, the value $(\texttt{pre}(E))_0$ is set to a special value *nil*.
- An initialization operator `->`, giving an initial value to an expression at the initial reaction step, defined as follows:

$$\begin{aligned} &(E\texttt{->}F)_0 = E_0 \\ &\forall t \geq 1, (E\texttt{->}F)_t = F_t \end{aligned}$$

A Lustre program is structured into nodes. A Lustre node is a subprogram implementing a reactive component, which defines output variables in terms of input variables, possibly containing memory variables.

Example 7.3.1. The `Sum` node represented in Figure 7.3 has an integer input `incr`, a boolean input `reset` and an integer output `s`. It computes the sum of the values of the `incr` input since the last instant at which `reset` was true, otherwise since the initial reaction step if `reset` has never been true.

The `Sum` node can be used in other nodes, such as in the equation `mod7 = Sum(1, (pre(mod7) = 6))` which instantiates the `Sum` node with 1 as the increment value and reset it when the previous value was 6. The variable `mod7` is thus the cyclic sequence of non-negative integers modulo 7.

```
node Sum (incr : int; reset : bool)
    returns (s : int);
let
    s = 0 -> if reset then
                0
             else
                pre(s) + incr;
tel;
```

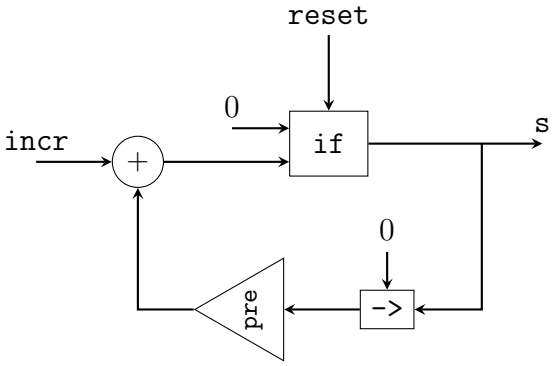


Figure 7.3: Definition of the `Sum` node with its representation as an operator network.

7.4 Compilation of Synchronous Languages

Synchronous programs are classically compiled to sequential code, with a single step function implementing a reaction of the entire program. Such a step function is intended to be called inside an infinite loop, which is in charge of triggering reactions, reading inputs and writing outputs. The machine implementation of a step function in a sequential imperative language is often called a step procedure.

Example 7.4.1. We show in Figure 7.4 the step procedure implementing in C the LUSTRE node `Sum` represented in Figure 7.3.

```
void sum_step(bool init, int incr, bool reset, int * s)
{
    if(init) {
        *s = 0;
    } else {
        if(reset) {
            *s = 0;
        } else {
            *s = *s + incr;
        }
    }
}
```

Figure 7.4: Step procedure implementing the `Sum` node represented in Figure 7.3.

The input variables `incr` and `reset` of the `Sum` node are passed by value to the `sum_step` procedure, while the output variable `s` is passed by pointer to allow imperative updates. As we can see in the definition of the `Sum` node, the variable `s` is defined as being equal to zero at the initial reaction step, through the use of the initialization operator. The step procedure `sum_step` tests the occurrence of the initial reaction step by a test on the value of a special boolean parameter `init`, which is true at the initial reaction step and false everafter. It is the responsibility of the external infinite loop to set appropriately the value of the `init` parameter.

Lustre allows the expression of hierarchy and node reuse by node calls in Lustre expressions, which can be used in the definition of other nodes. A single monolithic step procedure for the top-level node is classically obtained in the academic Lustre compiler by inlining all node calls.

7.4.1 Modular Compilation

Modular compilation is highly desirable in industrial compilers, such as in SCADE suite [7], to avoid an intolerable increase in the size of the generated code due to the inlining of internal nodes.

However, modular compilation of synchronous programs is not possible in general, as noticed by Gonthier in [51], and as exemplified by the `double_copy` node shown in Figure 7.5.

Example 7.4.2. The node `double_copy` is a valid Lustre node, since it is equivalent to the set of equations `z = x; y = z`, although it can not be compiled in a modular way.

```
node double_copy (t1, t2 : int)
    returns (v1, v2 : int);
let
    v1 = t2;
    v2 = t1;
tel;
```

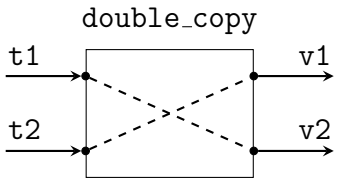


Figure 7.5: Definition and graphical representation of the Lustre node `double_copy`.

The equations of the `double_copy` node can be compiled either into the sequential statements `v1 = t2; v2 = t1;` or into `v2 = t1; v1 = t2;`, since there are two possible sequential orderings for the equations as they have no dependencies between each other. However, the ordering `v1 = t2; v2 = t1;` is not valid for the instantiation `(y,z) = double_copy(x,z)` resulting in the statements `y = z; z = x;` where `z` is used before being defined. Conversely, the ordering `v2 = t1; v1 = t1;` is not valid for the instantiation `(y,z) = double_copy(z,x)`.

Thus the `double_copy` node cannot be compiled independently of its usage and it is not possible to generate a single step procedure which can be called in every context.

In general, reactive components can not be compiled modularly, with a step procedure per component without considering the component environment.

Solutions have been proposed for the modular compilation of synchronous programs, either by enforcing restrictions on the ways components can be connected together, such as in SCADE, or by source code transformation [113, 86, 109, 110].

In SCADE, the rule enforced to enable modular compilation is that every feedback loop must cross an explicit external delay operator `pre`. Such a restriction rejects some causally correct programs, such as the `double_copy` node.

An alternative solution, between the inlining of all internal nodes and the rejection of some causally correct feedback loops was proposed in [113, 86, 109, 110]. The decision problem associated to the modular compilation of a synchronous data-flow network into

c atomic components, without restricting valid feedback loops, was shown to be NP-complete in [86]. However, according to [109], most real programs do not exhibit such a complexity. A symbolic encoding of the problem was proposed in [109, 110] in terms of input-output relations to obtain in polynomial time a decomposition of a node into atomic components which is optimal in some cases.

We will only consider in what follows structured reactive programs which can be compiled modularly.

7.5 Structured Reactive Programs

A reactive system can generally be considered to be made of a set of parallel components. Each component can be seen as a reactive system of its own, with its own memory, inputs and outputs, described by its own step function.

Structured reactive programs are sets of well-defined reactive components, where component implementations are independent from one another. We assume that components in a reactive program communicate solely by the means of inputs and outputs and a component can not manipulate directly the internal memory of a distinct component.

Formally, we consider a structured reactive program to be made of an ordered list $(C_1, ..., C_n)$ of reactive components. A reactive program has three pairwise-disjoint sets of global variables:

- A set X of global input variables
- A set Y of global output variables
- A set M of global memory variables

For a set U of variables, we denote as $\mathcal{V}(U)$ the set of valuations of variables in U. The step function $\mathcal{S}$ corresponding to the entire reactive program is such that $\mathcal{S} : \mathcal{V}(X) \times \mathcal{V}(M) \to \mathcal{V}(Y) \times \mathcal{V}(M)$.

Reactive Component

Each component C_i for $i = 1, ..., n$ of a reactive program has three pairwise-disjoint sets of variables:

- A set X_i of component input variables
- A set Y_i of component output variables
- A set M_i of component memory variables

Each reactive component C_i is implemented by a step function $\mathcal{S}_i : \mathcal{V}(X_i) \times \mathcal{V}(M_i) \to \mathcal{V}(Y_i) \times \mathcal{V}(M_i)$ which associates valuations of output variables and memory variables to a valuation of input variables and a previous valuation of memory variables.

The variables of a synchronous reactive data-flow component satisfy the following properties:

1. Each input of a component C_i is either a global input of the reactive program, a global memory or the output of a previous component:

$$\forall i \in \{1, ..., n\}, X_i \subseteq X \cup M \cup \bigcup_{1 \leq j < i} Y_j$$

2. The memory of a component C_i is local to that component. The set M_i is disjoint from any other set of variables.
3. The outputs of each component are disjoint:

$$\forall i, j \in \{1, ..., n\}, i \neq j \Rightarrow Y_i \cap Y_j = \emptyset$$

4. Each global output and each global memory variable is the output of a component:

$$Y \cup M \subseteq \bigcup_{1 \leq i \leq n} Y_i$$

5. Component output variables are used either as global output, global memory, or as input to another component:

$$\bigcup_{1 \leq i \leq n} Y_i \subseteq Y \cup M \cup \bigcup_{1 \leq i \leq n} X_i$$

These properties on variables of a reactive program and its constituting components give restrictions on the form of structured reactive programs. Components are considered to communicate only through variables, at least from a high-level point of view, and the actual communication mechanism used in a given implementation should be reducible to variable manipulations.

Property 1 ensures that each input variable of a component has been properly written or computed before being read by that component. All variables must be defined once and only once, by a single component. In particular, as a kind of well-formedness condition, property 3 ensures that no two distinct components can write to the same variable. Property 4 ensures that the global outputs of the program are well-defined and property 5 states that all outputs of a component are used.

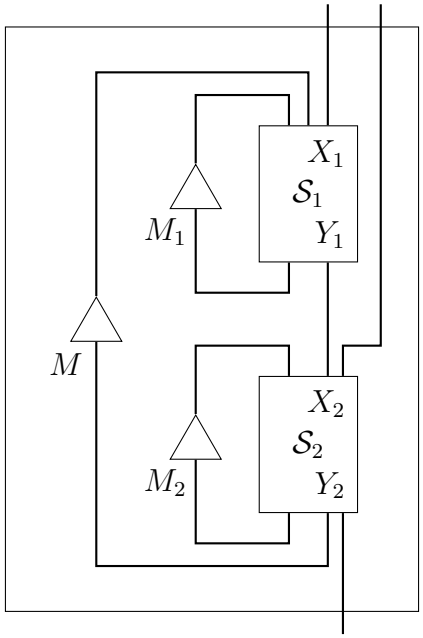


Figure 7.6: Structured reactive program with sub-components.

7.6 Analysis and Verification

Regarding reactive programs, experience shows as argued in [64], that we are mostly interested in proving safety properties, or bounded liveness properties that can be reduced to safety properties, which can be expressed as sets of possible valuations of component variables.

7.6.1 Verification of Synchronous Programs and Observers

Classically, safety properties can be expressed as *synchronous observers* [66], which are components receiving the inputs and the outputs of the component under verification and deciding at each reaction step if the property is satisfied or not, setting appropriately the value of an output variable giving the observer verdict.

A safety property can be verified on a reactive component by an analysis of the reachable states of the composition of the component to be verified with the observer of the safety property. The property is satisfied if the output variable of the observer is always true. Thus, the safety property is transformed into an invariant property of the composition of the component under verification with the observer.

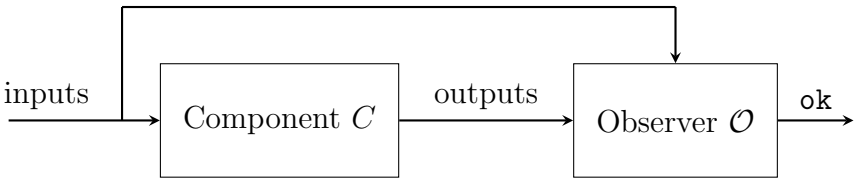


Figure 7.7: Composition of a reactive component C with an observer $\mathcal{O}$.

Observers have the advantage over temporal logics that they can be expressed in the same language as the program under verification, and thus can be executed and tested.

7.6.2 Analysis of Synchronous Programs

The analysis of synchronous programs is classically performed on a control structure called a *control automaton*, which is a way to compile synchronous programs using a partial evaluation of boolean memory variables, as pioneered by ESTEREL [16] v2 and v3 compilers. The academic LUSTRE compiler [114] is able to construct a boolean interpreted automaton and to minimize it on-the-fly by bisimulation [20]. The transitions of the control automaton are labeled by tests on boolean and numerical variables, and by assignments of numerical variables. Such a control structure, in the case of purely-boolean control programs, allows the model checking of safety properties.

Model Checking of the Boolean Control Automaton

The control automaton can also be used as a boolean abstraction of a general synchronous program, by introducing free logical variables to represent tests on numerical variables, representing an over-approximation of the program behavior, which can then be verified by a model checker.

Several tools based on model-checking have been developed (Lesar[112, 65], TempEst[105], Xeve[21]) to verify boolean safety properties on synchronous programs by an exploration of the state space of the control automaton, abstracting away the behavior of numerical variables. These techniques can fail to verify a property on synchronous programs with numerical variables, due to the over-approximation of the possible program behaviors.

More precisely, properties depending directly or indirectly on numerical variables can not be verified by model checking of the boolean control automaton. To alleviate these limitations, the Lesar model checker eliminates numerically unsatisfiable transitions in the control automaton before the model-checking itself. For linear constraints, efficient decision procedures are available, such as the emptiness test of convex polyhedra. This preprocessing restricts the boolean state-space to be explored by the model checker. However, the dynamic behavior of numerical variables is still ignored.

Linear Relation Analysis of Synchronous Programs

It was proposed in [67] to perform a Linear Relation Analysis on the boolean control automaton of a synchronous program. As a result, a convex polyhedron associated with each state of the control automaton is obtained, representing an over-approximation of the set of reachable variable valuations at a given state for a forward analysis. We can also perform a backward analysis and obtain an over-approximation of the variable valuations leading to an erroneous state or a property violation. Numerical tests and assignments of numerical variables on transitions of the control automaton can be used to derive a system of fixpoint equations, partitioned according to boolean states, which is then solved using the standard widening operator.

Linear Relation Analysis can be used either to verify a given property, on the control automaton representing an observer composed with the program under verification, or for the discovery of the possible reachable states without an explicit proof objective, which can be applied to prove the absence of runtime errors such as arithmetic overflows.

A dynamic refinement of the control automaton, with respect to a property to be verified, was proposed in [70, 73].

The Lack of Modularity

Either classical model checkers for synchronous languages, or Linear Relation Analysis for synchronous programs as presented in [67], consider a monolithic control automaton obtained after inlining all internal components. Thus due to inlining, the resulting boolean control automaton can have an intractable size for large synchronous programs. The analysis results for a given component are not reused, and a sub-component can be analyzed many times if it is instantiated in several different contexts. Therefore a modular verification approach for synchronous reactive programs would be highly desirable.

SAT/SMT-based Model Checking

More recently, several approaches have been proposed for the model checking of safety properties of Lustre programs using SAT/SMT-based techniques, namely k-induction [17, 28].

Let N be a LUSTRE node modeled as a formula $\Delta(n)$ expressing an equational system of constraints describing the valuation of the variables of N at a given instant n in terms of the valuations at instants n, $n-1$,...,$n-d$ where d is the maximum nesting depth of the `pre` operator in N.

Let P be a property expressed as a quantifier-free formula $P(n)$ over the variables of N. We denote as Δ_t the formula obtained by replacing every occurrence of n by t in $\Delta(n)$ and as P_t the formula obtained by replacing every occurrence of n by t in $P(n)$.

We can prove that P is an invariant for N by k-induction if we prove that the two following statements are correct for some $k \geq 0$:

$$\Delta_0 \wedge \Delta_1 \wedge ... \wedge \Delta_k \models P_0 \wedge P_1 \wedge ... \wedge P_k$$
$$\begin{cases} \Delta_n \wedge \Delta_{n+1} \wedge ... \wedge \Delta_{n+(k+1)} \\ P_n \wedge P_{n+1} \wedge ... \wedge P_{n+k} \end{cases} \models P_{n+(k+1)}$$

The approach presented in [119, 1] based on SAT-based k-induction is used for the model checking of boolean safety properties of LUSTRE programs in the PROVER tool integrated with the SCADE suite.

KIND 2 [57, 59, 58, 30] is an SMT-based model checker to verify safety properties of finite and infinite state-space LUSTRE programs, using first-order logic with linear integer and real arithmetic. KIND 2 relies on several SMT solvers as backends to prove quantifier-free properties, which are expressed by the user in an extended dialect of LUSTRE, either as assume-guarantee contracts or as invariants. Backends are run concurrently and can cooperate to achieve a given proof goal. KIND 2 can also perform compositional reasoning over nodes, by allowing the user to give assume-guarantee contracts on some nodes, and by reusing the proven properties in these contracts during the verification of a hierarchy of nodes. KIND 2 also takes advantage of the capabilities of modern SMT solvers, which are incremental and backtrackable, able to return models and to compute unsatisfiable cores. An earlier experiment of the SMT-based model checking of LUSTRE programs was described in [49].

Modular Abstractions of Lustre Nodes

The lack of modularity of earlier verification approaches, based on model checking or Linear Relation Analysis on a monolithic control automaton, can be tackled by computing abstractions of the behavior of LUSTRE nodes in a modular fashion, and to use these abstractions to compute invariants of LUSTRE programs. Then, a LUSTRE program can be verified for a given property of interest by computing an invariant of the program composed with an observer representing that property.

A modular abstraction of LUSTRE nodes by disjunctive invariants was proposed in [102], based on predicate abstraction, using quantifier elimination and SMT-solving. A finite set of predicates $\Pi = \{p_1, ..., p_m\}$ over the variables of a LUSTRE program is considered. The approach computes disjunctive invariants of the form $C_1 \vee ... \vee C_n$ where each disjunct C_i is a conjunction of predicates of Π with a bounded size. Such a disjunctive invariant follows the template:

$$\bigvee_i \bigwedge_{j=1}^{m} b_{i,j} \Rightarrow p_j$$

and can be obtained by instantiating the $b_{i,j}$ booleans. The problem of finding a disjunctive invariant over the finite set Π of predicates is reduced to finding suitable values for the boolean variables $b_{i,j}$. This problem is encoded into a quantifier-free formula F over the $b_{i,j}$ variables, which can be solved using an SMT-solver. A minimal disjunctive invariant, for a given template, can be obtained by iterative refinement.

Modular invariants of LUSTRE nodes encoded as Horn clauses are obtained in [50] based on property-directed reachability [68], either by encoding modularly a synchronous program as a set of Horn clauses or by synthesizing a modular invariant from a monolithic invariant.

7.7 Conclusion

The analysis and verification of synchronous reactive programs is classically based on a monolithic boolean control automaton describing the entire program. It is constructed by inlining all node instantiations to get a single flat component. The modular structure of a reactive program into sub-components is completely lost. Using model-checking or Linear Relation Analysis, the analysis of the boolean control automaton involves rapidly for large programs an explosion of the automaton size and of the number of abstract states. Several approaches using SAT or SMT solvers for the modular *verification* of LUSTRE nodes have been proposed. They require a proof objective or a property to be verified.

To prove the absence of errors at runtime like arithmetic overflows, we need to prove automatically that program variables are always bounded, and it would be quite painful for users to provide those bounds manually in large systems. Thus we are interested in the automatic *discovery* of invariant properties of reactive programs, without requiring a proof objective, in a modular way.

In the next chapter, we propose a new approach toward a modular analysis of reactive systems, based on the computation of disjunctive relational summaries of step procedures. The summaries of step procedures will be used to construct a representation of components called *Relational Mode Automata*, designed specifically for analysis and providing several levels of abstraction of component behavior. The analysis results of a relational mode automaton can be used modularly to analyze larger systems made of other components without computing a product automaton. We will also give a way to tune the level of detail of a relational mode automaton to achieve different tradeoffs in analysis precision and performance.

Chapter 8

Towards a Modular Analysis of Reactive Systems using Relational Mode Automata

Reactive programs are usually structured as sets of components, cooperating together to implement the system behavior. If the implementation of a reactive program is not explicitly structured as such, it is quite convenient and natural to consider such a program as being made of distinct components. Reactive programs involve concurrency between the components, at the level of their logical structure, even if the actual implementation is not concurrent itself. Classically, synchronous programs are compiled into a single step procedure, in which all logical concurrency has been dealt away by a sequential scheduling of component reactions.

As we saw in Chapter 7, synchronous programs are classically analyzed in a monolithic fashion, either in the form of a step procedure implementing the whole program, or as a single control automaton, resulting from the synchronous composition of its components.

The structure of a reactive program as a system of components is completely lost in those approaches. There is no reuse of analysis results for components, which may be instantiated in multiple different contexts.

Existing approaches for the modular analysis of synchronous programs consider only Lustre programs, with an observer specifying explicitly the property to be checked. On one hand, they do not handle reactive programs where, as in real-world industrial cases, some components are implemented externally in a different language, such as C. This is especially necessary to access native features of the implementation platform, such as input-output devices. On the other hand, we would like to be able to not only verify, but also automatically discover properties of reactive programs. For example, discovering linear relations between the variables of a reactive program can be used to check if these variables are bounded at all times, for every possible input event, without having to give the actual bound manually. It can also be applied to the analysis of the Worst-Case Execution Time (WCET) of reactive programs [25, 115].

We propose in this chapter an approach for the modular analysis of reactive programs, where each component can be analyzed separately, to discover automatically relations between input, output and memory variables.

8.1 The Bounded Event Counter

We consider a very simple reactive program, which implements a bounded event counter, defined by the `counter_step` procedure given in Figure 8.1.

```
void counter_step(int n, int init, int event, reset, int * cnt)
{
    assert(n >= 1);
1:  if(init){
        *cnt = event;
2:  } else {
3:      if(reset){
            *cnt = event;
4:      } else if(*cnt < n) {
            *cnt = *cnt + event;
5:      } else {
6:      }
7:  }
8:
}
```

Figure 8.1: Step procedure of the bounded event counter.

The bounded event counter receives two boolean inputs, `reset` and `event`, and returns an integer output `cnt` counting the number of occurrences of an event, denoted by the `event` variable, since the last reset, or since the first reaction step if no reset has been received yet. The output `cnt` stays at the same value when it reaches the counter capacity `n`, which is assumed to be a positive symbolic constant, until a reset is received. For simplicity, boolean variables are implemented as integer variables between 0 and 1. Alternatively, we could use an abstract domain combining boolean properties with convex polyhedra [70, 10].

This may seem to be a simplistic example, however overflows of counter components have been the cause of a number of notable software bugs in critical systems. The Boeing Dreamliner 787 [2] had to be rebooted every 248 days, due to a counter overflow in the firmware of the Generator Control Units (GCUs), which are in charge of AC power management in the airplane.

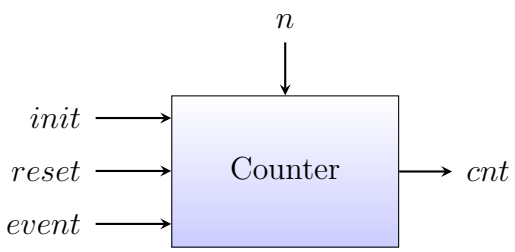


Figure 8.2: The bounded event counter component.

In order to prove the absence of arithmetic overflows in the `counter_step` procedure, we must first guarantee that the output variable `cnt` remains bounded at all times. More specifically, the value of *cnt* must be always between 0 and n, for every possible value of n with $n \geq 1$. We would like to obtain $0 \leq cnt \leq n$ at each reaction step in a modular fashion, without having to consider other components and without inlining of step procedures.

We can start by computing a classical Linear Relation Analysis on the step procedure and see what properties can be discovered.

The parameters *init* and *event* are boolean variables in the high-level definition of the `counter` component. From the knowledge of the high-level type of variables and the user-provided assertion $n \geq 1$, we get the precondition $I^\sharp$ for the `counter_step` procedure:

$$I^\sharp = (0 \leq init \leq 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1)$$

We define the following system of fixpoint equations:

$$\begin{array}{lcl} Q_1 & = & I^\sharp \\ Q_2 & = & (Q_1 \sqcap (init = 1))[cnt := event] \\ Q_3 & = & Q_1 \sqcap (init = 0) \\ Q_4 & = & (Q_3 \sqcap (reset = 1))[cnt := event] \\ Q_5 & = & (Q_3 \sqcap (reset = 0 \wedge cnt < n))[cnt := cnt + event] \\ Q_6 & = & Q_3 \sqcap (cnt \geq n) \\ Q_7 & = & Q_4 \sqcup Q_5 \sqcup Q_6 \\ Q_8 & = & Q_2 \sqcup Q_7 \end{array}$$

Using a classical Linear Relation Analysis, we get the following result for Q_8 corresponding to the exit point of the `counter_step` procedure:

$$Q_8 = (0 \leq init \leq 1 \wedge 0 \leq reset \leq 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1)$$

It is an utterly imprecise result, since we get nothing more than the precondition $I^\sharp$ itself. This not surprising, due to the convex hull operator applied for Q_7 and Q_8, and more so since we did not take into account that the `counter_step` procedure is called repeatedly inside an infinite loop, with the `init` variable being `true` at the first iteration and `false` everafter.

Thus it is interesting to compute a disjunctive relational summary of the `counter_step` procedure, to obtain a more precise representation of its behavior, and in the meantime, to consider the specific nature of step procedures.

