Automatic Verification with Abstraction and Theorem Proving

THÈSE N^{O} 5828 (2013)

PRÉSENTÉE LE 15 AOUT 2013

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Hossein HOJJAT

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. C. Koch, rapporteur
Prof. D. Monniaux, rapporteur
Dr A. Rybalchenko, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2013

Everything has its tax and the tax of knowledge is to teach others.
— Ali bin Abu-Talib

To the memory of Christophe Paus
who saved lives after lives
whose life ended tragically in the Flight 447

Acknowledgements

In a Ph.D thesis in the area of formal methods, the only place left to be “informal” sounds to be the Acknowledgements section. This section gives a unique chance to recognize the importance of the helps received from others. Regrettably, due to space limitations, it is impossible to include all the nice people without whom none of this would be possible. Looking back at my first months at EPFL, I and my wife were unlucky to experience an unfortunate situation. My wife had a massive brain stroke just three months after I joined the Ph.D program of EPFL. Her rehabilitation program took a very long time; all along my doctoral studies. I am happy to be finishing the thesis write-up as she has mostly recovered. Here, I'll do my best to include the name of the people who were of help to me in research and in facing the challenging problems of life.

I would like to extend my sincerest thanks and appreciation to Viktor and Barbara. During the first year of my Ph.D I was working with Barbara. After that I became a student of Viktor. I am indebted to the proportionate, careful and tactical supervision of Viktor during my research in the last four years. He allowed me to choose a topic on program analysis. My initial attempts with developing verification tools and my first research papers during Ph.D were not so successful. Viktor was always incredibly encouraging by persuading me to think about the weaknesses of the failed attempts and to collaborate with the other researchers. Being a student of Viktor gave the feeling that I was always being accompanied in the research. I'll always remember the day that Viktor and Barbara kindly offered to drive me in their own car to Grenoble. The productive discussion with the researchers in Verimag led to new interesting joint research topics.

I am delighted to have the opportunity to work with different people during my Ph.D. Most notably, working with Philipp Rümmer was a valuable experience for me. Philipp's profound insight into both theory and practice of software verification made him an ideal collaborator for me. I was always very enthusiastic to discuss different problems with him and I enjoyed our cooperation in implementing the new ideas in practice. I also had the chance to visit his group at the Uppsala University and to work with his student Paul. Paul is the kind of friend you never get tired to discuss different subjects with him and we had interesting debates over the semantics of timed systems. The Chapters 5 and 6 of this thesis are done in collaboration with Philipp and Paul. I'm extremely glad to have known and have worked with Radu Iosif since the first year of my Ph.D. I had the chance the take the class “Logic and Automata Theory” of him and Barbara. With Radu and his then-student Filip Konečný we could combine and connect the approaches of predicate abstraction and loop acceleration. This result is presented in

Chapter 4. Filip's tool pushed me to improve the performance of Eldarica since Flata was always a strong competitor in solving benchmarks.
I'd like to thank the jury members of my thesis defense, Rachid Guerraoui, Christoph Koch, David Monniaux and Andrey Rybalchenko for accepting to be a member of this committee and for their fruitful discussion during the defense. I appreciate the constructive comments of David Monniaux on the draft of this thesis.
I am thankful to the current and previous LARA members for all the wonderful moments that we had together: Andrej, Etienne, Eva, Filip, Giuliano, Mikael, Philippe, Pierre-Emmanuel, Ravi, Regis, Ruzica, Swen and Tihomir. In particular I thank Regis and Philippe for helping me to write the French abstract of this thesis. I am appreciative to the efforts of our secretary Yvette to handle the administrative issues and Fabien for solving the problems of our computers.
The Persian community in Lausanne was beyond my initial imagination. It is unbelievable to have such great people in thousand kilometers from home who never let me and my wife feel the absence of our families. Our friends were remarkably supportive during the period of hospitalization of my wife. It will be difficult for us to leave these nice people after my graduation and we'll miss them all. If I want to just to name a few of them here I can think of Mohammad Hossein Manshaei, Fatemeh HosseinZadeh and Venus Sharifi. Mohammad Hossein was really like a brother to me and Mrs. HosseinZadeh and Mrs. Sharifi loved my wife as their own daughter.
My special thank to my family and family-in-law. My father is the one who shaped my education and career from my childhood with his great excitement in introducing me to the wonderful world of programming. When I was in primary school he bought me a laptop with an Intel 80386 processor and 640KB of RAM and he taught me about programming in BASIC. My enjoyment of programming started from those days and it appears that it has not lessened until today. I'd like to thank my mother for showing me how patience and hope can make the tough moments in life more tolerable. Nahal, your return from the probable death opened new windows into my life and I will be thankful to God during the rest of my life that you did not leave me in such a tragic manner. Indeed, I believe that finishing your bachelor's degree after having four brain surgeries is a far greater achievement than my graduation. I also thank my father-in-law and my mother-in-law for their unconditional love and support. I am fortunate to have two lovely elder sisters, who have always stood by me. I'm really glad to have both of you.
Finally, I finish this part with remembering the memories that I have from Dr. Christophe Paus, the late neurosurgeon of the CHUV hospital. If I am now writing this thesis and finishing my graduation is because of you, Christophe. You saved many people including my wife in this world, but unfortunately your lifetime was very short to save more. I hope wherever you are you are in peace.

Lausanne, June 2013 Hossein Hojjat.

Preface

> Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability
>
> Bill Gates, April 18, 2002

Software verification, rigorously ensuring that software meets the desired properties for all execution, remains a grand challenge for computer science. One place to look for inspiration for solution is *hardware* verification, which, after decades of research, proved immensely fruitful. Indeed, major hardware manufacturers today routinely verify their microprocessor designs using model checking and theorem proving. Successful companies such as Jasper Design Automation provide high-quality verification tools as a main product.

Can we hope for a similar success in software verification? The emergence of software model checking, pioneered by Susanne Graf and Hassen Saidi in 1996, provided hope and started an exciting direction within automated software verification. SLAM tool developed in Microsoft Research by, among others, Thomas Ball, Rupak Majumdar, Todd D. Millstein, Sriram K. Rajamani led to one of the first successful examples of using software verification to impose software quality standards in a large software company, and was closely followed by further advances such as the ones in the BLAST tool by Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. The idea of predicate abstraction is to automatically *abstract* an *infinite*-state software and obtain a simpler *finite*-state problem, which can then be tackled by successful methods of hardware verification.

Notwithstanding the brilliance of the original idea, a large number of fundamental issues still remained opened regarding predicate abstraction in practice. A basic question is the very meaning of verifying a C program, given that even a single unverified memory access in a C program technically renders all other verification results meaningless. A more technical question includes a good mechanism for refining the abstraction in the face of false counterexamples. Indeed, it is the refinement step that hides many of the undecidability aspects of the verification problem. This thesis makes important contributions to both of these questions.

To address the first question, of rigorous semantics, the thesis adopts a logical representation as the verification starting point, instead of a particular programming language with messy semantics. This representation is in essence a recursive integer transition system; it is semantically unambiguous, rich enough to capture the notion of infinite-state system verification, and also related to fundamental models of computation such as Minsky machines and Petri nets. For practice it is crucial that such logical descriptions can be generated automatically, and the tool developed by the author does this for a subset of Scala, as well as for programs in a Numerical Transition Systems format developed by Radu Iosif, which can be automatically generated from C programs. Recent efforts (of which this thesis results are part) have led to a library of rigorously described verification problems, which puts automated software verification on a sound semantic and experimental basis, arguably for a first time.

To address the second question counterexample refinement (and the related termination of the verification process), the thesis builds on an observation by Henzinger, Jhala, Majumdar, and Kenneth L. McMillan that connected the refinement of abstract counterexample paths, a crucial part of predicate abstraction, to the logical notion of *interpolation*. This results provide further structure that help address the undecidable verification problem: interpolation problem remains decidable for many logical constraints describing program paths, and suggests that the refinements for counterexamples, even if not uniquely given, can be often found by examining unsatisfiability proofs.

Further crucial to the results of this thesis is an observation (which we learned from Andrey Rybalchenko) that considering inter-procedural and concurrent programs requires generalizing the notions of paths to inter-procedural trees, and that interpolation in such generalized case can be explained by solving recursion-free Horn clauses. The design space of verification algorithms becomes an interplay between solving recursion-free constraints (which approximate a recursive constraint) and using their solutions to constructively make progress in solving the overall recursive problem. What emerges is a hierarchy of recursive and recursion-free constraints whose solution has different algorithmic difficulty: more complex constraints are more difficult to solve, but also allow a verification tool to make more rapid progress in the overall verification problem. Two classes of constraints presented in the thesis are particularly worth emphasizing: *disjunctive interpolation problems*, and *accelerable recursive constraints*.

Disjunctive interpolation problems arise as a useful generalization of tree interpolants that, remarkably enough, does not increase the theoretical complexity of the satisfiability of constraints but can results in handling exponentially many tree interpolation problems at once. The evaluation and implementation of this functionality in a verification system was made possible thanks to a unique interpolation theorem prover Princess, developed by Philipp Rümmer.

A second class of constraints are accelerable constraints, which correspond to loops whose transitive closure can be solved in an exact way. They describe some of the situations where it is possible to algorithmically verify systems with the infinite state space and infinitely many traces. It is extremely valuable to incorporate such exact analysis into a general-purpose verification system, whenever an opportunity arises. The new concept of accelerated interpolants shows that it is indeed possible to include such exact computation into a predicate

abstraction engine. In the cases when it applies, the method can exclude infinitely many paths at once. Moreover, the method also applies when programs are under-approximated or over-approximated, and can be used to deliver final correct results thanks to the flexibility of the predicate abstraction approach.
These techniques have been implemented in the tool Eldarica, which is publicly available and has shown promising results in this remarkably difficult and important domain of algorithmic software verification. What is particularly remarkable that these techniques are general enough to apply to a wide range of domains, from hardware descriptions, to C and Scala programs, to concurrent timed systems.

Lausanne, June 2013

Viktor Kuncak

Abstract

Software verification is an important and complex discipline of research. Among different software verification techniques, predicate abstraction has shown potentials in practice, notably in verification of device drivers. The predicate abstraction technique works in an incremental and iterative fashion. It constructs an abstraction of the given program with respect to a set of predicates. With the help of spurious counter-examples it refines the coarse abstractions to recover the missing precision introduced during abstraction. It keeps on performing the abstraction refinement cycle until it finally proves the (in-)correctness of a program or it continues forever. The refinement step is commonly carried out with the help of Craig interpolants. There are major fundamental bottlenecks and restrictions facing the predicate abstraction approach. An important obstacle is the number of required refinement steps that can possibly grow to infinity. This thesis introduces two novel techniques, acceleration of interpolants and disjunctive interpolants to reduce the refinement steps and to increase convergence. The idea behind the acceleration of interpolants is to strengthen the generation of interpolants by computing the transitive closure of the loops. In the disjunctive interpolation technique the refinement step can exclude several counter-examples together by just a single call to the interpolating theorem prover.

I have implemented both of these novel techniques as part of a new predicate abstraction framework called Eldarica. To be able to compare to other approaches, Eldarica supports (among its other input formats) the intermediate languages of Integer Numerical Transition Systems (INTS) and Horn clauses. This thesis classifies the recursion-free Horn clauses with respect to the corresponding interpolation problem. The problem of solving a set of recursion-free Horn clauses arises in the refinement phase of the general case of recursive Horn clauses. The classification of recursion-free Horn clauses opens up new perspectives in building predicate abstraction solvers for Horn clauses. As a challenging domain for verification we finally introduce a compositional approach to verify the timed concurrent systems using translation to Horn clauses. In the concurrent timed systems not only the parallel computation of data matters but the exact time of generating the result is vital.

We have tried the Eldarica framework on several benchmarks coming from several sources. The technical contributions of this dissertation offer novel algorithms that enable Eldarica to verify some benchmarks that the earlier methods, based on the classical predicate abstraction approach could not handle.

Keywords: software verification, predicate abstraction, interpolation, Horn clause, timed systems.

Résumé

La vérification logicielle est un domaine de recherche important et complexe. Une technique de vérification qui s'est avérée efficace en pratique dans l'analyse de systèmes, et en particulier de pilotes de périphériques, est l'abstraction par prédicats. Cette technique fonctionne de manière itérative et incrémentale ; elle construit une abstraction d'un programme donné par rapport à ensemble de prédicats sur ses états. À l'aide de contre-exemples, elle raffine les approximations grossières et corrige les erreurs de précision introduites par l'abstraction. La procédure itère le cycle d'abstraction et de raffinement jusqu'à finalement prouver la validitié (ou l'invalidité) du programme ou, si cela s'avère impossible, ne termine pas. L'étape de raffinement est habituellement calculée à l'aide d'interpolants de Craig. L'abstraction par prédicats fait face à certaines limites fondamentales. Un obstacle en particulier est le nombre d'étapes de raffinement qui peut croître à l'infini.

La présente thèse introduit deux nouvelles techniques pour réduire le nombre d'étapes de raffinement et ainsi améliorer la convergence : l'accélération des interpolants, et les interpolants disjonctifs. Le principe de l'accélération est de renforcer la génération d'interpolants en calculant la fermeture transitive des boucles. En utilisant la technique d'interpolants disjonctifs, l'étape de raffinement peut exclure plusieurs contre-exemples en un seul appel à une procédure d'interpolation. J'ai implémenté ces deux nouvelles techniques au sein d'un nouveau outil de raisonnement par abstraction des prédcats appelé Eldarica. Afin de pouvoir se comparer à d'autres approches, Eldarica supporte (parmi divers formats d'entrée) le langage intermédiaire des «Integer Numerical Transition Systems (INTS)» et une représentation en clauses de Horn.

Une autre contribution de cette thèse est la classification des systèmes de clauses de Horn sans récursion par rapport aux problèmes d'interpolations correspondants. La résolution de systèmes de clauses de Horn sans récursion est un problème qui se pose lors de la phase de raffinement dans le cas général de systèmes de clauses de Horn récursives. La classification des clauses de Horn sans récursion ouvre de nouvelles perspectives pour la construction de systèmes d'abstraction par prédicats.

Finalement, adressant un difficile problème de vérification, je propose une approche compositionnelle pour la vérification de systèmes concurrents synchrones qui se base sur une traduction vers des clauses de Horn. Dans ces systèmes concurrents synchrones, il faut non seulement considérer la calcul de données en parallèle, mais également le minutage précis de la génération des résultats. J'ai évalué le système Eldarica sur plusieurs benchmarks obtenus de sources variées.

Les contributions techniques de cette thèse résultent en de nouveaux algorithmes qui permettent à Eldarica de vérifier certains benchmarks hors de portée des approches classiques d'abstraction par prédicats.

Mots-clés : vérification logicielle, abstraction par prédicats, interpolation, clauses de Horn, systèmes temporisés.

Contents

List of Figures

1 Introduction

> Pay attention to zeros. If there is a zero, someone will divide by it.
>
> Cem Kaner

Software is everywhere. In one form or another programs are integrated in everyday life. There are numerous examples of ways in which we are under the influence of software. Take, for example, trading. The financial transactions from smallest, such as buying a book online to largest, such as transactions in the level of banks and finance companies are basically done by software. Likewise, software plays an essential role in our transportation. It is estimated that a premium automobile nowadays runs around 100 million lines of code [Cha09]. The level of software presence in jet airliner can go well beyond. Just the avionics systems in the Airbus A380 include more than 100 millions lines of code [WDD+12]. To bring it even closer to ourselves, we are probably carrying a piece of code in our pockets right now. Cell phones usually include some sort of software; even the most basic models include an address book program. Many models of mobile phones these days are able to execute software on their operating systems. The Android system which is one of the most successful operating systems in the market of cell phones runs more than a million lines of code in version 4.0 [Rub11].

In the examples above and in many more cases, we are placing our confidence in software: we trust the computer programs not to miscalculate the financial transactions, not to crash our cars or airplanes and not to reveal the personal data from our cell phones. The defects in software not only impose serious economic losses to a company to recall the faulty products but it can also lead to possible damages and human casualties. The creator of the web-page [Der] has gathered more than 100 horror stories in which software problems has resulted in disastrous accidents. The purpose of the formal program verification research is to assure the reliability of software. Along those lines of research, this dissertation proposes new techniques in proving the safety of critical systems. Given the source code of a program, the goal of the techniques in this thesis is to determine whether a program can hit the error state.

The formal software verification approach that this thesis adopts is based on the successful method of predicate abstraction with counterexample-guided abstraction refinement, sometimes abbreviated as CEGAR [CGJ+00]. The effectiveness of this method is widely proved in practice [HJMM04, GPR11a, BLR11, BHJM07]. Although much success has been achieved with regard to the verification of complicated programs there are still few fundamental shortcomings affecting both the performance and the ability of the CEGAR tools to handle some challenging benchmarks. One deficiency is due to the repetitive nature of the procedure. CEGAR is basically a loop of abstraction and refinement which continues until the program is proved to be correct or incorrect. Having a complete procedure that always terminates is theoretically impossible, but, this thesis presents heuristics to reduce the number of required iterations and to increase convergence. The established framework in this thesis is also able to verify recursive and concurrent programs. To put the new techniques in a suitable perspective and to make clear the claims I start by giving a short background on the history of software verification.

1.1 Software verification: From Its Origins to Predicate Abstraction

The ideas behind formal program verification were developed at the same time by the birth of computer science. Alan Turing, the father of modern computer science, in one of his pioneering papers [Tur49] suggests the programmers put assertions about the correctness of programs at various points. The condition of an assertion should always be satisfied when the execution reaches a program point. He describes a checker to verify the assertions in order to prove the correctness of a particular program. The assertions remain to be one of the most prevalent ways of describing the correctness of software. Tony Hoare, another pioneer of computer science, reported that more than a quarter million of assertions were used in the code of Microsoft Office at the time of publication of the paper [Hoa03].

On verification of programs, a natural question arises. Namely, is it possible to devise an oracle that can verify every program? To put it in other words, can we have a general and complete procedure that can determine if a particular assertion fails at some point in an arbitrary program? The answer to this question came again early in the beginning days of computer science. In his seminal paper of 1936, Alan Turing proved that there is no general algorithm that would always terminate and solve the halting problem for all programs. Rice's theorem [Ric53] states that any nontrivial property of a program can be reduced to the halting problem. A direct impact of this impossibility result in practice is that for an automatic verifier targeting general programs there exits challenging programs for which the verifier loops endlessly on them.

In spite of theoretical limitations in achieving an ultimate verifier, the research on developing formal verification techniques started from the early ages of computer science. The initial approaches of proving correctness were not mechanized in the sense that the programmer had to come up with proofs manually. The formal system of Hoare logic [Hoa69] is one of the initial

proof approaches to verify programs. This system uses a set of logical rules, known as Hoare triples, for reasoning about programs. There are basically three components in a Hoare triple: pre-condition, program fragment and post-condition. A Hoare triple is an assertion about the behavior of a program fragment. The assertion expresses that the terminating executions of the program fragment which start in a state satisfying the pre-condition should terminate in a state satisfying the post-condition. There are Hoare triple rules associated to each program command in the programming language. There are also rules describing the composition of commands. The set of all rules together forms an inductive set of proof rules for a program. In the Hoare logic rules, the rule corresponding to iteration or loop represents a formidable challenge. Reasoning about a loop requires the existence of an invariant in a suitable logic. The loop invariant is an assertion that holds before and after each iteration of the loop. Finding the invariant of the loop requires the insight from the programmer. The programmer has to come up with a suitable invariant formula described in a suitable logic to be able to prove the programs containing loops.

Amir Pnueli used temporal logic [Pnu77] for specification and reasoning about sequential and concurrent programs. Temporal logic is an extension of classical logic by operators relating to time. The time operators include operators for specifying the next moment, every future moment and some future moment in time. The behavior of concurrent programs is commonly reactive: they have an ongoing behavior through interactions between the concurrent components and the program environment. The language of temporal logic defines predicates over infinite sequences of the program executions. The general proof principle in [Pnu77] for proving program safety is by induction which depends on the existence of an invariant. The invariant describes a property that holds initially and is transferred along all the valid transitions of the system. In temporal logic some other properties related to time are also considered such as termination which are beyond the scope of this thesis.

Soon after computers gained a tremendous popularity there was inevitably an increase in the number of software developers. It may not be straightforward for an average programmer to carry out a formal proof of correctness by finding suitable invariants. Furthermore, the size of the typical programs increases constantly as the time passes. This makes the software verification task beyond the human capabilities. Exploiting automatic techniques and tools has become a necessary force in the formal verification research after the advent of initial manual proof systems. Model checking appeared in early 1980s as one of the first successful attempts to solve the formal verification problem algorithmically. It was first observed in [Pnu77] that checking the validity of temporal logic on a finite state system is decidable. The model checking approach inspects a finite state system automatically in order to determine if the system conforms to a temporal logic specification.

The model checking method was invented by Clarke and Emerson [CE81] and independently by Queille and Sifakis [QS82]. It checks the satisfaction of a specification logical formula over a system represented by a graph. The predominant specification logic that is used is temporal logic. The algorithm exhaustively searches the state space of the program and checks the

satisfaction of the desired formula. If one of the states violates the desired property a counter example is generated and reported to the user. Noteworthy, the key restriction is the size of state space. One of the main research directions in the model checking community has been developing heuristics to tackle the state space explosion problem. No matter how big the state space becomes, model checking was designed as a verification technique for finite state systems. This is in particular the case with hardware systems in which the size of the memory and registers is fixed. In contrast to hardware, the model for software systems is usually considered to be "infinite-state", since they contain variables and data structures over unbounded domains such as Integers [MCF+97]. Even in the simple case of a program with neither recursive functions nor dynamic allocations which only uses bounded scalar types the state space is so enormously large that it should be treated as infinite for all practical purposes. A reasonable way to come up with verifiers for programs is to approximate the verification problem. To make sense the approximation should be sound here. The verifier checking an approximated version of the program is not allowed to make misleading reports on the (in-)correctness.

Cousot and Cousot formalized the approximative program verifiers in the framework of Abstract Interpretation [CC77]. The purpose of this framework is to correctly approximate a program in a conservative way such that the approximation does not lead to erroneous conclusions. To put it simple, abstract interpretation transfers the concrete program to an abstract program which is decidable and the analysis techniques can fully investigate it. By neglecting some details from the program during construction of an approximation a set of new execution paths are possibly included in the abstraction. The original program may not be able to necessarily perform the newly included traces in its execution. If the outcome of the over-approximation of the program is safe we can conclude that the original concrete program was also a safe program due to conservative abstraction. In the case that the verification procedure locates an error in the abstract program we cannot confidently determine that the concrete program was unsafe. We have to examine the error trace in the original program. If the error trace turns out to be genuine then a true error is found. Otherwise the error trace was just a result of over-approximation. Such error is usually known as "spurious counter-example".