8.2 Disjunctive Summaries of Step Procedures

We are interested in computing summaries of reactive components implemented by a step procedure. Step procedures can be either generated automatically as a result of the compilation of a reactive program, such as for synchronous languages, or implemented externally in a host language such as C.

8.2.1 General Principle

Following the approach presented in Chapter 5, we compute the disjunctive relational summary of a step procedure p by iterative refinement of its global precondition $I^\sharp$.

Intuition

Reactive components update the value of variables based on the previous value of memory variables and the current value of input variables. Thus it is interesting to compute the summary of a component by distinguishing behaviors according to preconditions on memory variables. Additionally, to obtain precise component summaries, we introduce an additional level of disjunction, by partitioning further according to preconditions on input variables. Partitioning with respect to constraints on memory variables is designed to separate behaviors according to high-level component states, whereas partitioning with respect to constraints on input variables separate the possible ways in which the next state and outputs are computed.

Two-Level Partition Refinement

We use a *two-level partition refinement* scheme for step procedures. First, we refine the global precondition $I^\sharp$ of a step procedure according to constraints on memory variables. This first refinement phase produces an abstract partition $\delta^\sharp_M = \{I^\sharp_1, ..., I^\sharp_n\}$ of the global precondition $I^\sharp$. In a second phase, we refine each precondition $I^\sharp_k \in \delta^\sharp_M$ according to constraints on input variables. Finally, we obtain an abstract partition $\delta^\sharp = \{I^\sharp_{1,1}, .., I^\sharp_{1,m_1}, ..., I^\sharp_{n,1}, .., I^\sharp_{n,m_n}\}$ of the global precondition $I^\sharp$.

The abstract partition $\delta^\sharp$ can be seen as made of two levels, in the sense that for each $k = 1..n$, the sets $\{I^\sharp_{k,1}, .., I^\sharp_{k,m_k}\} \subseteq \delta^\sharp$ of preconditions produced by the second phase are themselves abstract partitions of a precondition $I^\sharp_k \in \delta^\sharp_M$ given by the first refinement phase.

Refinement Heuristics

Let $I^\sharp_k$ be a precondition of a step procedure p. Let $Q_i(I^\sharp_k) \in D^\sharp$ be the abstract value discovered at a control location ν_i of p by a relational analysis under the precondition $I^\sharp_k$.

Definition 8.2.1 (Refinement with respect to memory variables)**.** Let M_0 be the set of variables representing initial values of memory variables. If $s^\sharp$ is a complementable abstract value such that:

1. $Q_i(I^\sharp_k) \downarrow M_0 \sqsubseteq s^\sharp$
2. $I^{\sharp'}_k = I^\sharp_k \sqcap \overline{s^\sharp} \neq \bot$ and $I^{\sharp''}_k = I^\sharp_k \sqcap s^\sharp \neq \bot$

then $I^{\sharp'}_k$ and $I^{\sharp''}_k$ are refining the precondition $I^\sharp_k$ according to memory variables.

Definition 8.2.2 (Refinement with respect to input variables)**.** Let X be the set of input variables. If $s^\sharp$ is a complementable abstract value such that:

1. $Q_i(I^\sharp_k) \downarrow \{M_0, X\} \sqsubseteq s^\sharp$
2. $I^{\sharp'}_k = I^\sharp_k \sqcap \overline{s^\sharp} \neq \bot$ and $I^{\sharp''}_k = I^\sharp_k \sqcap s^\sharp \neq \bot$

then $I^{\sharp'}_k$ and $I^{\sharp''}_k$ are refining the precondition $I^\sharp_k$ according to input variables.

We should note that most step procedures in practice have no loops. Loops may appear in step procedures generated from synchronous languages like Lustre when array constructs are used.

8.2.2 Example

We compute a disjunctive relational summary over the convex polyhedra abstract domain for the `counter_step` procedure shown in Figure 8.1. The `counter_step` procedure is represented by a system of fixpoint equations:

$$\begin{array}{lcl} I^\sharp & = & (0 \leq init \leq 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ Q_1 & = & I^\sharp \sqcap (cnt = cnt_0) \\ Q_2 & = & (Q_1 \sqcap (init = 1))[cnt := event] \\ Q_3 & = & Q_1 \sqcap (init = 0) \\ Q_4 & = & (Q_3 \sqcap (reset = 1))[cnt := event] \\ Q_5 & = & (Q_3 \sqcap (reset = 0 \wedge cnt < n))[cnt := cnt + event] \\ Q_6 & = & Q_3 \sqcap (cnt \geq n) \\ Q_7 & = & Q_4 \sqcup Q_5 \sqcup Q_6 \\ Q_8 & = & Q_2 \sqcup Q_7 \end{array}$$

We duplicate only the variable cnt with cnt_0 since it is the only parameter modified by the procedure. Let $M_0 = \{init, n, cnt_0\}$ be the set of variables representing initial values of memory variables.

Refinement According To Memory Variables

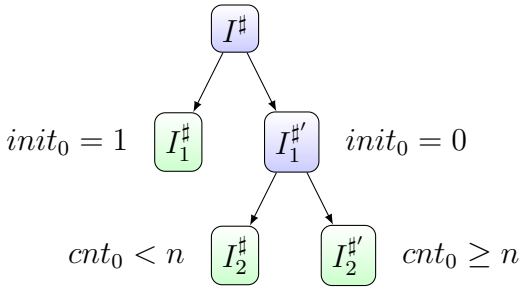


Figure 8.3: Partitioning of $I^\sharp$ according to preconditions on memory variables.

Refinement of $I^\sharp$ A Linear Relation Analysis of the system of fixpoint equations gives for Q_2 at program point 2:

$$\begin{array}{l} Q_2(I^\sharp) = (init = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ Q_2 \downarrow M_0 = (init = 1 \wedge n \geq 0) \end{array}$$

The inequality constraint $s^\sharp = (init \geq 1)$ from $Q_2 \downarrow M_0$ is a complementable abstract value. The complement of $s^\sharp$ with respect to $I^\sharp$ is $\overline{s^\sharp} = (init \leq 0) \sqcap I^\sharp = (init = 0)$. The global precondition $I^\sharp$ is refined into $I_1^\sharp$ and $I_1^{\sharp'}$:

$$\begin{array}{lcl} I_1^\sharp & = & I^\sharp \sqcap s^\sharp = (init = 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ I_1^{\sharp'} & = & I^\sharp \sqcap \overline{s^\sharp} = (init = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \end{array}$$

Refinement of $I_1^{\sharp'}$ A Linear Relation Analysis of the `counter_step` procedure under the precondition $I_1^{\sharp'}$ gives at program point 4:

$$Q_4(I_1^{\sharp'}) \downarrow M_0 = (init = 0 \wedge n \geq 1)$$

We gained no interesting property on memory variables, besides the constraints that we already had in $I_1^{\sharp'}$. This was expected since the program point 4 is only guarded by the condition $reset = 1$ concerning only input variables. We get at program point 5:

$$\begin{array}{lcl} Q_5(I_1^{\sharp'}) & = & (init = 0 \wedge reset = 0 \wedge cnt = cnt_0 + event \wedge cnt_0 \leq cnt \leq cnt_0 + 1 \\ & & \wedge\ n \geq 1 \wedge cnt_0 < n) \\ Q_5(I_1^{\sharp'}) \downarrow M_0 & = & (init = 0 \wedge cnt_0 < n \wedge n \geq 1) \end{array}$$

We can refine $I_1^{\sharp'}$ according to the constraint $cnt_0 < n$ into $I_2^{\sharp}$ and $I_2^{\sharp'}$:

$$\begin{array}{lcl} I_2^{\sharp} & = & (init = 0 \wedge cnt_0 < n \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ I_2^{\sharp'} & = & (init = 0 \wedge cnt_0 \geq n \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \end{array}$$

Both $I_2^{\sharp}$ and $I_2^{\sharp'}$ can not be refined further according to memory variables. We get an abstract partition $\delta_M^{\sharp} = \{I_1^{\sharp}, I_2^{\sharp}, I_2^{\sharp'}\}$ of the global precondition $I^{\sharp}$:

$$\begin{array}{lcl} I_1^{\sharp} & = & (init = 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ I_2^{\sharp} & = & (init = 0 \wedge cnt_0 < n \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ I_2^{\sharp'} & = & (init = 0 \wedge cnt_0 \geq n \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \end{array}$$

Refinement According To Input Variables

We denote as $X = \{event, reset\}$ the set of input variables of the bounded event counter. In the second refinement phase, we refine each precondition obtained in $\delta_M^{\sharp}$ according to constraints on input variables.

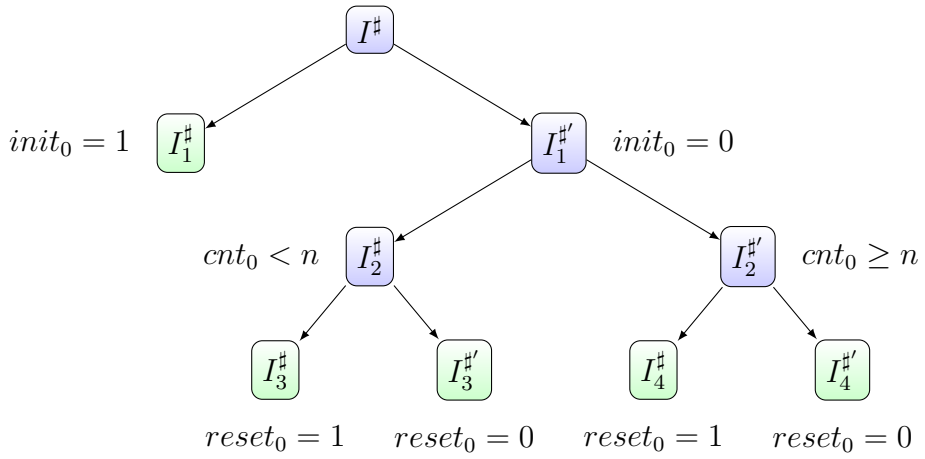


Figure 8.4: Partitioning of $I^{\sharp}$.

Refinement of $I_2^\sharp$ A Linear Relation Analysis of the `counter_step` procedure under the precondition $I_2^\sharp$ gives at program point 4:

$$Q_4(I_2^\sharp) = (init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1)$$
$$Q_4(I_2^\sharp) \downarrow \{M_0, X\} = (init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1)$$

The precondition $I_2^\sharp$ can be refined according to the constraint $reset \geq 1$ in $Q_4(I_2^\sharp) \downarrow X$ and its complement $reset = 0$ into $I_3^\sharp$ and $I_3^{\sharp'}$:

$$\begin{aligned} I_3^\sharp &= (init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ I_3^{\sharp'} &= (init = 0 \wedge cnt_0 < n \wedge reset = 0 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \end{aligned}$$

The preconditions $I_3^\sharp$ and $I_3^{\sharp'}$ can not be refined any further.

Refinement of $I_2^{\sharp'}$ A Linear Relation Analysis of the procedure under $I_2^{\sharp'}$ gives:

$$\begin{aligned} Q_4(I_2^{\sharp'}) &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \\ &\quad \wedge n \geq 1) \\ Q_4(I_2^{\sharp'}) \downarrow \{M_0, X\} &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge 0 \leq event \leq 1) \end{aligned}$$

As for $I_2^\sharp$, we can refine $I_2^{\sharp'}$ according to the constraint $reset \geq 0$ into $I_4^\sharp$ and $I_4^{\sharp'}$:

$$\begin{aligned} I_4^\sharp &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ I_4^{\sharp'} &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 0 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \end{aligned}$$

Abstract Partition of $I^\sharp$

We get an abstract partition $\delta^\sharp = \{I_1^\sharp, I_3^\sharp, I_3^{\sharp'}, I_4^\sharp, I_4^{\sharp'}\}$ of the global precondition $I^\sharp$ of the `counter_step` procedure:

$$\begin{aligned} I_1^\sharp &= (init = 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ I_3^\sharp &= (init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ I_3^{\sharp'} &= (init = 0 \wedge cnt_0 < n \wedge reset = 0 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ I_4^\sharp &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ I_4^{\sharp'} &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 0 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \end{aligned}$$

Disjunctive Summary of the `counter_step` Procedure

The disjunctive summary $\mathcal{S}_c = \{r_1^\sharp, r_2^\sharp, r_3^\sharp, r_4^\sharp, r_5^\sharp\}$ of the `counter_step` procedure is the collection of the abstract relations discovered at the exit of the procedure under each precondition in the abstract partition $\delta^\sharp$, with $r_1^\sharp = Q_8(I_1^\sharp)$, $r_2^\sharp = Q_8(I_3^\sharp)$, $r_3^\sharp = Q_8(I_3^{\sharp'})$, $r_4^\sharp = Q_8(I_4^\sharp)$, $r_5^\sharp = Q_8(I_4^{\sharp'})$. The summary $\mathcal{S}_c$ is as follows:

$$\begin{aligned} r_1^\sharp &= (init = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ r_2^\sharp &= (init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\ r_3^\sharp &= (init = 0 \wedge cnt_0 < n \wedge reset = 0 \wedge cnt = cnt_0 + event \\ &\quad \wedge cnt_0 \leq cnt \leq cnt_0 + 1 \wedge n \geq 1) \\ r_4^\sharp &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\ r_5^\sharp &= (init = 0 \wedge cnt_0 \geq n \wedge reset = 0 \wedge cnt = cnt_0 \wedge 0 \leq event \leq 1 \wedge n \geq 1) \end{aligned}$$

8.2.3 From Disjunctive Summaries To Modes

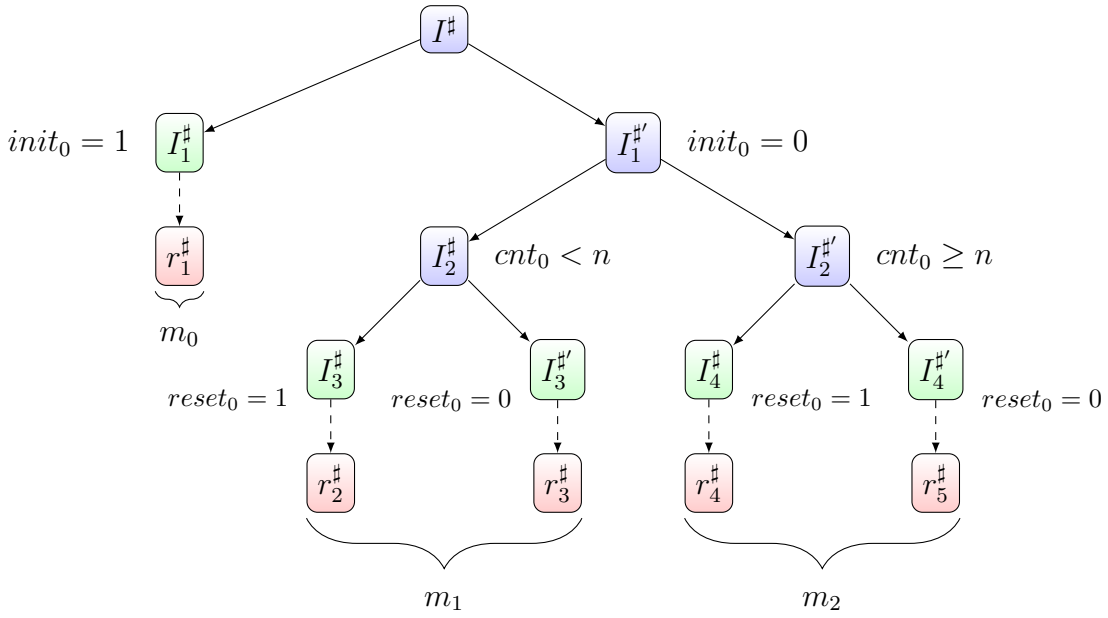


Figure 8.5: Partitioning of $I^\sharp$ and the associated members $r_1^\sharp, r_2^\sharp, r_3^\sharp, r_4^\sharp, r_5^\sharp$ of the summary $\mathcal{S}_c$ regrouped into disjunctions m_0, m_1, m_2 of linear relations.

Summary members derived from the same precondition on memory variables can be grouped into disjunctions $r_2^\sharp \vee r_3^\sharp$ and $r_4^\sharp \vee r_5^\sharp$. The summary $\mathcal{S}_c$ of the `counter_step` procedure can be written as a set $\mathcal{S}_c = \{m_0, m_1, m_2\}$ of disjunctions of linear relations:

$$
\begin{array}{rcl}
m_0 & = & (init = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\
m_1 & = & ((init = 0 \wedge cnt_0 < n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\
 & & \vee\ (init = 0 \wedge cnt_0 < n \wedge reset = 0 \wedge cnt = cnt_0 + event \\
 & & \quad \wedge\ cnt_0 \leq cnt \leq cnt_0 + 1 \wedge n \geq 1)) \\
m_2 & = & ((init = 0 \wedge cnt_0 \geq n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\
 & & \vee\ (init = 0 \wedge cnt_0 \geq n \wedge reset = 0 \wedge cnt = cnt_0 \wedge 0 \leq event \leq 1 \wedge n \geq 1))
\end{array}
$$

The disjunctive summary of a step procedure becomes a set $\mathcal{S} = \{m_i\}$ of disjunctions $m_i = T_{i,1} \vee ... \vee T_{i,n}$ of linear relations $(T_{i,k})_{k=1..n} \in D^\sharp$. Each disjunction m_i is represented symbolically, without applying the convex hull operator. The summary members $m_i \in \mathcal{S}$ can be seen as the *modes* of the reactive component.

We assume that there is a unique summary member $m_0 \in \mathcal{S}$ containing the constraint $init_0 = 1$ where $m_0 \sqsubseteq (init = 1)$, that we identify as the *initial mode* of the component.

The sequence of reaction steps makes a component to transition from one mode to another. Thus a reactive component can be represented by a control structure made of modes, defined by disjunctions of abstract relations. These structures will be termed *relational mode automata*.

8.3 Relational Mode Automata

We propose a new representation of reactive components, allowing different levels of abstraction and designed specifically for the analysis of their behavior, called Relational Mode Automata (RMA).

8.3.1 Definitions

Let Σ be the set of values with $\Sigma = \mathbb{Z} \cup \mathbb{B}$ and $\mathbb{B} = \{0, 1\}$. Given a finite set V of variables, a valuation of variables over V is a function $\sigma : V \to \Sigma$. We denote as $\mathcal{V}(V)$ the set of all variable valuations over V. The projection $\sigma \downarrow U$ of a valuation σ on $U \subseteq V$ is defined as $\sigma \downarrow U = \{v \mapsto \sigma(v) \mid v \in U\}$.

Definition 8.3.1. (Relational Mode Automaton) A Relational Mode Automaton (RMA) is a tuple $\mathcal{R} = (X, Y, Z, M, I, \mathit{Loc}, \ell_0, \phi, \tau)$ defined as follows:

- X, Y, Z, M are the finite disjoint sets of input variables, output variables, local variables and memory variables of the relational automaton $\mathcal{R}$. We denote collectively as $V = X \cup Y \cup Z \cup M$ the set of variables of $\mathcal{R}$.
- $I \subseteq \mathcal{V}(V)$ is the set of initial variable valuations.
- Loc is a finite set of control locations.
- $\ell_0 \in \mathit{Loc}$ is the initial control location.
- $\phi : \mathit{Loc} \to \mathit{Modes}$, with $\mathit{Modes} = \mathcal{P}(\mathcal{V}(V) \times \mathcal{V}(V))$, is a map associating a binary relation $\phi(\ell) \in \mathit{Modes}$ on variable valuations to each control location $\ell \in \mathit{Loc}$. The relation $\phi(\ell)$ is called the *mode* associated to the control location ℓ.
- $\tau \subseteq \mathit{Loc} \times \mathit{Loc}$ is the control transition relation.

The values of variables are updated according to the binary relation $\phi(\ell)$ associated to each control location ℓ. Input variables and output variables are the external interface of a relational mode automaton. Relational Mode Automata can be seen as a relational extension of Mode-Automata [91].

States

Quite classically, a state of a relational mode automaton is a pair (ℓ, σ) made of a control location $\ell \in \mathit{Loc}$ and a variable valuation $\sigma \in \mathcal{V}(V)$. We denote as $S = \mathit{Loc} \times \mathcal{V}(V)$ the set of states of a RMA. We will say that control *resides* at a given control location $\ell \in \mathit{Loc}$ for a state s if s is of the form $s = (\ell, \sigma)$ for some variable valuation $\sigma \in \mathcal{V}(V)$.

The set Loc of control locations should not be mistaken with the set S of automaton states, as control locations are only abstract places where control may reside during an execution. They are merely a notational facility to define the control structure of a relational mode automaton.

Domain of a Mode

The source $\mathit{src}(\phi(\ell))$ of a relation $\phi(\ell)$ defines the domain of applicability of a mode $\phi(\ell)$. The domain $\mathit{dom}(\ell) \subseteq \mathcal{V}(V)$ of a mode $\phi(\ell)$ is defined as $\mathit{dom}(\ell) = \mathit{src}(\phi(\ell))$.

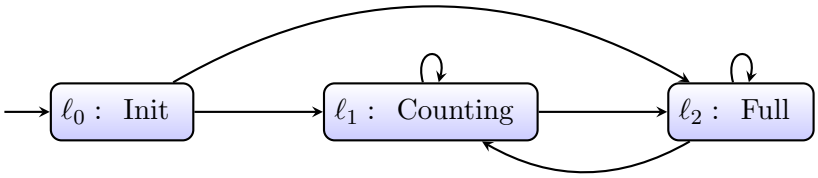


Figure 8.6: Relational mode automaton $\mathcal{R}_{ev}$ representing the bounded event counter component.

When modes are known to be disjunctions of elements of a relational abstract domain $D^\sharp$, such as disjunctions of linear relations, the domain of a mode is more specifically defined as:

$$dom(\ell) = \bigsqcup_{T_i \in \phi(\ell)} src^\sharp(T_i)[V_0/V]$$

Control Transitions

Each pair $(\ell_j, \ell_i) \in \tau$ in the control transition relation represents a possible *control change*, from a location ℓ_j to a location ℓ_i, which is realized if the variable valuation σ of a state $s = (\ell_j, \sigma)$ satisfies the domain of the destination mode $\phi(\ell_i)$, with $\sigma \in dom(\ell_i)$. Control transitions of the form $(\ell, \ell) \in \tau$, when control can possibly stay at a given control location ℓ, are called *self-transitions*.

Example 8.3.1 (Bounded event counter)**.** We define the relational mode automaton $\mathcal{R}_{ev} = (X, Y, Z, M, I, Loc, \ell_0, \phi, \tau)$ representing the bounded event counter implemented by the `counter_step` procedure in Figure 8.1. The automaton $\mathcal{R}_{ev}$ is shown in Figure 8.6. The variables of the automaton $\mathcal{R}_{ev}$ are:

$$\begin{array}{lcl} X & = & \{event, reset\} \\ Y & = & \{cnt\} \\ Z & = & \emptyset \\ M & = & \{cnt, n\} \end{array}$$

The addition of an explicit *init* variable is not necessary, since relational mode automata have an explicit initial control location ℓ_0.

The set Loc of control locations is $Loc = \{\ell_0, \ell_1, \ell_2\}$. The modes of the relational mode automaton $\mathcal{R}_{ev}$ are relations on variable valuations over V, given by the function $\phi : Loc \to \mathcal{P}(\mathcal{V}(V) \times \mathcal{V}(V))$ defined formally as follows:

$$\begin{array}{lcl} \phi(\ell_0) & = & (cnt = event \wedge 0 \leq cnt \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ \phi(\ell_1) & = & ((cnt_0 < n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\ & & \vee\ (cnt_0 < n \wedge reset = 0 \wedge cnt = cnt_0 + event \\ & & \quad \wedge\ cnt_0 \leq cnt \leq cnt_0 + 1 \wedge n \geq 1)) \\ \phi(\ell_2) & = & ((cnt_0 \geq n \wedge reset = 1 \wedge cnt = event \wedge 0 \leq cnt \leq 1 \wedge n \geq 1) \\ & & \vee\ (cnt_0 \geq n \wedge reset = 0 \wedge cnt = cnt_0 \wedge 0 \leq event \leq 1 \wedge n \geq 1)) \end{array}$$

The set I of initial valuations is:

$$I = (cnt = 0 \wedge n \geq 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

The control transition relation τ is defined as:

$$\tau \;=\; \{(\ell_0,\ell_1),(\ell_1,\ell_1),(\ell_1,\ell_2),(\ell_2,\ell_2),(\ell_2,\ell_1)\}$$

8.3.2 Semantics

State Transition System

For a given relational mode automaton $\mathcal{R}$, we define a state transition system $T_{\mathcal{R}} = (S, I, \rightarrow_{\mathcal{R}})$, where the set S of states is $S = \mathit{Loc} \times \mathcal{V}(V)$ and the state transition relation $\rightarrow_{\mathcal{R}} \subseteq S \times S$ is defined as follows:

$$\frac{(\ell_j,\ell_i) \in \tau \quad (\sigma,\sigma') \in \phi(\ell_j) \quad \sigma' \in \mathit{dom}(\ell_i)}{(\ell_j,\sigma) \rightarrow_{\mathcal{R}} (\ell_i,\sigma')}$$

The state transition system $T_{\mathcal{R}}$ is called the *state semantics* of the relational mode automaton $\mathcal{R}$.