Over the last decade a great deal of effort has been given to design verification methods that benefit from the advantages of both model checking and abstract interpretation. These techniques target the infinite state space of software yet are fully automatic in a push-button style of model checking. A notable approach is bounded model checking. Bounded model checking emerged as a technique for falsification of the properties of finite state hardware systems [BCCZ99]. The basic idea is to search for counterexamples within a certain depth of execution. If the algorithm finds a counterexample, it encodes the counterexample symbolically and checks the satisfiability of the resulted formula using a SAT solver. If the path turns out to be genuine the algorithm gets a satisfying assignment from the SAT solver and reports it to the user. Otherwise it increases the bound of search and continues. The bounded model checking is useful for finding bugs up to certain lengths. The authors of the C Bounded Model Checker

(CBMC) [CKL04] applied bounded model checking to software systems. CBMC unwinds the loops up to a certain depth. To handle variables it "bit blasts" the variables before passing the formula to a SAT solver. The bit blasting approach considers the bit-vector representation of a variable and represents each bit by a propositional variable. It then replaces the arithmetic operators by their equivalent circuits in order to obtain a propositional formula. With the progress in SMT solvers the variables are not unavoidably bit blasted to their bit-vector representation in the newer software bounded model checkers. The new tools provide freedom in choosing data types over the mathematical domains or bit-vectors [SFS11, AMP06].

The CEGAR approach that was mentioned in the beginning of this chapter is one of the predominant approaches in automatic software model checking. In the most recent Competition on Software Verification (SV-COMP 2013) [Bey13] the CEGAR approach and bounded model checking were the most popular techniques among the software verifiers. The insights from three specific advances in formal verification gave rise the idea behind the modern CEGAR-based tools. The first inspiration was the predicate abstraction technique [GS97] which is an abstract interpretation approach for abstracting a program with respect to a set of predicates defined over the program variables. The choice of the predicates in [GS97] is manual and is inspired by the guards and assignments of the program. The second invention was the CEGAR [CGJ$^+$00] methodology that extended the predicate abstraction framework by introducing an abstraction refinement loop. The main difference between the original predicate abstraction approach and CEGAR is in refinement: if the abstraction is too coarse and generates spurious counterexamples the abstraction is refined automatically. The refinement procedure of [CGJ$^+$00] considers an abstraction as an equivalence relation on the states of the original program. It then tries to divide the equivalence classes to smaller classes with the goal of excluding the unreachable states. The suitability of the existing refinement techniques for software systems was a question until interpolants appeared as the third novel conception. Craig Interpolants are shown to be a reasonable mechanism to refine the abstractions in software verification [HJMM04]. Nowadays the common-sense meaning of "predicate abstraction" is usually the CEGAR approach with interpolation as the refinement phase.

1.2 Predicate Abstraction

For the purpose of this chapter I describe the predicate abstraction methodology by using the small program in Figure 1.1. Chapter 3 gives a formal overview of the method. The program in Figure 1.1(a) contains two integer variables x and y. It starts by assuming the condition that the variables are greater than or equal to 0. Then in a while loop with a non-deterministic condition (represented by *) it decrements x by y if x is greater than y . Otherwise it decrements y by x. After the exit from the loop there is an assertion to ensure that the variable x is not equal to -1. The control flow graph of the program is in Figure 1.1(b). In the transitions of the control flow graph the primed variables represent the values of the variables in the destination vertex.

Intuitively, a predicate is simply a statement on the variables of the program. For example,

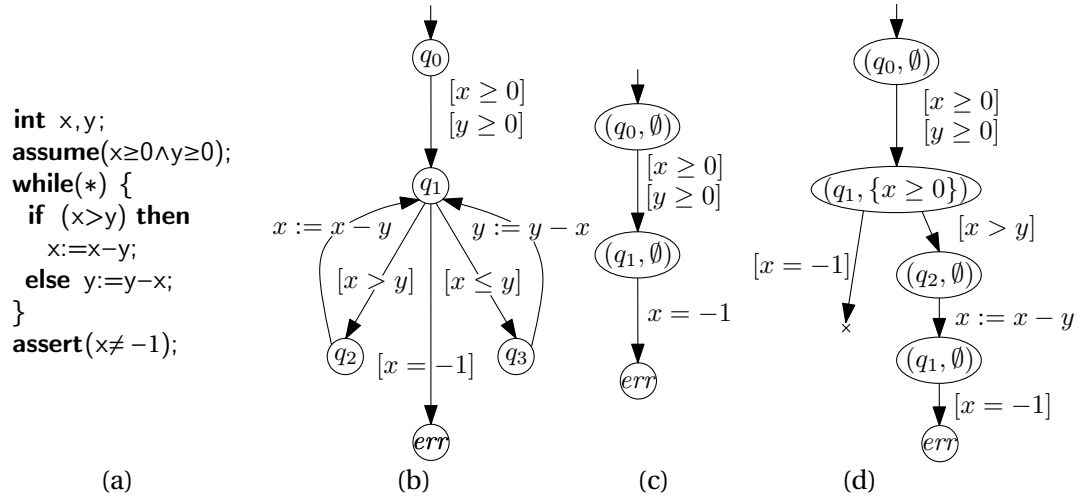


Figure 1.1: (a) The source code of a program. (b) The control flow graph. (c),(d). The reachability graphs for two possible iterations of CEGAR.

the formula $x > y$ or $x = 0$ are all proper predicates. The algorithm starts with an initial set of predicates. This can be empty, here we also assume that no predicates are given at the beginning. The basic task of predicate abstraction is to construct an abstract reachability graph (ARG). This graph is obtained by unwinding the control flow graph from its start state in the abstract space. To find the children of a node in the ARG we compute the abstract successors with respect to the outgoing transitions. The abstract successor basically assumes the abstraction of the current node, computes the set of predicates in the destination that are valid under the outgoing transition to that destination. The set of all valid predicates in the destination child constitutes the abstraction of the child. Since initially we do not have any predicate there are no predicates (the abstraction is *true*) in every node in the first reachability tree in Figure 1.1(c) and we immediately hit the error state.

After observing an error state in a path in the abstract program we have to check its legitimacy. For this purpose we convert the path to a mathematical formula by converting it to static single assignment form (SSA) [CFR+91]. A formula for the path in Figure 1.1(c) is $x_1 \geq 0 \wedge y_1 \geq 0 \wedge x_1 = -1$. A theorem prover call confirms the unsatisfiability of the formula hence the spuriousness of the path. Now we enter the refinement phase by calling an interpolating theorem prover. The interpolation at each step of the path allows us to remove irrelevant information that is not needed to maintain the unsatisfiability of path. In other words, the interpolant expresses the essence of the reason for unsatisfiability of a trace in the program. There can be generally multiple explanations for unsatisfiability of a formula. In practice, the result of a query to an interpolating theorem prover depends on the implemented interpolation technique and on the syntax of the given formula. Assuming that the interpolating theorem prover returns $x_1 \geq 0$ as an interpolant we add the predicate $x \geq 0$ as a candidate predicate for the state q_1.

Now it is the time to rerun the CEGAR loop, this time with a nonempty set of predicates. After constructing the initial abstraction for q_0 which is the empty set $\emptyset$, we look into the transition from q_0 to q_1. There is a candidate predicate for q_1 which is $x \geq 0$. Indeed this predicate is valid after the transition **assume**$(x \geq 0 \wedge y \geq 0)$ so we keep it as an abstraction for q_1. Having the assumption $x \geq 0$ in the state $(q_1, \{x \geq 0\})$ prevents us from again hitting the error state. So we cut the direct transition to error and continue with another possibility in q_1, the transition to q_2. This is the right branch in the reachability graph of Figure 1.1(d). Since there is no candidate predicate for q_2, by the transition to q_2 we construct the ARG node $(q_2, \emptyset)$ with the assumption true (empty set). Making the transition $x := x - y$ with the assumption $true$ from $(q_2, \emptyset)$ to q_1 again makes an abstract state with no predicates $(q_1, \emptyset)$. Nothing prevents this state from making a transition to error. In general, after hitting the error state it suffices to just consider a suffix of the path which is still infeasible. The last three steps of the path to error is an infeasible path itself $[x > y]; x := x - y; [x = -1]$. The SSA formula for this suffix of the path to error is $(x_1 > y_1) \wedge (x_2 = x_1 - y_1) \wedge (x_2 = -1)$. We get the two interpolants $x_1 > y_1$, $x_2 \geq 0$ from the interpolating theorem prover. We had the predicate $x \geq 0$ for the state q_1 before. We just add the predicate $x > y$ to q_2 and rerun the abstraction. The algorithm similarly traverses the loop q_1—q_3 and finally constructs a complete reachability graph to show that the program is safe.

1.2.1 Challenges and Limitations

There are some fundamental key questions to ask about predicate abstraction even in the small example of this section. We proved the problem using three refinement steps; first by removing the direct path to error, then a path going through the q_1—q_2 loop and finally a path going though the q_1—q_3 loop. One might ask if it is possible to remove the three discovered counter-examples of the abstract program by just a single refinement step. This can reduce the calls to the interpolating theorem prover to a single call. This thesis systematically studies the problem of reducing the number of interpolation calls as one of its goal. It introduces a new notion of *disjunctive interpolants* for handling several counter-examples together. Another fundamental question is the generality of the generated interpolants. In this example we got general interpolants from the interpolating theorem prover that removed all the possible unfolding of the loops. In general it is possible that the interpolating theorem prover computes specific interpolants for each traversal of the loop. To address this problem we present an acceleration methodology that basically prevents the system from diverging. One of the ultimate challenges for the CEGAR tools is being able to analyze inter-procedural and concurrent programs. This dissertation uses the intermediate language of Horn clauses as a useful technique in proving such programs. Researchers have observed that Horn clauses are suitable for representation of many complicated domains.

1.3 Technical Achievements

The technical contributions of this dissertation are summarized as the following.

- We propose disjunctive interpolants as a new form of interpolation which removes several counter-examples together to increase convergence and improve performance. We present a predicate abstraction based solver for recursive Horn clauses that uses disjunctive interpolants for refinement.
- We introduce a taxonomy for recursion-free Horn clauses and show the corresponding interpolation problem that they solve. We give computational complexities in each of the proposed classes for the underlying languages of Boolean logic and Presburger arithmetic.
- We introduce the counter-example guided accelerated abstraction refinement (CEGAAR) algorithm as an improved CEGAR approach. The novelty of the proposed algorithm is in combining the two approaches of predicate abstraction and the computation of transitive closure of loops in a program. The CEGAAR algorithm is able to solve problems on which the traditional tools mostly diverge.

1.3.1 Tools and Applications

We have implemented all the techniques presented in this thesis in the framework of Eldarica [1] that works in the domain of Integer Presburger arithmetic. Eldarica is able to input several languages. The input languages are the following: A subset of Scala, the numerical transition system [HKG+12], Horn clauses in Prolog format [GLPR12], Horn clauses in SMT-LIB2 format and UPPAAL [2] benchmarks. We have used Eldarica in the verification of different benchmarks. The benchmarks come from a wide variety of sources. They include benchmarks from verification conditions for programs with arrays, C programs with challenging loops, models extracted from programs with singly-linked lists, C programs provided as benchmarks in the NECLA static analysis suite, C programs with asynchronous procedure calls, models extracted from VHDL models of circuits, benchmarks from the HSF library and benchmarks from the International Competition on Software Verification. The experiments sections of Chapters 4 and 5 give more information about the benchmarks.

1.4 Outline

This thesis opens with an overview of the program model in Chapter 2 that is used in the rest of the thesis. It then continues to present a background on predicate abstraction in Chapter 3, the main technique that is exploited and pursued in the thesis. Chapter 4 is devoted to the

[1] http://lara.epfl.ch/w/eldarica
[2] http://www.uppaal.org/

idea of combining predicate abstraction with computation of transitive closures of the loops, namely accelerating the interpolants. The entire Chapter 5 is about solving Horn clauses and the correspondence between recursion-free Horn clauses and interpolation. Our initial steps on verification of timed systems using Horn clauses are the subject of Chapter 6. This thesis concludes with Chapter 7 on related works and conclusions in Chapter 8.

2 Precise Modeling of Software

> There's no sense in being precise when you don't even know what you're talking about.
>
> John von Neumann

In this section we define our model for programs. The analysis in the later sections is based on the model defined in this section. Unless explicitly stated otherwise, this dissertation focuses on verification of programs which their underlying transition relation is expressed using Presburger arithmetic. Such transition relation are known as counter automata, counter systems, counter machines or Integer Numerical Transition Systems (INTS) [HKG+12]. This model is an infinite-state extension of the model of finite-state *boolean transition systems*. In principle any Turing-complete class of systems can be simulated by counter systems [Min67].

The interest for transitions systems manipulating Integers comes from the fact that they can encode various classes of systems with unbounded (or very large) data domains, such as hardware circuits, cache memories, or software systems with variables of non-primitive types, such as integer arrays, pointers and/or recursive data structures. A number of recent works have revealed cost-effective approximate reductions of verification problems for several classes of complex systems to decision problems phrased in terms of integer transition systems. Examples of systems that can be effectively verified by means of integer programs include: specifications of hardware components [SV07], programs with singly-linked lists [BBH+06], trees [HIRV07], and integer arrays [BHI+09].

We introduce two common representation languages in this section that can essentially encode different programs: INTS and Horn clauses. Common representation languages offer many potential benefits to software verification research by providing a robust communication mechanism for exchanging information and comparison among verification tools.

2.1 Program Model

In the following, let $\mathbb{Z}$ denote the set of integer numbers. Presburger arithmetic is the first-order logic theory of integer addition. Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a set of variables. Without loss of generality, a Presburger formula $\phi(\mathbf{x})$ is a quantified Boolean combination of atomic propositions of the form $t_1 \leq t_2$, where t_1 and t_2 are linear terms over the variables $\mathbf{x}$. A *linear term* t over a set of variables in $\mathbf{x}$ is a linear combination of the form $a_0 + \sum_{i=1}^{n} a_i x_i$, where $a_0, a_1, \ldots, a_n \in \mathbb{Z}$. A variable not occurring under the scope of a quantifier is said to be free. If $FV(\phi) \subseteq \mathbf{x}$ is the set of free variables in ϕ and $\nu : \mathbf{x} \to \mathbb{Z}$ is an interpretation of the variables in $\mathbf{x}$, we say that ν satisfies ϕ, written $\nu \models \phi$, if the formula resulting by replacing each variable $x \in \mathbf{x}$ by $\nu(x)$ in ϕ is logically equivalent to true. Notice that each formula defines a set of integer tuples which satisfies it. Let $\mathbf{x}'$ denote the set $\{x'_1, \ldots, x'_n\}$. A relation $R \in \mathbb{Z}^n \times \mathbb{Z}^n$ is defined by a formula $R(\mathbf{x}, \mathbf{x}')$. We denote by $\vec{\mathbf{x}}$ an ordered sequence $\langle x_1, \cdots x_n \rangle$ of variables and $|\vec{\mathbf{x}}|$ is the length of sequence.

We represent a program using a control-flow graph (or control-flow automaton) with nodes representing program points and the edges are annotated with relations between initial and final values of variables. In a relation between the initial and final values of variables the unprimed variables $\mathbf{x}$ denote the values of the variables at the source whereas the primed variables $\mathbf{x}'$ denote the values of the variables at the destination of a transition.

Definition 2.1.1. *A program is a tuple* $P = \langle \mathbf{x_g}, \{S_1, \cdots, S_n\}, S_m \rangle$ *where* $\mathbf{x_g}$ *represents a set of global variables,* $S_1, \cdots, S_n$ *are procedures, and* S_m *is the main procedure* $(1 \leq m \leq n)$*. Each procedure is a tuple* $S_i = \langle \mathbf{x_i}, \vec{\mathbf{x}}_\mathbf{i}^{\,in}, \vec{\mathbf{x}}_\mathbf{i}^{\,out}, Q_i, q_i^{init}, F_i, E_i, \rightarrow_i \rangle$*, where:*

- $\mathbf{x_i}$ *is the set of variables in* S_i *including the global variables.*
- $\vec{\mathbf{x}}_\mathbf{i}^{\,in} \subseteq \mathbf{x_i}$ *is the set of input variables and* $\vec{\mathbf{x}}_\mathbf{i}^{\,out} \subseteq \mathbf{x}$ *is the set of output variables.*
- Q_i *is the set of control states of* S_i*. We require that the sets of control states are pairwise disjoint, i.e., for all* $i \neq j$*,* $Q_i \cap Q_j = \emptyset$*.*
- $q_i^{init} \in Q_i$ *is the initial state,* $F_i, E_i \subseteq Q_i$ *are the final and error states.*
- $\rightarrow_i$ *is a set of transition rules of the following forms.*
  - $q \xrightarrow{R(\mathbf{x_i}, \mathbf{x'_i})} q'$ *is an internal transition, where* $q, q' \in Q_i$ *are the source and destination states, and* $R(\mathbf{x_i}, \mathbf{x'_i})$ *is a Presburger arithmetic formula defining a relation.*
  - $q \xrightarrow{\vec{\mathbf{z}}' = call_j(\vec{\mathbf{t}})} q'$ *is a procedure call transition, where* $q, q' \in Q_i$ *are the source and destination states,* $1 \leq j \leq n$ *is the index of the callee,* $\vec{\mathbf{t}}$ *is a sequence of linear terms over* $\mathbf{x_i}$ *called the actual parameters,* $\vec{\mathbf{z}} \subseteq \mathbf{x_i}$ *is a sequence of variables called the return variables. We require that the numbers of parameters and return variables of the call transition match the numbers of input and output variables of the callee,* $|\vec{\mathbf{t}}| = |\vec{\mathbf{x}}_\mathbf{j}^{\,in}|$ *and* $|\vec{\mathbf{z}}| = |\vec{\mathbf{x}}_\mathbf{j}^{\,out}|$*.*

In the case that the set of procedures is empty we deal with the single procedure S_m without any procedure call. Analyzing such programs is within the scope of *intraprocedural* analysis. For a program with only the main procedure S_m the *configuration* of the program is a pair $\langle q, v\rangle$ where $q \in Q_m$ is a control state, and $v : \mathbf{x_m} \to \mathbb{Z}$ is a valuation of the variables. A *run* of S_m is a finite sequence $c_0, c_1, \ldots, c_k$ of configurations such that $c_i = \langle q_i, v_i\rangle$ and $q_0 = q_m^{init}$, $q_k \in E \cup F$ and for all $0 \leq i < k$, $q_i \xrightarrow{\mathscr{R}_i} q_{i+1}$ and $v_i \cup v_{i+1} \models \mathscr{R}_i$. A run is said to be *safe* if $q_k \in F$. The system represented by S_m is said to be *safe* if all its runs are safe. We sometimes represent a program with a single main procedure S_m with only the model of the its main procedure S_m: $\langle \mathbf{x}, Q, q^{init}, F, E, \to\rangle$.

In the inter-procedural case the transitions can be either internal or procedure calls. In a run of a system with procedure calls whenever a callee reaches a final state, the execution returns back to the caller. A path in the program is a sequence of transitions in which the calls and returns matches. A *call graph* is a directed graph that represents the calling relationships between procedures of a program. Specifically, each node in a call graph represents a procedure and existence of the edge (i, j) indicates that procedure P_i calls the procedure P_j. A program is *recursive* if there exists a cycle in the call graph.

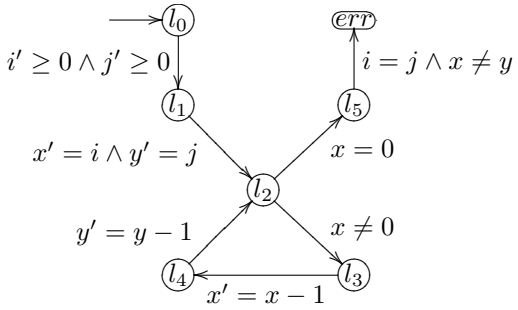
Consider the program in Figure 2.1(a). This program contains a single non-recursive procedure. The model for the main procedure is the following.

$$
\begin{array}{ll}
\Big\langle \{x, y, i, j\}\ ,\ \langle\rangle\ ,\ \langle\rangle\ , & \{l_0, l_1, l_2, l_3, l_4, l_5, err\}\ ,\ l_0\ ,\ \emptyset\ ,\ \{err\}, \\
\big\{ l_0 \to l_1 & (i' \geq 0 \wedge j' \geq 0) \\
l_1 \to l_2 & (x' = i \wedge y' = j \wedge i' = i \wedge j' = j) \\
l_4 \to l_2 & (y' = y - 1 \wedge x' = x \wedge i' = i \wedge j' = j) \\
l_3 \to l_4 & (x' = x - 1 \wedge y' = y \wedge i' = i \wedge j' = j) \\
l_2 \to l_3 & (x \neq 0 \wedge x' = x \wedge y' = y \wedge i' = i \wedge j' = j) \\
l_2 \to l_5 & (x = 0 \wedge x' = x \wedge y' = y \wedge i' = i \wedge j' = j) \\
l_5 \to err & (i = j \wedge x \neq y \wedge x' = x \wedge y' = y \wedge i' = i \wedge j' = j)\big\}\Big\rangle
\end{array}
$$

Figure 2.1(b) shows the model graphically. Most programmers would have little difficulty observing that the assertion will always succeed, but many tools, including non-relational abstract interpretation, as well as predicate abstraction with arbitrary interpolation can fail to prove the assertion to hold [JM06]. The program in Figure 2.1(c) is a recursive program generally known as the McCarthy 91 function. The havoc function assigns an arbitrary number to its argument non-deterministically. The Figure 2.1(d) is the corresponding integer numerical transition system. This program has a complex recursion pattern and it is usually considered as a challenging problem for formal verification [Man74]. The ELDARICA framework in fact succeeds for verification of both of these examples.

```
     def main() {
       var i,j: Int
l0:    havoc(i: Int >= 0)
       havoc(j: Int >= 0)
l1:    var x: Int = i
       var y: Int = j
l2:    while (x != 0) {
l3:      x = x - 1
l4:      y = y - 1
       }
l5:    if (i == j)
         assert(x == y)
       }
```

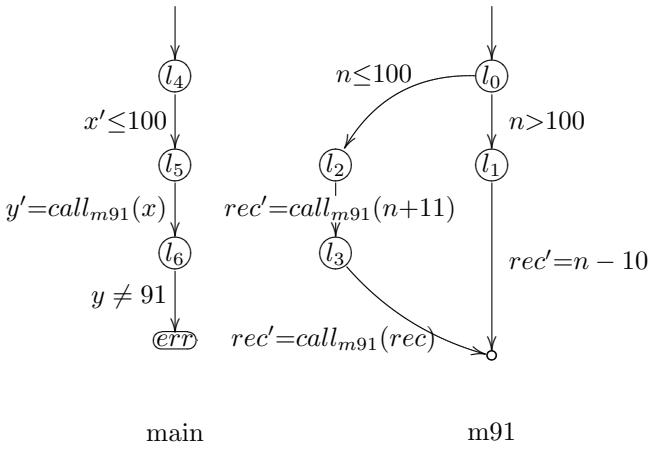
(a)



(b)

```
     def mc91(n: Int)
       returns rec: Int = {
l0:    if (n > 100)
l1:      rec = n - 10
       else {
l2:      rec = mc91(n+11)
l3:      rec = mc91(rec)
       }
     }
     def main() {
l4:    havoc(x: Int <= 100)
l5:    var y: Int = mc91(x)
l6:    assert(y == 91)
     }
```

(c)



(d)

Figure 2.1: Example of a non-recursive program (a) and a recursive program (c) with corresponding integer transition systems (b), (d). By convention, if a variable v does not appear in the transition relation formula, we implicitly assume that the frame condition $v = v'$ is conjoined.