Post-Condition Operator

For each pair (ℓ,ℓ') of control locations, we define a post-condition operator $\mathit{post}(\ell,\ell') : \mathcal{P}(\mathcal{V}(V)) \rightarrow \mathcal{P}(\mathcal{V}(V))$ giving the image of a set $U \subseteq \mathcal{V}(V)$ of valuations by the mode $\phi(\ell)$, assuming that the next mode is $\phi(\ell')$. When modes can be arbitrary relations, the $\mathit{post}(\ell,\ell')$ operator is defined as follows:

$$\forall U \subseteq \mathcal{V}(V), \mathit{post}(\ell,\ell')(U) = \mathit{tgt}(\phi(\ell) \cap (U \times \mathit{src}(\phi(\ell'))))$$

When modes $\phi(\ell), \phi(\ell') \in \mathcal{P}(D^\sharp)$ are disjunctions of abstract values in $D^\sharp$ the $\mathit{post}(\ell,\ell')$ operator is more specifically defined as:

$$\forall d^\sharp \in D^\sharp, \mathit{post}(\ell,\ell')(d^\sharp) = \bigsqcup_{T_i \in \phi(\ell)} \exists V_0, (T_i \sqcap (\exists X, d^\sharp[V/V_0]) \sqcap \mathit{dom}(\ell'))$$

Reachable Valuations

The reachable states at a given control location $\ell \in \mathit{Loc}$ are the states (ℓ,σ) which are reachable from the initial control location ℓ_0 in a finite number of computation steps of the state transition system $T_{\mathcal{R}}$. We define the set $\mathit{reach}(\ell)$ of reachable valuations at a control location $\ell \in \mathit{Loc}$ as:

$$\mathit{reach}(\ell) = \{\sigma \in \mathcal{V}(V) \mid \exists \sigma_0 \in I, \sigma_0 \in \mathit{dom}(\ell_0) \wedge (\ell_0,\sigma_0) \rightarrow_{\mathcal{R}}^\star (\ell,\sigma)\}$$

The set $\mathit{reach}(\ell)$ can be expressed as the least solution of the following system of fixpoint equations:

$$\mathit{reach}(\ell_0) = I \cup \bigcup_{(\ell,\ell_0)\in\tau} \mathit{post}(\ell,\ell_0)(\mathit{reach}(\ell)) \cap \mathit{dom}(\ell_0)$$

$$\mathit{reach}(\ell) = \bigcup_{(\ell',\ell)\in\tau} \mathit{post}(\ell',\ell)(\mathit{reach}(\ell')) \cap \mathit{dom}(\ell)$$

Substitution

We define a way to get a new relational mode automaton from an already defined one, by variable substitution. It is intended as a templating mechanism for relational mode automata.

Let $\mathcal{R} = (X, Y, Z, M, I, Loc, l_0, \phi, \tau)$ be a relational mode automaton and $V = X \cup YZ \cup M$. Let $W = (w_1, ..., w_m)$ and $W' = (w'_1, .., w'_m)$ be two distinct tuples of variables of equal length with $V \cap W' = \emptyset$.

Definition 8.3.2 (Variable Renaming)**.** We denote as $K[W/W']$ the renaming in K of each variable $w_i \in W$ by the variable $w'_i \in W'$, such that $K[W/W'] = (K \setminus W) \cup W'$.

Definition 8.3.3 (Renaming of Valuations)**.** For a valuation $\sigma \in \mathcal{V}(V)$, we define the renaming $\sigma[W/W'] \in \mathcal{V}(V[W/W'])$ as follows:

$$\forall \sigma \in \mathcal{V}(V), \forall v \in V[W/W'], \quad \sigma[W/W'](v) = \begin{cases} \sigma(w_i) & \text{if } v = w_i \in W \\ \sigma(v) & \text{otherwise} \end{cases}$$

Definition 8.3.4 (Variable substitution in RMA)**.** We define the renamed relational mode automaton $\mathcal{R}[W/W'] = (X', Y', Z', M', I', Loc, \ell_0, \phi', \tau)$ as follows:

- $X' = X[W/W']$, $Y' = Y[W/W']$, $Z' = Z[W/W']$, $M' = M[W/W']$
- $I' = \{\sigma[W/W'] \mid \sigma \in I\}$
- $\forall \ell \in Loc, \phi'(\ell) = \phi(\ell)[W/W']$

8.3.3 Parallel Composition

Two relational mode automata $\mathcal{R}_1, \mathcal{R}_2$ can have distinct sets V_1, V_2 of variables and thus states with valuations defined over different sets of variables. The parallel composition of $\mathcal{R}_1$ and $\mathcal{R}_2$ must have states with valuations keeping track of the variables of both automata, defined over a greater set $V = V_1 \cup V_2$ of variables. In order to properly define the parallel composition of $\mathcal{R}_1$ and $\mathcal{R}_2$, we must extend the modes of $\mathcal{R}_1$ and $\mathcal{R}_2$ to the greater set V of variables.

Definition 8.3.5 (Extension of valuations)**.** Let V_k and V be finite sets of variables such that $V_k \subseteq V$. We define the extension $\sigma_k \uparrow V \subseteq \mathcal{V}(V)$ of a valuation $\sigma_k \in \mathcal{V}(V_k)$ to a set of valuations over V as:

$$\forall \sigma_k \in \mathcal{V}(V_k), \quad \sigma_k \uparrow V = \{\sigma \in \mathcal{V}(V) \mid \sigma \downarrow V = \sigma_k\}$$

Each extended valuation $\sigma \in \sigma_k \uparrow V$ coincides with the original valuation σ_k on the variables of V_k, preserving their values, such that $\sigma \downarrow V = \sigma_k$. The extension $\sigma_k \uparrow V$ is the set of all possible valuations σ of V satisfying this criterion.

The extension of valuations over a greater set V can be seen, in some sense, as introducing *degrees of freedom* for the variables of V which are not in V_k. The other variables are allowed to to range freely over the set of all possible values.

Example 8.3.2. Let $V_1 = \{x, y\}$ and $V_2 = \{z, t\}$ be sets of variables. Let $V = V_1 \cup V_2$. Let $\sigma_1 = [x \mapsto 1, y \mapsto 2]$ be a valuation over V_1 and let $\sigma_2 = [z \mapsto 3, t \mapsto 4]$ be a valuation over V_2. The extensions to V of the valuations σ_1 and σ_2 are:

$$\sigma_1 \uparrow V = \{[x \mapsto 1, y \mapsto 2, z \mapsto a, t \mapsto b] \mid a, b \in \Sigma\}$$

$$\sigma_2 \uparrow V = \{[x \mapsto c, y \mapsto d, z \mapsto 3, t \mapsto 4] \mid c, d \in \Sigma\}$$

We should remark that the intersection $(\sigma_1 \uparrow V) \cap (\sigma_2 \uparrow V)$ is a singleton:

$$(\sigma_1 \uparrow V) \cap (\sigma_2 \uparrow V) = \{[x \mapsto 1, y \mapsto 2, z \mapsto 3, t \mapsto 4]\}$$

We define straightforwardly the lifting of extension to sets of variable valuations.

Definition 8.3.6 (Extension of sets of valuations)**.** Let V_k and V be finite sets of variables such that $V_k \subseteq V$. We define the extension $U_k \uparrow V \subseteq \mathcal{V}(V)$ of $U_k \subseteq \mathcal{V}(V_k)$ to a set of valuations over V as:

$$\forall U_k \subseteq \mathcal{V}(V_k), \quad U_k \uparrow V = \bigcup_{\sigma_k \in U_k} \sigma_k \uparrow V$$

Using the extension of valuations, we define the extension of modes, which are relations over variable valuations.

Definition 8.3.7 (Relational extension)**.** Let V_k and V be finite sets of variables such that $V_k \subseteq V$. We define the extension $\rho \uparrow V \subseteq \mathcal{V}(V) \times \mathcal{V}(V)$ of a relation $\rho \subseteq \mathcal{V}(V_k) \times \mathcal{V}(V_k)$ to a relation over valuations of V as:

$$\forall \rho \subseteq \mathcal{V}(V_k)^2, \rho \uparrow V = \{(\sigma, \sigma') \in \mathcal{V}(V)^2 \mid (\sigma_k, \sigma'_k) \in \rho \wedge \sigma \in \sigma_k \uparrow V \wedge \sigma' \in \sigma'_k \uparrow V\}$$

Example 8.3.3. Let $V_1 = \{x, y, n\}$ and $V_2 = \{x, z, n\}$ be sets of variables. Let $V = V_1 \cup V_2$. Let $r_1 \subseteq \mathcal{V}(V_1) \times \mathcal{V}(V_1)$ and $r_2 \subseteq \mathcal{V}(V_2) \times \mathcal{V}(V_2)$ be two relations defined as:

$$r_1 = ((y = y_0 + 1 \wedge x < n) \vee (y = y_0 \wedge x \geq n))$$

$$r_2 = ((z = z_0 + 1 \wedge x < n) \vee (z = z_0 \wedge x \geq n))$$

The relations r_1 and r_2 can be seen as the modes of two relational mode automata $\mathcal{R}_1$ and $\mathcal{R}_2$ incrementing their own output variable, respectively y and z, when a common input variable x is strictly lower than n. The extensions to V of r_1 and r_2 are formally defined as:

$$\begin{aligned}
r_1 \uparrow V \;=\; & \{(\sigma, \sigma') \in \mathcal{V}(V) \times \mathcal{V}(V) \mid \\
& \quad (\sigma'(y) = \sigma(y) + 1 \wedge \sigma'(x) < \sigma'(n) \wedge \sigma'(z) \in \Sigma \wedge \sigma \in \mathcal{V}(V)) \\
& \quad \vee (\sigma'(y) = \sigma(y) \wedge \sigma'(x) \geq \sigma'(n) \wedge \sigma'(z) \in \Sigma \wedge \sigma \in \mathcal{V}(V))\} \\
r_2 \uparrow V \;=\; & \{(\sigma, \sigma') \in \mathcal{V}(V) \times \mathcal{V}(V) \mid \\
& \quad (\sigma'(z) = \sigma(z) + 1 \wedge \sigma'(x) \leq \sigma'(n) \wedge \sigma'(y) \in \Sigma \wedge \sigma \in \mathcal{V}(V)) \\
& \quad \vee (\sigma'(z) = \sigma(z) \wedge \sigma'(x) \geq \sigma'(n) \wedge \sigma'(y) \in \Sigma \wedge \sigma \in \mathcal{V}(V))\}
\end{aligned}$$

The intersection $(r_1 \uparrow V) \sqcap (r_2 \uparrow V)$ of the extended relations is:

$$(r_1 \uparrow V) \sqcap (r_2 \uparrow V) = ((y = y_0 + 1 \wedge z = z_0 + 1 \wedge x < n) \vee (y = y_0 \wedge z = z_0 \wedge x \geq n))$$

The intersection $(r_1 \uparrow V) \sqcap (r_2 \uparrow V)$ represents the parallel composition of the modes r_1 of $\mathcal{R}_1$ and r_2 of $\mathcal{R}_2$.

Definition 8.3.8 (Parallel composition of RMA)**.**
Let $\mathcal{R}_1 = (X_1, Y_1, Z_1, M_1, I_1, Loc_1, \ell_0^1, \phi_1, \tau_1)$ and $\mathcal{R}_2 = (X_2, Y_2, Z_2, M_2, I_2, Loc_2, \ell_0^2, \phi_2, \tau_2)$ be relational mode automata. Let $V = Vars(\mathcal{R}_1) \cup Vars(\mathcal{R}_2)$. The parallel product $\mathcal{R}_1 \times \mathcal{R}_2$ is the relational mode automaton $(X, Y, Z, M_1 \cup M_2, I, Loc_1 \times Loc_2, (\ell_0^1, \ell_0^2), \phi, \tau)$ where:

- $X = (X_1 \setminus Y_2) \cup (X_2 \setminus Y_1)$
- $Y = (Y_1 \setminus X_2) \cup (Y_2 \setminus X_1)$
- $Z = Z_1 \cup Z_2 \cup (X_1 \cap Y_2) \cup (Y_1 \cap X_2)$
- $I = (I_1 \uparrow V) \cap (I_2 \uparrow V)$
- The function $\phi : Loc_1 \times Loc_2 \to \mathcal{P}(\mathcal{V}(V) \times \mathcal{V}(V))$ associating modes to control locations is defined as:

$$\forall \ell_1 \in Loc_1, \forall \ell_2 \in Loc_2, \quad \phi((\ell_1, \ell_2)) = (\phi_1(\ell_1) \uparrow V) \cap (\phi_2(\ell_2) \uparrow V)$$

- $((\ell_1, \ell_2), (\ell_1', \ell_2')) \in \tau \Leftrightarrow (\ell_1, \ell_1') \in \tau_1 \wedge (\ell_2, \ell_2') \in \tau_2$

The parallel composition extends the modes of the original automata $\mathcal{R}_1$ and $\mathcal{R}_2$ to get modes updating the variables of both automata, defined over valuations of the global set V of variables.

The intersection $(\phi_1(\ell_1) \uparrow V) \cap (\phi_2(\ell_2) \uparrow V)$ of extended modes defines the parallel composition of each pair $(\phi_1(\ell_1), \phi_2(\ell_2))$ of modes. It allows, in a simple way, the parallel composition of relational mode automata and the combination of the original modes.

In accordance with the semantics of synchronous languages, relational mode automata have simultaneously the control residing in their respective initial locations at the first reaction step.

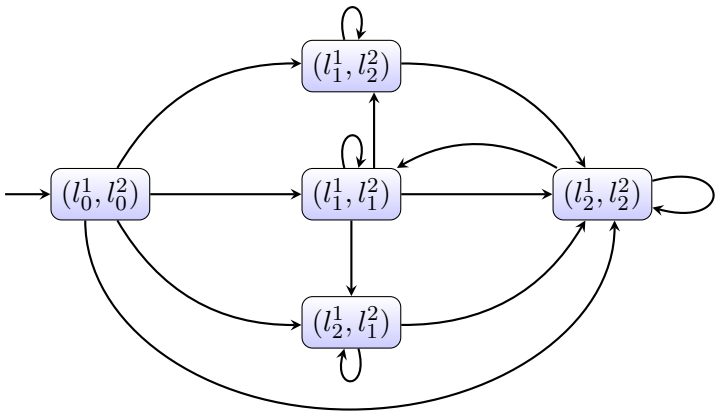


Figure 8.7: Transition diagram of the product automaton $\mathcal{R}_s = \mathcal{R}_{ev}^1 \times \mathcal{R}_{ev}^2$.

Example 8.3.4 (Parallel composition of bounded counters)**.** We consider the parallel composition of two relational mode automata $\mathcal{R}_1$ and $\mathcal{R}_2$ representing bounded event counters reacting to the same event and reset commands, where $\mathcal{R}_1 = \mathcal{R}_{ev}[cnt/cnt_1]$ and $\mathcal{R}_2 = \mathcal{R}_{ev}[cnt/cnt_2]$.

We define the parallel product $\mathcal{R}_s = \mathcal{R}_1 \times \mathcal{R}_2$. The relational mode automaton $\mathcal{R}_s = (X_s, Y_s, Z_s, M_s, I_s, Loc_1 \times Loc_2, (\ell_0^1, \ell_0^2), \phi_s, \tau_s)$ is defined as follows. The sets of

variables are:

$$\begin{array}{lcl} X_s & = & \{event, reset\} \\ Y_s & = & \{cnt_1, cnt_2\} \\ Z_s & = & \emptyset \\ M_s & = & \{cnt_1, cnt_2, n\} \end{array}$$

The set I_s of initial variable valuations is:

$$I_s = (cnt_1 = 0 \wedge cnt_2 = 0 \wedge n \geq 1 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

The function $\phi_s : Loc_1 \times Loc_2 \to \mathcal{P}(\mathcal{V}(V_s) \times \mathcal{V}(V_s))$ is defined as:

$$\begin{array}{lcl}
\phi_s((\ell_0^1, \ell_0^2)) & = & (cnt_1 = event \wedge cnt_1 = cnt_2 \wedge 0 \leq cnt_1 \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\
\phi_s((\ell_1^1, \ell_1^2)) & = & (cnt_{1,0} < n \wedge cnt_{2,0} < n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\
 & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\
 & & \vee\ (cnt_{1,0} < n \wedge cnt_{2,0} < n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} + event \\
 & & \quad \wedge\ cnt_2 = cnt_{2,0} + event \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\
\phi_s((\ell_1^1, \ell_2^2)) & = & (cnt_{1,0} < n \wedge cnt_{2,0} \geq n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\
 & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\
 & & \vee\ (cnt_{1,0} < n \wedge cnt_{2,0} \geq n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} + event \\
 & & \quad \wedge\ cnt_2 = cnt_{2,0} \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\
\phi_s((\ell_2^1, \ell_2^2)) & = & (cnt_{1,0} \geq n \wedge cnt_{2,0} < n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\
 & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\
 & & \vee\ (cnt_{1,0} \geq n \wedge cnt_{2,0} < n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} \\
 & & \quad \wedge\ cnt_2 = cnt_{2,0} + event \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\
\phi_s((\ell_2^1, \ell_2^2)) & = & (cnt_{1,0} \geq n \wedge cnt_{2,0} \geq n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\
 & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\
 & & \vee\ (cnt_{1,0} \geq n \wedge cnt_{2,0} \geq n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} \wedge cnt_2 = cnt_{2,0} \\
 & & \quad \wedge\ 0 \leq event \leq 1 \wedge n \geq 1)
\end{array}$$

The control transition relation $\tau_s \subseteq Loc_s \times Loc_s$ is defined as:

$$\tau = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}\}$$

$$\begin{array}{lcllcl}
t_1 & = & ((l_0^1, l_0^2), (l_1^1, l_1^2)) & t_8 & = & ((l_2^1, l_1^2), (l_2^1, l_1^2)) \\
t_2 & = & ((l_0^1, l_0^2), (l_2^1, l_1^2)) & t_9 & = & ((l_2^1, l_1^2), (l_2^1, l_2^2)) \\
t_3 & = & ((l_0^1, l_0^2), (l_1^1, l_2^2)) & t_{10} & = & ((l_1^1, l_2^2), (l_1^1, l_2^2)) \\
t_4 & = & ((l_1^1, l_1^2), (l_1^1, l_1^2)) & t_{11} & = & ((l_1^1, l_2^2), (l_2^1, l_2^2)) \\
t_5 & = & ((l_1^1, l_1^2), (l_1^1, l_2^1)) & t_{12} & = & ((l_2^1, l_2^2), (l_2^1, l_2^2)) \\
t_6 & = & ((l_1^1, l_1^2), (l_2^1, l_1^2)) & t_{13} & = & ((l_2^1, l_2^2), (l_1^1, l_1^2)) \\
t_7 & = & ((l_1^1, l_2^1), (l_2^1, l_2^2)) & t_{14} & = & ((l_0^1, l_0^2), (l_2^1, l_2^2))
\end{array}$$

The modes $\phi_s((l_1^1, l_2^2))$ and $\phi_s((l_2^1, l_1^2))$ are not reachable in any concrete execution of the product automaton $\mathcal{R}_s$. As both counters are reacting to the same event and initialized to the same value, they must be equal at each step. Although intuitively simple and quite obvious, the equality of counters involves, in some sense, the *dynamics* of the product automaton, which are not considered in the definition of the parallel product.

The actually reachable structure, made of reachable modes only, should be much smaller. Since the reachability of modes in a relational mode automaton is undecidable in general, we turn once more to the abstract world.

8.4 Reachability Analysis of Relational Mode Automata

We define a reachability analysis for relational mode automata which computes, for each control location $\ell \in Loc$, approximations $reach^\sharp(\ell) \in D^\sharp$ of the sets of reachable valuations. The approximation $reach^\sharp(\ell)$ is defined as the least solution of a system of fixpoint equations:

$$reach^\sharp(\ell_0) = I \sqcup \bigsqcup_{(\ell_j,\ell_0)\in\tau} post(\ell_j,\ell_0)(reach^\sharp(\ell_j)) \sqcap dom(\ell_0)$$

$$\forall \ell_i \in Loc, \ell_i \neq \ell_0 \Rightarrow reach^\sharp(\ell_i) = \bigsqcup_{(\ell_j,\ell_i)\in\tau} post(\ell_j,\ell_i)(reach^\sharp(\ell_j)) \sqcap dom(\ell_i)$$

8.4.1 Computation Strategy

We compute a local increasing and decreasing sequence at each control location ℓ where there is a self-transition $(\ell,\ell) \in \tau$, starting from the least upper bound of the abstract values computed at predecessor locations. Local sequences are computed inside a global sequence, until convergence of all abstract values associated to control locations. The computation of local sequences at each location is designed to avoid a loss of precision, by first computing the effect of a mode applied repeatedly in isolation, before the propagation to other modes.

Widening Limited by Preconditions of modes

The domain $dom(\ell)$ of a mode $\phi(\ell)$ is an obvious invariant of the control location ℓ. The standard widening operator ∇ does not take into consideration $dom(\ell)$ and does not preserve it in general. We use a widening $\nabla_\ell : D^\sharp \times D^\sharp \to D^\sharp$ limited by $dom(\ell)$ in the local sequences at each control location ℓ. The limited widening ∇_ℓ is defined as:

$$Q_1 \nabla_\ell Q_2 = (Q_1 \nabla Q_2) \sqcap dom(\ell)$$

Local Increasing Sequences

The local increasing sequence $(Y_i^\ell)_{k\geq 0}$ at a control location ℓ, starting with an abstract value $d^\sharp \in D^\sharp$, is defined as:

$$\begin{array}{lcl} Y_0^\ell & = & d^\sharp \\ Y_{i+1}^\ell & = & Y_i^\ell \nabla_\ell (Y_i^\ell \sqcup F_\ell(Y_i^\ell)) \end{array}$$

where the local transfer function $F_\ell : D^\sharp \to D^\sharp$ gives the image of an abstract value by the mode $\phi(\ell)$ as follows:

$$\forall d^\sharp \in D^\sharp, F_\ell(d^\sharp) = \begin{cases} post(\ell,\ell)(d^\sharp) \sqcap dom(\ell) & \text{if } (\ell,\ell) \in \tau \\ d^\sharp & \text{if } (\ell,\ell) \notin \tau \end{cases}$$

We denote as Y_∇^ℓ the limit of a local increasing sequence $(Y_i^\ell)_{i\geq 0}$.

Local Decreasing Sequences

The decreasing sequence $(Z_i^\ell)_{i\geq 0}$ at a control location ℓ starting with the limit Y_∇^ℓ of the increasing sequence $(Y_i^\ell)_{i\geq 0}$ is defined as:

$$\begin{array}{lcl} Z_0^\ell & = & Y_\nabla^\ell \\ Z_{i+1}^\ell & = & Z_i^\ell \triangle F_\ell(Z_i^\ell) \end{array}$$

We denote as $\Omega_\ell(d^\sharp) = Z_\triangle^\ell$ the limit of the local decreasing sequence at ℓ, computed from the limit of an increasing sequence $(Y_i^\ell)_{i\geq 0}$ starting with $d^\sharp \in D^\sharp$.

Global Increasing Sequence

We denote as $(G_k)_{k\geq 0}$ the global increasing sequence computed over the entire relational mode automaton. The terms of $(G_k)_{k\geq 0}$ belongs to the product domain $(D^\sharp)^L$, where $L = |Loc|$ is the number of control locations. For each term G_k, we denote as $G_k^\ell \in D^\sharp$ the component of G_k associated to a location ℓ.

Incoming Value At each step of the global sequence, local sequences are started on values obtained at predecessor locations. The incoming abstract value $in_\ell(G_k) \in D$ to a control location ℓ is:

$$in_\ell(G_k) = \begin{cases} I \sqcup \bigsqcup\limits_{(\ell',\ell_0)\in\tau} post(\ell',\ell_0)(G_i^{\ell'}) \sqcap dom(\ell_0) & \text{if } \ell = \ell_0 \\ \bigsqcup\limits_{(\ell',\ell)\in\tau} post(\ell',\ell)(G_i^{\ell'}) \sqcap dom(\ell) & \text{if } \ell \neq \ell_0 \end{cases}$$

Global Transfer Function The global transfer function $F^G : (D^\sharp)^L \to (D^\sharp)^L$ computes a local increasing and decreasing sequence at each control location ℓ starting with $in_\ell(G_k)$:

$$F^G(G_k)^\ell = \Omega_\ell(in_\ell(G_k))$$

The global increasing sequence $(G_k)_{k\geq 0}$ is defined as follows:

$$\begin{array}{lcl} G_0 & = & \bot^L \\ G_{k+1} & = & G_k \nabla (G_k \sqcup F^G(G_k)) \end{array}$$

We denote as G_∇ the limit of the global increasing sequence $(G_k)_{k\geq 0}$.