2.2 The INTS Infrastructure

INTS (Integer Numerical Transition System) is a common representation language for representing integer transition systems. The INTS syntax is a textual description of a control flow graph labeled by Presburger arithmetic formulae, as in Figure 2.1 (b). We have developed a toolkit for rigorous automated verification of programs in INTS format. The unifying component is the INTS library[1], which defines the syntax of the INTS representation by providing a parser and a library of abstract syntax tree classes.

At this point, there are several tools supporting the INTS format, as input and/or output language. The INTS library is designed for easy bridging with new tools, which can be either front-

[1] http://richmodels.epfl.ch/ntscomp/ntslib

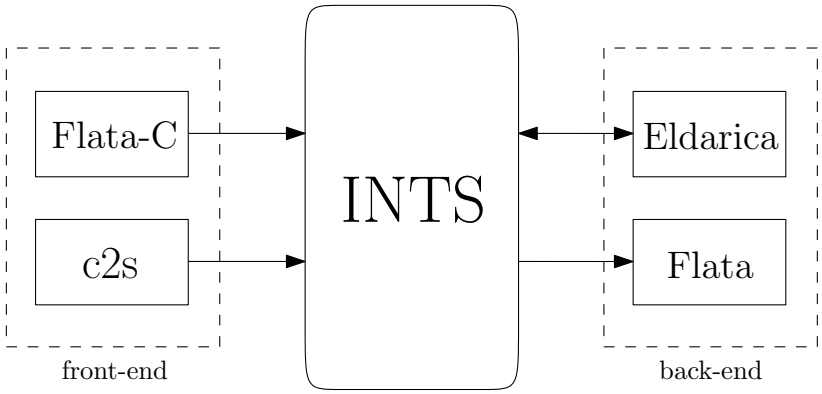


Figure 2.2: Integer Numerical Transition System Infrastructure

ends (translators from mainstream programming languages into INTS), back-ends(verification tools), or both. Figure 2.2 shows a schematic diagram of the INTS framework. Currently, there exist tools to generate INTS from sequential and concurrent C (Flata-C), Scala (Eldarica front-end), and Boogie (c2s). The acceleration based tool Flata and the predicate abstraction based tool Eldarica can verify INTS benchmarks [HKG+12].

Flata verifier. FLATA[2] is a verification tool for hierarchical non-recursive INTS models. The tool computes the summary relation for each INTS independently of its calling context, thus avoiding the overhead of procedure inlining. The verification is based on computing transitive closure of loops. Classes of integer relations for which transitive closures can be computed precisely include: (1) *difference bounds relations*, (2) *octagons*, and (3) *finite monoid affine transformations*. For these three classes, the transitive closures can be effectively defined in Presburger arithmetic. FLATA integrates the transitive closure computation method for difference bounds and octagonal relations from [BIK10] in a semi-algorithm computing the summary relation incrementally, by eliminating control states and composing incoming with outgoing relations.

Eldarica verifier. ELDARICA[3] implements predicate abstraction with Counter-Example Guided Abstraction Refinement (CEGAR). It generates an abstract reachability tree (ART) of the system on demand, using lazy abstraction with Cartesian abstraction, and uses interpolation to refine the set of predicates [HJMM04]. For checking the feasibility of paths, and constructing abstractions, ELDARICA employs the provers Z3[4] and Princess.[5] In addition, ELDARICA uses caching of previously explored states and formulae to prevent unnecessary reconstruction of trees. Large block encoding can be performed to reduce the number of calls to the interpolating theorem prover.

Eldarica refines abstractions with the help of *Craig Interpolants*, extracted from infeasibility

[2] http://www-verimag.imag.fr/FLATA.html
[3] http://lara.epfl.ch/w/eldarica
[4] http://research.microsoft.com/en-us/um/redmond/projects/z3/
[5] http://www.philipp.ruemmer.org/princess.shtml

(1) r0(N,Rec) ← **true**
(2) r1(N,Rec) ← r0(N,Rec) ∧ N > 100
(3) r2(N,Rec) ← r0(N,Rec) ∧ N ≤ 100
(4) r3(N,Rec') ← r2(N,Rec) ∧ rf(N + 11,Rec')
(5) rf(N,Rec') ← r1(N,Rec) ∧ Rec' = N − 10
(6) rf(N,Rec') ← r3(N,Rec) ∧ rf(Rec,Rec')

(7) r4(X,Y) ← **true**
(8) r5(X',Y) ← r4(X,Y) ∧ X' ≤ 100
(9) r6(X,Y') ← r5(X,Y) ∧ rf(X,Y')
(10) **false** ← r6(X,Y) ∧ Y ≠ 91

Figure 2.3: The encoding of the program in Figure 2.1 (c) into a set of recursive Horn clauses.

proofs for spurious counterexamples. The complete interpolation procedure for Presburger arithmetic was proposed in [BKRW11], and is implemented as part of Princess. We refer to Chapter 3 for a more detailed description of predicate abstraction.

2.3 Horn Clauses

The problem of verifying the correctness of a program reduces to finding the solution to a set of recursive predicates [BMR12]. Recent advances in theorem proving and interpolation in different subsets of logic gave rise to new software model checking tools which accept recursive predicates as their native input format. Most notably, HSF [GGL+12] inputs recursive predicates encoded as Prolog style Horn clauses and μZ [HB12] uses an extension of the SMT-LIB format with recursive predicates. The predicate abstraction framework Eldarica [HKG+12] is also able to verify Horn clauses in both Prolog style and SMT-LIB formats.

We treat Horn clauses in greater detail later in Chapter 5. Informally, a Horn clause is a disjunction of literals with at most one positive literal. Fixing a vocabulary of relation symbols $\mathscr{R}$ a Horn clause is a formula of the following form.

$$\forall \mathbf{x}.\ l_0(\mathbf{x}) \wedge \cdots \wedge l_n(\mathbf{x}) \longrightarrow l(\mathbf{x})$$

The literals $l_i(\mathbf{x})$ in the Horn clause are either a relation symbol $r(\mathbf{x})$ from $\mathscr{R}$ or an arithmetic formula over $\mathbf{x}$ without any symbol of $\mathscr{R}$. We describe the mapping of the recursive program in Figure 2.1 (c) to recursive Horn clauses in Figure 2.3. For translation to Horn clauses we assign an uninterpreted relation symbol $\mathtt{r}i$ to each state l_i of the control flow graph. The arguments of the relation symbol $\mathtt{r}i$ act as placeholders of the visible variables in the state l_i. The relation symbol `rf` corresponds to the summary of the function `mc91`. In the relation symbol for function summaries we do not include the local variables since they are invisible from outside of the function. The first argument of `rf` is the input and the second one is the output. We do not dedicate any relation symbol to the error state err.

$$
\begin{aligned}
r_0(N, Rec) &\equiv true \\
r_1(N, Rec) &\equiv (N \geq 101) \\
r_2(N, Rec) &\equiv (N \leq 100) \\
r_3(N, Rec) &\equiv (Rec \leq 101) \\
r_4(X, Y) &\equiv true \\
r_5(X, Y) &\equiv (X \leq 100) \\
r_6(X, Y) &\equiv (Y = 91) \\
r_f(N, Rec) &\equiv ((Rec = 91) \vee (((N - Rec) \geq 10) \wedge (N \geq 102)))
\end{aligned}
$$

Figure 2.4: Solution of the Horn clauses in Fig. 2.3.

The initial states of the functions are not constrained at the beginning; they are just implied by *true*. The clause that has *false* as its head corresponds to the assertion in the program. In order to satisfy the assertion with the head *false*, the body of the clause should also be evaluated to *false*. We put the condition leading to error in the body of this clause to ensure the error condition is not happening. The rest of the clauses are one to one translation of the edges in the control flow graph.

For the edges with no function calls we merely relate the variables in the previous state to the variables in the next state using the transfer functions on the edges. For example, the clause (2) expresses that *rec* is kept unchanged in the transition from l_0 to l_1 and the value of `n` is greater than 100 in l_1. For the edges with function call we should also take care of the passing arguments and the return values. For example, the clause (4) corresponds to the edge containing a function call from l_2 to l_3. This clause sets the value of `rec` in the state l_3 to the return value of the function `mc91` called with $n + 11$. Note that the only clauses in this example that have more than one relation symbols in the body are the ones related to edges with function calls.

The solution of the obtained system of Horn clauses demonstrates the correctness of the program. In a solution each relation symbol is mapped to an expression over its arguments. If we replace the relation symbols in the clauses by the expressions in the solution we should obtain only valid clauses. In a system with a genuine path to error we cannot find any solution to the system since we have no way to satisfy the assertion clause. Fig. 2.4 gives one possible solution of the Horn clauses in terms of concrete formulae, found by our verification tool Eldarica.

2.4 Discrete vs. Dense Domains

An important trend in verification of numerical systems assumes that all variables range over rational (real) numbers. This results in an overapproximation of the set of behaviors of the

system, which, in turn, can be verified by cost-effective methods on rational (real) numbers, such as: invariant discovery using polyhedra and widening, constraint solving using linear programming, template-based linear and polynomial invariant inference, linear interpolation, etc.

However, using rational (real) domains results in a loss of precision and spurious error reports that are impossible to revert by typical classical refinement methods. Moreover, in some situations, one explicitly needs to reason about modular constraints (i.e., congruence modulo an integer constant), which is not possible within the rational domain. These situations typically occur when verifying pointer arithmetic properties—here one needs to check if all memory accesses are aligned with respect to the machine-dependent integer size.

3 Background on Predicate Abstraction

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> Edsger W. Dijkstra

Predicate abstraction with Counter-Example Guided Abstraction Refinement (CEGAR) is one of the most prominent technologies for software verification. This technique has been used in a number of successful software model checkers including BLAST [HJMS02]. The approach proceeds in a three-step loop: (i) Compute a conservative over-approximation of the original state space with respect to a set of predicates. The conservative abstraction guarantees that, for every execution in the concrete system, there is a corresponding execution in the abstract system. (ii) Model check the abstract model. If the abstract model satisfies the (reachability) property, then the concrete system also satisfies the property and the verification stops. Otherwise, the verifier checks whether the counter-example trace is genuine by concretizing an abstract trace and calling a theorem prover. If the example is real, the verifier reports the counterexample and stops. (iii) Otherwise the process continues and the counter-example suggests additional predicates to refine the abstraction, avoiding that particular spurious trace. The loop restarts from the beginning with the extended set of predicates. The loop stops as soon as it finds a real bug or it proves the correctness, but may run forever, because the general verification problem is undecidable. Figure 3.3 depicts the high level structure of CEGAR.

3.1 Predicate Abstraction

Predicate abstraction computes an overapproximation of the transition system generated by a program and verifies whether an error state is reachable in the abstract system. If no error occurs in the abstract system, the algorithm reports that the original system is safe. Otherwise, if a path to an error state (counterexample) has been found in the abstract system, the corresponding concrete path is checked. If this latter path corresponds to a real execution

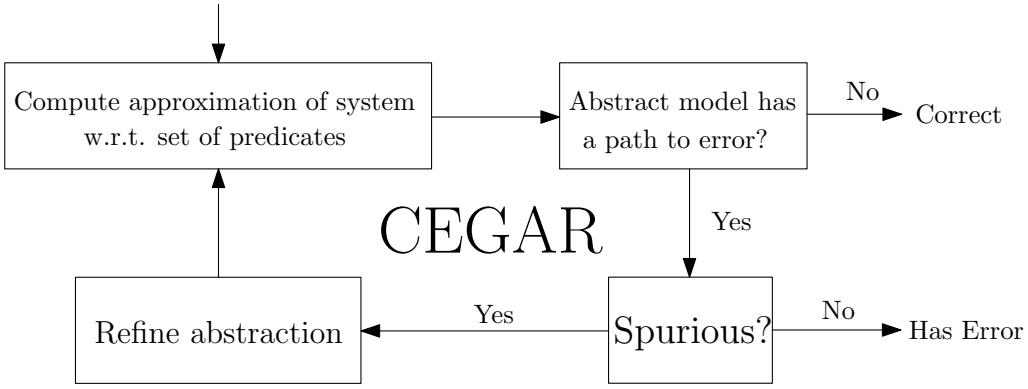


Figure 3.1: Counter-Example Guided Abstraction Refinement (CEGAR)

of the system, then a real error has been found. Otherwise, the abstraction is refined in order to exclude the counterexample, and the procedure continues.

Consider the program model $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow \rangle$ introduced in Chapter 2. A *predicate* P is a first-order arithmetic formula on $\mathbf{x}$. Assume that we have a set of predicates $\mathscr{P}$. For a set of n-tuples $S \subseteq \mathbb{Z}^n$ and a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, let $sp(S,R) = \{\vec{v} \in \mathbb{Z}^n \mid \exists \vec{u} \in S \,.\, (\vec{u}, \vec{v}) \in R\}$ denote the *strongest postcondition* of S via R. We use sp for sets and relations, as well as for logical formulae defining them.

Definition 3.1.1. *An* abstract reachability tree *(ART) for G is a tuple $T = \langle S, \pi, r, e \rangle$ where $S \subseteq Q \times 2^{\mathscr{P} \setminus \{\bot\}}$ is a set of nodes (we do not introduce any node for empty set of predicates $\langle q, \emptyset \rangle$), $\pi : Q \rightarrow 2^{\mathscr{P}}$ is a mapping associating control states with sets of predicates, $r = \langle q^{init}, \{\top\} \rangle$ is the root node, $e \subseteq S \times S$ is a tree-structured edge relation:*

- *all nodes in S are reachable from the root r*
- *for all $n, m, p \in S$, $e(n,p) \wedge e(m,p) \Rightarrow n = m$*
- $e(\langle q_1, \Phi_1 \rangle, \langle q_2, \Phi_2 \rangle) \;\Rightarrow\; q_1 \xrightarrow{R} q_2$ *and* $\Phi_2 = \{P \in \pi(q_2) \mid sp(\bigwedge \Phi_1, R) \rightarrow P\}$

We say that an ART node $\langle q_1, \Phi_1 \rangle$ is *subsumed* by another node $\langle q_2, \Phi_2 \rangle$ if and only if $q_1 = q_2$ and $\bigwedge \Phi_1 \rightarrow \bigwedge \Phi_2$. During the construction of the abstract reachability tree whenever the model checker discovers a node that is subsumed by an existing node in the tree it backtracks and prunes the searching state space from the subsumed node.

It can be easily checked that each path $\sigma : r = \langle q_1^{init}, \{\top\} \rangle, \langle q_2, \Phi_2 \rangle, \ldots, \langle q_k, \Phi_k \rangle$, starting from the root in T, can be mapped into a trace $\theta : q_1^{init} \xrightarrow{R_1} q_2 \ldots q_{k-1} \xrightarrow{R_{k-1}} q_k$ of G, such that $sp(\top, R_1 \circ R_2 \circ \ldots \circ R_{n-1}) \rightarrow \bigwedge \Phi_k$. The composition of two relations $R_1, R_2 \in \mathbb{Z}^n \times \mathbb{Z}^n$ is denoted by $R_1 \circ R_2 = \{(\vec{u}, \vec{v}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \exists \vec{t} \in \mathbb{Z}^n \,.\, (\vec{u}, \vec{t}) \in R_1 \text{ and } (\vec{t}, \vec{v}) \in R_2\}$. We say that θ is a *concretization* of σ, or that σ concretizes to θ. A path in an ART is said to be *spurious* if none of its concretizations is feasible.

3.2 Interpolation-Based Abstraction Refinement

By *refinement* we understand the process of enriching the predicate mapping π of an ART $T = \langle S, \pi, r, e\rangle$ with new predicates. The goal of refinement is to prevent spurious counterexamples (paths to an error state) from appearing in the ART. A key difficulty in the predicate abstraction approach is to automatically find predicates to make the abstraction sufficiently precise [BPR02]. A breakthrough technique is to generate predicates based on *Craig interpolants* [Cra57] derived from the proof of unfeasibility of a spurious trace [HJMM04]. To this end, an effective technique used in many predicate abstraction tools is that of *interpolation*.

Given an unsatisfiable conjunction $A \wedge B$, an interpolant I is a formula using the common variables of A and B, such that $A \rightarrow I$ is valid and $I \wedge B$ is unsatisfiable. Intuitively, I is the explanation behind the unsatisfiability of $A \wedge B$. Below we introduce a slightly more general definition of a *trace interpolant*, also known as inductive sequences of interpolants.

Definition 3.2.1 (Trace Interpolant([HJMM04, McM06, JM06]))**.** *Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow\rangle$ be a CFG and*

$$\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$$

be an infeasible trace of G. An interpolant *for θ is a sequence of predicates $\langle I_1, I_2, \ldots, I_n\rangle$ with free variables in $\mathbf{x}$, such that: $I_1 = \top$, $I_n = \bot$, and for all $i = 1, \ldots, n-1$, $sp(I_i, R_i) \rightarrow I_{i+1}$.*

Interpolants exist for many theories, including all theories with quantifier elimination, and thus for Presburger arithmetic. Moreover, a trace is infeasible if and only if it has an interpolant. Including any interpolant of an infeasible trace into the predicate mapping of an ART suffices to eliminate any abstraction of the trace from the ART. We can thus refine the ART and exclude an infeasible trace by including the interpolant that proves the infeasibility of the trace.

Note that the refinement technique using Definition 3.2.1 only guarantees that *one* spurious counterexample is eliminated from the ART with each refinement step. This fact hinders the efficiency of predicate abstraction tools, which must rely on the ability of theorem provers to produce interpolants that are general enough to eliminate more than one spurious counterexample at the time. The theorem provers however have limited knowledge of the structure of the system and producing general interpolants that exclude several (or all) spurious traces is merely a matter of luck.

Theorem 3.2.1. *Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow\rangle$ be a CFG and $\theta : q_1 = q^{init} \xrightarrow{R_1} q_2 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$ be an infeasible trace of G. If $T = \langle S, \pi, r, e\rangle$ is an ART for G such that there exists an interpolant $\langle I_i \in \pi(q_i)\rangle_{i=1}^{n}$ for θ, then no path in T concretizes to θ.*

Proof. By contradiction, suppose that there exists a path

$$\sigma : \langle q_1, \Phi_1\rangle, \langle q_2, \Phi_2\rangle, \ldots, \langle q_n, \Phi_n\rangle$$

in T, that concretizes to θ. We show by induction on i, that $I_i \in \Phi_i$, for all $i = 1, \ldots, n$. By the

definition of T, $I_1 = \top \in \Phi_1$, always. For the induction step, assume that $I_{i-1} \in \Phi_{i-1}$, for some $i > 1$. By the definition of T we have $\Phi_i = \{P \in \pi(q_i) \mid sp(\bigwedge \Phi_{i-1}, R_i) \rightarrow P\}$. Since $sp(I_{i-1}, R_i) \rightarrow I_i$, by Definition 3.2.1 and $I_{i-1} \in \Phi_{i-1}$, we have $\bigwedge \Phi_{i-1} \rightarrow I_{i-1}$, and by monotonicity of the sp operator, $sp(\bigwedge \Phi_{i-1}, R_i) \rightarrow I_i$. But $I_i \in \pi(q_i)$ which implies $I_i \in \Phi_i$, by the definition of T. Consequently $I_n = \bot \in \Phi_n$, which is in contradiction with the fact that no node in T may contain $\bot$ in its second component. □

3.3 Example for Interpolation

As an example for interpolation, consider the infeasible path l_0, l_1, l_2, l_5, err from the example program in Figure 2.1(a). By converting the statements and guards into a formula, we extract the following path constraint:

$$\underbrace{i_0 \geq 0 \wedge j_0 \geq 0 \wedge x_0 = i_0 \wedge y_0 = j_0 \wedge x_0 = 0}_{\phi(i_0, j_0, x_0, y_0)} \wedge \underbrace{i_0 = j_0 \wedge x_0 \neq y_0}_{\psi(i_0, j_0, x_0, y_0)}$$

We derive the inconsistency of constraints like this by linear combination of the equations, as shown in Figure 3.2, forming the unsatisfiable consequence $0 \neq 0$. For the given partitioning of the constraint into $\phi(i_0, j_0, x_0, y_0), \psi(i_0, j_0, x_0, y_0)$, an interpolant can be computed by projecting this linear combination to the equations originating from the left partition:

$$I(i_0, j_0, x_0, y_0) \equiv -1 \cdot (x_0 - i_0 = 0) + 1 \cdot (y_0 - j_0 = 0) \equiv (y_0 - x_0 + i_0 - j_0 = 0)$$

The resulting predicate, $i_0 - j_0 = x_0 - y_0$, enables Eldarica to refine the abstract reachability tree and construct an inductive invariant for the loop in the example program, proving its safety.

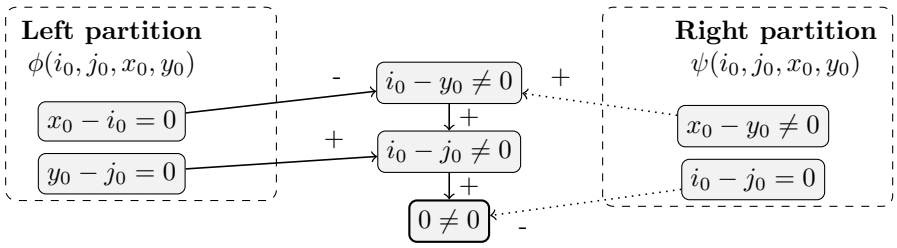


Figure 3.2: Proof about path constraints. All atoms are normalized to have right-hand side 0.

The paper [BKRW11] provides an elaborate discussion on interpolation in the Presburger arithmetic theory.

```
1  input CFG G = ⟨x, Q, q^init, F, E, →⟩
2  output ART T = ⟨S, π, Root, e⟩
3  WorkList = [], S, π, e = ∅, Root = nil
4  def ConstructART(q^init, initialAbstraction) {
5    node = newARTnode(q^init, initialAbstraction)
6    if (Root = nil) Root = node
7    WorkList.add(⟨q^init, node⟩)
8    while (!(WorkList.empty)) {
9      ⟨nextCFGvertex, nextARTnode⟩ = WorkList.remove()
10     for (child = children(nextCFGVertex)) {
11       Let R be such that nextCFGvertex --R--> child in G
12       Φ = {p ∈ π(child) | SP(⋀nextARTnode.abstraction, R) ⊢ p}
13       if (⊥ ∈ Φ or (∃ ⟨child, Ψ⟩ ∈ S ∧ Φ ⊢ Ψ))
14         continue
15       node = newARTnode(child, Φ)
16       S = S ∪ {node}
17       e = e ∪ {(nextARTnode, node)}
18       if (child ∈ E and checkRefineError(node))
19         report "ERROR"
20       WorkList.add(⟨child, node⟩)
21       WorkList.removeAll(nodes from WorkList subsumed by node)
22     }
23   }
24 }
```

Figure 3.3: The CEGAR algorithm

3.4 Algorithm for Constructing an ART

A set of predicates $\mathscr{P}$ can define $2^{|\mathscr{P}|}$ abstract states. To compute the abstraction of the program we need to call the theorem prover totally $(2^{|\mathscr{P}|})^2$ times. In the algorithm of this section in order to make a polynomial number of calls we use the idea of Cartesian abstraction [BPR01]. In Cartesian abstraction we ignore the correlation between the predicates and treat each predicate separately. Basically in each node we compute the set of predicates that hold individually. We then compute the conjunction of the valid predicates as an abstraction for the node. The Cartesian abstraction will decrease our accuracy in making the abstraction. We rely on the refinement step to compensate the disregarded accuracy in abstraction by providing inclusive predicates.