Global Decreasing Sequence

The global decreasing sequence is defined as:

$$\begin{array}{lcl} G'_0 & = & G_\nabla \\ G'_{i+1} & = & G'_{i+1} \triangle F^G(G'_{i+1}) \end{array}$$

We denote as $G'_\triangle$ the limit of the decreasing sequence $(G'_i)_{i\geq 0}$. The approximation of the set of reachable valuations at a control location ℓ is $reach^\sharp(\ell) = G'^\ell_\triangle$.

8.4.2 Reachability Analysis of the Bounded Event Counter

We compute a reachability analysis of the relational mode automaton $\mathcal{R}_{ev}$ defined in Example 8.3.1. The global increasing sequence starts with $G_0^\ell = \bot$ for each location ℓ.

Global Iteration 1

Location ℓ_0 The incoming abstract value $in_{\ell_0}(G_0)$ to the initial location ℓ_0 is:

$$in_{\ell_0}(G_0) = I = (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1)$$

We obtain immediately a new value $G_1^{\ell_0}$ associated to ℓ_0:

$$\begin{array}{rcl} G_1^{\ell_0} & = & \Omega_{\ell_0}(in_{\ell_0}(G_0)) \\ & = & (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \end{array}$$

Location ℓ_1 The incoming value $in_{\ell_1}(G_1)$ at ℓ_1 is:

$$\begin{array}{rcl} in_{\ell_1}(G_1) & = & post(\ell_0, \ell_1)(G_1^{\ell_0}) \sqcap dom(\ell_1) \\ & = & (cnt = event \wedge 0 \leq cnt \leq 1 \wedge cnt < n \wedge 0 \leq reset \leq 1) \end{array}$$

We compute a local increasing sequence at ℓ_1, starting with $in_{\ell_1}(G_1)$, as follows:

$$\begin{array}{rcl} Y_0^{\ell_1} & = & in_{\ell_1}(G_1) \\ Y_i^{\ell_1} & = & Y_i^{\ell_1} \nabla_{\ell_1} (Y_i^{\ell_1} \sqcup F_{\ell_1}(Y_i^{\ell_1})) \end{array}$$

The decreasing sequence converges immediately. The abstract value $G_1^{\ell_1}$ at ℓ_1 is:

$$G_1^{\ell_1} = (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

Location ℓ_2 The incoming value $in_{\ell_2}(G_1)$ at ℓ_2 is:

$$\begin{array}{rcl} in_{\ell_2}(G_1) & = & post(\ell_0, \ell_2)(G_1^{\ell_0}) \sqcap dom(\ell_2) \sqcup post(\ell_1, \ell_2)(G_1^{\ell_1}) \sqcap dom(\ell_2) \\ & = & (cnt = n \wedge event = 1 \wedge cnt \geq 1 \wedge 0 \leq reset \leq 1) \end{array}$$

The local increasing and decreasing sequence at ℓ_2 gives:

$$G_1^{\ell_2} = (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge reset \geq 0 \wedge event \leq 1)$$

Global Iteration 2

Location ℓ_1 The incoming value at ℓ_1 is:

$$\begin{array}{rcl} in_{\ell_1}(G_1) & = & post(\ell_0, \ell_1)(G_1^{\ell_0}) \sqcap dom(\ell_1) \sqcup post(\ell_1, \ell_1)(G_1^{\ell_1}) \sqcap dom(\ell_1) \\ & & \sqcup\ post(\ell_2, \ell_1)(G_1^{\ell_2}) \sqcap dom(\ell_1) \\ & = & (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1) \end{array}$$

The local sequences at ℓ_1 gives:

$$G_2^{\ell_1} = (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

We have $G_2^{\ell_1} = G_1^{\ell_1}$, thus the global increasing sequence has converged. We get the following approximations of sets of reachable valuations:

$$\begin{array}{rcl} reach^\sharp(\ell_0) & = & (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ reach^\sharp(\ell_1) & = & (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1) \\ reach^\sharp(\ell_2) & = & (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge reset \geq 0 \wedge event \leq 1) \end{array}$$

8.4.3 Iterative Construction of the Control Transition Relation

Whereas the modes of a relational mode automaton can be obtained from the members of the disjunctive summary of a step procedure, the control transition relation τ can be constructed iteratively by a reachability analysis.

We compute a transition relation τ_k at each iteration $k \geq 0$ of the global increasing sequence $(G_k)_{k\geq 0}$, as follows:

$$\begin{array}{lcl} \tau_0 & = & \emptyset \\ \tau_{k+1} & = & \{(\ell,\ell') \in \tau_k \mid \ell' \neq \ell_0 \wedge post(\ell,\ell')(G_k^\ell) \sqcap dom(\ell') \neq \bot\} \end{array}$$

In the first global term G_0, the initial control location ℓ_0 is the only location known to be reachable with $G_0^\ell = I$. We start with $\tau_0 = \emptyset$. Then, at each global iteration k, we add a control transition (ℓ,ℓ') to τ_{k+1} when a location ℓ is discovered to be reachable and ℓ' can be a direct successor of ℓ, which is when $post(\ell,\ell')(G_k^\ell) \sqcap dom(\ell') \neq \bot$. Finally, τ is obtained from the limit G_∇ of the global increasing sequence.

Example 8.4.1. We construct iteratively the control transition relation τ of the bounded event counter that we defined manually in Example 8.3.1.



Figure 8.8: Initially, only ℓ_0 is known to be reachable and $\tau = \emptyset$.

Location ℓ_0 We start with an empty transition relation $\tau = \emptyset$. The value associated to the initial control location is $G_0^{\ell_0} = I$. We examine possible transitions from ℓ_0:

$$\begin{array}{lcl} post(\ell_0,\ell_1)(G_0^{\ell_0}) \sqcap dom(\ell_1) & = & (cnt = event \wedge 0 \leq cnt \leq 1 \wedge cnt < n \wedge 0 \leq reset \leq 1) \\ post(\ell_0,\ell_2)(G_0^{\ell_0}) \sqcap dom(\ell_2) & = & (cnt = 1 \wedge n = 1 \wedge event = 1 \wedge 0 \leq reset \leq 1) \end{array}$$

Thus we add transitions from ℓ_0 to ℓ_1 and ℓ_2. The control transition relation becomes $\tau = \{(\ell_0,\ell_1),(\ell_0,\ell_2)\}$.

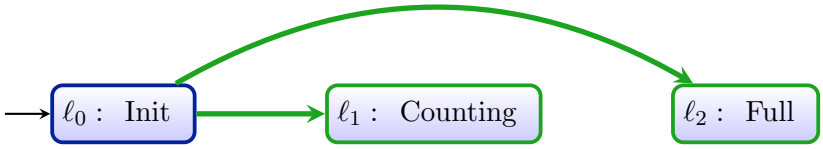


Figure 8.9: We add transitions (ℓ_0,ℓ_1) and (ℓ_0,ℓ_2).

Location ℓ_1 The value $G_1^{\ell_1}$ computed at ℓ_1 is:

$$G_1^{\ell_1} = (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

We examine possible transitions from ℓ_1:

$$\begin{array}{lcl} post(\ell_1,\ell_1)(G_1^{\ell_1}) \sqcap dom(\ell_1) & = & (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \\ & & \wedge\ 0 \leq reset \leq 1) \\ post(\ell_1,\ell_2)(G_1^{\ell_1}) \sqcap dom(\ell_2) & = & (cnt = n \wedge event = 1 \wedge reset = 0 \wedge cnt \geq 1) \end{array}$$

We add the self-transition (ℓ_1, ℓ_1) and (ℓ_1, ℓ_2). The control transition relation becomes:

$$\tau = \{(\ell_0, \ell_1), (\ell_0, \ell_2), (\ell_1, \ell_1), (\ell_1, \ell_2)\}$$

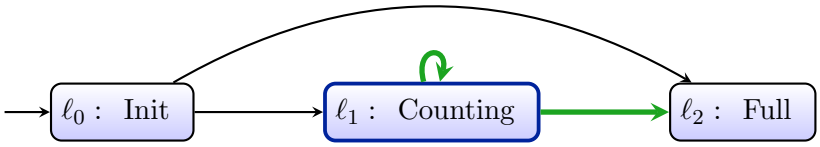


Figure 8.10: We add the self-transition on ℓ_1 and the transition (ℓ_1, ℓ_2).

Location ℓ_2 The value $G_1^{\ell_2}$ computed at ℓ_2 is:

$$G_1^{\ell_2} = (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge 0 \leq reset \leq 1)$$

$$\begin{aligned} post(\ell_2, \ell_1)(G_1^{\ell_2}) \sqcap dom(\ell_1) &= (cnt = event \wedge reset = 1 \wedge 0 \leq cnt \leq 1 \wedge cnt < n) \\ post(\ell_2, \ell_2)(G_1^{\ell_2}) \sqcap dom(\ell_2) &= (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge event \leq 1 \wedge reset \geq 0) \end{aligned}$$

We add the self-transition (ℓ_2, ℓ_2) and (ℓ_2, ℓ_1).

The global increasing sequence has converged. The control transition relation of the bounded event counter is:

$$\tau = \{(\ell_0, \ell_1), (\ell_0, \ell_2), (\ell_1, \ell_1), (\ell_1, \ell_2), (\ell_2, \ell_1), (\ell_2, \ell_2)\}$$

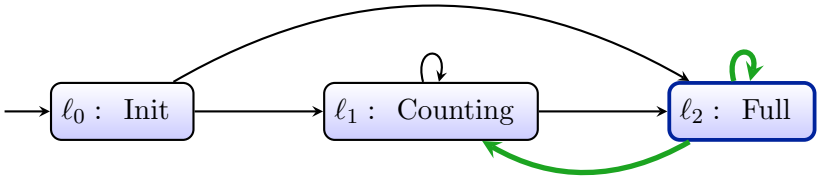


Figure 8.11: We add the self-transition on ℓ_2 and a transition (ℓ_2, ℓ_1).

8.5 Analysis of a Parallel Composition of Relational Mode Automata

We can compute a reachability analysis on a parallel composition of relational mode automata in different ways.

A rather crude way is to construct explicitly the parallel product automaton as defined in 8.3.8 and to compute a reachability analysis of the product automaton. It has a quadratic number of modes with respect to the original automata. Furthermore, the parallel product does not consider the dynamic behavior of relational mode automata and many control locations of the product can be unreachable, as shown in Example 8.3.4. Thus to mitigate the cost of the explicit parallel product, we construct only the reachable part of the parallel product automaton, called the *reduced product automaton*, as a side-product of the reachability analysis.

8.5.1 Computation Strategy

Let $\mathcal{R}_1 = (X_1, Y_1, Z_1, M_1, I_1, Loc_1, \ell_0^1, \phi_1, \tau_1)$ and $\mathcal{R}_2 = (X_2, Y_2, Z_2, M_2, I_2, Loc_2, \ell_0^2, \phi_2, \tau_2)$ be relational mode automata. The reduced product automaton of $\mathcal{R}_1$ and $\mathcal{R}_2$ is denoted as $\mathcal{R} = (X, Y, Z, M, I_1 \sqcap I_2, Loc_1 \times Loc_2, (\ell_0^1, \ell_0^2), \phi, \tau)$.

We adapt the reachability analysis defined in 8.4 to pairs of parallel relational mode automata. We compute local increasing and decreasing sequences at pairs $(\ell_1, \ell_2) \in Loc_1 \times Loc_2$ of control locations inside a global sequence. At the beginning, only the initial pair (ℓ_0^1, ℓ_0^2) is known to be reachable. In each global iteration $k \geq 0$, we explore only the pairs (ℓ_1, ℓ_2) of locations which were found to be reachable in earlier iterations.

We construct the control transition relation τ of the reduced product by adding the newly reachable pairs (ℓ_1, ℓ_2) at the end of each iteration. When the global sequence has converged, the reduced product automaton is only made of locations which have been found to be reachable during the analysis.

We denote as $(G_k)_{k\geq 0}$ the global increasing sequence over the entire relational mode automaton. For each term G_k, we denote as $G_k^{(\ell_1,\ell_2)} \in D^\sharp$ the abstract value associated to a pair (ℓ_1, ℓ_2) of locations.

Domain The domain $dom((\ell_1, \ell_2))$ of a pair $(\phi(\ell_1), \phi(\ell_2))$ of modes is defined as:

$$\forall (\ell_1, \ell_2) \in Loc_1 \times Loc_2, dom((\ell_1, \ell_2)) = dom(\ell_1) \sqcap dom(\ell_2)$$

Limited Widening We define a widening operator $\nabla_{(\ell_1,\ell_2)} : D^\sharp \times D^\sharp \to D^\sharp$ limited by $dom((\ell_1, \ell_2))$ for each pair (ℓ_1, ℓ_2) of locations, as:

$$\forall P, Q \in D^\sharp, \quad P \nabla_{(\ell_1,\ell_2)} Q = (P \nabla Q) \sqcap dom((\ell_1, \ell_2))$$

Post-Condition Operator The $post((\ell_1, \ell_2)) : D^\sharp \to D^\sharp$ operator giving the image of an abstract value $d^\sharp$ by a pair $(\phi(\ell_1), \phi(\ell_2))$ of modes is defined as:

$$\forall d^\sharp \in D^\sharp, post((\ell_1, \ell_2))(d^\sharp) = post(\ell_1)(d^\sharp) \sqcap post(\ell_2)(d^\sharp)$$

Control Transition Relation The control transition relation $(\tau_k)_{k\geq 0}$ of the reduced product automaton at an iteration k of the global increasing sequence $(G_k)_{k\geq 0}$ is:

$$\begin{array}{lcl} \tau_0 & = & \emptyset \\ \tau_{k+1} & = & \{((\ell_1, \ell_2), (\ell_1', \ell_2')) \mid (\ell_1', \ell_2') \neq (\ell_0^1, \ell_0^2) \\ & & \wedge\ post((\ell_1, \ell_2), (\ell_1', \ell_2'))(G_k^{(\ell_1,\ell_2)}) \sqcap dom((\ell_1', \ell_2')) \neq \bot\} \end{array}$$

Local Increasing Sequence

A local increasing sequence $(Y_i)_{i\geq 0}$ at a pair (ℓ_1, ℓ_2) of locations starting with an abstract value $d^\sharp \in D^\sharp$ is defined as:

$$\begin{array}{lcl} Y_0^{(\ell_1,\ell_2)} & = & d^\sharp \\ Y_{i+1}^{(\ell_1,\ell_2)} & = & Y_i^{(\ell_1,\ell_2)} \nabla_{(\ell_1,\ell_2)} (Y_i^{(\ell_1,\ell_2)} \sqcup F_{(\ell_1,\ell_2)}(Y_i^{(\ell_1,\ell_2)}) \end{array}$$

where the local transfer function $F_{(\ell_1,\ell_2)} : D^\sharp \to D^\sharp$ gives the image of an abstract value by a pair $(\phi(\ell_1), \phi(\ell_2))$ of modes, as follows:

$$\forall d^\sharp \in D^\sharp, F_{(\ell_1,\ell_2)}(d^\sharp) = \begin{cases} post((\ell_1,\ell_2),(\ell_1,\ell_2))(d^\sharp) \sqcap dom((\ell_1,\ell_2)) & \text{if } ((\ell_1,\ell_2),(\ell_1,\ell_2)) \in \tau_k \\ d^\sharp & \text{if } ((\ell_1,\ell_2),(\ell_1,\ell_2)) \notin \tau_k \end{cases}$$

We denote as $Y_\nabla^{(\ell_1,\ell_2)}$ the limit of a local increasing sequence $(Y_i^{(\ell_1,\ell_2)})_{i\geq 0}$.

Local Decreasing Sequence

A local decreasing sequence $(Z_i^{(\ell_1,\ell_2)})_{i\geq 0}$ at a pair (ℓ_1,ℓ_2) of locations starting with the limit $Y_\nabla^{(\ell_1,\ell_2)}$ of the increasing sequence is defined as:

$$\begin{array}{lcl} Z_0^{(\ell_1,\ell_2)} & = & Y_\nabla^{(\ell_1,\ell_2)} \\ Z_{i+1}^{(\ell_1,\ell_2)} & = & Z_i^{(\ell_1,\ell_2)} \triangle F_{(\ell_1,\ell_2)}(Z_i^{(\ell_1,\ell_2)}) \end{array}$$

We denote as $\Omega_{(\ell_1,\ell_2)}(d^\sharp) = Z_\triangle^{(\ell_1,\ell_2)}$ the limit of the local decreasing sequence at (ℓ_1,ℓ_2), computed from the limit of an increasing sequence $(Y_i)_{i\geq 0}$ starting with $d^\sharp$.

Incoming Value The incoming abstract value $in_{(\ell_1,\ell_2)}(G_k) \in D^\sharp$ to a pair (ℓ_1,ℓ_2) of locations is:

$$in_{(\ell_0^1,\ell_0^2)}(G_k) = I \sqcup \bigsqcup_{((\ell_1,\ell_2),(\ell_0^1,\ell_0^2))\in\tau_k} post((\ell_1,\ell_2),(\ell_0^1,\ell_0^2))(G_k^{(\ell_1,\ell_2)}) \sqcap dom((\ell_0^1,\ell_0^2))$$

for $(\ell_1,\ell_2) \neq (\ell_0^1,\ell_0^2)$:

$$in_{(\ell_1,\ell_2)}(G_k) = \bigsqcup_{((\ell_1',\ell_2'),(\ell_1,\ell_2))\in\tau_k} post((\ell_1',\ell_2'),(\ell_1,\ell_2))(G_k^{(\ell_1',\ell_2')}) \sqcap dom((\ell_1,\ell_2))$$

Global Transfer Function The global transfer function $F^G : (D^\sharp)^L \to (D^\sharp)^L$ computes a local increasing and decreasing sequence at each pair (ℓ_1,ℓ_2) of control locations starting with $in_{(\ell_1,\ell_2)}(G_k)$:

$$F^G(G_k)^{(\ell_1,\ell_2)} = \Omega_{(\ell_1,\ell_2)}(in_{(\ell_1,\ell_2)}(G_k))$$

We denote as $G'_\triangle$ the limit of the global decreasing sequence. The approximation of the set of reachable valuations at (ℓ_1,ℓ_2) is $reach^\sharp((\ell_1,\ell_2)) = G'^{(\ell_1,\ell_2)}_\triangle$.

8.5.2 Construction of the Reduced Product Automaton

The sets of variables of the reduced product automaton are defined as:

- $X = (X_1 \setminus Y_2) \cup (X_2 \setminus Y_1)$
- $Y = (Y_1 \setminus X_2) \cup (Y_2 \setminus X_1)$
- $Z = Z_1 \cup Z_2 \cup (X_1 \cap Y_2) \cup (Y_1 \cap X_2)$

The control transition relation τ is obtained from the limit G_∇ of the global increasing sequence. The function $\phi : Loc_1 \times Loc_2 \to D^\sharp$ associating modes to control locations is defined as:

$$\phi = \{\phi_1(\ell_1) \wedge \phi_2(\ell_2) \mid \exists (\ell_1',\ell_2'), ((\ell_1,\ell_2),(\ell_1',\ell_2')) \in \tau \vee ((\ell_1',\ell_2'),(\ell_1,\ell_2)) \in \tau\}$$

where $\phi_1(\ell_1) \wedge \phi_2(\ell_2)$ denotes the disjunction formed by the product of the disjunctions $\phi_1(\ell_1)$ and $\phi_2(\ell_2)$.

8.5.3 Example

We analyze the parallel product $\mathcal{R}_s = \mathcal{R}_1 \times \mathcal{R}_2$ of the bounded event counters $\mathcal{R}_1$ and $\mathcal{R}_2$ defined in Example 8.3.4. We construct the reduced product automaton $\mathcal{R}_s$ during the analysis. The initial term of the global increasing sequence is $G_0 = \bot$ and $\tau = \emptyset$.

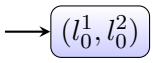


Figure 8.12: Initially, only (ℓ_0^1, ℓ_0^2) is known to be reachable and $\tau = \emptyset$.

Global Iteration 1

Location (ℓ_0^1, ℓ_0^2) The incoming abstract value to the initial location (ℓ_0^1, ℓ_0^2) of $\mathcal{R}_s$ is:

$$in_{(\ell_0^1,\ell_0^2)}(G_0) = I_1 \sqcap I_2 = (cnt_1 = 0 \wedge cnt_2 = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1)$$

The abstract value $G_1^{(\ell_0^1,\ell_0^2)}$ associated to (ℓ_0^1, ℓ_0^2) is:

$$\begin{array}{rcl} G_1^{(\ell_0^1,\ell_0^2)} & = & \Omega_{(\ell_0^1,\ell_0^2)}(in_{(\ell_0^1,\ell_0^2)}(G_0)) \\ & = & (cnt_1 = 0 \wedge cnt_2 = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \end{array}$$

We examine the possible transitions from (ℓ_0^1, ℓ_0^2):

$$\begin{array}{l} post((\ell_0^1, \ell_0^2), (\ell_1^1, \ell_1^2))(in_{(\ell_0^1,\ell_0^2)}(G_0)) \sqcap dom((\ell_1^1, \ell_1^2)) \neq \bot \\ post((\ell_0^1, \ell_0^2), (\ell_2^1, \ell_2^2))(in_{(\ell_0^1,\ell_0^2)}(G_0)) \sqcap dom((\ell_2^1, \ell_2^2)) \neq \bot \\ post((\ell_0^1, \ell_0^2), (\ell_1^1, \ell_2^2))(in_{(\ell_0^1,\ell_0^2)}(G_0)) \sqcap dom((\ell_1^1, \ell_2^2)) = \bot \\ post((\ell_0^1, \ell_0^2), (\ell_2^1, \ell_1^2))(in_{(\ell_0^1,\ell_0^2)}(G_0)) \sqcap dom((\ell_2^1, \ell_1^2)) = \bot \end{array}$$

The locations (ℓ_1^1, ℓ_2^1) and (ℓ_2^1, ℓ_2^2) are reachable from (ℓ_0^1, ℓ_0^2). The control transition relation becomes $\tau = \{((\ell_0^1, \ell_0^2), (\ell_1^1, \ell_1^2)), ((\ell_0^1, \ell_0^2), (\ell_2^1, \ell_2^2))\}$.

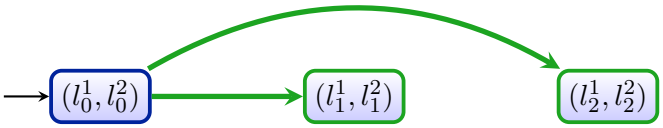


Figure 8.13: The locations (ℓ_1^1, ℓ_1^2) and (ℓ_2^1, ℓ_2^2) are reachable from the initial location (ℓ_0^1, ℓ_0^2).

Location (ℓ_1^1, ℓ_1^2) The incoming abstract value to (ℓ_1^1, ℓ_1^2) is:

$$in_{(\ell_1^1,\ell_1^2)}(G_1) = (cnt_1 = event \wedge cnt_1 = cnt_2 \wedge 0 \leq cnt_1 \leq 1 \wedge cnt_1 < n \wedge 0 \leq reset \leq 1)$$

We compute an increasing sequence at (ℓ_1^1, ℓ_1^2), followed by a decreasing sequence. We get the abstract value $G_1^{(\ell_1,\ell_1^2)}$ at (ℓ_1^1, ℓ_1^2):

$$G_1^{(\ell_1^1,\ell_1^2)} = (cnt_1 = cnt_2 \wedge cnt_1 < n \wedge cnt_1 \geq event \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

We examine possible transitions from (ℓ_1^1, ℓ_1^2):

$$post((\ell_1^1, \ell_1^2), (\ell_1^1, \ell_1^2))(in_{(\ell_1^1,\ell_1^2)}(G_1)) \sqcap dom((\ell_1^1, \ell_1^2)) \neq \bot$$
$$post((\ell_1^1, \ell_1^2), (\ell_1^1, \ell_2^2))(in_{(\ell_1^1,\ell_1^2)}(G_1)) \sqcap dom((\ell_1^1, \ell_2^2)) = \bot$$
$$post((\ell_1^1, \ell_1^2), (\ell_2^1, \ell_1^2))(in_{(\ell_1^1,\ell_1^2)}(G_1)) \sqcap dom((\ell_2^1, \ell_1^2)) = \bot$$
$$post((\ell_1^1, \ell_1^2), (\ell_2^1, \ell_2^2))(in_{(\ell_1^1,\ell_1^2)}(G_1)) \sqcap dom((\ell_2^1, \ell_2^2)) \neq \bot$$

We add the self-transition $((\ell_1^1, \ell_1^2), (\ell_1^1, \ell_1^2))$ and a transition to (ℓ_2^1, ℓ_2^2).

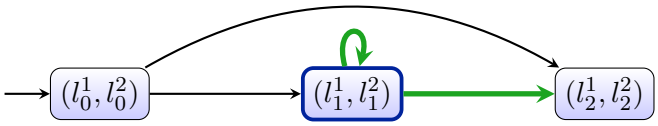


Figure 8.14: The location (ℓ_2^1, ℓ_2^2) is reachable from (ℓ_1^1, ℓ_1^2) and we add the self-transition on (ℓ_1^1, ℓ_1^2).

Location (ℓ_2^1, ℓ_2^2) The incoming abstract value to (ℓ_2^1, ℓ_2^2) is:

$$in_{(\ell_2^1,\ell_2^2)}(G_1) = (cnt_1 = n \wedge cnt_1 = cnt_2 \wedge cnt_1 \geq 1 \wedge event = 1 \wedge 0 \leq reset \leq 1)$$

We compute an increasing and decreasing sequence at (ℓ_2^1, ℓ_2^2). We get the abstract value $G_1^{(\ell_2^1,\ell_2^2)}$ associated to (ℓ_2^1, ℓ_2^2):

$$G_1^{(\ell_2^1,\ell_2^2)} = (cnt_1 = n \wedge cnt_1 = cnt_2 \wedge cnt_1 \geq 1 \wedge 0 \leq reset \leq event \wedge event \leq 1)$$

We examine the possible transitions:

$$post((\ell_2^1, \ell_2^2), (\ell_2^1, \ell_2^2))(in_{(\ell_2^1,\ell_2^2)}(G_1)) \sqcap dom((\ell_2^1, \ell_2^2)) \neq \bot$$
$$post((\ell_2^1, \ell_2^2), (\ell_1^1, \ell_1^2))(in_{(\ell_2^1,\ell_2^2)}(G_1)) \sqcap dom((\ell_1^1, \ell_1^2)) \neq \bot$$

We add the self-transition $((\ell_2^1, \ell_2^2), (\ell_2^1, \ell_2^2))$ and a transition back to (ℓ_1^1, ℓ_1^2).

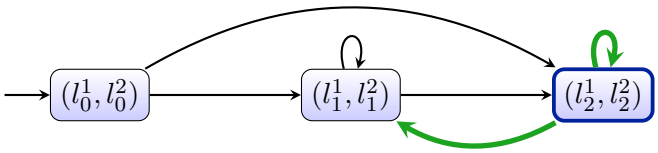


Figure 8.15: We add a self-transition on (ℓ_2^1, ℓ_2^2) and a transition from (ℓ_2^1, ℓ_2^2) back to (ℓ_1^1, ℓ_1^2).

Global Iteration 2

Location (ℓ_1^1, ℓ_1^2) The incoming abstract value to (ℓ_1^1, ℓ_1^2) is:

$$in_{(\ell_1^1,\ell_1^2)}(G_1) = (cnt_1 = cnt_2 \wedge cnt_1 < n \wedge cnt_1 \geq event \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

We get the abstract value $G_2^{(\ell_1^1,\ell_1^2)}$ associated to (ℓ_1^1, ℓ_1^2):

$$G_2^{(\ell_1^1,\ell_1^2)} = (cnt_1 = cnt_2 \wedge cnt_1 < n \wedge cnt_1 \geq event \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1)$$

We have $G_1^{(\ell_1^1,\ell_1^2)} = G_2^{(\ell_1^1,\ell_1^2)}$. The global increasing sequence has converged.

Reduced Product Automaton

We construct the reduced product automaton $\mathcal{R}_s = (X, Y, Z, M, I_1 \sqcap I_2, Loc, (\ell_0^1, \ell_0^2), \phi, \tau)$ from the analysis results. The sets of variables are $X = \{event, reset\}$, $Y = \{cnt_1, cnt_2\}$, $Z = \emptyset$, $M = \{cnt_1, cnt_2\}$. The set of control locations is $Loc = \{(\ell_0^1, \ell_0^2), (\ell_1^1, \ell_1^2), (\ell_2^1, \ell_2^2)\}$. The control transition relation τ is:

$$\begin{array}{rcl} \tau & = & \{((\ell_0^1, \ell_0^2), (\ell_1^1, \ell_1^2)), ((\ell_0^1, \ell_0^2), (\ell_2^1, \ell_2^2))((\ell_1^1, \ell_1^2), (\ell_1^1, \ell_1^1)), ((\ell_1^1, \ell_1^2), (\ell_2^1, \ell_2^2)), \\ & & ((\ell_2^1, \ell_2^2), (\ell_2^1, \ell_2^2)), ((\ell_2^1, \ell_2^2), (\ell_1^1, \ell_1^1))\} \end{array}$$

The function $\phi : Loc \to \mathcal{P}(\mathcal{V}(V) \times \mathcal{V}(V))$ associating modes to control locations is:

$$\begin{array}{rcl} \phi((\ell_0^1, \ell_0^2)) & = & (cnt_1 = event \wedge cnt_1 = cnt_2 \wedge 0 \leq cnt_1 \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ \phi((\ell_1^1, \ell_1^2)) & = & (cnt_{1,0} < n \wedge cnt_{2,0} < n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\ & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\ & & \vee\ (cnt_{1,0} < n \wedge cnt_{2,0} < n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} + event \\ & & \quad \wedge\ cnt_2 = cnt_{2,0} + event \wedge 0 \leq event \leq 1 \wedge n \geq 1) \\ \phi((\ell_2^1, \ell_2^2)) & = & (cnt_{1,0} \geq n \wedge cnt_{2,0} < n \wedge reset = 1 \wedge cnt_1 = event \wedge cnt_1 = cnt_2 \\ & & \quad \wedge\ 0 \leq cnt_1 \leq 1 \wedge n \geq 1) \\ & & \vee\ (cnt_{1,0} \geq n \wedge cnt_{2,0} < n \wedge reset = 0 \wedge cnt_1 = cnt_{1,0} \\ & & \quad \wedge\ cnt_2 = cnt_{2,0} + event \wedge 0 \leq event \leq 1 \wedge n \geq 1) \end{array}$$

8.6 Reduction of Relational Mode Automata

We are interested in the analysis of reactive components represented by relational mode automata at different levels of precision. The precision of a relational mode automaton can be reduced at two different levels, either by merging modes, producing an automaton with fewer modes, or from the inside of modes, by merging disjuncts for modes defined by disjunctions of abstract relations.

8.6.1 Reduction by Merging Modes

Principle A relational mode automaton $\mathcal{R} = (X, Y, Z, M, I, Loc, \ell_0, \phi, \tau)$ can be reduced by merging some of its modes. Two modes $\phi(\ell_1)$ and $\phi(\ell_2)$ of $\mathcal{R}$ are merged by replacing the locations ℓ_1, ℓ_2 by a fresh location $\ell' \notin Loc$ where the mode associated to ℓ' is the relation $\phi(\ell_1) \cup \phi(\ell_2)$. The control transitions originating from or leading to ℓ_1 or ℓ_2 are rewritten in terms of ℓ'. If ℓ_1 or ℓ_2 is the initial location ℓ_0, then ℓ' becomes the new initial control location.

Reduction $\rightsquigarrow_{\ell_1,\ell_2}$ For $\ell_1, \ell_2 \in Loc$, the reduction relation $\mathcal{R} \rightsquigarrow_{\ell_1,\ell_2} \mathcal{R}'$ associating a reduced relational mode automaton $\mathcal{R}'$ to $\mathcal{R}$ by merging the modes $\phi(\ell_1), \phi(\ell_2)$ is defined

as:

$$\begin{array}{rcl}\mathcal{R} \rightsquigarrow_{\ell_1,\ell_2} \mathcal{R}' & \Leftrightarrow & \mathcal{R}' = (X, Y, Z, M, I, Loc', \ell_0', \phi', \tau') \\ & & \wedge\ Loc' = (Loc \setminus \{\ell_1, \ell_2\}) \cup \{\ell'\} \wedge \ell' \notin Loc \\ & & \wedge\ \ell_0' = \begin{cases} \ell' & \text{if } (\ell_1 = \ell_0) \vee (\ell_2 = \ell_0) \\ \ell_0 & \text{otherwise} \end{cases} \\ & & \wedge\ \phi'(\ell') = \phi(\ell_1) \cup \phi(\ell_2) \wedge \forall \ell \neq \ell', \phi'(\ell) = \phi(\ell) \\ & & \wedge\ \tau' = (\tau \cap (Loc' \times Loc')) \cup \{(\ell, \ell') \mid (\ell, \ell_1) \in \tau \vee (\ell, \ell_2) \in \tau\} \\ & & \qquad\qquad \cup \{(\ell', \ell) \mid (\ell_1, \ell) \in \tau \vee (\ell_2, \ell) \in \tau\}\end{array}$$

When modes $\phi(\ell_1), \phi(\ell_2)$ are disjunctions of elements of a relational abstract domain $D^\sharp$, the merged mode is $\phi'(\ell') = \mathit{merge}(\phi(\ell_1), \phi(\ell_2))$ where $\mathit{merge} : \mathcal{P}(D^\sharp) \times \mathcal{P}(D^\sharp) \to \mathcal{P}(D^\sharp)$ can be any sound merging function such that:

$$\bigcup_{\substack{T_1 \in \phi(\ell_1) \\ T_2 \in \phi(\ell_2)}} \gamma(T_1) \cup \gamma(T_2) \subseteq \bigcup_{m \in \mathit{merge}(\phi(\ell_1), \phi(\ell_2))} \gamma(m)$$

In particular, *merge* can keep separated the disjuncts of both modes such that:

$$\mathit{merge}(\phi(\ell_1), \phi(\ell_2)) = \phi(\ell_1) \vee \phi(\ell_2)$$

or in another extreme, be the join of all disjuncts:

$$\mathit{merge}(\phi(\ell_1), \phi(\ell_2)) = \bigsqcup_{T_i \in \phi(\ell_1) \cup \phi(\ell_2)} T_i$$

8.6.2 Internal Reduction

When the modes of a relational mode automaton are disjunctive relations, the precision of modes can be reduced by merging some disjuncts. For example, we can replace some disjuncts, or all of them, by their least upper bound. More generally, a mode $\phi(\ell) = T_1 \vee ... \vee T_n$ is reduced into a mode $\phi'(\ell)$ with respect to a mode-reducing function $\mathit{mred} : \mathcal{P}(D^\sharp) \to \mathcal{P}(D^\sharp)$ as:

$$\phi'(\ell) = \mathit{mred}(\phi(\ell))$$

where *mred* can be any sound mode-reducing function, satisfying:

$$\bigcup_{T_i \in \phi(\ell)} \gamma(T_i) \subseteq \bigcup_{m_i \in \mathit{mred}(\phi(\ell))} \gamma(m_i)$$

8.7 Invariants

8.7.1 Invariant of a Relational Mode Automaton

Let $\mathit{reach}^\sharp : Loc \to D^\sharp$ be the function associating approximations of the sets of reachable variable valuations at each control location of a relational mode automaton $\mathcal{R}$. The invariant $\mathit{inv}(\mathcal{R})$ is defined as:

$$\mathit{inv}(\mathcal{R}) = \bigsqcup_{\ell \in Loc} \mathit{reach}^\sharp(\ell)$$

Example 8.7.1. For $\mathcal{R}_{ev}$ representing the bounded event counter, we computed in 8.4.2 the following approximations of sets of reachable valuations:

$$\begin{array}{lcl} reach^\sharp(\ell_0) & = & (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ reach^\sharp(\ell_1) & = & (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1) \\ reach^\sharp(\ell_2) & = & (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge reset \geq 0 \wedge event \leq 1) \end{array}$$

The invariant of $\mathcal{R}_{ev}$ is:

$$\begin{array}{rcl} inv(\mathcal{R}_{ev}) & = & reach^\sharp(\ell_0) \sqcup reach^\sharp(\ell_1) \sqcup reach^\sharp(\ell_2) \\ & = & (0 \leq cnt \leq n \wedge cnt + reset \leq event + n \wedge 0 \leq reset \leq 1 \wedge event \geq 0 \\ & & \wedge\ n \geq 1) \end{array}$$

8.7.2 Weaker invariants

We can get a weaker and less-precise invariant of $\mathcal{R}$ by eliminating variables. The choice of variables to eliminate is made heuristically, according to a proof objective or high-level user-specified goals.

Example 8.7.2. If we are only interested in the bounds on the output variable *cnt* of the event counter, in terms of the counter capacity parameter n, we can use instead a weaker invariant $\mathcal{W}_{ev}$:

$$\begin{array}{rcl} \mathcal{W}_{ev} & = & inv(\mathcal{R}_{ev}) \downarrow \{cnt, n\} \\ & = & (0 \leq cnt \leq n \wedge n \geq 1) \end{array}$$

The weaker invariant $\mathcal{W}_{ev}$ has fewer constraints than $inv(\mathcal{R}_{ev})$ although it is still useful to prove that the variables of a collection of counter components are bounded. However, $\mathcal{W}_{ev}$ is not expressive enough to discover the equality of parallel event counters reacting to the same event, such as in 8.5.3.

8.7.3 Disjunctive Invariant

Conversely, we can get a more precise invariant of a relational mode automaton by keeping as separate disjuncts some or all the abstract values $reach^\sharp(\ell)$ discovered at each control location.

Example 8.7.3. For the bounded event counter, we can either keep the full disjunctive invariant:

$$\begin{array}{rcl} \mathcal{J}_{ev} & = & reach^\sharp(\ell_0) \vee reach^\sharp(\ell_1) \vee reach^\sharp(\ell_2) \\ & = & (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ & & \vee\ (cnt < n \wedge cnt + reset \geq event \wedge cnt \geq 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1) \\ & & \vee\ (cnt = n \wedge cnt \geq 1 \wedge event \geq reset \wedge reset \geq 0 \wedge event \leq 1) \end{array}$$

or just distinguish the abstract value $reach^\sharp(\ell_0)$ discovered at the initial location from the other disjuncts:

$$\begin{array}{rcl} \mathcal{J}'_{ev} & = & reach^\sharp(\ell_0) \vee (reach^\sharp(\ell_1) \sqcup reach^\sharp(\ell_2)) \\ & = & (cnt = 0 \wedge 0 \leq event \leq 1 \wedge 0 \leq reset \leq 1 \wedge n \geq 1) \\ & & \vee\ (0 \leq cnt \leq n \wedge cnt + reset \leq event + n \wedge 0 \leq reset \leq 1 \wedge event \geq 0 \wedge n \geq 1) \end{array}$$

A disjunctive invariant of the relational mode automaton representing a given reactive component can be used to compute the abstract effect of a call to the step procedure of the component, when the component is instantiated inside a larger component. The abstract effect of a call to the step procedure is computed as for disjunctive summaries in 5.2.4. An example using disjunctive invariants is given in Chapter 9.

8.7.4 Relational Augmentation

An invariant $\mathcal{J} \in \mathcal{P}(D^\sharp)$ of a relational mode automaton $\mathcal{R}$ can be augmented by adding back into $\mathcal{J}$ some constraints from the modes of $\mathcal{R}$, describing how the values of memory variables are computed in terms of their previous values and input variables. It is intended as a way to reintroduce some of the relational information given in the modes of $\mathcal{R}$, by combining it with the constraints discovered in the invariant $\mathcal{J}$.

We compute an abstract relation $\Phi \in D^\sharp$ on memory and input variables from the modes of $\mathcal{R}$ as follows:

$$\Phi = \bigsqcup_{\ell \in Loc} \left(\bigsqcup_{T_i \in \phi(\ell)} T_i \downarrow \{M, X\} \right)$$

The augmented invariant $\mathcal{Q}_\mathcal{R} \in \mathcal{P}(D^\sharp)$ is defined as:

$$\mathcal{Q}_\mathcal{R} = \bigvee_{Q_i \in \mathcal{J}} Q_i \sqcap \Phi$$

8.8 Conclusion

We proposed a new representation of the behavior of reactive components called *Relational Mode Automata* (RMA). Relational mode automata can be constructed automatically from a disjunctive summary of the step procedure of a reactive component. They are designed specifically for the analysis of reactive programs, allowing different levels of abstraction and precision.

We described a reachability analysis of RMA, which is able to analyze the parallel composition of several reactive components without requiring an explicit construction of the synchronous product of automata prior to the analysis. Instead, only the part of the composition found to be reachable during the analysis is explored and constructed.

We shown that the analysis results of a RMA representing a given reactive component can be used in the form of a disjunctive invariant to compute the effect of a call to the step procedure implementing the component, in order to analyze larger components. Thus our approach can be used for the modular analysis of reactive programs.

Our approach has been implemented for small examples using the PyApron library providing high-level Python bindings for convex polyhedra to the Apron library. In the next chapter, we present an application to a larger example which is a simplification of a real proposal of subway control system.

Chapter 9

Example: A Subway Control System

We are interested in the analysis of a larger reactive program which models the behavior of a subway control system. This is a classical example [67] for the verification of synchronous programs which has been greatly simplified from a real proposition. The control system of a simple subway train is implemented by the `train_step` procedure shown in Figure 9.1.

In our subway system, all trains on the same track receive a common time signal from a central clock, represented in each train by a boolean input variable s. Signaling beacons are installed along the tracks. The boolean input variable b is true when a beacon is detected by a train. Ideally, a train should cross one beacon per s signal. A train is controlled according to the difference *delta* between the number of received clock ticks and the number of detected beacons. A train is considered to be late when the difference *delta* becomes greater or equal to 10 and to be early when *delta* becomes lower or equal to -10.

We assume that trains have a stopping distance equal to 10 beacons. Trains are controlled using an hysteresis to avoid shaking for passenger comfort. A train begins to brake when $delta = -10$ and keeps braking until the train is completely stopped or $delta \geq 0$. A train stays stopped until $delta \geq 0$. Similarly, a train is considered to be late when $delta = 10$, and becomes ontime only when it has gained enough advance, which is when $delta \geq 0$.

When a train is stopped, no new beacons are detected. When a train is late, in order to avoid collisions with other trains on the same track, the central clock disables the time signal for the track. In other trains, it causes *delta* to decrease, compelling trains to brake when *delta* becomes lower or equal to -10. When the slower train resumes its course and returns to its normal pace, more beacons are encountered and the difference *delta* starts to decrease. The train becomes on time again when *delta* becomes lower or equal to zero. The central clock then resumes the time signal for the entire track, allowing other trains to return to their normal speed. Thus collision avoidance is achieved by *suspending time*.

In order to guarantee the safety of the subway, we want to discover bounds on the difference $nb_1 - nb_2 = delta2 - delta1$ in number of encountered beacons between any pair of trains. These bounds can then be used for the initial placement of trains to ensure the absence of collisions. We also want to prove that numerical variables in each train are bounded at all times, to guarantee the absence of arithmetic overflows at runtime.

We start by analyzing a single train as implemented by the `train_step` procedure.

Then, we show that the invariant discovered for a single train can be used in a modular way to prove the absence of collisions in a subway system made of several trains.

9.1 Detailed Summary of the `train_step` Procedure

We construct a relational mode automaton representing a single train by computing the disjunctive relational summary of the `train_step` procedure. The `train_step` procedure is represented by the following system of fixpoint equations:

$$
\begin{array}{lcl}
I^\sharp & = & (0 \leq init \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq ontime \leq 1 \wedge 0 \leq late \leq 1 \\
 & & \wedge\ 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1) \\
Q_1 & = & I \sqcap (init_0 = init \wedge ontime_0 = ontime \wedge late_0 = late \wedge onbrake_0 = onbrake \\
 & & \wedge\ stopped_0 = stopped \wedge nbrake_0 = nbrake \wedge delta_0 = delta) \\
Q_2 & = & (Q_1 \sqcap (init = 1))[init := 0][delta := 0][ontime := 1][late := 0] \\
 & & [onbrake := 0][stopped := 0][nbrake := 0] \\
Q_3 & = & (Q_1 \sqcap (init = 0 \wedge ontime = 1))[delta := delta + s - b] \\
Q_4 & = & (Q_3 \sqcap (delta \geq 10))[ontime := 0][late := 1][nbrake := 0] \\
Q_5 & = & (Q_3 \sqcap (delta \leq -10))[ontime := 0][onbrake := 1][nbrake := b] \\
Q_6 & = & (Q_3 \sqcap (-9 \leq delta \leq 9))[nbrake := 0] \\
Q_7 & = & Q_4 \sqcup Q_5 \sqcup Q_6 \\
Q_8 & = & ((Q_1 \sqcap (init = 0 \wedge ontime = 0 \wedge late = 1)) \sqcap (s = 0))[delta := delta - b] \\
 & & [nbrake := 0] \\
Q_9 & = & (Q_8 \sqcap (delta \leq 0))[ontime := 1][late := 0] \\
Q_{10} & = & (Q_8 \sqcap (delta \geq 1)) \sqcup Q_9 \\
Q_{11} & = & (Q_1 \sqcap (init = 0 \wedge ontime = 0 \wedge late = 0 \wedge onbrake = 1))[delta := delta + s - b] \\
Q_{12} & = & (Q_{11} \sqcap (delta \geq 0))[ontime := 1][onbrake := 0][nbrake := 0] \\
Q_{13} & = & (Q_{11} \sqcap (delta \leq -1 \wedge nbrake \geq 10))[stopped := 1][onbrake := 0] \\
Q_{14} & = & (Q_{11} \sqcap (delta \leq -1 \wedge nbrake \leq 9))[nbrake := nbrake + b] \\
Q_{15} & = & Q_{12} \sqcup Q_{13} \sqcup Q_{14} \\
Q_{16} & = & ((Q_1 \sqcap (init = 0 \wedge ontime = 0 \wedge late = 0 \wedge onbrake = 0 \wedge stopped = 1)) \sqcap (b = 0)) \\
 & & [delta := delta + s] \\
Q_{17} & = & (Q_{16} \sqcap (delta \geq 0))[ontime := 1][stopped := 0][nbrake := 0] \\
Q_{18} & = & (Q_{16} \sqcap (delta \leq -1)) \sqcup Q_{17} \\
Q_{19} & = & Q_1 \sqcap (init = 0 \wedge ontime = 0 \wedge late = 0 \wedge onbrake = 0 \wedge stopped = 0) \\
Q_{20} & = & Q_2 \sqcup Q_7 \sqcup Q_{15} \sqcup Q_{18} \sqcup Q_{19}
\end{array}
$$

The disjunctive summary of the `train_step` procedure is computed according to the two-level refinement scheme presented in 8.2. The global precondition $I^\sharp$ of the procedure is first refined according to preconditions on memory variables and then according to preconditions on input variables. The sets of input, output, local and memory variables of a train component are:

$$
\begin{array}{lcl}
X & = & \{s, b\} \\
Y & = & \{ontime, late, onbrake, stopped, delta\} \\
Z & = & \emptyset \\
M & = & \{init, ontime, late, onbrake, stopped, delta, nbrake\}
\end{array}
$$

```
void train_step(int * init, int s, int b, int * ontime, int * late,
                int * onbrake, int * stopped, int * nbrake, int * delta)
{
1:  if(*init == 1){
        *init = 0; *delta = 0; *ontime = 1; *late = 0;
        *onbrake = 0; *stopped = 0; *nbrake = 0;
2:  } else if(*ontime == 1){
        *delta = *delta + s - b;
3:      if(*delta >= 10){
            *ontime = 0; *late = 1; *nbrake = 0;
4:      } else if(*delta <= -10){
            *ontime = 0; *onbrake = 1; *nbrake = b;
5:      } else {
            *nbrake = 0;
6:      }
7:  } else if(*late == 1){
        assert(s == 0);
        *delta = *delta - b; *nbrake = 0;
8:      if(*delta <= 0){
            *ontime = 1; *late = 0;
9:      }
10: } else if(*onbrake == 1){
        *delta = *delta + s - b;
11:     if(*delta >= 0){
            *ontime = 1; *onbrake = 0; *nbrake = 0;
12:     } else if(*nbrake >= 10){
            *stopped = 1; *onbrake = 0;
13:     } else {
            *nbrake = *nbrake + b;
14:     }
15: } else if(*stopped == 1){
        assert(b == 0);
        *delta = *delta + s;
16:     if(*delta >= 0){
            *ontime = 1; *stopped = 0; *nbrake = 0;
17:     }
18: } else { 19: }
20:
}
```

Figure 9.1: The train_step procedure.