Figure 3.3 presents the pseudocode of the CEGAR algorithm. The main procedure is ConstructART which builds an ART for a given CFG and an abstraction of the set of initial values. ConstructART is a worklist algorithm that expands the ART according to a certain exploration strategy (depth-first, breadth-first, etc.) determined by the type of the structure used as a worklist. We assume without loss of generality that the CFG has exactly one initial vertex. The ConstructART procedure starts with q^{init} and expands the tree according to the definition of the ART (lines 11 and 12). New ART nodes are constructed using newARTnode, which receives

a CFG state and a set of predicates as arguments. The algorithm backtracks from expanding the ART when either the current node contains $\bot$ in its set of predicates, or it is subsumed by another node in the ART (line 13). In the algorithm (Fig. 3.3), we denote logical entailment by $\phi \vdash \psi$ in order to avoid confusion. The refinement step is performed by the checkRefineError function. This function returns true if and only if a feasible error trace has been detected; otherwise, further predicates are generated to refine the abstraction.

4 Accelerating Interpolants

> In order to understand recursion, you must understand recursion.
>
> Anonymous

While empirically successful on a variety of domains, abstraction refinement using interpolants suffers from the unpredictability of interpolants computed by provers, which can cause the verification process to diverge and never discover a sufficient set of predicates (even in case such predicates exist). The failure of such a refinement approach manifests in a sequence of predicates that rule out longer and longer counterexamples, but still fail to discover inductive invariants.

Following another direction, researchers have been making continuous progress on techniques for computing the transitive closure of useful classes of relations on integers [BIK10, FL02, Boi99]. These *acceleration* techniques can compute closed form representation of certain classes of loops using Presburger arithmetic.

We present Counterexample-Guided *Accelerated* Abstraction Refinement (CEGAAR), a new algorithm for verifying infinite-state transition systems. CEGAAR combines *interpolation-based predicate discovery* in counterexample-guided predicate abstraction with *acceleration* technique for computing the transitive closure of loops. CEGAAR applies acceleration to dynamically discovered looping patterns in the unfolding of the transition system, and combines overapproximation with underapproximation. It constructs inductive invariants that rule out an infinite family of spurious counterexamples, alleviating the problem of divergence in predicate abstraction without losing its adaptive nature.

A key contribution of this chapter is an algorithmic solution to apply these specialized analyses for particular classes of loops to rule out an infinite family of counterexamples during predicate abstraction refinement. An essential ingredient of this approach are interpolants that not only rule out one path, but are also *inductive* with respect to loops along this path. We observe

that we can start from any interpolant for a path that goes through a loop in the control-flow graph, and apply a postcondition (or, equivalently a weakest precondition) with respect to the transitive closure of the loop (computed using acceleration) to generalize the interpolant and make it inductive. Unlike previous theoretical proposals [CFLZ08], our method treats interpolant generation and transitive closure computation as black boxes: we can start from any interpolant and strengthen it using any loop acceleration. We call the resulting technique Counterexample-Guided *Accelerated* Abstraction Refinement, or CEGAAR for short. Our experience indicates that CEGAAR works well in practice.

We present theoretical and experimental justification for the effectiveness of CEGAAR, showing that inductive interpolants can be computed from classical Craig interpolants and transitive closures of loops. We present an implementation of CEGAAR that verifies integer transition systems. We show that the resulting implementation robustly handles a number of difficult transition systems that cannot be handled using interpolation-based predicate abstraction or acceleration alone.

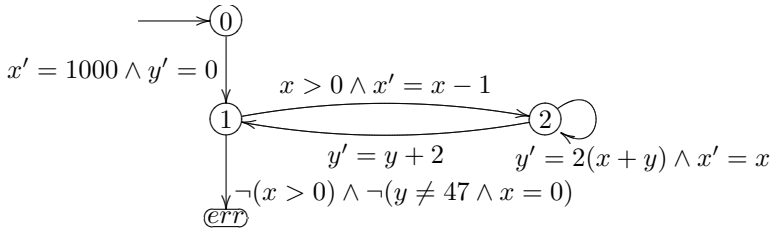
4.1 Motivating Example

To illustrate the power of the technique that we propose, consider the example in Figure 6.1. The example is smaller than the examples we consider in our evaluation (Section 4.6), but already illustrates the difficulty of applying existing methods.

Note that the innermost loop requires a very expressive logic to describe its closed form, so that standard techniques for computing exact transitive closure of loops do not apply. In particular, the acceleration technique does not apply to the innermost loop, and the presence of the innermost loop prevents the application of acceleration to the outer loop. On the other hand, predicate abstraction with interpolation refinement also fails to solve this example. Namely, it enters a very long refinement loop, considering increasingly longer spurious paths with CFG node sequences of the form $\mathbf{0}(\mathbf{12})^i\mathbf{1e}$, for $0 \leq i < 1000$. The crux of the problem

```
int x,y;
x = 1000; y = 0;
while(x > 0){
  x--;
  while(*) {
    y = 2*(x + y);
  }
  y = y + 2;
}
assert(y != 47 && x
   == 0);
```

(a)



(b)

Figure 4.1: Example Program and its Control Flow Graph with Large Block Encoding

is that the refinement eliminates each of these paths one by one, constructing too specific interpolants.

Our combined CEGAAR approach succeeds in proving the assertion of this program by deriving the loop invariant $y\%2 == 0 \wedge x \geq 0$. Namely, once predicate abstraction considers a path where the CFG node **1** repeats (such as **0121e**), it applies acceleration to this path. CEGAAR then uses the accelerated path to construct an inductive interpolant, which eliminates an infinite family of spurious paths. This provides predicate abstraction with a crucial predicate $y\%2 = 0$, which enables further progress, leading to the discovery of the predicate $x \geq 0$. Together, these predicates allow predicate abstraction to construct the invariant that proves program safety. Note that this particular example focuses on proving the absence of errors, but our experience suggests that CEGAAR can, in many cases, find long counterexamples faster than standard predicate abstraction.

4.2 Preliminaries

The program model that we use in this chapter is the model of Chapter 2. We consider an intraprocedural program with the model $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow \rangle$. A *path* in the program is a sequence $\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$, where $q_1, q_2, \ldots, q_n \in Q$ and $q_i \xrightarrow{R_i} q_{i+1}$ is an edge in G, for each $i = 1, \ldots, n-1$. We assume without loss of generality that all variables in $\mathbf{x} \cup \mathbf{x}'$ appear free in each relation labeling an edge of G[1]. The path θ is said to be a *cycle* if $q_1 = q_n$, and a *trace* if $q_1 = q^{init}$. The path θ is said to be *feasible* if and only if there exist valuations $\nu_1, \ldots, \nu_n : \mathbf{x} \rightarrow \mathbb{Z}$ such that $\nu_i, \nu_{i+1} \models R_i$, for all $i = 1, \ldots, n-1$. A control state is said to be *reachable* in G if it occurs on a feasible trace. We denote the relation $R_1 \circ R_2 \circ \ldots \circ R_{n-1}$ by $\rho(\theta)$ and assume that the set of free variables of $\rho(\theta)$ is $\mathbf{x} \cup \mathbf{x}'$.

To define the algorithm in this Chapter we use concepts from binary relations. The composition of two binary relations $R_1, R_2 \in \mathbb{Z}^n \times \mathbb{Z}^n$ is denoted by $R_1 \circ R_2 = \{(\vec{u}, \vec{v}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \exists \vec{t} \in \mathbb{Z}^n \,.\, (\vec{u}, \vec{t}) \in R_1 \text{ and } (\vec{t}, \vec{v}) \in R_2\}$. Let ϵ be the identity relation $\{(\vec{u}, \vec{u}) \mid \vec{u} \in \mathbb{Z}^n \times \mathbb{Z}^n\}$. We define $R^0 = \epsilon$ and $R^i = R^{i-1} \circ R$, for any $i > 0$. With these notations, $R^+ = \bigcup_{i=1}^{\infty} R^i$ denotes the *transitive closure* of R, and $R^* = R^+ \cup \epsilon$ denotes the *reflexive and transitive closure* of R. We sometimes use the same symbols to denote a relation and its defining formula.

For a set of n-tuples $S \subseteq \mathbb{Z}^n$ and a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, let $sp(S, R) = \{\vec{v} \in \mathbb{Z}^n \mid \exists \vec{u} \in S \,.\, (\vec{u}, \vec{v}) \in R\}$ denote the *strongest postcondition* of S via R, and $wp(S, R) = \{\vec{u} \in \mathbb{Z}^n \mid \forall \vec{v} \,.\, (\vec{u}, \vec{v}) \in R \rightarrow \vec{v} \in S\}$ denote the *weakest precondition* of S with respect to R. We use sp and wp for sets and relations, as well as for logical formulae defining them. The operations sp and wp form a Galois connection: $sp(S, R) \subseteq T \iff S \subseteq wp(T, R)$.

In this chapter we are interested in checking *safety properties*, which can be expressed in terms of control state reachability. Assume I represents the initial condition of the program in q^{init}.

[1] For variables that are not modified by a transition, this can be achieved by introducing an explicit update $x' = x$.

A program is said to be *safe* if and only if $sp(I, \rightarrow^*) \cap E = \emptyset$, or equivalently iff $I \cap wp(E, \rightarrow^*) = \emptyset$. Computing $sp(I, \rightarrow^*)$ is usually referred to as *forward analysis*, whereas computing $wp(E, \rightarrow^*)$ is referred to as *backward analysis*.

4.2.1 Acceleration

The goal of acceleration is, given a relation R in a fragment of integer arithmetic, to compute its reflexive and transitive closure, R^*. In general, defining R^* in a decidable fragment of integer arithmetic is not possible, even when R is definable in a decidable fragment such as, e.g. Presburger arithmetic. There are two fragments of arithmetic in which transitive closures of relations are Presburger definable.

Definition 4.2.1. *An* octagonal relation *is a relation defined by a constraint of the form* $\pm x \pm y \leq c$*, where* x *and* y *range over the set* $\mathbf{x} \cup \mathbf{x}'$*, and* c *is an integer constant.*

The transitive closure of an octagonal relation has been shown to be Presburger definable and effectively computable [BIK10].

Definition 4.2.2. *A* linear affine relation *is a relation of the form* $\mathscr{R}(\vec{x}, \vec{x'}) \equiv C\vec{x} \geq \vec{d} \wedge \vec{x'} = A\vec{x} + \vec{b}$*, where* $A \in \mathbb{Z}^{n \times n}$*,* $C \in \mathbb{Z}^{p \times n}$ *are matrices and* $\vec{b} \in \mathbb{Z}^n$*,* $\vec{d} \in \mathbb{Z}^p$*.* $\mathscr{R}$ *is said to have the* finite monoid property *if and only if the set* $\{A^i \mid i \geq 0\}$ *is finite.*

It is known that the finite monoid condition is decidable [Boi99], and moreover that the transitive closure of a finite monoid affine relation is Presburger definable and effectively computable [FL02, Boi99].

4.3 Interpolation-Based Abstraction Refinement

In Section 3.2 we introduced an interpolation based approach for excluding a spurious counter example. The following is a stronger notion of an interpolant, which ensures generality with respect to an infinite family of counterexamples.

Definition 4.3.1. *([CFLZ08], Def. 2.4) Given a CFG* G*, a* trace scheme *in* G *is a sequence of the following form.*

$$\xi : q_0 \xrightarrow{O_1} \overset{L_1}{\curvearrowright} q_1 \xrightarrow{O_2} \dots \xrightarrow{O_{n-1}} \overset{L_{n-1}}{\curvearrowright} q_{n-1} \xrightarrow{O_n} \overset{L_n}{\curvearrowright} q_n \xrightarrow{O_{n+1}} q_{n+1} \tag{4.1}$$

where q_0 *is the initial state* q^{init} *and:*

- $O_i = \rho(\theta_i)$*, for some non-cyclic paths* θ_i *of* G*, from* q_{i-1} *to* q_i
- $L_i = \bigvee_{j=1}^{k_i} \rho(\lambda_{ij})$*, for some cycles* λ_{ij} *of* G*, from* q_i *to* q_i

Intuitively, a trace scheme represents an infinite regular set of traces in G. The trace scheme is said to be *feasible* if and only if at least one trace of G of the following form is feasible.

$$\theta_1;\lambda_{1i_1}\ldots\lambda_{1i_{j_1}};\theta_2;\ldots;\theta_n;\lambda_{ni_n}\ldots\lambda_{ni_{j_n}};\theta_{n+1}$$

The trace scheme is said to be *bounded* if $k_i = 1$, for all $i = 1,2,\ldots,n$. A bounded trace scheme is a regular language of traces, of the form $\sigma_1\cdot\lambda_1^*\cdot\ldots\cdot\sigma_n\cdot\lambda_n^*\cdot\sigma_{n+1}$, where σ_i are acyclic paths, and λ_i are cycles of G. Note that there is an analogy between the definition of bounded trace schemes and the notion of bounded languages [GS64].

Definition 4.3.2. *([CFLZ08], Def. 2.5) Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow\rangle$ be a CFG and ξ be an infeasible trace scheme of the form (4.1). An* interpolant *for ξ is a sequence of predicates $\langle I_0, I_1, I_2, \ldots, I_n, I_{n+1}\rangle$, with free variables in $\vec{x}$, such that:*

1. $I_0 = \top$ *and* $I_{n+1} = \bot$
2. $sp(I_i, O_{i+1}) \rightarrow I_{i+1}$, *for all* $i = 0,1,\ldots,n$
3. $sp(I_i, L_i) \rightarrow I_i$, *for all* $i = 1,2,\ldots,n$

The main difference with Definition 3.2.1 is the third requirement, namely that each interpolant predicate (except for the first and the last one) must be *inductive* with respect to the corresponding loop relation. It is easy to see that each of the following two sequences are interpolants for ξ, provided that ξ is infeasible (Lemma 2.6 in [CFLZ08]).

$$\langle \top,\ sp(\top, O_1\circ L_1^*),\ \ldots,\ sp(\top, O_1\circ L_1^*\circ O_2\circ\ldots O_n\circ L_n^*)\rangle \tag{4.2}$$

$$\langle wp(\bot, O_1\circ L_1^*\circ O_2\circ\ldots O_n\circ L_n^*),\ \ldots,\ wp(\bot, O_n\circ L_n^*),\ \bot\rangle \tag{4.3}$$

Just as for finite trace interpolants, the existence of an inductive interpolant suffices to prove the infeasibility of the entire trace scheme.

Lemma 4.3.1. *Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow\rangle$ be a CFG and ξ be an infeasible trace scheme of G of the form (4.1). If $T = \langle S, \pi, r, e\rangle$ is an ART for G, such that there exists an interpolant $\langle I_i \in \pi(q_i)\rangle_{i=0}^{n+1}$ for ξ, then no path in T concretizes to a trace in ξ.*

Proof. By contradiction, suppose that there exists a path σ:

$$\langle q_0,\Phi_0\rangle,\ \langle q_{11},\Phi_{11}\rangle,\ldots,\langle q_{1i_1},\Phi_{1i_1}\rangle,\ \ldots,\ \langle q_{n1},\Phi_{n1}\rangle,\ldots,\langle q_{ni_n},\Phi_{ni_n}\rangle,\ \langle q_{n+1},\Phi_{n+1}\rangle\rangle$$

in T which concretizes to a trace in ξ. In analogy with the proof of Lemma 3.2.1, one shows that:

- $I_0 \in \Phi_0$
- $I_k \in \Phi_{kj}$, for all $k = 1,\ldots,n$ and $j = 1,\ldots,i_k$
- $I_{n+1} \in \Phi_{n+1}$

The third condition of Definition 4.3.2 is needed for the proof of the second point above. Since $I_{n+1} = \bot$, this contradicts the fact that no node in T may contain $\bot$ in its second component. □

4.4 Counterexample-Guided Accelerated Abstraction Refinement

This section presents the CEGAAR algorithm for predicate abstraction with interpolant-based accelerated abstraction refinement. Since computing the interpolant of a trace scheme is typically more expensive than computing the interpolant of a finite counterexample, we apply acceleration in a demand-driven fashion. The main idea of the algorithm is to accelerate only those counterexamples in which some cycle repeats a certain number of times. For example, if the abstract state exploration has already ruled out the spurious counterexamples $\sigma \cdot \tau$, $\sigma \cdot \lambda \cdot \tau$ and $\sigma \cdot \lambda \cdot \lambda \cdot \tau$, when it sees next the spurious counterexample $\sigma \cdot \lambda \cdot \lambda \cdot \lambda \cdot \tau$, it will accelerate it into $\sigma \cdot \lambda^* \cdot \tau$, and rule out all traces which comply to this scheme. The maximum number of cycles that are allowed to occur in the acyclic part of an error trace, before computing the transitive closure, is called the *delay*, and is a parameter of the algorithm (here the delay was 2). A smaller delay results in a more aggressive acceleration strategy, whereas setting the delay to infinity is equivalent to performing predicate abstraction without acceleration.

The procedure for predicate abstraction is the same as ConstructART in Figure 3.3. The main difference lies in how checkRefineError function refines the abstraction. Figure 4.2 shows the pseudo-code of the checkRefineError function. This function returns true if and only if a feasible error trace has been detected; otherwise, further predicates are generated to refine the abstraction. Figure 4.4 shows the pseudo code of the interpolateRefine function that is responsible for refining abstractions. The checkRefineError function determines a minimal infeasible ART path to $node$ in line 4. This path is generalized into a trace scheme (line 6). The generalization function Fold takes $Path$ and the delay parameter δ as input and produces a trace scheme which contains $Path$. The Fold function creates a trace scheme of the form (4.1) out of the spurious path given as argument. The spurious path is traversed and control states are recorded in a list. When we encounter a control state which is already in the list, we have identified an elementary cycle λ. If the current trace scheme ends with at least δ occurrences of λ, where $\delta \in \mathbb{N}_\infty$ is the delay parameter, then λ is added as a loop to the trace scheme, provided that its transitive closure can be effectivelly computed. For efficiency reasons, we syntactically check the relation on the loop, namely we check whether the relation

is syntactically compliant with the definition of octagonal relations. Notice that a relation can be definable by an octagonal constraint even if it is not a conjunction of octagonal constraints, i.e. it may contain redundant atomic propositions which are not of this form.

Once the folded trace scheme is obtained, there are three possibilities:

1. If the trace scheme is not bounded (the test on line 7 passes), we compute a bounded overapproximation of it, in an attempt to prove its infeasibility (line 8). We describe the computation of bounded overapproximation of trace schemes in Section 4.5.2. If the test on line 9 succeeds, the original trace scheme is proved to be infeasible and the ART is refined using the interpolants for the overapproximated trace scheme.

2. Else, if the overapproximation was found to be feasible, it could be the case that the abstraction of the scheme introduced a spurious error trace. In this case, we compute a bounded underapproximation of the trace scheme, which contains the initial infeasible path, and replace the current trace scheme with it (line 10). We describe the computation of bounded underapproximation of trace schemes in Section 4.5.3. The only requirement we impose on the Underapprox function is that the returned bounded trace scheme contains $Path$, and is a subset of $newScheme$.

3. Finally, if the trace scheme is bounded (either because the test on line 7 failed, or because the folded path was replaced by a bounded underapproximation on line 10) and also infeasible (the test on line 13 passes) then the ART is refined with the interpolants computed for the scheme. If, on the other hand, the scheme is feasible, we continue searching for an infeasible trace scheme starting from the head of $Path$ upwards (line 14).

Example Let $\theta : q_1 \xrightarrow{P} q_2 \xrightarrow{Q} q_2 \xrightarrow{R} q_1 \xrightarrow{P} q_2 \xrightarrow{R} q_1$ be a path. The result of applying Fold to this path is the trace scheme ξ shown in the left half of Fig. 4.3. Notice that this path scheme is not bounded, due to the presence of two loops starting and ending with q_2. A possible bounded underapproximation of ξ, containing the original path θ, is shown in the right half of Fig. 4.3. □

The iteration stops either when a refinement is possible (lines 9,13), in which case checkRefineError returns false, or when the search reaches the root of the ART and the trace scheme is feasible, in which case checkRefineError returns true (line 16) and the main algorithm in Figure 3.3 reports a true counterexample. Notice that, since we update $node$ to the head of $Path$ (line 14), the position of $node$ is moved upwards in the ART. Since this cannot happen indefinitely, the main loop (lines 3-15) of the checkRefineError is bound to terminate.

The interpolateRefine function is used to compute the interpolant of the trace scheme, update the predicate mapping π of the ART, and reconstruct the subtree of the ART whose root is the first node on $Path$ (this is usually called the *pivot* node). The InterpolateRefine (Fig. 4.4)

```
1  def checkRefineError(node): Boolean {
2    traceScheme = []
3    while (the ART path Root → ··· → node is spurious) {
4      Let Path = ⟨q1,Φ1⟩ → ... → ⟨qn,Φn⟩ be the (unique) minimal ART path with
5      pivot = ⟨q1,Φ1⟩ and ⟨qn,Φn⟩ = node such that the CFG path q1 → ··· → qn is infeasible
6      newScheme =Fold(Path,delay)
7      if (!isBounded(newScheme)) {
8        absScheme =Concat(Overapprox(newScheme),traceScheme)
9        if ( interpolateRefine (absScheme,pivot)) return false
10         else newScheme =Underapprox(newScheme,Path)
11     }
12     traceScheme=Concat(newScheme,traceScheme)
13     if ( interpolateRefine (traceScheme,pivot)) return false
14     node = Path.head
15   }
16   return true
17 }
```

Figure 4.2: The CEGAAR algorithm - Accelerated Refinement

function returns true if and only if its argument represents an infeasible trace scheme. In this case, new predicates, obtained from the interpolant of the trace scheme, are added to the nodes of the ART. This function uses internally the TransitiveClosure procedure (line 2) in order to generate the meta-trace scheme (4.5). The AccelerateInterpolant function (line 5) computes the interpolant for the trace scheme, from the resulting meta-trace scheme. Notice that the refinement algorithm is recursive, as ConstructART calls checkRefineError (line 18), which in turn calls InterpolateRefine (lines 9,13), which calls back ConstructART (line 10). Our procedure is *sound,* in the sense that whenever function ConstructART terminates with a non-error result, the input program does not contain any reachable error states. Vice versa, if a program contains a reachable error state, ConstructART is guaranteed to eventually discover a feasible path to this state, since the use of a work list ensures fairness when exploring ARTs.