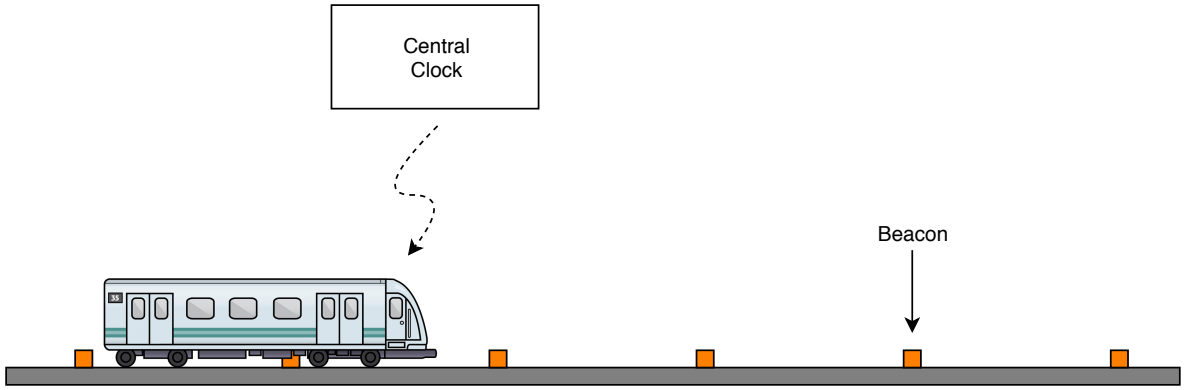


Figure 9.2: A subway track.

We denote as $M_0 = \{init_0, delta_0, nbrake_0, ontime_0, late_0, onbrake_0, stopped_0\}$ the set of variables representing previous values of memory variables.

9.1.1 Refinement on Memory Variables

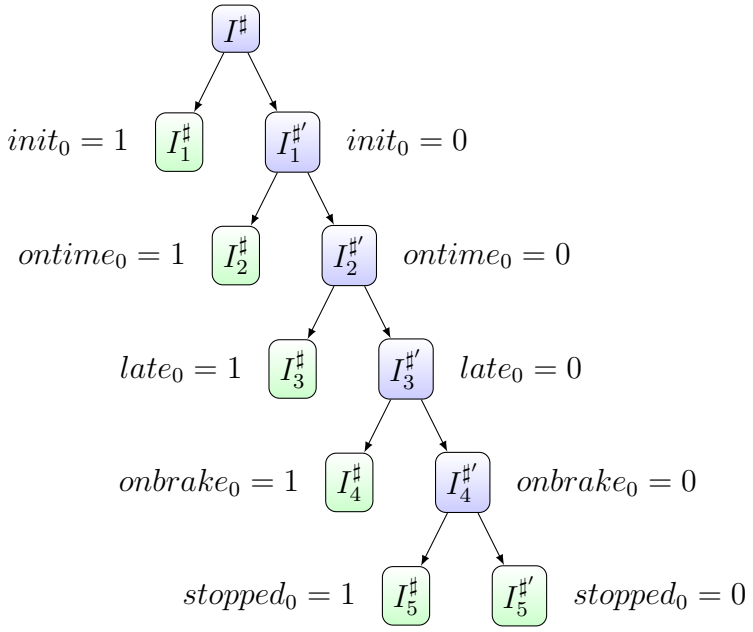


Figure 9.3: Partitioning tree of $I^\sharp$ according to preconditions on memory variables.

Refinement of $I^\sharp$ A Linear Relation Analysis of the `train_step` procedure under the global precondition $I^\sharp$ gives for Q_2:

$$\begin{aligned} Q_2 \quad &= \quad (init_0 = 1 \wedge 0 \leq ontime_0 \leq 1 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ &\quad \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1 \wedge init = 0 \wedge delta = 0 \\ &\quad \wedge\ ontime = 1 \wedge late = 0 \wedge onbrake = 0 \wedge stopped = 0 \wedge nbrake = 0) \\ Q_2 \downarrow M_0 \quad &= \quad (\mathbf{init_0 = 1} \wedge 0 \leq ontime_0 \leq 1 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ &\quad \wedge\ 0 \leq stopped_0 \leq 1) \end{aligned}$$

$I^\sharp$ can be refined according to the constraint $init_0 = 1$ and its complement $init_0 = 0$ into two new preconditions $I_1^\sharp$ and $I_1^{\sharp'}$:

$$\begin{array}{rcl} I_1^\sharp & = & I^\sharp \sqcap (init_0 = 1) \\ & = & (\mathbf{init_0 = 1} \wedge 0 \leq ontime_0 \leq 1 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_1^{\sharp'} & = & I^\sharp \sqcap (init_0 = 0) \\ & = & (\mathbf{init_0 = 0} \wedge 0 \leq ontime_0 \leq 1 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

Refinement of $I_1^\sharp$ We compute a Linear Relation Analysis of the `train_step` procedure under the precondition $I_1^\sharp$. We get $Q_3(I_1^\sharp) = \bot$, $Q_8(I_1^\sharp) = \bot$, $Q_{11}(I_1^\sharp) = \bot$, $Q_{16}(I_1^\sharp) = \bot$ and $Q_{19}(I_2^\sharp) = \bot$. The precondition $I_1^\sharp$ can not be refined further.

Refinement of $I_1^{\sharp'}$ We compute a Linear Relation Analysis under the precondition $I_1^{\sharp'}$. We can refine $I_1^{\sharp'}$ at program point 3, according to the constraint $ontime_0 = 1$ and its complement $ontime_0 = 0$ into $I_2^\sharp$ and $I_2^{\sharp'}$:

$$\begin{array}{rcl} I_2^\sharp & = & I_1^{\sharp'} \sqcap (ontime_0 = 1) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_2^{\sharp'} & = & I_1^{\sharp'} \sqcap (ontime_0 = 0) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

Refinement of $I_2^\sharp$ Under precondition $I_2^\sharp$, we get $Q_8(I_2^\sharp) = \bot$, $Q_{11}(I_2^\sharp) = \bot$, $Q_{16}(I_2^\sharp) = \bot$ and $Q_{19}(I_2^\sharp) = \bot$. We do not consider the program points 4, 5, 6 since they are guarded by tests depending on inputs. The test `*delta >= 10` depends on the current value of *delta* which is computed by the statement `*delta = *delta + s - b` involving input variables s and b. Thus $I_2^\sharp$ can not be refined further according to memory variables.

Refinement of $I_2^{\sharp'}$ We can refine $I_2^{\sharp'}$ at program point 8 according to the constraint $late_0 = 1$ and its complement $late_0 = 0$ into $I_3^\sharp$ and $I_3^{\sharp'}$:

$$\begin{array}{rcl} I_3^\sharp & = & I_2^{\sharp'} \sqcap (late_0 = 1) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 1} \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_3^{\sharp'} & = & I_2^{\sharp'} \sqcap (late_0 = 0) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 0} \wedge 0 \leq onbrake_0 \leq 1 \\ & & \wedge\ 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

Refinement of $I_3^\sharp$ Under precondition $I_3^\sharp$, we get $Q_{11}(I_3^\sharp) = \bot$, $Q_{16}(I_3^\sharp) = \bot$ and $Q_{19}(I_3^\sharp) = \bot$. We do not consider the program point 9 as it is guarded by the test `*delta <= 0` and the current value of *delta* is computed by the statement `*delta = *delta - b` which depends on the input variable b. Thus $I_3^\sharp$ can not be refined according to memory variables.

Refinement of $I_3^{\sharp'}$ We can refine $I_3^{\sharp'}$ at program point 11 according to the constraint $onbrake_0 = 1$ and its complement $onbrake_0 = 0$ into $I_4^\sharp$ and $I_4^{\sharp'}$:

$$\begin{aligned} I_4^\sharp &= I_3^{\sharp'} \sqcap (onbrake_0 = 1) \\ &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 1} \wedge \mathbf{onbrake_0 = 1} \\ &\quad \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_4^{\sharp'} &= I_3^{\sharp'} \sqcap (onbrake_0 = 0) \\ &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 0} \wedge \mathbf{onbrake_0 = 0} \\ &\quad \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{aligned}$$

Refinement of $I_4^\sharp$ Under precondition $I_4^\sharp$, we get $Q_{16}(I_4^\sharp) = \bot$ and $Q_{19}(I_4^\sharp) = \bot$. Tests at program points 12, 13 and 14 depends on input variables. The `else if(*nbrake >= 10)` branch is entered if $delta \leq -1$ and Q_{13} is defined in the system of equations as being guarded by the condition $delta \leq -1 \wedge nbrake \geq 10$. Thus $I_4^\sharp$ can not be refined according to memory variables only.

Refinement of $I_4^{\sharp'}$ We can refine $I_4^{\sharp'}$ at program point 16 according to the constraint $stopped_0 = 1$ and its complement $stopped_0 = 0$ into $I_5^\sharp$ and $I_5^{\sharp'}$:

$$\begin{aligned} I_5^\sharp &= I_4^{\sharp'} \sqcap (stopped_0 = 1) \\ &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 1} \wedge \mathbf{onbrake_0 = 1} \\ &\quad \wedge \mathbf{stopped_0 = 1} \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_5^{\sharp'} &= I_4^{\sharp'} \sqcap (stopped_0 = 0) \\ &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 0} \wedge \mathbf{onbrake_0 = 0} \\ &\quad \wedge \mathbf{stopped_0 = 0} \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{aligned}$$

Refinement of $I_5^\sharp$ Under precondition $I_5^\sharp$, we get $Q_{19}(I_5^\sharp) = \bot$. As for other preconditions, $I_5^\sharp$ can not be refined at Q_{16} guarded by `*delta >= 0`.

Abstract Partition on Memory Variables We get an abstract partition $\delta_m^\sharp = \{I_1^\sharp, I_2^\sharp, I_3^\sharp, I_4^\sharp, I_5^\sharp, I_5^{\sharp'}\}$ of the global precondition $I^\sharp$.

9.1.2 Refinement on Input Variables

Refinement of $I_2^\sharp$ We get for $Q_4(I_2^\sharp)$ under precondition $I_2^\sharp$:

$$\begin{aligned} Q_4(I_2^\sharp) &= (init_0 = 0 \wedge ontime_0 = 1 \wedge delta = delta_0 + s - b \wedge onbrake = onbrake_0 \\ &\quad \wedge stopped = stopped_0 \wedge ontime = 0 \wedge nbrake = 0 \wedge late = 1 \wedge init = 0 \\ &\quad \wedge b + delta \leq delta_0 + 1 \wedge b + delta \geq delta_0 \wedge 0 \leq late_0 \leq 1 \\ &\quad \wedge 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1 \wedge delta \geq 10 \wedge b \geq 0) \\ Q_4 \downarrow \{M_0, X\} &= (init_0 = 0 \wedge ontime_0 = 1 \wedge \mathbf{delta_0 + s \geq 10 + b} \wedge 0 \leq b \leq 1 \wedge 0 \leq s \leq 1 \\ &\quad \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1) \end{aligned}$$

We can refine $I_2^\sharp$ into $I_6^\sharp$ and $I_6^{\sharp'}$ at program point 2, according to the constraint $delta_0 + s \geq 10 + b$ and its complement $delta_0 + s < 10 + b$ which involve input variables.

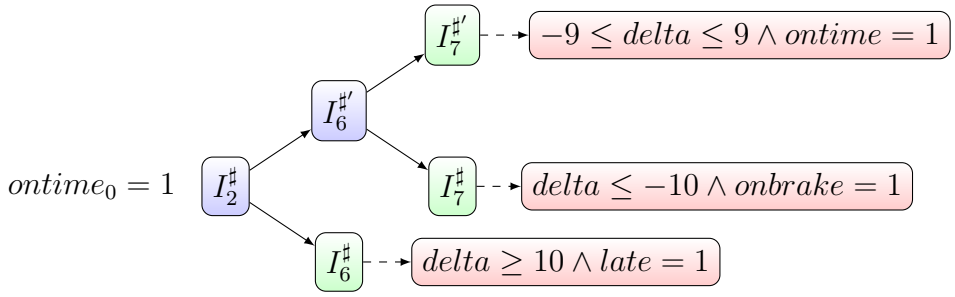


Figure 9.4: Partitioning of $I_2^\sharp$ according to input variables, with the ranges of possible values for *delta* under preconditions $I_7^{\sharp'}$, $I_7^\sharp$, $I_6^\sharp$.

It corresponds to the condition `*delta >= 10` on the current value of *delta*.

$$\begin{array}{rcl} I_6^\sharp & = & I_2^\sharp \sqcap (delta_0 + s \geq 10 + b) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{delta_0 + s \geq b + 10} \wedge 0 \leq late_0 \leq 1 \\ & & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_6^{\sharp'} & = & I_2^\sharp \sqcap (delta_0 + s < 10 + b) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{delta_0 + s < b + 10} \wedge 0 \leq late_0 \leq 1 \\ & & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

Refinement of $I_6^{\sharp'}$ We get for $Q_5(I_6^{\sharp'})$:

$$\begin{array}{rcl} Q_5(I_6^{\sharp'}) \downarrow \{M_0, X\} & = & (init_0 = 0 \wedge ontime_0 = 1 \wedge \mathbf{delta_0 + s + 10 \leq b} \wedge 0 \leq late_0 \leq 1 \\ & & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

$I_6^{\sharp'}$ can be refined into $I_7^\sharp$ and $I_7^{\sharp'}$ at program point 5, according to the constraint $delta_0 + s + 10 \leq b$, which corresponds to the condition `*delta <= -10`.

$$\begin{array}{rcl} I_7^\sharp & = & I_6^{\sharp'} \sqcap (delta_0 + s + 10 \leq b) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{delta_0 + s + 10 \leq b} \wedge 0 \leq late_0 \leq 1 \\ & & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ I_7^{\sharp'} & = & I_6^{\sharp'} \sqcap (delta_0 + s + 10 > b) \\ & = & (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{delta_0 + s + 10 > b} \wedge 0 \leq late_0 \leq 1 \\ & & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

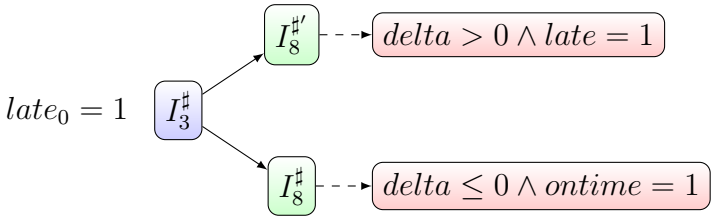


Figure 9.5: Partitioning of $I_3^\sharp$ according to input variables, with the ranges of possible values for *delta* under preconditions $I_8^\sharp$ and $I_8^{\sharp'}$.

Refinement of $I_3^\sharp$

$$\begin{aligned} Q_9(I_3^\sharp) \downarrow \{M_0, X\} \quad = \quad & (init_0 = 0 \wedge late_0 = 1 \wedge ontime_0 = 0 \wedge \mathbf{b \geq delta_0} \wedge s = 0 \\ & \wedge\ 0 \leq onbrake_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq b \leq 1) \end{aligned}$$

$I_3^\sharp$ is refined into $I_8^\sharp$ and $I_8^{\sharp'}$ according to the constraint $b \geq delta_0$:

$$\begin{aligned} I_8^\sharp &= I_3^\sharp \sqcap (b \geq delta_0) \\ I_8^{\sharp'} &= I_3^\sharp \sqcap (b < delta_0) \end{aligned}$$

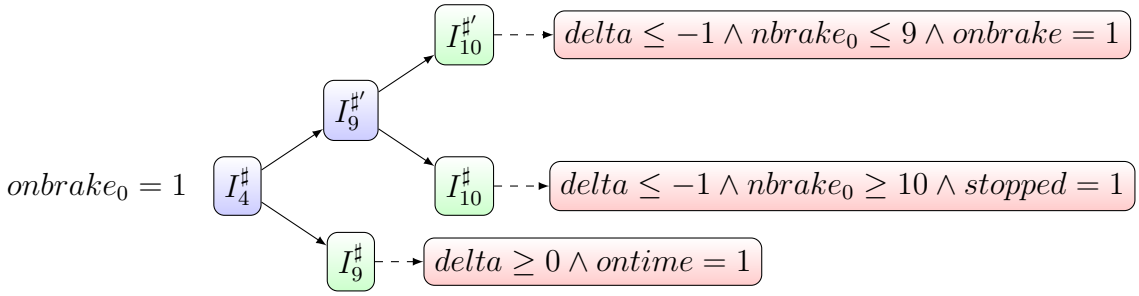


Figure 9.6: Partitioning of $I_4^\sharp$ according to input variables, with the ranges of possible values for *delta* under preconditions $I_9^\sharp, I_{10}^\sharp, I_{10}^{\sharp'}$.

Refinement of $I_4^\sharp$

$$\begin{aligned} Q_{12}(I_4^\sharp) \downarrow \{M_0, X\} \quad = \quad & (init_0 = 0 \wedge onbrake_0 = 1 \wedge late_0 = 0 \wedge ontime_0 = 0 \\ & \wedge\ \mathbf{delta_0 + s \geq b} \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq s \leq 1) \end{aligned}$$

$I_4^\sharp$ is refined into $I_9^\sharp$ and $I_9^{\sharp'}$ according to the constraint $delta_0 + s \geq b$:

$$\begin{aligned} I_9^\sharp &= I_4^\sharp \sqcap (delta_0 + s \geq b) \\ I_9^{\sharp'} &= I_4^\sharp \sqcap (delta_0 + s < b) \end{aligned}$$

Refinement of $I_9^{\sharp'}$

$$\begin{aligned} Q_{13}(I_9^{\sharp'}) \downarrow \{M_0, X\} \quad = \quad & (init_0 = 0 \wedge onbrake_0 = 1 \wedge late_0 = 0 \wedge ontime_0 = 0 \\ & \wedge\ 0 \leq stopped_0 \leq 1 \wedge \mathbf{nbrake_0 \geq 10} \wedge b \geq delta_0 + s + 1 \\ & \wedge\ 0 \leq b \leq 1 \wedge 0 \leq s \leq 1) \end{aligned}$$

$I_9^{\sharp'}$ is refined into $I_{10}^\sharp$ and $I_{10}^{\sharp'}$ according to the constraint $nbrake_0 \geq 10$:

$$\begin{aligned} I_{10}^\sharp &= I_9^{\sharp'} \sqcap (nbrake_0 \geq 10) \\ I_{10}^{\sharp'} &= I_9^{\sharp'} \sqcap (nbrake_0 < 10) \end{aligned}$$

Refinement of $I_5^\sharp$

$$\begin{aligned} Q_{17}(I_5^\sharp) \downarrow \{M_0, X\} \quad = \quad & (init_0 = 0 \wedge stopped_0 = 1 \wedge ontime_0 = 0 \wedge onbrake_0 = 0 \\ & \wedge\ late_0 = 0 \wedge b = 0 \wedge 0 \leq s \leq 1 \wedge \mathbf{delta_0 + s \geq 0}) \end{aligned}$$

$I_5^\sharp$ is refined into $I_{11}^\sharp$ and $I_{11}^{\sharp'}$ according to the constraint $delta_0 + s \geq 0$.

$$\begin{aligned} I_{11}^\sharp &= I_5^\sharp \sqcap (delta_0 + s \geq 0) \\ I_{11}^{\sharp'} &= I_5^\sharp \sqcap (delta_0 + s < 0) \end{aligned}$$

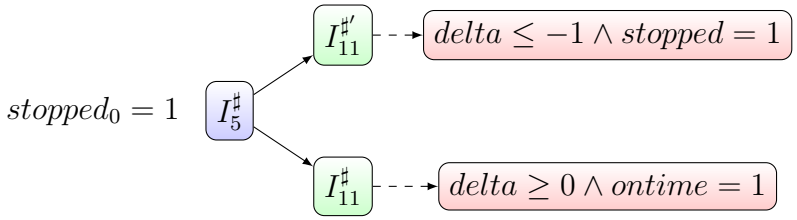


Figure 9.7: Partitioning of $I_5^\sharp$ according to input variables, with the ranges of possible values for *delta* under preconditions $I_{11}^\sharp$ and $I_{11}^{\sharp'}$.

9.1.3 Summary

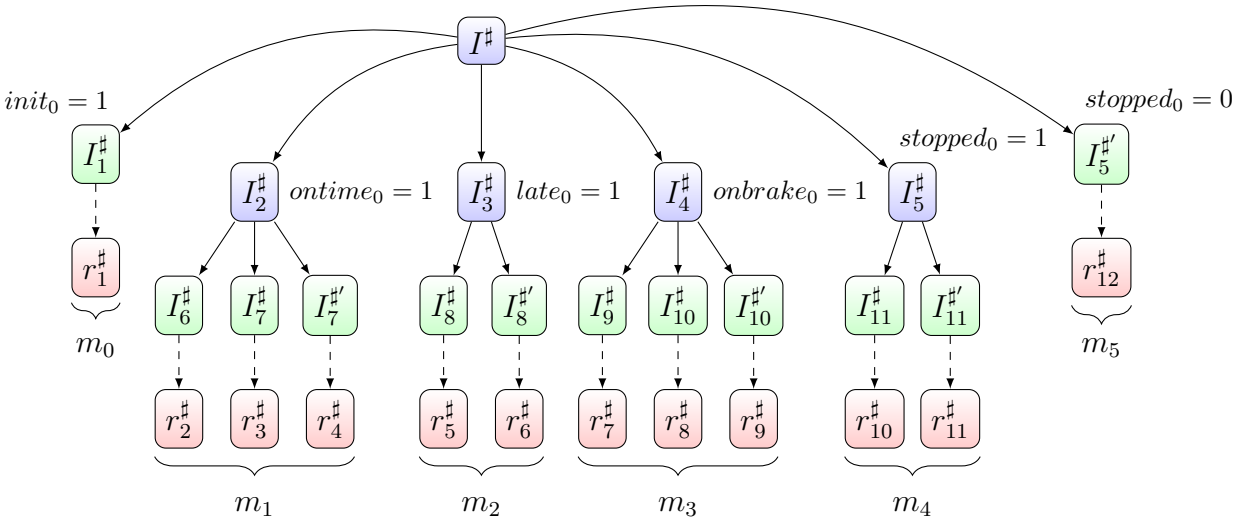


Figure 9.8: Disjunctive summary of the `train_step` procedure and grouping of summary members into modes $m_0, m_1, m_2, m_3, m_4, m_5$.

Preconditions obtained by considering constraints on previous values of memory variables have been refined according to constraints depending on input variables. The set of preconditions $\delta = \{I_1^\sharp, I_6^\sharp, I_7^\sharp, I_7^{\sharp'}, I_8^\sharp, I_8^{\sharp'}, I_9^\sharp, I_{10}^\sharp, I_{10}^{\sharp'}, I_{11}^\sharp, I_{11}^{\sharp'}, I_5^{\sharp'}\}$ is an abstract partition of the global precondition $I^\sharp$ of the `train_step` procedure.

The disjunctive relational summary $\mathcal{S}_{train}$ of the `train_step` procedure is:

$$\mathcal{S}_{train} = \{r_1^\sharp, r_2^\sharp, r_3^\sharp, r_4^\sharp, r_5^\sharp, r_6^\sharp, r_7^\sharp, r_8^\sharp, r_9^\sharp, r_{10}^\sharp, r_{11}^\sharp, r_{12}^\sharp\}$$

where $r_1^\sharp = Q_{20}(I_1^\sharp)$, $r_2^\sharp = Q_{20}(I_6^\sharp)$, $r_3^\sharp = Q_{20}(I_7^\sharp)$, $r_4^\sharp = Q_{20}(I_7^{\sharp'})$, $r_5^\sharp = Q_{20}(I_8^\sharp)$, $r_6^\sharp = Q_{20}(I_8^{\sharp'})$, $r_7^\sharp = Q_{20}(I_9^\sharp)$, $r_8^\sharp = Q_{20}(I_{10}^\sharp)$, $r_9^\sharp = Q_{20}(I_{10}^{\sharp'})$, $r_{10}^\sharp = Q_{20}(I_{11}^\sharp)$, $r_{11}^\sharp = Q_{20}(I_{11}^{\sharp'})$, $r_{12}^\sharp = Q_{20}(I_5^{\sharp'})$.

$$\begin{aligned}
r_1^\sharp \;=\;& (\mathbf{init_0 = 1} \wedge \mathbf{init = 0} \wedge ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge late = 0 \\
& \wedge nbrake = 0 \wedge delta = 0 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \wedge 0 \leq ontime_0 \leq 1 \\
& \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq s \leq 1) \\
r_2^\sharp \;=\;& (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{late = 1} \wedge delta = delta_0 + s - b \wedge onbrake = onbrake_0 \\
& \wedge stopped = stopped_0 \wedge ontime = 0 \wedge nbrake = 0 \wedge init = 0 \wedge delta_0 + 1 \geq b + delta \\
& \wedge b + delta \geq delta_0 \wedge 0 \leq b \leq 1 \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq onbrake \leq 1 \\
& \wedge 0 \leq stopped \leq 1 \wedge delta \geq 10)
\end{aligned}$$

$$
\begin{aligned}
r_3^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{onbrake = 1} \wedge delta \leq -10 \wedge delta = delta_0 + s - b \\
&\wedge nbrake = b \wedge late = late_0 \wedge stopped = stopped_0 \wedge ontime = 0 \wedge init = 0 \\
&\wedge delta_0 + 1 \geq b + delta \wedge 0 \leq b \leq 1 \wedge 0 \leq late \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\
&\wedge 0 \leq stopped \leq 1 \wedge b + delta \geq delta_0) \\
r_4^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 1} \wedge \mathbf{ontime = 1} \wedge delta = delta_0 + s - b \wedge -9 \leq delta \leq 9 \\
&\wedge onbrake = onbrake_0 \wedge stopped = stopped_0 \wedge nbrake = 0 \wedge init = 0 \\
&\wedge delta_0 + 1 \geq b + delta \wedge 0 \leq b \leq 1 \wedge 0 \leq late \leq 1 \wedge 0 \leq onbrake \leq 1 \\
&\wedge 0 \leq stopped \leq 1 \wedge b + delta \geq delta_0) \\
r_5^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{late_0 = 1} \wedge \mathbf{ontime = 1} \wedge late = 0 \wedge delta = delta_0 - b \wedge s = 0 \\
&\wedge onbrake = onbrake_0 \wedge stopped = stopped_0 \wedge ontime_0 = 0 \wedge nbrake = 0 \wedge init = 0 \\
&\wedge 0 \leq b \leq 1 \wedge delta \leq 0 \wedge 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1) \\
r_6^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{late_0 = 1} \wedge \mathbf{late = 1} \wedge delta \geq 1 \wedge delta = delta_0 - b \wedge s = 0 \\
&\wedge ontime_0 = 0 \wedge ontime = 0 \wedge nbrake = 0 \wedge init = 0 \wedge 0 \leq b \leq 1 \\
&\wedge 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1) \\
r_7^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{onbrake_0 = 1} \wedge \mathbf{ontime = 1} \wedge onbrake = 0 \wedge delta \geq 0 \\
&\wedge delta = delta_0 + s - b \wedge stopped = stopped_0 \wedge ontime_0 = 0 \wedge nbrake = 0 \\
&\wedge late_0 = 0 \wedge late = 0 \wedge init = 0 \wedge delta_0 + 1 \geq b + delta \wedge b + delta \geq delta_0 \\
&\wedge 0 \leq b \leq 1 \wedge 0 \leq stopped \leq 1) \\
r_8^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{onbrake_0 = 1} \wedge \mathbf{stopped = 1} \wedge onbrake = 0 \wedge delta \leq -1 \\
&\wedge delta = delta_0 + s - b \wedge nbrake = nbrake_0 \wedge ontime_0 = 0 \wedge ontime = 0 \\
&\wedge late_0 = 0 \wedge late = 0 \wedge init = 0 \wedge delta_0 + 1 \geq b + delta \wedge b + delta \geq delta_0 \\
&\wedge 0 \leq b \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge nbrake \geq 10) \\
r_9^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{onbrake_0 = 1} \wedge \mathbf{onbrake = 1} \wedge delta = delta_0 + s - b \\
&\wedge stopped = stopped_0 \wedge ontime_0 = 0 \wedge ontime = 0 \wedge late_0 = 0 \wedge late = 0 \\
&\wedge init = 0 \wedge nbrake = nbrake_0 + b \wedge delta_0 + 1 \geq b + delta \wedge 0 \leq b \leq 1 \\
&\wedge delta \leq -1 \wedge 0 \leq stopped \leq 1 \wedge nbrake \leq b + 9 \wedge b + delta \geq delta_0) \\
r_{10}^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{stopped_0 = 1} \wedge \mathbf{ontime = 1} \wedge delta = delta_0 + s \wedge stopped = 0 \\
&\wedge ontime_0 = 0 \wedge onbrake_0 = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late_0 = 0 \wedge late = 0 \\
&\wedge init = 0 \wedge b = 0 \wedge delta_0 + 1 \geq delta \wedge delta \geq delta_0 \wedge delta \geq 0) \\
r_{11}^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{stopped_0 = 1} \wedge \mathbf{stopped = 1} \wedge nbrake = nbrake_0 \wedge delta = delta_0 + s \\
&\wedge ontime_0 = 0 \wedge ontime = 0 \wedge onbrake_0 = 0 \wedge onbrake = 0 \wedge late_0 = 0 \wedge late = 0 \\
&\wedge init = 0 \wedge b = 0 \wedge delta \leq -1 \wedge delta_0 + 1 \geq delta \wedge delta \geq delta_0) \\
r_{12}^\sharp &= (\mathbf{init_0 = 0} \wedge \mathbf{ontime_0 = 0} \wedge \mathbf{late_0 = 0} \wedge \mathbf{onbrake_0 = 0} \wedge \mathbf{stopped_0 = 0} \\
&\wedge delta = delta_0 \wedge nbrake = nbrake_0 \wedge stopped = stopped_0 \wedge ontime = ontime_0 \\
&\wedge onbrake = onbrake_0 \wedge late = late_0 \wedge init = 0 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1)
\end{aligned}
$$

The members of the disjunctive summary $\mathcal{S}_{train}$ can be grouped into modes according to which precondition on memory variables they have been derived from, as shown in Figure 9.8. The modes of a train are as follows:

$$
\begin{aligned}
m_0 &= r_1^\sharp \\
m_1 &= (r_2^\sharp \vee r_3^\sharp \vee r_4^\sharp) \\
m_2 &= (r_5^\sharp \vee r_6^\sharp) \\
m_3 &= (r_7^\sharp \vee r_8^\sharp \vee r_9^\sharp) \\
m_4 &= (r_{10}^\sharp \vee r_{11}^\sharp) \\
m_5 &= r_{12}^\sharp
\end{aligned}
$$

9.2 Relational Mode Automaton for a Train

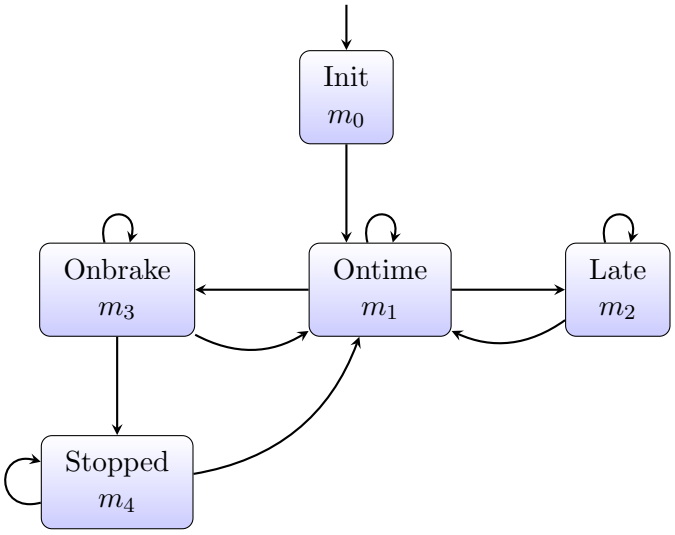


Figure 9.9: Relational Mode Automaton $\mathcal{R}_{train}$.

From the modes of a train given by the summary of the `train_step` procedure, we construct the relational mode automaton $R_{train} = (X, Y, Z, M, I, Loc, \ell_{init}, \phi, \tau)$ shown in Figure 9.9. The initial variable valuations are:

$$\begin{aligned} I \;=\; & (0 \leq s \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq ontime \leq 1 \wedge 0 \leq late \leq 1 \wedge 0 \leq onbrake \leq 1 \\ & \wedge\; 0 \leq stopped \leq 1 \wedge init = 1) \end{aligned}$$

The set of control locations is $Loc = \{\ell_{init}, \ell_{ontime}, \ell_{late}, \ell_{onbrake}, \ell_{stopped}\}$ and the modes are $\phi(\ell_{init}) = m_0$, $\phi(\ell_{ontime}) = m_1$, $\phi(\ell_{late}) = m_2$, $\phi(\ell_{onbrake}) = m_3$, $\phi(\ell_{stopped}) = m_4$.

We can write each mode $\phi(\ell)$ as a conjunction $A_\ell \wedge T$ of a precondition A_ℓ and a disjunction $T = T_{\ell,\ell_0} \vee ... \vee T_{\ell,\ell_n}$ of linear input-output relations $(T_i)_{i=1..n}$. The precondition A_ℓ gives the domain of the mode and the disjunction T gives the possible ways in which the variables can be updated. The function $\phi : Loc \to \mathcal{V}(V) \times \mathcal{V}(V)$ associating modes to control locations is:

$$\begin{aligned} \phi(\ell_{init}) \;=\; & (delta = 0 \wedge late = 0 \wedge stopped = 0 \wedge onbrake = 0 \\ & \wedge\; nbrake = 0 \wedge ontime = 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{aligned}$$

$$\begin{aligned} \phi(\ell_{ontime}) \;=\; & (\mathbf{ontime_0 = 1} \wedge 0 \leq late_0 \leq 1 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\ & \wedge\; 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ & \wedge\; ((-\mathbf{9} \leq \mathbf{delta} \leq \mathbf{9} \wedge \mathbf{ontime = 1} \wedge delta = delta_0 + s - b \wedge late = late_0 \\ & \quad \wedge\; stopped = stopped_0 \wedge onbrake = onbrake_0 \wedge nbrake = 0) \\ & \vee\; (\mathbf{delta \geq 10} \wedge \mathbf{late = 1} \wedge delta = delta_0 + s - b \wedge stopped = stopped_0 \\ & \quad \wedge\; onbrake = onbrake_0 \wedge nbrake = 0 \wedge ontime = 0) \\ & \vee\; (\mathbf{delta \leq -10} \wedge \mathbf{onbrake = 1} \wedge delta = delta_0 + s - b \\ & \quad \wedge\; stopped = stopped_0 \wedge nbrake = b \wedge late = late_0 \wedge ontime = 0)) \end{aligned}$$

$$\begin{aligned}
\phi(\ell_{late}) \;=\; & (\mathbf{late_0 = 1} \wedge ontime_0 = 0 \wedge 0 \leq stopped_0 \leq 1 \wedge 0 \leq onbrake_0 \leq 1 \\
& \wedge\; 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\
& \wedge\; ((\mathbf{delta \geq 1} \wedge \mathbf{late = 1} \wedge delta = delta_0 + s - b \wedge s = 0 \wedge ontime = ontime_0 \\
& \quad \wedge\; onbrake = onbrake_0 \wedge stopped = stopped_0 \wedge nbrake = 0) \\
& \vee\; (\mathbf{delta \leq 0} \wedge \mathbf{ontime = 1} \wedge delta = delta_0 + s - b \wedge stopped = stopped_0 \\
& \quad \wedge\; onbrake = onbrake_0 \wedge nbrake = 0 \wedge late = 0))
\end{aligned}$$

$$\begin{aligned}
\phi(\ell_{onbrake}) \;=\; & (\mathbf{onbrake_0 = 1} \wedge ontime_0 = 0 \wedge late_0 = 0 \wedge 0 \leq stopped_0 \leq 1 \\
& \wedge\; 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\
& \wedge\; ((\mathbf{delta \leq -1} \wedge \mathbf{onbrake = 1} \wedge delta = delta_0 + s - b \wedge late = late_0 \\
& \quad \wedge\; stopped = stopped_0 \wedge nbrake = nbrake_0 + b \wedge ontime = ontime_0 \\
& \quad \wedge\; nbrake_0 \leq 9) \\
& \vee\; (\mathbf{delta \geq 0} \wedge \mathbf{ontime = 1} \wedge delta = delta + s - b \wedge late = late_0 \\
& \quad \wedge\; stopped = stopped_0 \wedge onbrake = 0 \wedge nbrake = 0) \\
& \vee\; (\mathbf{nbrake_0 \geq 10} \wedge \mathbf{stopped = 1} \wedge delta = delta_0 + s - b \wedge late = late_0 \\
& \quad \wedge\; onbrake = 0 \wedge nbrake = nbrake_0 \wedge ontime = ontime_0))
\end{aligned}$$

$$\begin{aligned}
\phi(\ell_{stopped}) \;=\; & (\mathbf{stopped_0 = 1} \wedge ontime_0 = 0 \wedge onbrake_0 = 0 \wedge late_0 = 0 \\
& \wedge\; 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\
& \wedge\; ((\mathbf{delta \leq -1} \wedge \mathbf{stopped = 1} \wedge delta = delta_0 + s - b \wedge b = 0 \\
& \quad \wedge\; nbrake_0 \geq 10 \wedge late = late_0 \wedge onbrake = onbrake_0 \wedge nbrake = nbrake_0 \\
& \quad \wedge\; ontime = ontime_0) \\
& \vee\; (\mathbf{delta \geq 0} \wedge \mathbf{ontime = 1} \wedge delta = delta_0 + s - b \wedge late = late_0 \\
& \quad \wedge\; stopped = stopped_0 \wedge onbrake = onbrake_0 \wedge nbrake = 0))
\end{aligned}$$

The control transition relation $\tau \subseteq Loc \times \mathcal{V}(V) \times Loc$ is:

$$\begin{aligned}
\tau \;=\; & \{(\ell_{init}, \top, \ell_{ontime}), (\ell_{ontime}, \top, \ell_{ontime}), (\ell_{ontime}, \top, \ell_{late}), (\ell_{ontime}, \top, \ell_{onbrake}), \\
& (\ell_{late}, \top, \ell_{late}), (\ell_{late}, \top, \ell_{ontime}), (\ell_{onbrake}, \top, \ell_{onbrake}), (\ell_{onbrake}, \top, \ell_{ontime}), \\
& (\ell_{onbrake}, \top, \ell_{stopped}), (\ell_{stopped}, \top, \ell_{stopped}), (\ell_{stopped}, \top, \ell_{ontime})\}
\end{aligned}$$

9.3 Analysis of a Single Train

For each control location $\ell \neq \ell_{init}$, we denote as $stay_\ell$ a property which is satisfied when control stays at ℓ:

$$\begin{aligned}
stay_{ontime} &= (-9 \leq delta \leq 9) \\
stay_{late} &= (delta \geq 1) \\
stay_{onbrake} &= (delta \leq -1 \wedge nbrake \leq 10) \\
stay_{stopped} &= (delta \leq -1 \wedge nbrake \geq 10)
\end{aligned}$$

We define a limited widening operator ∇_ℓ for each control location ℓ as:

$$P \nabla_\ell Q = (P \nabla Q) \sqcap stay_\ell$$

Initially, only the control location ℓ_{init} is known to be reachable. The initial term of the global increasing sequence is trivially:

$$\begin{aligned}
G_{\ell_{init}} &= I \\
&= (0 \leq s \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq ontime \leq 1 \wedge 0 \leq late \leq 1 \\
&\quad \wedge\; 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1)
\end{aligned}$$

The incoming abstract value $in_{\ell_{ontime}}$ at ℓ_{ontime} is:

$$\begin{array}{rcl} in_{\ell_{ontime}} & = & post(\ell_{init}, \ell_{ontime})(G_{\ell_{init}}) \\ & = & (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ & & \wedge\ delta = 0 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

The abstract value $G_{\ell_{ontime}}$ is the limit of a local increasing and decreasing sequence at ℓ_{ontime} starting with $in_{\ell_{ontime}}$.

Local iteration at ℓ_{ontime}

We compute a local increasing sequence at ℓ_{ontime} as follows:

$$\begin{array}{rcl} Y_0^{\ell_{ontime}} & = & (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ & & \wedge\ delta = 0 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ Y_{k+1}^{\ell_{ontime}} & = & Y_k^{\ell_{ontime}} \nabla_{\ell_{ontime}} (Y_k^{\ell_{ontime}} \sqcup post(\ell_{ontime}, \ell_{ontime})(Y_k^{\ell_{ontime}})) \end{array}$$

After convergence, we get:

$$\begin{array}{rcl} G_{\ell_{ontime}} & = & (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ & & \wedge -9 \leq delta \leq 9 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \end{array}$$

Local iteration at $\ell_{onbrake}$

The incoming abstract value at $\ell_{onbrake}$ is:

$$\begin{array}{rcl} in_{\ell_{onbrake}} & = & post(\ell_{ontime}, \ell_{onbrake})(G_{\ell_{ontime}}) \\ & = & (onbrake = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge nbrake = 1 \wedge late = 0 \\ & & \wedge\ delta = -10 \wedge s = 0 \wedge b = 1) \end{array}$$

We compute a local increasing sequence at $\ell_{onbrake}$, we get:

$$\begin{array}{rcl} G_{\ell_{onbrake}} & = & (onbrake = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge late = 0 \\ & & \wedge\ b \leq 1 \wedge nbrake \geq b + s \wedge delta \leq -1 \wedge nbrake \leq 10 \wedge s \geq 0 \wedge nbrake \geq 1 \\ & & \wedge\ delta + nbrake \geq s - 9) \end{array}$$

Local iteration at $\ell_{stopped}$

The incoming abstract value at $\ell_{stopped}$ is:

$$\begin{array}{rcl} in_{\ell_{stopped}} & = & post(\ell_{onbrake}, \ell_{stopped})(G_{\ell_{onbrake}}) \\ & = & (stopped = 1 \wedge ontime = 0 \wedge onbrake = 0 \\ & & \wedge\ nbrake = 10 \wedge -19 + s \leq delta \leq s - 1 \\ & & \wedge\ late = 0 \wedge 0 \leq s \leq 1 \wedge b = 0) \end{array}$$

We compute a local increasing sequence at $\ell_{stopped}$, we get:

$$\begin{array}{rcl} G_{\ell_{stopped}} & = & (stopped = 1 \wedge ontime = 0 \wedge onbrake = 0 \wedge nbrake = 10 \\ & & \wedge\ late = 0 \wedge b = 0 \wedge delta \leq -1 \wedge 0 \leq s \leq 1 \wedge delta \geq s - 19) \end{array}$$

Local iteration at ℓ_{ontime}

We compute a new incoming value at ℓ_{ontime} with the contributions of $G_{\ell_{onbrake}}$ and $G_{\ell_{stopped}}$.

$$\begin{aligned} in_{\ell_{ontime}} &= post(\ell_{init}, \ell_{ontime})(G_{\ell_{init}}) \sqcup post(\ell_{onbrake}, \ell_{ontime})(G_{\ell_{onbrake}}) \\ &\quad \sqcup post(\ell_{stopped}, \ell_{ontime})(G_{\ell_{stopped}}) \sqcup post(\ell_{ontime}, \ell_{ontime})(G_{\ell_{ontime}}) \\ &= (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ &\quad \wedge -9 \leq delta \leq 9 \wedge 0 \leq b \leq 1 \wedge 0 \leq s \leq 1) \end{aligned}$$

The incoming value has not changed, thus the loop $\ell_{ontime}, \ell_{onbrake}, \ell_{stopped}$ has converged.

Local iteration at ℓ_{late}

The incoming abstract value at ℓ_{late} is:

$$\begin{aligned} in_{\ell_{late}} &= post(\ell_{ontime}, \ell_{late})(G_{\ell_{late}}) \\ &= (late = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge onbrake = 0 \wedge nbrake = 0 \\ &\quad \wedge delta = 10 \wedge s = 1 \wedge b = 0) \end{aligned}$$

We compute a local increasing sequence at ℓ_{late}, we get:

$$\begin{aligned} G_{\ell_{late}} &= (late = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge onbrake = 0 \wedge nbrake = 0 \\ &\quad \wedge b + delta = 10 \wedge b + s \leq 1 \wedge 0 \leq b \leq 9) \end{aligned}$$

Local iteration at ℓ_{ontime}

Finally, we can compute the incoming value at ℓ_{ontime} using the values associated to its predecessors:

$$\begin{aligned} in_{\ell_{ontime}} &= post(\ell_{init}, \ell_{ontime})(G_{\ell_{init}}) \sqcup post(\ell_{onbrake}, \ell_{ontime})(G_{\ell_{onbrake}}) \\ &\quad \sqcup post(\ell_{stopped}, \ell_{ontime})(G_{\ell_{stopped}}) \sqcup \ post(\ell_{late}, \ell_{ontime})(G_{\ell_{late}}) \\ &\quad \sqcup post(\ell_{ontime}, \ell_{ontime})(G_{\ell_{ontime}}) \\ &= (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ &\quad \wedge -9 \leq delta \leq 9 \wedge 0 \leq b \leq 1 \wedge 0 \leq s \leq 1) \end{aligned}$$

The incoming value at ℓ_{ontime} has not changed, thus the analysis of the relational mode automaton R_{train} has converged. We discovered for each control location:

$$\begin{aligned} G_{\ell_{init}} &= (0 \leq s \leq 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq ontime \leq 1 \wedge 0 \leq late \leq 1 \\ &\quad \wedge 0 \leq onbrake \leq 1 \wedge 0 \leq stopped \leq 1) \\ G_{\ell_{ontime}} &= (ontime = 1 \wedge stopped = 0 \wedge onbrake = 0 \wedge nbrake = 0 \wedge late = 0 \\ &\quad \wedge -9 \leq delta \leq 9 \wedge 0 \leq s \leq 1 \wedge 0 \leq b \leq 1) \\ G_{\ell_{late}} &= (late = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge onbrake = 0 \wedge nbrake = 0 \\ &\quad \wedge b + delta = 10 \wedge b + s \leq 1 \wedge 0 \leq b \leq 9) \\ G_{\ell_{onbrake}} &= (onbrake = 1 \wedge stopped = 0 \wedge ontime = 0 \wedge late = 0 \\ &\quad \wedge b \leq 1 \wedge nbrake \geq b + s \wedge delta \leq -1 \wedge nbrake \leq 10 \wedge s \geq 0 \wedge nbrake \geq 1 \\ &\quad \wedge delta + nbrake \geq s - 9) \\ G_{\ell_{stopped}} &= (stopped = 1 \wedge ontime = 0 \wedge onbrake = 0 \wedge nbrake = 10 \\ &\quad \wedge late = 0 \wedge b = 0 \wedge delta \leq -1 \wedge 0 \leq s \leq 1 \wedge delta \geq s - 19) \end{aligned}$$

The variable *delta* is bounded at every reaction step:

$$\begin{array}{lcl} G_{\ell_{ontime}} \downarrow delta & = & (-9 \leq delta \leq 9) \\ G_{\ell_{late}} \downarrow delta & = & (1 \leq delta \leq 10) \\ G_{\ell_{onbrake}} \downarrow delta & = & (-19 \leq delta \leq -1) \\ G_{\ell_{stopped}} \downarrow delta & = & (-19 \leq delta \leq -1) \end{array}$$

Let $\mathcal{J}_{train}$ be a disjunctive invariant of the $\mathcal{R}_{train}$ automaton defined as:

$$\mathcal{J}_{train} \quad = \quad post(\ell_{init}, \ell_{ontime})(G_{\ell_{init}}) \vee (G_{\ell_{ontime}} \sqcup G_{\ell_{onbrake}} \sqcup G_{\ell_{stopped}} \sqcup G_{\ell_{late}})$$

For clarity, we use a simpler and weaker disjunctive invariant $\mathcal{J}'_{train}$ of $\mathcal{R}_{train}$:

$$\mathcal{J}'_{train} \quad = \quad (init = 0 \wedge delta = 0 \wedge late = 0) \vee (init = 0 \wedge -19 \leq delta \leq 10 \wedge late = 0)$$

In order to increase moderately the precision, we augment $\mathcal{J}'_{train}$ by adding relations given by the modes of the automaton $\mathcal{R}_{train}$ as described in 8.7.4.

$$\begin{array}{rcl} \mathcal{Q}_{train} & = & (init_0 = 1 \wedge init = 0 \wedge delta = 0 \wedge late = 0) \\ & & \vee\ (init_0 = 0 \wedge init = 0 \wedge -19 \leq delta \leq 10 \wedge delta = delta_0 + s - b \\ & & \wedge\ 0 \leq late \leq 1) \end{array}$$

$\mathcal{Q}_{train}$ is an over-approximation of a reaction step of the relational mode automaton $\mathcal{R}_{train}$ and represents the effect of a call to the `train_step` procedure.

9.4 Modular Analysis of a Pair of Trains

We want to discover a bound on the difference $delta = nb_2 - nb_1$ in number of encountered beacons on a subway track between any pair of trains. A subway track with two trains is modeled by the `track_step` procedure given in Figure 9.10. As we did previously for a single train, we also want to prove that the variables of all trains are bounded. We analyze the `track_step` procedure in a modular way, using the results of the analysis of a single train.

The `track_step` procedure implements an instance of each train by calling the `train_step` procedure which updates the trains variables at each reaction step. Since the `track_step` procedure is written only for analysis purposes, the numbers of detected beacons are counted in an unbounded fashion and stored respectively in nb_1 and nb_2.