4.5 Computing Accelerated Interpolants

This section describes a method of refining an ART by excluding an infinite family of infeasible traces at once. Our method combines interpolation with acceleration in a way which is

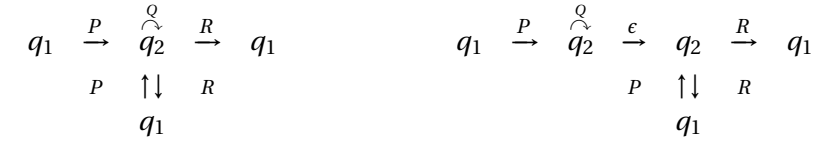


Figure 4.3: Underapproximation of unbounded trace schemes. ϵ stands for the identity relation.

```
1  def InterpolateRefine (traceScheme, Pivot) : Boolean {
2    metaTrace = TransitiveClosure(traceScheme)
3    interpolant = InterpolatingProverCall (metaTrace)
4    if ( interpolant = ∅) return false
5    I = AccelerateInterpolant ( interpolant )
6    for (ψ ∈ I) {
7      let ν be the CFG vertex corresponding to ψ
8      π = π[ν ← (π(ν) ∪ ψ)]
9    }
10   ConstructART(Pivot,Pivot.abstraction )
11   return true
12 }
```

Figure 4.4: The Interpolation Function

oblivious of the particular method used to compute interpolants. For instance, it is possible to combine proof-based [McM05b] or constraint-based [RSS07] interpolation with acceleration, whenever computing the precise transitive closure of a loop is possible. In cases when the precise computation fails, we may resort to both over- and under-approximation of the transitive closure. In both cases, the accelerated interpolants are at least as general (and many times more general) than the classical interpolants extracted from a finite counterexample trace.

4.5.1 Precise Acceleration of Bounded Trace Schemes

We consider first the case of bounded trace schemes of the form (4.1), where the control states $q_1, \ldots, q_n$ belong to some cycles labeled with relations $L_1, \ldots, L_n$. Under some restrictions on the syntax of the relations labeling the cycles L_i, the reflexive transitive closures L_i^* are effectively computable using acceleration algorithms [Boi99, FL02, BHI+09]. Among the known classes of relations for which acceleration is possible we consider: *octagonal relations* and *finite monoid affine transformations*. These are all conjunctive linear relations. We consider in the following that all cycle relations L_i belong to one of these classes. Under this restriction, any infeasible bounded trace scheme has an effectivelly computable interpolant of one of the forms (4.2),(4.3).

However, there are two problems with applying definitions (4.2),(4.3) in order to obtain interpolants of trace schemes. On one hand, relational composition typically requires expensive quantifier eliminations. The standard proof-based interpolation techniques (e.g. [McM05b]) overcome this problem by extracting the interpolants directly from the proof of infeasibility of the trace. Alternatively, constraint-based interpolation [RSS07] reduce the interpolant computation to a Linear Programming problem, which can be solved by efficient algorithms. Both methods apply, however, only to finite traces, and not to infinite sets of traces defined as trace schemes. Another, more important, problem is related to the sizes of the interpolant predicates from (4.2), (4.3) compared to the sizes of interpolant predicates obtained by proof-theoretic

methods (e.g. [KLR10]), as the following example shows.

Example Let $R(x,y,x',y') : x' = x+1 \wedge y' = y+1$ and $\phi(x,y,\ldots)$, $\psi(x,y,\ldots)$ be some complex Presburger arithmetic formulae. The trace scheme:

$$q_0 \xrightarrow{z=0\wedge z'=z\wedge\phi} q_1 \overset{z'=z+2\wedge R}{\curvearrowright} \xrightarrow{z=5\wedge\psi} q_2 \tag{4.4}$$

is infeasible, because z remains even, so it cannot become equal 5. One simple interpolant for this trace scheme has at program point q_1 the formula $z\%2 = 0$. On the other hand, the strongest interpolant has $(z = 0 \wedge z' = x \wedge \phi) \circ (z' = z+2 \wedge R)^*$ at q_1, which is typically a much larger formula, because of the complex formula ϕ. Note however that ϕ and R do not mention z, so they are irrelevant. □

We can also describe the difference between obtaining interpolants by quantifier elimination and by the standard proof-based approaches geometrically. Consider a simple case of unsatisfiability where the conjunction of two convex polyhedra P and Q is unsatisfiable. The quantifier elimination approach comes up with the polyhedron P as the interpolant. In fact any hyperplane containing P that has empty intersection with Q is a legitimate interpolant. The strongest or weakest interpolants obtained by quantifier elimination are usually too complex to be useful in practice.

To construct useful interpolants instead of the strongest or the weakest ones, we therefore proceed as follows. Let ξ be a bounded trace scheme of the form (4.1). For each control loop $q_i \xrightarrow{R_i} q_i$ of ξ, we define the corresponding *meta-transition* $q_i' \xrightarrow{R_i^*} q_i''$ labeled with the reflexive and transitive closure of R_i. Intuitively, firing the meta-transition has the same effect as iterating the loop an arbitrary number of times. We first replace each loop of ξ by the corresponding meta-transition. The result is the *meta-trace*:

$$\overline{\xi} : q_0 \xrightarrow{O_1} q_1' \xrightarrow{L_1^*} q_1'' \xrightarrow{O_2} q_2' \ \ldots\ q_{n-1}'' \xrightarrow{O_n} q_n' \xrightarrow{L_n^*} q_n'' \xrightarrow{O_{n+1}} q_{n+1} \tag{4.5}$$

Since we supposed that ξ is an infeasible trace scheme, the (equivalent) finite meta-trace $\overline{\xi}$ is infeasible as well, and it has an interpolant $\mathscr{I}_{\overline{\xi}} = \langle \top, I_1', I_1'', I_2', I_2'', \ldots, I_n', I_n'', \bot \rangle$ in the sense of Definition 3.2.1. This interpolant is not an interpolant of the trace scheme ξ, in the sense of Definition 4.3.2. In particular, none of I_i', I_i'' is guaranteed to be inductive with respect to the loop relations L_i. To define compact inductive interpolants based on $\mathscr{I}_{\overline{\xi}}$ and the transitive closures L_i^*, we consider the following sequences:

$$\begin{aligned} \mathscr{I}_\xi^{sp} &= \langle \top, sp(I_1', L_1^*), sp(I_2', L_2^*), \ldots, sp(I_n', L_n^*), \bot \rangle \\ \mathscr{I}_\xi^{wp} &= \langle \top, wp(I_1'', L_1^*), wp(I_2'', L_2^*), \ldots, wp(I_n'', L_n^*), \bot \rangle \end{aligned}$$

The following lemma proves the correctness of this approach.

Lemma 4.5.1. *Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow \rangle$ be a CFG and ξ be an infeasible trace scheme of the form (4.1). Then $\mathscr{I}_\xi^{sp}$ and $\mathscr{I}_\xi^{wp}$ are interpolants for ξ, and moreover $\mathscr{I}_{\xi_i}^{wp} \rightarrow \mathscr{I}_{\xi_i}^{sp}$, for all $i = 1, 2, \ldots, n$.*

Proof. To prove that $\mathscr{I}_\xi^{sp}$ is an interpolant for ξ, we show the three points of Definition 4.3.2. The first point holds by the construction of $\mathscr{I}_\xi^{sp}$. For the second point, we have to show $sp(\mathscr{I}_{\xi_i}^{sp}, O_{i+1}) \rightarrow \mathscr{I}_{\xi_{i+1}}^{sp}$.

$$\begin{array}{rcll} sp(I_i', L_i^*) & \rightarrow & I_i'' & \text{, since } \mathscr{I}_{\overline{\xi}} \text{ is an interpolant for } \overline{\xi} \\ sp(sp(I_i', L_i^*), O_{i+1}) & \rightarrow & sp(I_i'', O_{i+1}) & \text{, since } sp \text{ is monotone} \\ sp(\mathscr{I}_{\xi_i}^{sp}, O_{i+1}) & \rightarrow & I_{i+1}' & \text{, since } \mathscr{I}_{\overline{\xi}} \text{ is an interpolant for } \overline{\xi} \end{array}$$

We must show next that $I_{i+1}' \rightarrow \mathscr{I}_{\xi_{i+1}}^{sp}$. For this, we compute:

$$\begin{array}{rcl} sp(I_{i+1}', L_{i+1}^*) & = & \exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge L_{i+1}^*(\vec{z}, \vec{x}) \\ & = & \exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge \bigvee_{k=0}^{\infty} L_{i+1}^k(\vec{z}, \vec{x}) \\ & = & \bigvee_{k=0}^{\infty} \exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge L_{i+1}^k(\vec{z}, \vec{x}) \\ & = & \exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge \epsilon \ \vee \ \bigvee_{k=1}^{\infty} \exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge L_{i+1}^k(\vec{z}, \vec{x}) \end{array}$$

We have that $\exists \vec{z} \,.\, I_{i+1}'(\vec{z}) \wedge \epsilon$ is equivalent to I_{i+1}', which concludes the second point. For the third point, we compute:

$$\begin{array}{rcl} sp(\mathscr{I}_{\xi_i}^{sp}, L_i) & = & \exists \vec{z} \,.\, sp(I_i', L_i^*)(\vec{z}) \wedge L_i(\vec{z}, \vec{x}) \\ & = & \exists \vec{z} \exists \vec{t} \,.\, I_i'(\vec{t}) \wedge L_i^*(\vec{t}, \vec{z}) \wedge L_i(\vec{z}, \vec{x}) \\ & = & \exists \vec{t} \,.\, I_i'(\vec{t}) \wedge L_i^+(\vec{t}, \vec{x}) \\ & \rightarrow & \exists \vec{t} \,.\, I_i'(\vec{t}) \wedge L_i^*(\vec{t}, \vec{x}) \\ & = & sp(I_i', L_i^*) = \mathscr{I}_{\xi_i}^{sp} \end{array}$$

The proof for the $\mathscr{I}_\xi^{wp}$ interpolant is symmetric, using the fact that sp and wp form a Galois connection. Finally, we have $wp(I_i'', L_i^*) \rightarrow I_i' \rightarrow sp(I_i', L_i^*)$ which proves the last statement.

□

Notice that computing $\mathscr{I}_\xi^{sp}$ and $\mathscr{I}_\xi^{wp}$ requires n relational compositions, which is, in principle, just as expensive as computing directly one of the extremal interpolants (4.2),(4.3). However, by re-using the meta-trace interpolants, one potentially avoids the worst-case combinatorial explosion in the size of the formulae, which occurs when using (4.2), (4.3) directly.

Example Let us consider again the trace scheme (4.4). The corresponding infeasible finite trace $\overline{\xi}$ is:

$$q_0 \xrightarrow{z=0\wedge z'=z\wedge\phi} q_1' \xrightarrow{\exists k\geq 0\ .\ z'=z+2k\ \wedge\ x'=x+k\ \wedge\ y'=y+k} q_1'' \xrightarrow{z=5\wedge\psi} q_2$$

A possible interpolant for this trace is $\langle \top, z=0, \exists k\geq 0\ .\ z=2k, \bot\rangle$. An inductive interpolant for the trace scheme, derived from it, is $\mathscr{I}^{sp}_{\xi} = \langle \top, sp(z=0, \exists k\geq 0. z'=z+2k\wedge x'=x+k\wedge y'=y+k), \bot\rangle = \langle \top,\ z\%2=0,\ \bot\rangle$. □

4.5.2 Bounded Overapproximations of Trace Schemes

Consider a trace scheme (4.1), not necessarily bounded, where the transitive closures of the relations L_i labeling the loops are not computable by any available acceleration method [BHI+09, Boi99, FL02]. One alternative is to find abstractions $L_i^\sharp$ of the loop relations, i.e. relations $L_i^\sharp \leftarrow L_i$, for which transitive closures are computable. If the new abstract trace remains infeasible, it is possible to compute an interpolant for it, which is an interpolant for the original trace scheme. However, replacing the relations L_i with their abstractions $L_i^\sharp$ may turn an infeasible trace scheme into a feasible one, where the traces introduced by abstraction are spurious. In this case, we give up the overapproximation, and turn to the underapproximation technique described in the next section.

The overapproximation method computes an interpolant for a trace scheme ξ of the form (4.1) under the assumption that the abstract trace scheme:

$$\xi^\sharp : q_0 \xrightarrow{O_1} \overset{L_1^\sharp}{\curvearrowright} q_1 \xrightarrow{O_2} \dots \xrightarrow{O_{n-1}} \overset{L_{n-1}^\sharp}{\curvearrowright} q_{n-1} \xrightarrow{O_n} \overset{L_n^\sharp}{\curvearrowright} q_n \xrightarrow{O_{n+1}} q_{n+1} \tag{4.6}$$

is infeasible. In this case one can effectivelly compute the interpolants $\mathscr{I}^{sp}_{\xi^\sharp}$ and $\mathscr{I}^{wp}_{\xi^\sharp}$, since the transitive closures of the abstract relations labeling the loops are computable by acceleration. The following lemma proves that, under certain conditions, computing an interpolant for the abstraction of a trace scheme is sound.

Lemma 4.5.2. *Let G be a CFG and ξ be a trace scheme (4.1) such that the abstract trace scheme $\xi^\sharp$ (4.6) is infeasible. Then the interpolants $\mathscr{I}^{sp}_{\xi^\sharp}$ and $\mathscr{I}^{wp}_{\xi^\sharp}$ for $\xi^\sharp$ are also interpolants for ξ.*

Proof. We show that $\mathscr{I}^{sp}_{\xi^\sharp}$ meets the three conditions of Definition 4.3.2. The first condition is trivially true, while the proof of the second condition is essentially the same as in the proof of Lemma 4.5.1. For the third point, since $L_i \rightarrow L_i^\sharp$, we have:

$$\begin{aligned} sp(\mathscr{I}^{sp}_{\xi^\sharp}, L_i) &= sp(sp(I_i', {L_i^\sharp}^*), L_i) \\ &\rightarrow sp(sp(I_i', {L_i^\sharp}^*), L_i^\sharp) \\ &= sp(I_i', {L_i^\sharp}^+) \rightarrow \mathscr{I}^{sp}_{\xi^\sharp} \end{aligned}$$

The proof for $\mathscr{I}^{wp}_{\xi^\sharp}$ is symmetrical. □

4.5.3 Bounded Underapproximations of Trace Schemes

Let ξ be a trace scheme of the form (4.1), where each relation L_i labeling a loop is a disjunction $L_{i1} \vee \ldots \vee L_{ik_i}$ of relations for which the transitive closures are effectively computable and Presburger definable. A *bounded underapproximation scheme* of a trace scheme ξ is obtained by replacing each loop $q_i \xrightarrow{L_i} q_i$ in ξ by a bounded trace scheme of the form:

$$\overset{L_{i1}}{\curvearrowright} \quad \overset{L_{i2}}{\curvearrowright} \quad \overset{L_{ik_i}}{\curvearrowright}$$
$$q_i^1 \xrightarrow{\epsilon} q_i^2 \xrightarrow{\epsilon} \ldots q_i^{k_i}$$

where ϵ denotes the identity relation. Let us denote[2] the result of this replacement by $\xi^\flat$. It is manifest that the set of traces $\xi^\flat$ is included in ξ.

Since we assumed that the reflexive and transitive closures L_{ij}^* are effectivelly computable and Presburger definable, the feasibility of $\xi^\flat$ is a decidable problem. If $\xi^\flat$ is found to be feasible, this points to a real error trace in the system. On the other hand, if $\xi^\flat$ is found to be infeasible, let $\mathscr{I}_{\xi^\flat} = \langle \top, I_1^1, \ldots, I_1^{k_1}, \ldots, I_n^1, \ldots, I_n^{k_n}, \bot \rangle$ be an interpolant for $\xi^\flat$. A refinement scheme using this interpolant associates the predicates $\{I_i^1, \ldots, I_i^{k_i}\}$ with the control state q_i from the original CFG. As the following lemma shows, this guarantees that any trace that follows the pattern of $\xi^\flat$ is excluded from the ART, ensuring that a refinement of the ART using a suitable underapproximation (that includes a spurious counterexample) is guaranteed to make progress.

Lemma 4.5.3. *Let $G = \langle \mathbf{x}, Q, q^{init}, F, E, \rightarrow \rangle$ be a CFG, ξ be an infeasible trace scheme of G (4.1) and $\xi^\flat$ a bounded underapproximation of ξ. If $T = \langle S, \pi, r, e \rangle$ is an ART for G, such that $\{I_i^1, \ldots, I_i^{k_i}\} \subseteq \pi(q_i)$, then no path in T concretizes to a trace in $\xi^\flat$.*

Proof. By contradiction, suppose that there exists a path in T which concretizes to a trace in $\xi^\flat$, and let

$$\underbrace{\langle q_i, \Phi_{i1}^1 \rangle, \ldots, \langle q_i, \Phi_{i\ell_{i,1}}^1 \rangle}_{\overset{L_{i1}}{\curvearrowright} \; q_i}, \ldots, \underbrace{\langle q_i, \Phi_{i1}^{k_i} \rangle, \ldots, \langle q_i, \Phi_{i\ell_{i,k_i}}^{k_i} \rangle}_{\overset{L_{ik_i}}{\curvearrowright} \; q_i}$$

be the fragment of the path which corresponds to the unfolding of the sub-trace:

$$\overset{L_{i1}}{\curvearrowright} \quad \overset{L_{i2}}{\curvearrowright} \quad \overset{L_{ik_i}}{\curvearrowright}$$
$$q_i \xrightarrow{\epsilon} q_i \xrightarrow{\epsilon} \ldots q_i$$

One can show, among the lines of the proof of Lemma 4.5.1, that $I_i^j \in \Phi_{i\ell}^j$, for all $j = 1, \ldots, k_i$

[2] The choice of the name depends on the ordering of particular paths $L_{i1}, L_{i2}, \ldots, L_{ik_i}$, however we shall denote any such choice in the same way, in order to keep the notation simple.

and $\ell = 1, \ldots, \ell_{i,j}$. In this way, we obtain that the last set Φ contains $\bot$, which contradicts the definition of the ART. □

Notice that a refinement scheme based on underapproximation guarantees the exclusion of those traces from the chosen underapproximation trace scheme, and not of all traces from the original trace scheme. Since a trace scheme is typically obtained from a finite counterexample, an underapproximation-based refinement still guarantees that the particular counterexample is excluded from further searches. In other words, using underapproximation is still better than the classical refinement method, since it can potentially exclude an entire family of counterexamples (including the one generating the underapproximation) at once.

4.6 Experimental Results

We have implemented CEGAAR by building on the predicate abstraction engine Eldarica[3] [HKG+12], the FLATA verifier[4] [HKG+12] based on acceleration, and the Princess interpolating theorem prover [BKRW11, Rüm08]. Tables in Figure 4.5 compares the performance of the Flata, Eldarica, *static acceleration* and CEGAAR on a number of benchmarks (the platorm used for experiments is Intel® Core™2 Duo CPU P8700, 2.53GHz with 4GB of RAM).

The benchmarks are all in the Numerical Transition Systems format[5] (NTS). We have considered seven sets of examples, extracted automatically from different sources.

(a) C programs with arrays provided as examples of divergence in predicate abstraction [JM06]. We have used the tool FLATA-C[6] to extract the NTS models for these programs. For the array operations FLATA-C basically check that they happen within the array bounds.

(b) verification conditions for programs with arrays, expressed in the SIL logic of [BHI+09] and translated to NTS.

(c) small C programs with challenging loops.

(d) NTS extracted from programs with singly-linked lists by the L2CA tool. [BBH+06]

(e) C programs provided as benchmarks in the NECLA static analysis suite

(f) C programs with asynchronous procedure calls translated into NTS using the approach of [GM12] (the examples with extension .optim are obtained via an optimized translation method [Gan]

(g) models extracted from VHDL models of circuits following the method of [SV07].

[3] http://lara.epfl.ch/w/eldarica
[4] http://www-verimag.imag.fr/FLATA.html
[5] http://richmodels.epfl.ch/ntscomp_ntslib
[6] http://www-verimag.imag.fr/FLATA-C.html

The benchmarks are available from the home page of our tool. The results on this benchmark set suggest that we have arrived at a fully automated verifier that is robust in verifying automatically generated integer programs with a variety of looping control structure patterns. An important question we explored is the importance of dynamic application of acceleration, as well as of overapproximation and underapproximation. We therefore also implemented static acceleration [CFLZ08], a lightweight acceleration technique generalizing large block encoding (LBE) [BCG+09] with transitive closures. It simplifies the control flow graph prior

Model	Time [s] F.	E.	S.	D.
(a) Examples from [JM06]				
anubhav (C)	0.8	3.0	4.0	3.1
copy1 (E)	2.0	7.2	5.8	5.9
cousot (C)	0.6	-	6.2	5.9
loop1 (E)	1.7	7.1	5.2	5.4
loop (E)	1.8	5.9	4.8	5.4
scan (E)	3.3	-	5.1	5.0
string_concat1 (E)	5.3	-	10.1	7.3
string_concat (E)	4.9	-	7.0	7.5
string_copy (E)	4.6	-	6.3	5.7
substring1 (E)	0.6	9.4	18.2	8.3
substring (E)	2.1	3.3	6.3	3.5
(b) Verification conditions for array programs [BHI+09]				
rotation_vc.1 (C)	0.6	2.0	9.5	2.0
rotation_vc.2 (C)	1.6	2.2	18.5	2.2
rotation_vc.3 (C)	1.2	0.3	18.3	0.3
rotation_vc.1 (E)	1.1	1.3	10.2	1.3
split_vc.1 (C)	3.9	3.7	91.1	3.6
split_vc.2 (C)	3.0	2.3	74.1	2.2
split_vc.3 (C)	3.3	0.6	75.0	0.6
split_vc.1 (E)	28.5	2.3	185.6	2.4
(c) Examples from L2CA [BBH+06]				
bubblesort (E)	14.9	9.9	9.5	9.3
insdel (E)	0.1	1.3	2.5	1.4
insertsort (E)	2.0	4.2	5.0	4.0
listcounter (C)	0.3	-	1.9	3.7
listcounter (E)	0.3	1.4	1.6	1.4
listreversal (C)	4.5	3.0	6.0	3.3
listreversal (E)	0.8	2.7	8.1	2.8
mergesort (E)	1.2	7.7	21.3	7.4
selectionsort (E)	1.5	8.1	13.7	7.7

Model	Time [s] F.	E.	S.	D.
(d) Examples from [Mon]				
boustrophedon (C)	-	-	-	14.4
gopan (C)	0.4	-	-	6.4
halbwachs (C)	-	-	7.3	7.0
rate_limiter (C)	31.7	6.1	8.1	5.5
(e) NECLA benchmarks				
inf1 (E)	0.2	2.0	2.0	2.0
inf4 (E)	0.9	3.7	3.7	3.7
inf6 (C)	0.1	2.0	2.0	2.0
inf8 (C)	0.3	3.6	3.4	3.9
(f) VHDL models from [SV07]				
counter (C)	0.1	1.6	1.6	1.6
register (C)	0.2	1.1	1.1	1.1
synlifo (C)	16.6	22.1	21.4	22.0
(g) Examples from [GM12]				
h1 (E)	-	5.1	5.6	5.1
h1.optim (E)	0.8	2.9	5.5	2.9
h1h2 (E)	-	9.4	10.1	12.2
h1h2.optim (E)	1.1	3.3	4.4	3.4
simple (E)	-	6.4	7.0	8.4
simple.optim (E)	0.8	3.0	5.1	2.9
test0 (C)	-	23.0	23.4	29.2
test0.optim (C)	0.3	3.2	5.4	3.2
test0 (E)	-	5.4	5.9	5.7
test0.optim (E)	0.6	3.0	5.8	2.9
test1.optim (C)	0.9	4.7	5.9	7.8
test1.optim (E)	1.5	4.4	5.9	4.7
test2_1.optim (E)	1.6	5.2	5.5	5.6
test2_2.optim (E)	2.9	4.6	5.9	4.6
test2.optim (C)	6.4	27.2	30.1	30.0
wrpc.manual (C)	0.6	1.2	1.4	1.2
wrpc (E)	-	7.9	8.4	8.2
wrpc.optim (E)	-	5.1	8.5	5.2

Figure 4.5: Benchmarks for **F**lata, **E**ldarica without acceleration, Eldarica with acceleration of loops at the CFG level (**S**tatic) and CEGAAR (**D**ynamic acceleration). The letter after the model name distinguishes **C**orrect from models with a reachable **E**rror state. Items with "-" led to a timeout for the respective approach.

to predicate abstraction. In some cases, such as mergesort from the (d) benchmarks and split_ vc.1 from (b) benchmarks, the acceleration overhead does not pay off. The problem is that static acceleration tries to accelerate every loop in the CFG rather than accelerating the loops occurring on spurious paths leading to error. Acceleration of inessential loops generates large formulas as the result of combining loops and composition of paths during large block encoding. The CEGAAR algorithm is the only approach that could handle all of our benchmarks. There are cases in which the Flata tool outperforms CEGAAR such as test2.optim from (f) benchmarks. We attribute this deficiency to the nature of predicate abstraction, which tries to discover the required predicates by several steps of refinement. In the verification of benchmarks, acceleration was exact 11 times in total. In 30 case the over-approximation of the loops was successful, and in 15 cases over-approximation failed, so the tool resorted to under-approximation. This suggests that all techniques that we presented are essential to obtain an effective verifier.