Disjunctive Summary of `track_step`

The disjunctive summary $\mathcal{S}_{track} = \{r_1^\sharp, r_2^\sharp\}$ of the `track_step` procedure is:

$$\begin{array}{rcl} A & = & (0 \leq b1 \leq 1 \wedge 0 \leq b2 \leq 1 \wedge 0 \leq clk \leq 1 \wedge 0 \leq s \leq 1 \wedge 0 \leq ontime1 \leq 1 \\ & & \wedge\ 0 \leq late1 \leq 1 \wedge 0 \leq onbrake1 \leq 1 \wedge 0 \leq stopped1 \leq 1 \wedge 0 \leq ontime2 \leq 1 \\ & & \wedge\ 0 \leq late2 \leq 1\ \ \wedge 0 \leq onbrake2 \leq 1 \wedge 0 \leq stopped2 \leq 1 \wedge 0 \leq init \leq 1 \\ & & \wedge\ 0 \leq init_0 \leq 1 \wedge 0 \leq late1_0 \leq 1 \wedge 0 \leq late2_0 \leq 1) \\ r_1^\sharp & = & A \sqcap (init_0 = 1 \wedge init = 0 \wedge nb1 = 0 \wedge nb2 = 0 \wedge s = clk \wedge delta1 = 0 \\ & & \wedge\ delta2 = 0 \wedge late1 = 0 \wedge late2 = 0) \\ r_2^\sharp & = & A \sqcap (init_0 = 0 \wedge init = 0 \wedge nb1 = nb1_0 + b1 \wedge nb2 = nb2_0 + b2 \\ & & \wedge\ delta1 = delta1_0 + s - b1 \wedge delta2 = delta2_0 + s - b2 \\ & & \wedge\ late1_0 + late2_0 + s \geq clk \wedge late1_0 + s \leq 1 \wedge late2_0 + s \leq 1 \wedge clk \geq s) \end{array}$$

Relational Mode Automaton $\mathcal{R}_{track}$

Using the summary $\mathcal{S}_{track}$ we construct the relational mode automaton $\mathcal{R}_{track}$, shown in Figure 9.11, representing the `track_step` procedure. The set of control locations is $Loc = \{\ell_0, \ell_1\}$ and the modes are $\phi(\ell_0) = r_1^\sharp$ and $\phi(\ell_1) = r_2^\sharp$.

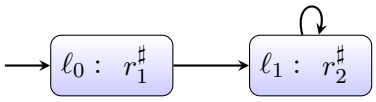


Figure 9.11: Relational mode automaton $\mathcal{R}_{track}$.

Reachability Analysis of $\mathcal{R}_{track}$

A reachability analysis of the relational mode automaton $\mathcal{R}_{track}$ gives:

$$\begin{array}{rcl} reach^\sharp(\ell_0) & = & A \sqcap (nb1 = 0 \wedge nb2 = 0 \wedge delta1 = 0 \wedge delta2 = 0 \wedge late1 = 0 \\ & & \wedge\ late2 = 0 \wedge s = clk) \\ reach^\sharp(\ell_1) & = & A \sqcap (delta2 - delta1 = nb1 - nb2 \wedge -19 \leq delta1 \leq 10 \\ & & \wedge\ -19 \leq delta2 \leq 10 \wedge nb1 \geq 0 \wedge nb2 \geq 0 \wedge late1 + late2 + s \geq clk \\ & & \wedge\ late1 + s \leq 1 \wedge late2 + s \leq 1 \wedge clk \geq s \wedge init = 0) \end{array}$$

The projection on the numbers of beacons detected by each train is:

$$G_{\ell_1} \downarrow \{nb1, nb2\} = (-29 \leq nb1 - nb2 \leq 29 \wedge nb1 \geq 0 \wedge nb2 \geq 0)$$

Thus the difference $nb1 - nb2$ in number of detected beacons by each train is bounded. We can use the constraint $-29 \leq nb1 - nb2 \leq 29$ for the initial placement of trains. It ensures that no collisions are possible if trains are initially separated by more than 29 beacons on a track.

Verification of the Assertion on *late*

We use the results of the reachability analysis of $\mathcal{R}_{track}$ to prove that the assertion $late = 1 \Rightarrow s = 0$ holds for each train. It ensures that a train do not receive the time signal when it is late, which is enforced by the implementation of the global clock of the track.

$$\begin{array}{rcl} reach^\sharp(\ell_0) \sqcap (late_1 = 1) & = & \bot \\ reach^\sharp(\ell_0) \sqcap (late_2 = 1) & = & \bot \\ reach^\sharp(\ell_1) \sqcap (late_1 = 1) & = & A \sqcap (delta2 - delta1 = nb1 - nb2 \wedge \mathbf{s = 0} \wedge \mathbf{late1 = 1} \\ & & \wedge\ init = 0 \wedge -19 \leq delta1 \leq 10 \wedge -19 \leq delta2 \leq 10 \\ & & \wedge\ nb1 \geq 0 \wedge delta1 - delta2 + nb1 \geq 0) \\ reach^\sharp(\ell_1) \sqcap (late_2 = 1) & = & A \sqcap (delta2 - delta1 = nb1 - nb2 \wedge \mathbf{s = 0} \wedge \mathbf{late2 = 1} \\ & & \wedge\ init = 0 \wedge -19 \leq delta1 \leq 10 \wedge -19 \leq delta2 \leq 10 \\ & & \wedge\ nb1 \geq 0 \wedge delta1 - delta2 + nb1 \geq 0) \end{array}$$

The assertions $late1 = 1 \Rightarrow s = 0$ and $late2 = 1 \Rightarrow s = 0$ are verified at each location of $\mathcal{R}_{track}$ and also hold when the `train_step` procedure is called to implement each train.

9.5 Conclusion

In this chapter, we were interested in the analysis of a classical example [67] of subway system inspired from a real industrial proposition.

We presented the detailed computation of the disjunctive relational summary of the `metro_step` procedure implementing a single train. The computed summary was used to construct automatically an abstraction of the behavior of a single train as a relational mode automaton. We analyzed the relational mode automaton representing a single train and discovered that all the variables of a train are bounded at runtime. In particular, we found using our approach the classical result that the difference *delta* between the number of received clock ticks and the number of detected beacons is always bounded with $-19 \leq delta \leq 10$. We computed a disjunctive relational invariant of an individual train from analysis results.

We used the disjunctive relational invariant of a single train to analyze a pair of trains on a subway track in a modular way. Reduction techniques presented in 8.6 were used to simplify the analysis and get rid of unnecessary details. We discovered that the difference $nb_1 - nb_2 = delta_2 - delta_1$ in number of encountered beacons between any pair of trains on a track is always bounded with $-29 \leq nb_1 - nb_2 \leq 29$ although neither 29 nor bounds are given explicitly.

The computations involved in this example have been performed using the PyApron library which provides a high-level Python binding to the Apron library for the convex polyhedra implementation. The implementation of the refinement process was written specifically for that example.

We chose to consider all variables as integers and to represent booleans as integer variables between 0 and 1. We could have used an abstract domain allowing the expression of boolean properties with convex polyhedra [70, 10].

```
void track_step(int * init, int clk, int b1, int b2,
                int * ontime1, int * late1, int * onbrake1,
                int * stopped1, int * delta1, int * nbrake1,
                int * ontime2, int * late2, int * onbrake2,
                int * stopped2, int * delta2, int * nbrake2,
                int * nb1, int * nb2)
{
    int s;

    if(*init){
        *nb1 = 0;
        *nb2 = 0;
        s = clk;
        train_step(init, s, b1, ontime1, late1,
                   onbrake1, stopped1, nbrake1, delta1);
        train_step(init, s, b2, ontime2, late2,
                   onbrake2, stopped2, nbrake2, delta2);
    } else {
        assert(*stopped1 == 0 || b1 == 0);
        assert(*stopped2 == 0 || b2 == 0);

        /* Counting beacons detected by each train */
        *nb1 = *nb1 + b1;
        *nb2 = *nb2 + b2;

        /* Local clock of the track */
        if((*late1 == 1) || (*late2 == 1)){
            s = 0;
        } else {
            s = clk;
        }

        /* Train 1 */
        train_step(init, s, b1, ontime1, late1,
                   onbrake1, stopped1, nbrake1, delta1);
        /* Train 2 */
        train_step(init, s, b2, ontime2, late2,
                   onbrake2, stopped2, nbrake2, delta2);
    }
}
```

Figure 9.10: The `track_step` procedure.

Chapter 10

Conclusion

We proposed a modular interprocedural analysis for numerical properties, as a solution to the cost of using expressive relational abstract domains in program analysis. An analysis using a relational abstract domain can be straightforwardly converted into a relational analysis computing an input-output relation. Such relations can be used as procedure summaries, computed once and for all, and used in a bottom-up fashion to compute the effect of procedure calls.

We described a general framework to compute relational procedure summaries by abstract interpretation, based on the forward propagation of abstract input-output relations. Even if it is especially applied to convex polyhedra, it can be used with any relational abstract domain. Although the idea is not new [69], we proposed a formalization of relational abstract interpretation that we did not find elsewhere.

Applying this idea to Linear Relation Analysis, we concluded that procedure summaries made of individual input-output relations represented by convex polyhedra are not precise enough and deserve to be refined disjunctively. We proposed a method based on precondition partitioning to compute disjunctive relational summaries. We gave partitioning heuristics for refining summaries, according to the reachability of control points or calling contexts of called procedures. We also identified several improvements to summary computation itself, like widening limited by preconditions and previously computed relations, and a more precise computation of abstract relations at loops exit points.

We shown that our approach can be used to compute disjunctive relational summaries of recursive procedures, as it is often a weakness of existing interprocedural analyses. Our analysis was able to uncover non-trivial procedure behaviors caused by recursion.

We provide an implementation of our approach in a new static analysis platform for C programs called MARS. It was the product of a significant effort to design a new tool to simplify the development of static analyses, while accepting a large subset of C and providing highly-precise traceability information. Using MARS, we conducted experiments of our modular analysis on a set of benchmark programs well-known in the WCET research community. We shown that our approach can significantly reduce the analysis time for Linear Relation Analysis, compared to a classical full context-sensitive approach where procedures are analyzed completely in each call context. This is especially the case for procedures which are called several times, like procedures which are part of a library or a framework. On the other hand, analysis precision is not significantly damaged and can be even improved due to the use of disjunction. This approach was published in [27]. Some

possible future works were given in 6.6.

During the time of this thesis, we also published the continuation of earlier works on the improvement [26] of Linear Relation Analysis and the application of Linear Relation Analysis to the computation of the Worst-Case Execution Time of programs [115].

Reactive systems can be generally structured as collections of components reacting to inputs, producing outputs and updating their internal memories to achieve the system functionality. The reaction of each component to its environment is implemented by a step procedure. Synchronous languages like LUSTRE are well-known for the implementation of synchronous reactive systems.

In a second part, we proposed an approach toward a modular analysis of reactive systems for numerical properties. We presented a flexible abstraction of reactive components called *Relational Mode Automata* (RMA), allowing the representation of reactive components at various levels of abstraction, to achieve different tradeoffs in analysis performance and precision. Relational mode automata can be constructed automatically from the disjunctive relational summary of the step procedure implementing a given reactive component. The core idea of relational mode automata is that component behavior can be described by a collection of component modes and the possible transitions between them.

We described a reachability analysis of RMA which is able to analyze a parallel composition of reactive components without constructing the synchronous product of automata prior to the analysis. Only the part of the product found to be reachable is constructed.

We shown that the analysis results of RMA can be computed once and for all and used to analyze component instantiations in a larger reactive system. We gave several heuristics to adapt the level of abstraction of RMA, either by merging modes or by the internal reduction of modes themselves, and through the use of disjunctive invariants.

We applied our approach to the analysis of an example of reactive system from a real proposition for an automated subway control system. Although it has been greatly simplified for clarity, it retains its core train regulation strategy.

Our objective was to guarantee the absence of arithmetic overflows at runtime by proving automatically that all variables in a train are bounded at all times. We were interested in the pure discovery of variable bounds, as they are not provided explicitly in the system definition. The analysis results of the RMA for an individual train were used in a modular way to discover constraints on the initial placement of trains on a track ensuring that the subway system is safe.

Although wider experiments should be conducted on a significant set of real-world reactive programs, we believe that our approach based on relational mode automata is humbly paving the way toward a modular analysis of reactive systems for numerical properties. In order to achieve this goal, some questions should be addressed by future works:

1. The automatic construction of RMA from disjunctive relational summaries of step procedures was described in 8.1 on the example of the `counter_step` procedure. Either the `counter_step` procedure or the `metro_step` procedure were written manually in C, in a particular form, where the values of memory variables are tested first and secondly input variables are tested to determine the current reaction of the component. It enables our two-level refinement scheme to produce clean and simple disjunctive summaries by identifying syntactically the tests on memory vari-

ables and the tests depending on input variables. Although this form may seem quite natural for a human being, it is not the case for step procedures generated automatically from synchronous languages like Lustre.
We could propose a way to construct relational mode automata from general step procedures by identifying semantically the tests depending only on memory variables and the tests possibly depending on input variables. However this would involve a fairly costly dependency analysis, for a benefit not really superior in practice to the syntactical identification of tests, as the construction of RMA from step procedures should remain as cheap as possible.
Instead, a more pragmatic way would be to address the compilation of synchronous programs into step procedures of the appropriate form for analysis. Classically, heuristics are used in the compilation of Lustre programs to decide where tests should be placed in the generated code. A new heuristic could be possibly devised to structure the control flow of the generated procedure in the preferred way for RMA construction.

2. We presented in 8.6 and 8.7 a catalogue of techniques to adapt the level of abstraction of RMA to analyze larger reactive systems: reduction by merging modes, internal reduction of modes, weakening of invariants, disjunctive invariants and relational augmentation. All these techniques can be combined in various ways. For the analysis of our subway, we computed a weakened disjunctive invariant of a relational mode automaton, which was augmented by linear relations given by the automaton modes.
Experiments should be conducted on the impact of these techniques on the precision of analysis results and analysis performance. Heuristics should be designed to select automatically the reduction techniques to apply during the analysis of a reactive program, depending on high-level goals: whether we want to perform the most precise analysis of a reactive program achievable with our approach, or if we want a modestly-precise analysis to put the emphasis on performance.

We devised two approaches for the modular analysis of *interprocedural phenomena*, being either procedures, possibly with recursion, or reactive components, which can be seen in a sense as a special case of procedures with persistent memories. Similarly, our approaches could be applied to the analysis of object-oriented programs, where objects have memory in the form of their data fields, which are persisted and can be modified by methods from one call to another.

Our approaches show that expressive relational abstract domains such as convex polyhedra can be used to compute precise procedure summaries which in turn can improve significantly analysis scalability. They give a direction toward the adoption of expressive relational abstract domains in static analysis tools for large programs.

Bibliography

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using scade. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods*, pages 115–129, Berlin, Heidelberg, 2006. Springer.

[2] F. A. Administration. Airworthiness directives; the Boeing company airplanes, 2015. Document Number 2015-10066, RIN 2120-AA64.

[3] F. Alexis. *Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof*. PhD thesis, Université Grenoble Alpes, Oct. 2015.

[4] F. Allen. Interprocedural analysis and the information derived by it. In *IBM Germany Scientific Symposium Series*, pages 291–321. Springer, 1974.

[5] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, 2010.

[6] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[7] I. Ansys. SCADE Suite. https://www.ansys.com/products/embedded-software/ansys-scade-suite, 2019.

[8] K. Apinis, H. Seidl, and V. Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 377–386, Seattle, WA, June 2013.

[9] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.

[10] A. Bakhirkin and D. Monniaux. Extending constraint-only representation of polyhedra with boolean constraints. In *International Static Analysis Symposium*, pages 127–145. Springer, 2018.

[11] F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *International Static Analysis Symposium*, pages 315–332. Springer, 2007.

[12] J. M. Barth. An interprocedural data flow analysis algorithm. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 119–131. ACM, 1977.

[13] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, 1994.

[14] F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1-2):259–271, 2005.

[15] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Readings in hardware/software co-design*, pages 147–159, 2002.

[16] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[17] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 409–426, Berlin, Heidelberg, 2000. Springer.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation*, pages 85–108. Springer, 2002.

[19] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.

[20] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In *International Conference on Computer Aided Verification*, pages 197–203. Springer, 1990.

[21] A. Bouali. Xeve, an esterel verification environment. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, pages 500–504, Berlin, Heidelberg, 1998. Springer.

[22] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *International Workshop on Programming Language Implementation and Logic Programming*, pages 307–323. Springer, 1990.

[23] F. Bourdoncle. *Sémantiques des langages imperatifs d'ordre superieur et interpretation abstraite.* PhD thesis, 1992.

[24] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[25] R. Boutonnet and M. Asavoae. The WCET analysis using counters - a preliminary assessment. In *Proceedings of 8th JRWRTC, in conjunction with RTNS14*, Versailles, France, Oct. 2014.

[26] R. Boutonnet and N. Halbwachs. Improving the results of program analysis by abstract interpretation beyond the decreasing sequence. *Formal Methods in System Design*, 53(3):384–406, Dec 2018.

[27] R. Boutonnet and N. Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In C. Enea and R. Piskac, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 136–159. Springer International Publishing, 2019.

[28] A. R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer.

[29] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[30] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.

[31] N. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[32] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*, pages 278–285, 1996.

[33] A. Cortesi. Widening operators for abstract interpretation. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40. IEEE, 2008.

[34] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[35] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada*. North-Holland Publishing Company, 1977.

[36] P. Cousot and R. Cousot. Relational abstract interpretation of higher order functional programs. In *JTASPEFT/WSA*, pages 33–36, 1991.

[37] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[38] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

[39] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proc. SSGRR*, pages 6–10, 2001.

[40] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, volume 2, pages 159–178. Springer, 2002.

[41] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[42] A. P. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[43] V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract interpretation of cellular signalling networks. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 83–97. Springer, 2008.

[44] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer, 1989.

[45] Euclid of Alexandria. Propositions 1–2. In *The Elements*, volume 7, c.300 BC.

[46] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 10–30, 2010.

[47] A. Flexeder, M. Müller-Olm, M. Petter, and H. Seidl. Fast interprocedural linear two-variable equalities. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):21, 2011.

[48] A. Fouilhé, D. Monniaux, and M. Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *International Static Analysis Symposium*, pages 345–365. Springer, 2013.

[49] A. Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. *Electronic Notes in Theoretical Computer Science*, 144(1):19 – 33, 2006. Proceedings of the Third International Workshop on Bounded Model Checking (BMC 2005).

[50] P. Garoche, A. Gurfinkel, and T. Kahsai. Synthesizing modular invariants for synchronous code. In *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014.*, pages 19–30, 2014.

[51] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones : application à Esterel*. PhD thesis, 1988. 1988PA112049.

[52] D. Gopan and T. Reps. Lookahead widening. In *International Conference on Computer Aided Verification*, pages 452–466. Springer, 2006.

[53] D. Gopan and T. Reps. Guided static analysis. In *International Static Analysis Symposium*, pages 349–365. Springer, 2007.

[54] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–458. Springer, 2008.

[55] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. *Programming Languages and Systems*, pages 253–267, 2007.

[56] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[57] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9, Nov 2008.

[58] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, 2008.

[59] G. E. Hagen. *Verifying Safety Properties of Lustre Programs: An SMT-based Approach*. PhD thesis, Iowa City, IA, USA, 2008. AAI3347220.

[60] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, Mar. 1979.

[61] N. Halbwachs. Delay analysis in synchronous programs. In *International Conference on Computer Aided Verification*, pages 333–346. Springer, 1993.

[62] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[63] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep. 1991.

[64] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[65] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[66] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[67] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[68] K. Hoder and N. Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.

[69] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ICS*, volume 91, pages 244–251, 1991.

[70] B. Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones*. PhD thesis, 2000.

[71] B. Jeannet. INTERPROC analyzer for recursive programs with numerical variables. *INRIA, software and documentation are available at the following URL: http://pop-art. inrialpes. fr/interproc/interprocweb. cgi. Last accessed*, pages 06–11, 2010.

[72] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *SAS*, volume 3672, pages 186–202. Springer, 2005.

[73] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *International Static Analysis Symposium*, pages 39–50. Springer, 1999.

[74] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.

[75] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *AMAST*, pages 258–273. Springer, 2004.

[76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[77] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega calculator and library, version 1.1.0. *College Park, MD*, 20742:18, 1996.

[78] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[79] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In *International Symposium on Automated Technology for Verification and Analysis*, pages 492–502. Springer, 2011.

[80] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[81] H. Le Verge. A note on Chernikova's algorithm. 1992.

[82] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, Sep. 1991.

[83] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *International Static Analysis Symposium*, pages 280–301. Springer, 2000.

[84] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[85] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[86] R. Lublinerman and S. Tripakis. Modular code generation from triggered and timed block diagrams. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 147–158, April 2008.

[87] V. Maisonneuve, O. Hermant, and F. Irigoin. Computing invariants with transformers: experimental scalability and accuracy. *Electronic Notes in Theoretical Computer Science*, 307:17–31, 2014.

[88] Z. Manna and J. McCarthy. Properties of programs and partial function logic. Technical report, Stanford University, California, Department of Computer Science, 1969.

[89] Z. Manna and A. Pnueli. Formalization of properties of functional programs. *Journal of the ACM (JACM)*, 17(3):555–569, 1970.

[90] F. Maraninchi and Y. Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages -Oxford-*, 27(1-3):61–92, Nov. 2001.

[91] F. Maraninchi and Y. Rémond. Mode-automata: A new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, Mar. 2003.

[92] A. Maréchal. *New Algorithmics for Polyhedral Calculus via Parametric Linear Programming*. PhD thesis, Grenoble Alpes University, France, 2017.

[93] A. Maréchal and M. Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pages 367–385, 2017.

[94] Mathworks. Polyspace code prover, mathworks.

[95] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, pages 5–20. Springer, 2005.

[96] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*, pages 155–172. Springer, 2001.

[97] A. Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, page 310, 2001.

[98] A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, Ecole Polytechnique X, Dec. 2004.

[99] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN Notices*, volume 41, pages 54–63. ACM, 2006.

[100] A. Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[101] A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, 2018.

[102] D. Monniaux and M. Bodin. Modular abstractions of reactive nodes using disjunctive invariants. *Programming Languages and Systems*, pages 19–33, 2011.

[103] M. Müller-Olm, H. Seidl, and B. Steffen. *Interprocedural analysis (almost) for free.* Dekanat Informatik, Univ., 2004.

[104] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[105] J. V. Olnhausen, L. J. Jagadeesan, L. J. Jagadeesan, C. Puchol, and J. E. V. Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. In *In Proceedings of the 7th International Conference on Computer Aided Verification, Volume 939 of the Lecture Notes in Computer Science*, pages 127–140, 1996.

[106] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Annual Asian Computing Science Conference*, pages 331–345. Springer, 2006.

[107] C. Popeea and W.-N. Chin. Dual analysis for proving safety and finding bugs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2137–2143. ACM, 2010.

[108] C. Popeea and W.-N. Chin. Dual analysis for proving safety and finding bugs. *Science of Computer Programming*, 78(4):390–411, 2013.

[109] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, Grenoble, France, October 12-16, 2009*, pages 215–224, 2009.

[110] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks. *Design Automation for Embedded Systems*, 14(3):165–192, 2010.

[111] V. Pratt. Origins of the calculus of binary relations. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 248–254. IEEE, 1992.

[112] C. Ratel. *Design and implementation of a formal verification tool for Lustre programs : the system Lesar*. PhD thesis, Université Joseph-Fourier - Grenoble I, July 1992.

[113] P. Raymond. Compilation séparée de programmes Lustre. Technical report, Master Thesis, Projet SPECTRE, IMAG, July 1988.

[114] P. Raymond. *Efficient Compilation of a Declarative Synchronous Language: the Lustre-V3 Code Generator*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, Nov. 1991. 141 pages.

[115] P. Raymond, C. Maiza, C. Parent-Vigouroux, E. Jahier, N. Halbwachs, F. Carrier, M. Asavoae, and R. Boutonnet. Improving WCET evaluation using linear relation analysis. *LITES*, 6(1):02–1, 2019.

[116] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[117] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[118] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. 1978.

[119] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[120] A. Simon and A. King. Widening polyhedra with landmarks. In *Asian Symposium on Programming Languages and Systems*, pages 166–182. Springer, 2006.

[121] P. Sotin and B. Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *European Symposium on Programming*, pages 459–479. Springer, 2011.

[122] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *IFIP Congress (1)*, pages 376–381, 1971.

[123] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[124] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941.

[125] C. Wang, Z. Yang, A. Gupta, and F. Ivančić. Using counterexamples for improving the precision of reachability computation with polyhedra. In *International Conference on Computer Aided Verification*, pages 352–365. Springer, 2007.

[126] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *ACM SIGPLAN Notices*, volume 43, pages 221–234. ACM, 2008.

[127] S. Yovine. Model checking timed automata. In *School organized by the European Educational Forum*, pages 114–152. Springer, 1996.

[128] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *ACM SIGPLAN Notices*, volume 49, pages 249–258. ACM, 2014.