5 Interpolation and Solving Horn Clauses

> If you want to increase your success rate,
> double your failure rate.
>
> Thomas J. Watson

Software model checking has greatly benefited from the combination of a number of seminal ideas: automated abstraction through theorem proving [GS97], exploration of finite-state abstractions, and counterexample-driven refinement [BPR02]. Even though these techniques can be viewed independently, the effectiveness of verification has been consistently improving by providing more sophisticated communication between these steps. Often, carefully chosen search aspects are being pushed into a learning-enabled constraint solver, resulting in better overall verification performance. An essential advance was to use interpolants derived from unsatisfiability proofs to refine the abstraction [HJMM04]. In recent years, we have seen significant progress in interpolating methods for different logical constraints [CGS10, BKRW11, MR13], and a wealth of more general forms of interpolation [HHP10, AGC12a, MR13, RHK13]. This chapter sheds light on computing different types of interpolation queries, going beyond tree interpolants and DAG interpolants towards recursion-free Horn clauses. The use of Horn constraints as intermediate representation has been recently proposed [GPR11a, GLPR12, MR13] as a promising direction to extend the reach of automated verification methods to a variety of areas such as programs with procedures and concurrent programs. We gave a model of a recursive program using Horn clauses in Section 2.3.

We systematically examine binary interpolation, inductive interpolant sequences, tree interpolants, restricted DAG interpolants and show the recursion-free Horn clause problems to which their correspond. We present an algorithm for solving the interpolation problem, relating it to a subclass of recursion-free Horn clauses [PGS98, MLNH07, GPR11a]. We also give a taxonomy of the various interpolation problems, and the corresponding systems of Horn clauses, in terms of their computational complexity. We identify a new notion, *disjunctive interpolants*, which are more general than tree interpolants and inductive sequences

of interpolants. Like tree interpolation [HHP10, MR13], a disjunctive interpolation query is a tree-shaped constraint specifying the interpolants to be derived; however, in disjunctive interpolation, branching in the tree can represent both conjunctions and disjunctions.

We then consider solving general recursion-free Horn clauses and show that this problem is solvable whenever the logic admits interpolation. We establish tight complexity bounds for solving recursion-free Horn clauses for propositional logic (PSPACE) and for integer linear arithmetic (co-NEXPTIME). In contrast, the disjunctive interpolation problem remains in coNP for these logics. We generalize our results from recursion-free Horn clauses to general well-founded constraints, i.e., to constraints without infinite resolution proofs. We also show how to use solvers for recursion-free Horn clauses to verify recursive Horn clauses using counterexample-driven predicate abstraction. We present a library of recursion-free Horn problems, designed for benchmarking Horn solvers and interpolation engines.

We finally devise a predicate abstraction based algorithm for solving general recursive Horn clauses. The refinement step uses interpolation to find new predicates. We have improved our predicate abstraction engine Eldarica [HKG+12] to solve recursive Horn clauses. We apply the algorithm on a set of publicly available benchmarks.

5.1 Example: Verification of Recursive Predicates

We start by showing how our approach can verify programs encoded as Horn clauses, by means of predicate abstraction and a theorem prover for Presburger arithmetic. Fig. 5.1 shows an example of a system of Horn clauses that compute the greatest common divisor of its first and its second argument in its third argument. After invoking the gcd operation on the equal positive numbers M and N, we wish to check whether it is possible for the result R to be more than the M. In general, we encode error conditions as Horn clauses with *false* in their head, and refer to such clauses as error clauses, although such clauses do not have a special semantic status in our system. When executed with these clauses as input, our verification tool automatically identifies that the definition of $\mathsf{gcd}(\mathsf{M},\mathsf{N},\mathsf{R})$ as the predicate $(M = N) \rightarrow (M \geq R)$ gives a solution to these Horn clauses. In terms of safety (partial correctness), this means that the error condition cannot be reached.

Our approach uses counterexample-driven refinement to perform verification. In this example, the abstraction of Horn clauses starts with a trivial set of predicates, containing only the predicate *false*, which is assumed to be a valid approximation until proven otherwise. Upon examining a clause that has a concrete satisfiable formula on the right-hand side (e.g. $M = N \wedge R = M$), we rule out *false* as the approximation of gcd. In the absence of other candidate predicates, the approximation of gcd becomes the conjunction of an empty set of predicates, which is *true*. Using this approximation the error clause is no longer satisfied. At this point the algorithm checks whether a true error is reached by directly chaining the clauses involved in computing the approximation of predicates. This amounts to checking whether the following recursion-free subset of clauses has a solution:

(1) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
(2) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} > \mathsf{N} \wedge \mathsf{M1} = \mathsf{M} - \mathsf{N} \wedge \mathsf{gcd(M1,N,R)}$
(3) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} < \mathsf{N} \wedge \mathsf{N1} = \mathsf{N} - \mathsf{M} \wedge \mathsf{gcd(M,N1,R)}$
(4) $\mathsf{false} \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M} = \mathsf{N} \wedge \mathsf{gcd(M,N,R)} \wedge \mathsf{R} > \mathsf{M}$

Figure 5.1: Horn clauses computing the greatest common divisor of two numbers and an assertion on result. Variables are universally quantified in each clause.

(1) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
(1') $\mathsf{gcd1(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
(2') $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} > \mathsf{N} \wedge \mathsf{M1} = \mathsf{M} - \mathsf{N} \wedge \mathsf{gcd1(M1,N,R)}$
(3') $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} < \mathsf{N} \wedge \mathsf{N1} = \mathsf{N} - \mathsf{M} \wedge \mathsf{gcd1(M,N1,R)}$
(4) $\mathsf{false} \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M} = \mathsf{N} \wedge \mathsf{gcd(M,N,R)} \wedge \mathsf{R} > \mathsf{M}$

Figure 5.2: Extended recursion-free approximation of the Horn clauses in Fig. 5.1.

(1) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
(4) $\mathsf{false} \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M} = \mathsf{N} \wedge \mathsf{gcd(M,N,R)} \wedge \mathsf{R} > \mathsf{M}$

The solution to above problem is any formula $I(M,N,R)$ such that

$\mathsf{I(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
$\mathsf{false} \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M} = \mathsf{N} \wedge \mathsf{I(M,N,R)} \wedge \mathsf{R} > \mathsf{M}$

This is precisely an interpolant of $M = N \wedge R = M$ and $M \geq 0 \wedge M = N \wedge R > M$. A valid interpolant is $P_1(M,N,R) \equiv M \geq R$. Choosing this interpolant eliminates the current contradiction for Horn clauses and P_1 is added into a list of abstraction predicates for the relation gcd. Because the predicates approximating gcd are now updated, we consider the abstraction of the system in terms of these predicates.

The predicate P_1 is not a conjunct in a valid approximation for gcd in clause (2), so the following recursion-free unfolding is not solved by the approximation so far:

(1) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M} = \mathsf{N} \wedge \mathsf{R} = \mathsf{M}$
(2') $\mathsf{gcd1(M,N,R)} \leftarrow \mathsf{M} > \mathsf{N} \wedge \mathsf{M1} = \mathsf{M} - \mathsf{N} \wedge \mathsf{gcd(M1,N,R)}$
(4') $\mathsf{false} \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M} = \mathsf{N} \wedge \mathsf{gcd1(M,N,R)} \wedge \mathsf{R} > \mathsf{M}$

This particular problem could be reduced to solving an interpolation sequence, but it is more natural to think of it simply as a solution for recursion-free Horn clauses. A solution is an interpretation of the relations gcd and gcd1 as ternary relations on integers, such that the clauses are true. Note that this problem could also be viewed as the computation of tree interpolants, which are also a special case of solving recursion-free Horn clauses, as are DAG interpolants and a new notion of disjunctive tree interpolants that we introduce. In line with [GPR11a, GPR11b, GLPR12] we observe that recursion-free clauses are a perfect fit for counterexample-driven verification: they allow us to provide the theorem proving procedure with much more information that they can use to refine abstractions. In the limit, the original set of clauses or its recursive unfoldings are its own approximations, some of them exact, but

the advantage of *recursion-free* Horn clauses is that their solvability is decidable under very general conditions. This provides us with a solid theorem proving building block to construct robust and predictable solvers for the undecidable recursive case. Our paper describes a new such building block: disjunctive interpolants, which correspond to a subclass of non-recursive Horn clauses.

To illustrate disjunctive interpolants, Fig. 5.2 provides another recursion-free approximations of the problem. In this approximation we can distinguish 3 different paths from the error clause (4) through the clauses (1'), (2') and (3') to ground formulae. The traditional refinement approach using e.g. tree interpolation typically removes the 3 instances of the spurious counter-examples using 3 interpolation calls. A novelty of disjunctive interpolation is removing the different choices of counter-examples altogether using a single call to the interpolating theorem prover. Eliminating more counter-examples at once can reduce the number of iterations and increase convergence.

5.2 Formulae and Horn Clauses

Constraint languages. Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set $\mathscr{F}$ of fixed-arity function symbols, and a set $\mathscr{P}$ of fixed-arity predicate symbols. Interpretation of $\mathscr{F}$ and $\mathscr{P}$ is determined by a class $\mathscr{S}$ of structures (U, I) consisting of non-empty universe U, and a mapping I that assigns to each function in $\mathscr{F}$ a set-theoretic function over U, and to each predicate in $\mathscr{P}$ a set-theoretic relation over U. As a convention, we assume the presence of an equation symbol "=" in $\mathscr{P}$, with the usual interpretation. Given a countably infinite set $\mathscr{X}$ of variables, a *constraint language* is a set *Constr* of first-order formulae over $\mathscr{F}, \mathscr{P}, \mathscr{X}$ For example, the language of quantifier-free Presburger arithmetic has $\mathscr{F} = \{+, -, 0, 1, 2, \ldots\}$ and $\mathscr{P} = \{=, \leq, |\}$).

A constraint is called *satisfiable* if it holds for some structure in $\mathscr{S}$ and some assignment of the variables $\mathscr{X}$, otherwise *unsatisfiable*. We say that a set $\Gamma \subseteq$ *Constr* of constraints *entails* a constraint $\phi \in$ *Constr* if every structure and variable assignment that satisfies all constraints in Γ also satisfies ϕ; this is denoted by $\Gamma \models \phi$.

$fv(\phi)$ denotes the set of free variables in constraint ϕ. We write $\phi[x_1, \ldots, x_n]$ to state that a constraint contains (only) the free variables $x_1, \ldots, x_n$, and $\phi[t_1, \ldots, t_n]$ for the result of substituting the terms $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$. Given a constraint ϕ containing the free variables $x_1, \ldots, x_n$, we write $Cl_\forall(\phi)$ for the *universal closure* $\forall x_1, \ldots, x_n.\phi$.

Positions. We denote the set of *positions* in a constraint ϕ by *positions*(ϕ). For instance, the constraint $a \wedge \neg a$ has 4 positions, corresponding to the sub-formulae $a \wedge \neg a, \neg a$, and the two occurrences of a. The sub-formula of a formula ϕ underneath a position p is denoted by $\phi \downarrow p$, and we write $\phi[p/\psi]$ for the result of replacing the sub-formula $\phi \downarrow p$ with ψ. Further, we write $p \leq q$ if position p is above q (that is, q denotes a position within the sub-formula $\phi \downarrow p$), and

$p < q$ if p is strictly above q.

Craig interpolation is the main technique used to construct and refine abstractions in software model checking. A binary interpolation problem is a conjunction $A \wedge B$ of constraints. A *Craig interpolant* is a constraint I such that $A \models I$ and $B \models \neg I$, and such that $fv(I) \subseteq fv(A) \cap fv(B)$. The existence of an interpolant implies that $A \wedge B$ is unsatisfiable. We say that a constraint language has the *interpolation property* if also the opposite holds: whenever $A \wedge B$ is unsatisfiable, there is an interpolant I.

5.2.1 Horn Clauses

To define the concept of Horn clauses, we fix a set $\mathscr{R}$ of uninterpreted fixed-arity *relation symbols,* disjoint from $\mathscr{P}$ and $\mathscr{F}$. A *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$ where

- C is a constraint over $\mathscr{F}, \mathscr{P}, \mathscr{X}$;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathscr{R}$ to first-order terms over $\mathscr{F}, \mathscr{X}$;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in \mathscr{R}$ to first-order terms, or is the constraint *false*.

H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = true$, we usually leave out C and just write $B_1 \wedge \cdots \wedge B_n \rightarrow H$. First-order variables (from $\mathscr{X}$) in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in \mathscr{S}$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

A *relation symbol assignment* is a mapping $sol : \mathscr{R} \rightarrow Constr$ that maps each n-ary relation symbol $p \in \mathscr{R}$ to a constraint $sol(p) = C_p[x_1, \ldots, x_n]$ with n free variables. The *instantiation* $sol(h)$ of a Horn clause h is defined by:

$$\begin{aligned} sol\big(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t})\big) &= C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow sol(p)[\bar{t}] \\ sol\big(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow false\big) &= C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow false \end{aligned}$$

Definition 5.2.1 (Solvability)**.** *Let $\mathscr{HC}$ be a set of Horn clauses over relation symbols $\mathscr{R}$.*

1. *$\mathscr{HC}$ is called* semantically solvable *if for every structure $(U, I) \in \mathscr{S}$ there is an interpretation of the relation symbols $\mathscr{R}$ as set-theoretic relations over U such that the universally quantified closure $Cl_\forall(h)$ of every clause $h \in \mathscr{HC}$ holds in (U, I).*
2. *$\mathscr{HC}$ is called* syntactically solvable *if there is a relation symbol assignment sol such that for every structure $(U, I) \in \mathscr{S}$ and every clause $h \in \mathscr{HC}$ it is the case that $Cl_\forall(sol(h))$ is satisfied.*

Note that, in the special case when $\mathscr{S}$ contains only one structure, $\mathscr{S} = \{(U, I)\}$, semantic solvability reduces to the existence of relations interpreting $\mathscr{R}$ that extend the structure (U, I) in such a way to make all clauses true. In other words, Horn clauses are solvable in a structure if and only if the extension of the theory of (U, I) by relation symbols $\mathscr{R}$ in the vocabulary and by given Horn clauses as axioms is consistent.

Clearly, if a set of Horn clauses is syntactically solvable, then it is also semantically solvable. The converse is not true in general, because the solution need not be expressible in the constraint language.

For instance, the following Horn clauses are formulated over the theory of linear arithmetic, and semantically solvable, but no solution can be represented in terms of Presburger arithmetic constraints:

(1) $\mathsf{sq}(1,0) \leftarrow \mathbf{true}$
(2) $\mathsf{sq}(n+2,x+n) \leftarrow \mathsf{sq}(n,x)$
(3) $\mathsf{false} \leftarrow x > 1 \wedge \mathsf{sq}(n,x) \wedge \mathsf{sq}(n,x+1)$

The first two clauses imply that $sq(n, x)$ holds (for some n) whenever x is a square number. The third clause states that no two consecutive numbers greater 1 are squares. Since one-dimensional sets definable in Presburger arithmetic are ultimately periodic [GS66], no Presburger arithmetic over-approximation of sq can satisfy all the clauses.

Definition 5.2.2 (Dependence Relation)**.** *A set $\mathscr{HC}$ of Horn clauses induces a* dependence relation $\rightarrow_{\mathscr{HC}}$ *on $\mathscr{R}$, defining $p \rightarrow_{\mathscr{HC}} q$ if there is a Horn clause in $\mathscr{HC}$ that contains p in its head, and q in the body.*

The set $\mathscr{HC}$ is called *recursion-free* if $\rightarrow_{\mathscr{HC}}$ is acyclic, and *recursive* otherwise. In the next sections we study the solvability problem for recursion-free Horn clauses; in particular, Theorem 5.5.1 below characterises the relationship between syntactic and semantic solvability for recursion-free Horn clauses. This case is relevant, since solvers for recursion-free Horn clauses form a main component of many general Horn-clause-based verification systems [GPR11a, GLPR12].

Definition 5.2.3 (Normal Form)**.** *A set $\mathscr{HC}$ of Horn clauses is in normal form [GLPR12] iff every relation symbol has a unique and fixed pairwise distinct vector of variables and every non-argument variable occurs in at most one clause.*

In order to guarantee a fixed vector of parameters for relation symbols we may need to duplicate the relation symbols and all the clauses defining them and solve the resulting system. As an exmple consider the following set of clauses.

$$p(x, y) \wedge p(y, z) \rightarrow r(x, z), \quad q(x, y) \rightarrow p(x, y), \quad x \geq 0 \rightarrow q(x, x)$$

We first expand the system to

$$\begin{aligned} p_1(x,y) \wedge p_2(y,z) &\rightarrow r(x,z) \\ q_1(x,y) &\rightarrow p_1(x,y) \\ x \geq 0 &\rightarrow q_1(x,x) \\ q_2(x,y) &\rightarrow p_2(x,y) \\ x \geq 0 &\rightarrow q_2(x,x) \end{aligned}$$

To make the parameters unique we rename the arguments as the following.

$$\begin{aligned} p_1(x_{p_1}, y_{p_1}) \wedge p_2(y_{p_2}, z_{p_2}) \wedge (y_{p_1} = y_{p_2}) \wedge (x_{p_1} = x_r) \wedge (z_{p_2} = z_r) &\rightarrow r(x_r, z_r) \\ q_1(x_{q_1}, y_{q_1}) \wedge (x_{q_1} = x_{p_1}) \wedge (y_{q_1} = y_{p_1}) &\rightarrow p_1(x_{p_1}, y_{p_1}) \\ (x_{q_1} \geq 0) \wedge (x_{q_1} = y_{q_1}) &\rightarrow q_1(x_{q_1}, y_{q_1}) \\ q_2(y_{q_2}, z_{q_2}) \wedge (y_{q_2} = y_{p_2}) \wedge (z_{q_2} = z_{p_2}) &\rightarrow p_2(y_{p_2}, z_{p_2}) \\ (y_{q_2} \geq 0) \wedge (y_{q_2} = z_{q_2}) &\rightarrow q_2(y_{q_2}, z_{q_2}) \end{aligned}$$

We now solve the expanded system. Afterwards we construct a solution of the original system as

$$C_p[x,y] \;=\; C_{p_1}[x,y] \wedge C_{p_2}[x,y], \quad C_q[x,y] \;=\; C_{q_1}[x,y] \wedge C_{q_2}[x,y]$$

This is possible because the space of (syntactic) solutions of a Horn clause is closed under conjunction.

5.3 The Relationship between Craig Interpolation and Horn Clauses

It has become common to work with generalised forms of Craig interpolation, such as inductive sequences of interpolants, tree interpolants, and restricted DAG interpolants. We show that a variety of such interpolation approaches can be reduced to recursion-free Horn clauses. Recursion-free Horn clauses thus provide a general framework unifying and subsuming a number of earlier notions. As a side effect, we can formulate a general theorem about existence of the individual kinds of interpolants in Sect. 5.6, applicable to any constraint language with the (binary) interpolation property.

An overview of the relationship between specific forms of interpolation and specific fragments of recursions-free Horn clauses is given in Figure 5.3, and will be explained in more detail in the rest of this section. The table in Figure 5.3 refers to the following fragments of recursion-free Horn clauses:

Definition 5.3.1 (Horn clause fragments)**.** *We say that a finite, recursion-free set $\mathscr{HC}$ of Horn clauses*

Form of interpolation	Fragment of Horn clauses
Binary interpolation [Cra57, McM03] $A \wedge B$	Pair of Horn clauses $A \rightarrow p(\bar{x}),\ B \wedge p(\bar{x}) \rightarrow \mathit{false}$ with $\{\bar{x}\} = \mathit{fv}(A) \cap \mathit{fv}(B)$
Inductive interpolant seq. [HJMM04, McM06] $T_1 \wedge T_2 \wedge \cdots \wedge T_n$	Linear tree-like Horn clauses $T_1 \rightarrow p_1(\bar{x}_1),\ p_1(\bar{x}_1) \wedge T_2 \rightarrow p_2(\bar{x}_2),\ \ldots$ with $\{\bar{x}_i\} = \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$
Tree interpolants [McM, HHP10]	Tree-like Horn clauses
(Restricted) DAG interpolants [AGC12a]	Linear Horn clauses
Disjunctive interpolants [RHK13]	Body disjoint Horn clauses

Figure 5.3: Equivalence of interpolation problems and systems of Horn clauses.

1. *is* **linear** *if the body of each Horn clause contains at most one relation symbol,*
2. *is* **body-disjoint** *if for each relation symbol p there is at most one clause containing p in its body; furthermore, every clause contains p at most once;*
3. *is* **head-disjoint** *if for each relation symbol p there is at most one clause containing p in its head;*
4. *is* **tree-like** *[GPR11b] if it is body-disjoint and head-disjoint.*

Theorem 5.3.1 (Interpolation and Horn clauses)**.** *For each line of Figure 5.3 it holds that:*

1. *an interpolation problem of the stated form can be polynomially reduced to (syntactically) solving a set of Horn clauses, in the stated fragment;*
2. *solving a set of Horn clauses (syntactically) in the stated fragment can be polynomially reduced to solving a sequence of interpolation problems of the stated form.*

5.3.1 Binary Craig Interpolants [Cra57, McM03]

The simplest form of Craig interpolation is the derivation of a constraint I such that $A \models I$ and $I \models \neg B$, and such that $\mathit{fv}(I) \subseteq \mathit{fv}(A) \cap \mathit{fv}(B)$. Such derivation is typically constructed by efficiently processing the proof of unsatisfiability of $A \wedge B$. To encode a binary interpolation problem into Horn clauses, we first determine the set $\bar{x} = \mathit{fv}(A) \cap \mathit{fv}(B)$ of variables that can possibly occur in the interpolant. We then pick a relation symbol p of arity $|\bar{x}|$, and define two Horn clauses expressing that $p(\bar{x})$ is an interpolant:

$$A \rightarrow p(\bar{x}), \qquad B \wedge p(\bar{x}) \rightarrow \mathit{false}$$

It is clear that every syntactic solution for the two Horn clauses corresponds to an interpolant of $A \wedge B$.

5.3.2 Inductive Sequences of Interpolants [HJMM04, McM06]

Recall the definition of trace interpolants in 3.2.1. We repeat it here to place it in correspondence with solving the appropriate recursion-free Horn clauses. Given an unsatisfiable conjunction $T_1 \wedge \ldots \wedge T_n$ (in practice, often corresponding to an infeasible path in a program), an *inductive sequence of interpolants* is a sequence $I_0, I_1, \ldots, I_n$ of formulae such that

1. $I_0 = \mathit{true}$, $I_n = \mathit{false}$,
2. for all $i \in \{1, \ldots, n\}$, the entailment $I_{i-1} \wedge T_i \models I_i$ holds, and
3. for all $i \in \{0, \ldots, n\}$, it is the case that $\mathit{fv}(I_i) \subseteq \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$.

While inductive sequences can be computed by repeated computation of binary interpolants [HJMM04], more efficient solvers have been developed that derive a whole sequence of interpolants simultaneously [CGS10, BKRW11, McM].

Inductive sequences as linear tree-like Horn clauses. An inductive sequence of interpolants can straightforwardly be encoded as a set of linear Horn clauses, by introducing a fresh relation symbol p_i for each interpolant I_i to be computed. The arguments of the relation symbols have to be chosen reflecting condition 3 of the definition of interpolant sequences: for each $i \in \{0, \ldots, n\}$, we assume that $\bar{x}_i = \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$ is the vector of variables that can occur in I_i. Conditions 1 and 2 are then represented by the following Horn clauses:

$$p_0(\bar{x}_0),\ p_0(\bar{x}_0) \wedge T_1 \rightarrow p_1(\bar{x}_1),\ p_1(\bar{x}_1) \wedge T_2 \rightarrow p_2(\bar{x}_2),\ \ldots,\ p_n(\bar{x}_n) \rightarrow \mathit{false}$$

Linear tree-like Horn clauses as inductive sequences. Suppose $\mathscr{HC}$ is a finite, recursion-free, linear, and tree-like set of Horn clauses. We can solve the system of Horn clauses by computing one inductive sequence of interpolants for every connected component of the $\rightarrow_{\mathscr{HC}}$-graph. First, we first normalize the set of Horn clauses (Definition 5.2.3) so that for every relation symbol p, we have a unique vector of variables $\bar{x}_p$, and rewrite $\mathscr{HC}$ such that p only occurs in the form $p(\bar{x}_p)$. This is straightforward since $\mathscr{HC}$ is recursion-free and body-disjoint. We then ensure, through renaming, that every variable x that is not argument of a relation symbol occurs in at most one clause. A connected component then represents Horn clauses

$$C_1 \rightarrow p_1(\bar{x}_1),\ C_2 \wedge p_1(\bar{x}_1) \rightarrow p_2(\bar{x}_2),\ C_3 \wedge p_2(\bar{x}_2) \rightarrow p_3(\bar{x}_3),\ \ldots,\ C_n \wedge p_n(\bar{x}_n) \rightarrow \mathit{false}\,.$$

(If the first or the last of the clauses is missing, we assume that its constraint is *false*). Any inductive sequence of interpolants for $C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_n$ solves the clauses.

5.3.3 Tree Interpolants [McM, HHP10]

Tree interpolants strictly generalise inductive sequences of interpolants, and are designed with the application of inter-procedural verification in mind: in this context, the tree structure of the interpolation problem corresponds to (a part of) the call graph of a program. Tree interpolation problems correspond to recursion-free tree-like sets of Horn clauses.

Suppose (V, E) is a finite directed tree, writing $E(v, w)$ to express that the node w is a direct child of v. Further, suppose $\phi : V \to \mathit{Constr}$ is a function that labels each node v of the tree with a formula $\phi(v)$. A labelling function $I : V \to \mathit{Constr}$ is called a *tree interpolant* (for (V, E) and ϕ) if the following properties hold:

1. for the root node $v_0 \in V$, it is the case that $I(v_0) = \mathit{false}$,
2. for any node $v \in V$, the following entailment holds:

$$\phi(v) \wedge \bigwedge_{(v,w)\in E} I(w) \models I(v) ,$$

3. for any node $v \in V$, every non-logical symbol (in our case: variable) in $I(v)$ occurs both in some formula $\phi(w)$ for w such that $E^*(v, w)$, and in some formula $\phi(w')$ for some w' such that $\neg E^*(v, w')$. (E^* is the reflexive transitive closure of E).

Since the case of tree interpolants is instructive for solving recursion-free sets of Horn clauses in general, we give a result about the existence of tree interpolants. The proof of the lemma computes tree interpolants by repeated derivation of binary interpolants; however, as for inductive sequences of interpolants, there are solvers that can compute all formulae of a tree interpolant simultaneously [McM, GPR11a, GPR11b].

Lemma 5.3.2. *Suppose the constraint language Constr that has the interpolation property. Then a tree (V, E) with labelling function $\phi : V \to \mathit{Constr}$ has a tree interpolant I if and only if $\bigwedge_{v\in V} \phi(v)$ is unsatisfiable.*

Proof. "$\Rightarrow$" follows from the observation that every interpolant $I(v)$ is a consequence of the conjunction $\bigwedge_{(v,w)\in E^+} \phi(w)$.

"$\Leftarrow$": let $v_1, v_2, \ldots, v_n$ be an inverse topological ordering of the nodes in (V, E), i.e., an ordering such that $\forall i, j.\ (E(v_i, v_j) \Rightarrow i > j)$. We inductively construct a sequence of formulae $I_1, I_2, \ldots, I_n$, such that for every $i \in \{1, \ldots, n\}$ the following properties hold:

1. the following conjunction is unsatisfiable:

$$\bigwedge\{I_k \mid k \leq i,\ \forall j.\,(E(v_j, v_k) \Rightarrow j > i)\} \ \wedge\ \Big(\phi(v_{i+1}) \wedge \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n)\Big) \tag{5.1}$$

2. the following entailment holds:

$$\phi(v_i) \wedge \bigwedge_{(v_i, v_j) \in E} I_j \ \models\ I_i$$

3. every non-logical symbol in I_i occurs both in a formula $\phi(w)$ with $E^*(v_i, w)$, and in a formula $\phi(w')$ with $\neg E^*(v_i, w')$.

Assume that the formulae $I_1, I_2, \ldots, I_i$ have been constructed, for $i \in \{0, \ldots, n-1\}$. We then derive the next interpolant I_{i+1} by solving the binary interpolation problem

$$\Big(\phi(v_{i+1}) \wedge \bigwedge_{E(v_{i+1}, v_j)} I_j\Big) \wedge \Big(\bigwedge\{I_k \mid k \leq i,\ \forall j.\,(E(v_j, v_k) \Rightarrow j > i+1)\} \ \wedge\ \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n)\Big) \tag{5.2}$$

That is, we construct I_{i+1} so that the following entailments hold:

$$\phi(v_{i+1}) \wedge \bigwedge_{E(v_{i+1}, v_j)} I_j \ \models\ I_{i+1},$$
$$\bigwedge\{I_k \mid k \leq i,\ \forall j.\,(E(v_j, v_k) \Rightarrow j > i+1)\} \ \wedge\ \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n) \ \models\ \neg I_{i+1}$$

Furthermore, I_{i+1} only contains non-logical symbols that are common to the left and the right side of the conjunction.

Note that (5.2) is equivalent to (5.1), therefore unsatisfiable, and a well-formed interpolation problem. It is also easy to see that the properties 1–3 hold for I_{i+1}. Also, we can easily verify that the labelling function $I : v_i \mapsto I_i$ is a solution for the tree interpolation problem defined by (V, E) and ϕ. □

Tree interpolation as tree-like Horn clauses. The encoding of a tree interpolation problem as a tree-like set of Horn clauses is very similar to the encoding for inductive sequences of interpolants. We introduce a fresh relation symbol p_v for each node $v \in V$ of a tree interpolation problem $(V, E), \phi$, assuming that for each $v \in V$ the vector $\bar{x}_v = \bigcup_{E^*(v,w)} fv(\phi(w)) \cap \bigcup_{\neg E^*(v,w)} fv(\phi(w))$ represents the set of variables that can occur in the interpolant $I(v)$. The interpolation problem is then represented by the following clauses:

$$p_0(\bar{x}_0) \rightarrow \mathit{false}, \quad \Big\{\, \phi(v) \wedge \bigwedge_{(v,w) \in E} p_w(\bar{x}_w) \rightarrow p_v(\bar{x}_v) \,\Big\}_{v \in V}$$

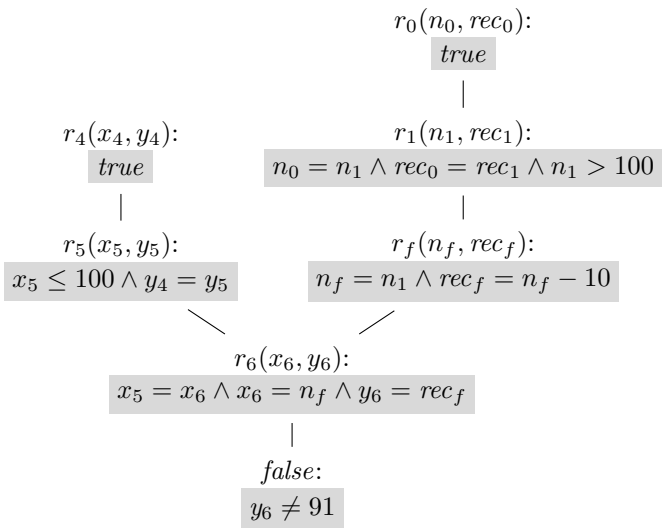


Figure 5.4: Tree interpolation problem for the clauses in Example 5.3.3

Tree-like Horn clauses as tree interpolation. Suppose $\mathscr{HC}$ is a finite, recursion-free, and tree-like set of Horn clauses. We can solve the system of Horn clauses by computing a tree interpolant for every connected component of the $\rightarrow_{\mathscr{HC}}$-graph. As before, we first normalise the Horn clauses by fixing, for every relation symbol p, a unique vector of variables $\bar{x}_p$, and rewriting $\mathscr{HC}$ such that p only occurs in the form $p(\bar{x}_p)$. We also ensure that every variable x that is not argument of a relation symbol occurs in at most one clause. The tree interpolation graph (V, E) is then defined by choosing the set $V = \mathscr{R} \cup \{false\}$ of relation symbols as nodes, and the child relation $E(p, q)$ to hold whenever p occurs as head, and q within the body of a clause. The labelling function ϕ is defined by $\phi(p) = C$ whenever there is a clause with head symbol p and constraint C, and $\phi(p) = false$ if p does not occur as head of any clause.

Example We consider a subset of the Horn clauses given in Figure 2.3:

(1) r0(N,Rec) ← **true**
(2) r1(N,Rec) ← r0(N,Rec) ∧ N > 100
(5) rf(N,Rec') ← r1(N,Rec) ∧ Rec' = N − 10
(7) r4(X,Y) ← **true**
(8) r5(X',Y) ← r4(X,Y) ∧ X' ≤ 100
(9) r6(X,Y') ← r5(X,Y) ∧ rf(X,Y')
(10) **false** ← r6(X,Y) ∧ Y ≠ 91

Note that this recursion-free subset of the clauses is body-disjoint and head-disjoint, and thus tree-like. Since the complete set of clauses in Figure 2.3 is solvable, also any subset is; in order to compute a (syntactic) solution of the clauses, we set up the corresponding tree interpolation problem. Figure 5.4 shows the tree with the labelling ϕ to be interpolated (in grey), as well as the head literals of the clauses generating the nodes of the tree. A tree interpolant solving the

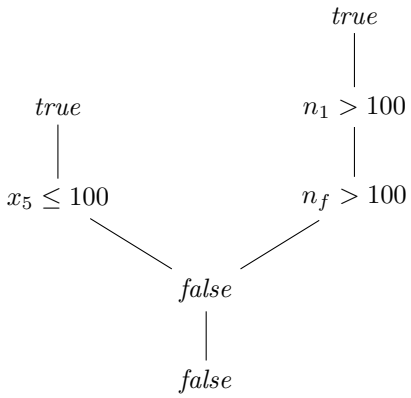


Figure 5.5: Tree interpolant solving the interpolation problem in Figure 5.4

interpolation problem is given in Figure 5.5. The tree interpolant can straightforwardly be mapped to a solution of the original tree-like Horn, for instance we set $r_f(n_f, rec_f) = (n_f > 100)$ and $r_5(x5, y_5) = (x5 \leq 100)$.

□

Symmetric Interpolants

A special case of tree interpolants, *symmetric interpolants,* was introduced in [McM05a]. Given an inconsistent set of formulas $V = \{v_0, \cdots, v_n\}$ a symmetric interpolant is a set of formulas $\mathscr{I} = \{I_0, \cdots, I_n\}$ such that each v_i implies I_i, and $\mathscr{I}$ is inconsistent, and each I_i is over the symbols common to v_i and $V \setminus v_i$. Symmetric interpolants are equivalent to tree interpolants with a flat tree structure (V, E), i.e., $V = \{root, v_1, \ldots, v_n\}$, where the nodes $v_1, \ldots, v_n$ are the direct children of *root*.

5.3.4 Restricted (and Unrestricted) DAG Interpolants [AGC12a]

Restricted DAG interpolants are a further generalisation of inductive sequence of interpolants, introduced for the purpose of reasoning about multiple paths in a program simultaneously [AGC12a]. Suppose (V, E, en, ex) is a finite connected DAG with entry node $en \in V$ and exit node $ex \in V$, further $\mathscr{L}_E : E \to Constr$ a labelling of edges with constraints, and $\mathscr{L}_V : V \to Constr$ a labelling of vertices. A *restricted DAG interpolant* is a mapping $I : V \to Constr$ with

1. $I(en) = true$, $I(ex) = false$,

2. for all $(v, w) \in E$ the entailment $I(v) \wedge \mathscr{L}_V(v) \wedge \mathscr{L}_E(v, w) \models I(w) \wedge \mathscr{L}_V(w)$ holds, and

3. for all $v \in V$ it is the case that[1]

$$fv(I(v)) \subseteq \Big(\bigcup_{(a,v)\in E} fv(\mathscr{L}_E(a,v)) \Big) \cap \Big(\bigcup_{(v,a)\in E} fv(\mathscr{L}_E(v,a)) \Big).$$

The UFO verification system [ALGC12] is able to compute DAG interpolants, based on the interpolation functionality of MathSAT [CGS10]. We can observe that DAG interpolants (despite their name) are incomparable in expressiveness to tree interpolation. This is because DAG interpolants correspond to *linear* Horn clauses, and might have shared relation symbol in bodies, while tree interpolants correspond to *possibly nonlinear tree-like* Horn clauses, but do not allow shared relation symbols in bodies. Nevertheless, it is possible to reduce DAG interpolants to tree interpolants, but only at the cost of a potentially exponential growth in the number of clauses. Considering the paths in the DAG as program paths this blowup corresponds to enumerating all the possible syntactic traces of the program.

Encoding of restricted DAG interpolants as linear Horn clauses. For every $v \in V$, let

$$\{\bar{x}_v\} = \Big(\bigcup_{(a,v)\in E} fv(\mathscr{L}_E(a,v)) \Big) \cap \Big(\bigcup_{(v,a)\in E} fv(\mathscr{L}_E(v,a)) \Big)$$

be the variables allowed in the interpolant to be computed for v, and p_v be a fresh relation symbol of arity $|\bar{x}_v|$. The interpolation problem is then defined by the following set of linear Horn clauses:

$$\begin{aligned}
\text{For each } (v,w) \in E: \quad & \mathscr{L}_V(v) \wedge \mathscr{L}_E(v,w) \wedge p_v(\bar{x}_v) \rightarrow p_w(\bar{x}_w), \\
& \mathscr{L}_V(v) \wedge \neg\mathscr{L}_V(w) \wedge \mathscr{L}_E(v,w) \wedge p_v(\bar{x}_v) \rightarrow \mathit{false}, \\
\text{For } en, ex \in V: \quad & \mathit{true} \rightarrow p_{en}(\bar{x}_{en}), \qquad p_{ex}(\bar{x}_{ex}) \rightarrow \mathit{false}
\end{aligned}$$

Encoding of linear Horn clauses as DAG interpolants. Suppose $\mathscr{HC}$ is a finite, recursion-free, and linear set of Horn clauses. We can solve the system of Horn clauses by computing a DAG interpolant for every connected component of the $\rightarrow_{\mathscr{HC}}$-graph. We first normalize the set of Horn clauses (Definition 5.2.3) by fixing a unique vector $\bar{x}_p$ of argument variables for each relation symbol p, and ensure that every non-argument variable x occurs in at most one clause. We also assume that multiple clauses $C \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$ and $D \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$ with the same relation symbols are merged to $(C \vee D) \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$.

Let $\{p_1, \ldots, p_n\}$ be all relation symbols of one connected component. We then define the DAG interpolation problem $(V, E, en, ex), \mathscr{L}_E, \mathscr{L}_V$ by

- the vertices $V = \{p_1, \ldots, p_n\} \cup \{en, ex\}$, including two fresh nodes en, ex,

[1] The definition of DAG interpolants in [AGC12a, Def. 4] implies that $fv(I(v)) = \emptyset$ for every interpolant $I(v)$, $v \in V$, i.e., only trivial interpolants are allowed. We assume that this is a mistake in [AGC12a, Def. 4], and corrected the definition as shown here.

- the edge relation

$$\begin{aligned} E \;=\; & \{(p,q) \mid \text{there is a clause } C \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q) \in \mathscr{HC}\} \\ & \cup \{(en,p) \mid \text{there is a clause } D \rightarrow p(\bar{x}_p) \in \mathscr{HC}\} \\ & \cup \{(p,ex) \mid \text{there is a clause } E \wedge p(\bar{x}_p) \rightarrow \mathit{false} \in \mathscr{HC}\}\,, \end{aligned}$$

- for each $(v,w) \in E$, the edge labelling

$$\mathscr{L}_E(v,w) \;=\; \begin{cases} C \wedge \bar{x}_v = \bar{x}_v \wedge \bar{x}_w = \bar{x}_w & \text{if } C \wedge v(\bar{x}_v) \rightarrow w(\bar{x}_w) \in \mathscr{HC} \\ D \wedge \bar{x}_w = \bar{x}_w & \text{if } v = en \text{ and } D \rightarrow w(\bar{x}_w) \in \mathscr{HC} \\ E \wedge \bar{x}_v = \bar{x}_v & \text{if } w = ex \text{ and } E \wedge v(\bar{x}_v) \rightarrow \mathit{false} \in \mathscr{HC} \end{cases}$$

  Note that the labels include equations like $\bar{x}_v = \bar{x}_v$ to ensure that the right variables are allowed to occur in interpolants.

- for each $v \in V$, the node labelling $\mathscr{L}_V(v) = \mathit{true}$.

By checking the definition of DAG interpolants, it can be verified that every interpolant solving the problem $(V,E,en,ex),\mathscr{L}_E,\mathscr{L}_V$ is also a solution of the linear Horn clauses.

5.4 Disjunctive Interpolants and Body-Disjoint Horn Clauses

Disjunctive interpolants were introduced in [RHK13] as a generalisation of tree interpolants. Disjunctive interpolants resemble tree interpolants in the sense that the relationship of the components of an interpolant is defined by a tree; in contrast to tree interpolants, however, this tree is an and/or-tree: branching in the tree can represent either *conjunctions* or *disjunctions*. Disjunctive interpolation problems can specify both conjunctive and disjunctive relationships between interpolants, and are thus applicable for simultaneous analysis of multiple paths in a program, but also tailored to inter-procedural analysis or verification of concurrent programs [GLPR12].

Disjunctive interpolants correspond to sets of body-disjoint Horn clauses; in this representation, and-branching is encoded by clauses with multiple body literals (like with tree interpolants), while or-branching is interpreted as multiple clauses sharing the same head symbol. The definition of disjunctive interpolation is chosen deliberately to be as general as possible, while still avoiding the high computational complexity of solving general systems of recursion-free Horn clauses. Computational complexity is discussed in Sect. 5.6.

We introduce disjunctive interpolants as a form of *sub-formula abstraction*. For example, given an unsatisfiable constraint $\phi[\alpha]$ containing α as a sub-formula in a positive position, the goal is to find an abstraction α' such that $\alpha \models \alpha'$ and $\alpha[\alpha'] \models \mathit{false}$, and such that α' only contains variables common to α and $\phi[\mathit{true}]$. Generalizing this to any number of subformulas,

we obtain the following.

Definition 5.4.1 (Disjunctive interpolant)**.** *Let ϕ be a constraint, and $pos \subseteq positions(\phi)$ a set of positions in ϕ that are only underneath the connectives $\wedge$ and $\vee$. A* disjunctive interpolant *is a map $I : pos \to Constr$ from positions to constraints such that:*

1. *For each position $p \in pos$, with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid p < q \text{ and } \neg\exists r \in pos.\ p < r < q\}$ we have*

$$\big(\phi[q_1 / I(q_1), \ldots, q_n / I(q_n)]\big)\downarrow p \models I(p)\,,$$

2. *For the topmost positions $\{q_1, \ldots, q_n\} = \{q \in pos \mid \neg\exists r \in pos.\ r < q\}$ we have*

$$\phi[q_1 / I(q_1), \ldots, q_n / I(q_n)] \models \mathit{false}\,,$$

3. *For each position $p \in pos$, we have $fv(I(p)) \subseteq fv(\phi \downarrow p) \cap fv(\phi[p / \mathit{true}])$.*

Example Consider $A_p \wedge B$, with position p pointing to the sub-formula A, and $pos = \{p\}$. The disjunctive interpolants for $A \wedge B$ and *pos* coincide with the ordinary binary interpolants for $A \wedge B$. □

Example Consider the formula $\phi = \big(\cdots\big(\big(\big(T_1\big)_{p_1} \wedge T_2\big)_{p_2} \wedge T_3\big)_{p_3} \wedge \cdots\big)_{p_{n-1}} \wedge T_n$ and positions $pos = \{p_1, \ldots, p_{n-1}\}$. Disjunctive interpolants for ϕ and *pos* correspond to inductive sequences of interpolants [HJMM04, McM06]. Note that we have the entailments $T_1 \models I(p_1)$, $I(p_1) \wedge T_2 \models I(p_2)$, …, $I(p_{n-1}) \wedge T_n \models \mathit{false}$. □

Example Tree interpolation problems correspond to disjunctive interpolation with a set *pos* of positions that are only underneath $\wedge$ (and never underneath $\vee$). □

Example We consider the example given in Fig. 5.2, Sect. 6.3. To compute a solution for the Horn clauses, we first *expand* the Horn clauses into a constraint, by means of exhaustive

inlining/resolution (see Sect. 5.5), obtaining a disjunctive interpolation problem:

$$
\begin{aligned}
\mathit{false} &\rightsquigarrow M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M,N,R) \wedge R > M \\
&\rightsquigarrow \begin{pmatrix} & M \geq 0 \\ \wedge & M = N \\ \wedge & R > M \end{pmatrix} \wedge \begin{pmatrix} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge \mathsf{gcd1}(M_1, N, R) \\ \vee \\ M < N \wedge N_1 = N - M \wedge \mathsf{gcd1}(M, N_1, R) \end{pmatrix} \\
&\rightsquigarrow \begin{pmatrix} & M \geq 0 \\ \wedge & M = N \\ \wedge & R > M \end{pmatrix} \wedge \begin{pmatrix} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge (M_1 = N \wedge R = M_1)_q \\ \vee \\ M < N \wedge N_1 = N - M \wedge (M = N_1 \wedge R = M)_r \end{pmatrix}_p
\end{aligned}
$$

In the last formula, the positions p, q, r corresponding to the relation symbol gcd and the two occurrences of gcd1 are marked. It can be observed that the last formula is unsatisfiable, and that $I = \{p \mapsto ((M = N) \rightarrow (M \geq R)),\ q \mapsto \mathit{true},\ r \mapsto \mathit{true}\}$ is a disjunctive interpolant. A solution for the Horn clauses can be derived from the interpolant by conjoining the constraints derived for the two occurrences of gcd1:

$$\mathsf{gcd}(M,N,R) \ = \ ((M = N) \rightarrow (M \geq R)), \quad \mathsf{gcd1}(M,N,R) \ = \ \mathit{true}$$

□

Theorem 5.4.1. *Suppose ϕ is a constraint, and suppose $pos \subseteq positions(\phi)$ is a set of positions in ϕ that are only underneath the connectives $\wedge$ and $\vee$. If Constr is a constraint language that has the interpolation property, then a disjunctive interpolant I exists for ϕ and pos if and only if ϕ is unsatisfiable.*

Proof. "$\Rightarrow$" By means of simple induction, we can derive that $\phi \downarrow p \models I(p)$ holds for every disjunctive interpolant I for ϕ and *pos*, and for every $p \in pos$. From Def. 5.4.1, it then follows that ϕ is unsatisfiable.

"$\Leftarrow$" Suppose ϕ is unsatisfiable. We encode the disjunctive interpolation problem into a (conjunctive) tree interpolation problem by adding auxiliary Boolean variables. [2] Wlog, we assume that *pos* contains the root position *root* of ϕ. The graph of the tree interpolation problem is (pos, E), with the edge relation $E = \{(p,q) \mid p < q \text{ and } \neg\exists r.\, p < r < q\}$. For every $p \in pos$, let a_p be a fresh Boolean variable. We label the nodes of the tree using the function ϕ_L : $pos \rightarrow Constr$. For each position $p \in pos$, with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid E(p,q)\}$

[2] The concept of auxiliary Boolean variables to represent interpolation problems has also been used in [SFS11] and [AGC12b], for the purpose of extracting function summaries in model checking.

we define

$$\phi_L(p) \;=\; \begin{cases} \phi[q_1/a_{q_1},\ldots,q_n/a_{q_n}] & \text{if } p = \mathit{root} \\ \neg a_p \vee \big(\phi[q_1/a_{q_1},\ldots,q_n/a_{q_n}]\big)\downarrow p & \text{otherwise} \end{cases}$$

Observe that $\bigwedge_{p\in \mathit{pos}} \phi_L(p)$ is unsatisfiable. According to Lemma 5.3.2 a tree interpolant I_T exists for this labelling function. By construction, for non-root positions $p \in \mathit{pos} \setminus \{\mathit{root}\}$ the interpolant labelling is equivalent to $I_T(p) \equiv \neg a_p \vee I_p$, where I_p does not contain any further auxiliary Boolean variables. We can then construct a disjunctive interpolant I for the original problem as

$$I(p) \;=\; \begin{cases} \mathit{false} & \text{if } p = \mathit{root} \\ I_p & \text{otherwise} \end{cases}$$

To see that I is a disjunctive interpolant, observe that for each position $p \in \mathit{pos}$ with direct children $\{q_1,\ldots,q_n\} \;=\; \{q \in \mathit{pos} \mid E(p,q)\}$ the following entailment holds (since I_T is a tree interpolant): $\phi_L(p) \wedge (\neg a_{q_1} \vee I_{q_1}) \wedge \cdots \wedge (\neg a_{q_n} \vee I_{q_n}) \;\models\; I_T(p)$
Via Boolean reasoning this implies: $\big(\phi[q_1/I_{q_1},\ldots,q_n/I_{q_n}]\big)\downarrow p \;\models\; I(p)$.

□

The proof provides a constructive method to solve disjunctive interpolation problems, by means of transformation to a tree interpolation problem. This is also the algorithm that we used in our experiments in Sect. 5.9; practical aspects of this approach are discussed in the beginning of Sect. 5.8.

5.4.1 Solvability of Body-Disjoint Horn Clauses

Disjunctive interpolation corresponds to *body disjoint* recursion-free Horn clauses. An example for body-disjoint clauses is the subset {(1), (4)} of clauses in Fig. 5.1. Syntactic solutions of a set $\mathscr{HC}$ of body-disjoint Horn clauses can be computed by solving a disjunctive interpolation problem; vice versa, every disjunctive interpolation problem can be translated into an equivalent set of body-disjoint clauses.

In order to extract an interpolation problem from $\mathscr{HC}$, we first normalise the clauses ((Definition 5.2.3): for every relation symbol $p \in \mathscr{R}$, we fix a unique vector of variables $\bar{x}_p$, and rewrite $\mathscr{HC}$ such that p only occurs in the form $p(\bar{x}_p)$. This is possible due to the fact that $\mathscr{HC}$ is body disjoint. The translation from Horn clauses to a disjunctive interpolation problem is done recursively, similar in spirit to inlining of function invocations in a program; thanks to body-disjointness, the encoding is polynomial.

$$enc(\mathcal{HC}) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow false) \in \mathcal{HC}} C \wedge enc'(B_1) \wedge \cdots \wedge enc'(B_n)$$

$$enc'(p(\tilde{x}_p)) = \left(\bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow p(\tilde{x}_p)) \in \mathcal{HC}} C \wedge enc'(B_1) \wedge \cdots \wedge enc'(B_n) \right)_{l_p}$$

Note that the resulting formula $enc(\mathcal{HC})$ contains a unique position l_p at which the definition of a relation symbol p is inlined; in the second equation, this position is marked with l_p. Any disjunctive interpolant I for this set of positions represents a syntactic solution of $\mathcal{HC}$, and vice versa.

The solution of body-disjoint Horn clauses can be computed by solving a sequence of tree-like sets of Horn clauses:

Lemma 5.4.2. *Let $\mathcal{HC}$ be a finite set of recursion-free body-disjoint Horn clauses. $\mathcal{HC}$ has a syntactic/semantic solution if and only if every maximum tree-like subset of $\mathcal{HC}$ has a syntactic/semantic solution.*

Proof. We outline direction "$\Leftarrow$" for syntactic solutions. Solving the tree-like subsets of $\mathcal{HC}$ yields, for each relation symbol $p \in \mathcal{R}$, a set SC_p of solution constraints. A global solution of $\mathcal{HC}$ can be constructed by forming a positive Boolean combination of the constraints in SC_p for each $p \in \mathcal{R}$. □

Example We consider a recursion-free unwinding of the Horn clauses in Figure 2.3. To make the set of clauses body-disjoint, some of the clause were duplicated, introducing primed copies of all relation symbols involved. The clauses are not head-disjoint, since (5) and (6) share the same head symbol.

(1) $\mathsf{r0(N,Rec)} \leftarrow \mathbf{true}$
(2) $\mathsf{r1(N,Rec)} \leftarrow \mathsf{r0(N,Rec)} \wedge \mathsf{N} > 100$
(3) $\mathsf{r2(N,Rec)} \leftarrow \mathsf{r0}^a\mathsf{(N,Rec)} \wedge \mathsf{N} \leq 100$
(4) $\mathsf{r3(N,Rec')} \leftarrow \mathsf{r2(N,Rec)} \wedge \mathsf{rf}^b\mathsf{(N + 11,Rec')}$
(5) $\mathsf{rf(N,Rec')} \leftarrow \mathsf{r1(N,Rec)} \wedge \mathsf{Rec' = N - 10}$
(6) $\mathsf{rf(N,Rec')} \leftarrow \mathsf{r3(N,Rec)} \wedge \mathsf{rf}^c\mathsf{(Rec,Rec')}$

(1^a) $\mathsf{r0}^a\mathsf{(N,Rec)} \leftarrow \mathbf{true}$

(1^b) $\mathsf{r0}^b\mathsf{(N,Rec)} \leftarrow \mathbf{true}$
(2^b) $\mathsf{r1}^b\mathsf{(N,Rec)} \leftarrow \mathsf{r0}^b\mathsf{(N,Rec)} \wedge \mathsf{N} > 100$
(5^b) $\mathsf{rf}^b\mathsf{(N,Rec')} \leftarrow \mathsf{r1}^b\mathsf{(N,Rec)} \wedge \mathsf{Rec' = N - 10}$

(1^c) $\mathsf{r0}^c\mathsf{(N,Rec)} \leftarrow \mathbf{true}$
(2^c) $\mathsf{r1}^c\mathsf{(N,Rec)} \leftarrow \mathsf{r0}^c\mathsf{(N,Rec)} \wedge \mathsf{N} > 100$
(5^c) $\mathsf{rf}^c\mathsf{(N,Rec')} \leftarrow \mathsf{r1}^c\mathsf{(N,Rec)} \wedge \mathsf{Rec' = N - 10}$

(7) r4(X,Y) ← **true**
(8) r5(X',Y) ← r4(X,Y) ∧ X' ≤ 100
(9) r6(X,Y') ← r5(X,Y) ∧ rf(X,Y')
(10) **false** ← r6(X,Y) ∧ Y ≠ 91

There are two maximum tree-like subsets: $T_1 = \{(1),(2),(5),(7),(8),(9),(10)\}$, and $T_2 = \{(3),(4),(6),(1^a),(1^b),(2^b),(5^b),(1^c),(2^c),(5^c),(7),(8),(9),(10)\}$. The subset T_1 has been discussed in Example 5.3.3. In the same way, it is possible to construct a solution for T_2 by solving a tree interpolation problem. The two solutions can be combined to construct a solution of $T_1 \cup T_2$:

	T_1	T_2	$T_1 \cup T_2$
$r_0(n,r)$	*true*	–	*true*
$r_1(n,r)$	$n > 100$	–	$n > 100$
$r_2(n,r)$	–	$n \leq 100$	$n \leq 100$
$r_3(n,r)$	–	$r \leq 101$	$r \leq 101$
$r_4(x,y)$	*true*	*true*	*true*
$r_5(x,y)$	$x \leq 100$	*true*	*true*
$r_6(x,y)$	$false$	$y = 91$	$y = 91$
$r_f(n,r)$	$n > 100$	$r = 91$	$n > 100 \vee r = 91$
$r_0^a(n,r)$	–	*true*	*true*
$r_0^b(n,r)$	–	*true*	*true*
$r_1^b(n,r)$	–	*true*	*true*
$r_f^b(n,r)$	–	$n - r \geq 10$	$n - r \geq 10$
$r_0^c(n,r)$	–	*true*	*true*
$r_1^c(n,r)$	–	$n \geq 101$	$n \geq 101$
$r_f^c(n,r)$	–	$n \geq 102 \vee (n = 101 \wedge r = 91)$	$n \geq 102 \vee (n = 101 \wedge r = 91)$

□

5.5 Solvability of Recursion-free Horn Clauses

The previous section discussed how the class of recursion-free body-disjoint Horn clauses can be solved by reduction to disjunctive interpolation. We next show that this construction can be generalised to arbitrary systems of recursion-free Horn clauses. In absence of the body-disjointness condition, however, the encoding of Horn clauses as interpolation problems can incur a potentially exponential blowup. We give a complexity-theoretic argument in Section 5.6 justifying that this blowup cannot be avoided in general. This puts disjunctive interpolation (and, equivalently, body-disjoint Horn clauses) at a sweet spot: preserving the relatively low complexity of ordinary binary Craig interpolation, while carrying much of the flexibility of the Horn clause framework.

We first introduce the exhaustive *expansion* $exp(\mathscr{HC})$ of a set $\mathscr{HC}$ of Horn clauses, which generalises the Horn clause encoding from the previous section. We write $C' \wedge B'_1 \wedge \cdots \wedge B'_n \rightarrow H'$

for a fresh variant of a Horn clause $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$, i.e., the clause obtained by replacing all free first-order variables with fresh variables. Expansion is then defined by the following recursive functions:

$$exp(\mathcal{HC}) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow false) \in \mathcal{HC}} C' \wedge exp'(B'_1) \wedge \cdots \wedge exp'(B'_n)$$

$$exp'(p(\bar{t})) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow p(\bar{s})) \in \mathcal{HC}} C' \wedge exp'(B'_1) \wedge \cdots \wedge exp'(B'_n) \wedge \bar{t} = \bar{s}'$$

Note that *exp* is only well-defined for finite and recursion-free sets of Horn clauses, since the expansion might not terminate otherwise.

Theorem 5.5.1 (Solvability of recursion-free Horn clauses)**.** *Let $\mathcal{HC}$ be a finite, recursion-free set of Horn clauses. If the underlying constraint language has the interpolation property, then the following statements are equivalent:*

1. *$\mathcal{HC}$ is semantically solvable;*
2. *$\mathcal{HC}$ is syntactically solvable;*
3. *$exp(\mathcal{HC})$ is unsatisfiable.*

Proof. $2 \Rightarrow 1$ holds because a syntactic solution gives rise to a semantic solution by interpreting the solution constraints. $\neg 3 \Rightarrow \neg 1$ holds because a model of $exp(\mathcal{HC})$ witnesses domain elements that every semantic solution of $\mathcal{HC}$ has to contain, but which violate at least one clause of the form $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow \mathit{false}$, implying that no semantic solution can exist.

$3 \Rightarrow 2$ is shown by encoding $\mathcal{HC}$ into a disjunctive interpolation problem (Sect. 5.4), which can solved with the help of Theorem 5.4.1. To this end, clauses are first duplicated to obtain a problem that is body disjoint, and subsequently normalised as described in Sect. 5.4.1.

The outline of the proof for direction $3 \Rightarrow 2$ is the following. Suppose the expansion $exp(\mathcal{HC})$ of a set $\mathcal{HC}$ of recursion-free Horn clauses is unsatisfiable. We compute a solution of the Horn clauses separately for every connected component of the $\rightarrow_{\mathcal{HC}}$-graph. Wlog we can therefore assume that the $\rightarrow_{\mathcal{HC}}$-graph is connected.

Renaming of first-order variables and normalisation. We normalise the resulting clauses like in Sect. 5.4.1: for every relation symbol p, we fix a unique vector of variables $\bar{x}_p$, and rewrite $\mathcal{HC}$ such that p only occurs in the form $p(\bar{x}_p)$; by renaming variables, we then ensure that every variable x that is not argument of a relation symbol occurs in at most one clause.

Encoding into a disjunctive interpolation problem. The translation from Horn clauses to a disjunctive interpolation problem is done by adapting the expansion function *exp*:

$$enc(\mathscr{HC}) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow false) \in \mathscr{HC}} C' \wedge enc(B_1) \wedge \cdots \wedge enc(B_n)$$

$$enc(p(\bar{x}_p)) = \left(\bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow p(\bar{x}_p)) \in \mathscr{HC}} C' \wedge enc(B_1) \wedge \cdots \wedge enc(B_n) \right)_{l_p}$$

Note that the resulting formula $enc(\mathscr{HC})$ contains a unique position l_p at which the definition of a relation symbol p is inlined; in the second equation, this position is marked with l_p. We then derive a disjunctive interpolant I for this set of positions in $enc(\mathscr{HC})$. A syntactic solution of $\mathscr{HC}$ is then given by the definition $\forall \bar{x}_p.\big(p(\bar{x}_p) \leftrightarrow I(l_p)\big)$, for all relation symbols p.

□

5.6 The Complexity of Recursion-free Horn Clauses

Theorem 5.5.1 gives rise to a general algorithm for (syntactically) solving recursion-free sets $\mathscr{HC}$ of Horn clauses, over constraint languages for which interpolation procedures are available. The general algorithm requires, however, to generate and solve the expansion $exp(\mathscr{HC})$ of the Horn clauses, which can be exponentially bigger than $\mathscr{HC}$ (in case $\mathscr{HC}$ is not body disjoint), and might therefore require exponential time. This leads to the question whether more efficient algorithms are possible for solving Horn clauses. We give a number of complexity results about (semantic) Horn clause solvability. It is of crucial importance to observe that solvability is PSPACE-hard, for every non-trivial constraint language *Constr*. The authors of [LQL12] conjecture a similar complexity result for the case of programs with procedures.

Lemma 5.6.1. *Suppose a constraint language can distinguish at least two values, i.e., there are two ground terms t_0 and t_1 such that $t_0 \neq t_1$ is satisfiable. Then the semantic solvability problem for recursion-free Horn clauses is PSPACE-hard.*

Proof. We reduce the unsatisfiability problem of Quantified Boolean Formulae (QBF) to solvability of recursion-free Horn clauses. The QBF problem is known to be PSPACE-hard, the original proof of PSPACE-completeness of QBF is in [SM73]. Assume an arbitrary QBF of the shape $\phi = Q_1 x_1.Q_2 x_2....Q_n x_n.F$, where $Q_i \in \exists, \forall$ are quantifiers, x_i are all variables occurring in the formula, and F is a quantifier-free Boolean formula in CNF.

We translate ϕ into a recursion-free set of Horn clauses:

- a literal x_i of a clause C_j in F becomes a Horn clause
  $x_i = t_1 \rightarrow C_{i,j}(x_1, x_2, \ldots, x_{i-1}, t_1, x_{i+1}, \ldots, x_n)$

- a literal $\neg x_i$ of a clause C_j in F becomes a Horn clause
  $x_i = t_0 \to C_{i,j}(x_1, x_2, \ldots, x_{i-1}, t_0, x_{i+1}, \ldots, x_n)$
- a clause C_j in F becomes a set of Horn clauses
  $C_{1,j}(x_1, \ldots) \to C_j(x_1, \ldots), \quad C_{2,j}(x_1, \ldots) \to C_j(x_1, \ldots), \quad \ldots$
- the body F becomes the Horn clause
  $C_1(x_1, \ldots) \wedge C_2(x_1, \ldots) \wedge \cdots \to F_n(x_1, \ldots)$
- a quantifier $Q_i = \exists$ is translated as the two clauses
  $F_{i+1}(x_1, \ldots, x_{i-1}, 0) \to F_i(x_1, \ldots, x_{i-1}), \quad F_{i+1}(x_1, \ldots, x_{i-1}, 1) \to F_i(x_1, \ldots, x_{i-1})$
- a quantifier $Q_i = \forall$ is translated as the clause
  $F_{i+1}(x_1, \ldots, x_{i-1}, 0) \wedge F_{i+1}(x_1, \ldots, x_{i-1}, 1) \to F_i(x_1, \ldots, x_{i-1})$
- finally, we add the clause $F_1() \wedge t_0 \neq t_1 \to \mathit{false}$.

It is now easy to see that the expansion $\mathit{exp}(\mathscr{HC})$ of the Horn clauses coincides with the result of expanding all quantifiers in ϕ. By Theorem 5.5.1, unsatisfiability of the expansion is equivalent to solvability of the set of Horn clauses. □

Theorem 5.6.2. *Semantic solvability of recursion-free Horn clauses over the constraint language of Booleans is PSPACE-complete.*

In combination with Lemma 5.6.1, it suffices to show that solvability is in PSPACE. This is done by encoding a set $\mathscr{HC}$ of Horn clauses into a QBF formula that is equivalent to the full expansion $\mathit{exp}(\mathscr{HC})$, but only grows linearly in the size of $\mathscr{HC}$.

The following lemma implies that solvability of recursion-free Horn clauses over the theory of Booleans is PSPACE-complete:

Lemma 5.6.3 (Succinct expansion)**.** *Let $\mathscr{HC}$ be a finite, recursion-free set of Horn clauses. If the underlying constraint language provides quantifiers, in (deterministic) linear time a formula $\mathit{sexp}(\mathscr{HC})$ can be extracted that is equivalent to $\mathit{exp}(\mathscr{HC})$. The number of quantifier alternations in $\mathit{sexp}(\mathscr{HC})$ is at most two times the number of relation symbols in $\mathscr{HC}$.*

Proof. We assume that the Horn clauses are connected, i.e., the $\to_{\mathscr{HC}}$-graph consists of a single connected component. Further, we assume that the first-order variables in any two clauses in $\mathscr{HC}$ are disjoint. The encoding of Horn clause as a QBF formula is then defined by the following algorithm in pseudo-code. The algorithm maintains a list *quantifiers* of quantifiers that have to be added in front of the formula.

quantifiers $\leftarrow \epsilon$, *checksRequired* $\leftarrow \emptyset$

function ENCODE($\mathscr{HC}$)
  Order clauses $\mathscr{HC}$ in topological order, starting from clauses with head *false*

matrix ← ENCODEBODIES($\{C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow \mathit{false} \in \mathcal{HC}\}, \epsilon$)
remaining ← $\{C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t}) \in \mathcal{HC}\}$
while *remaining* $\neq \emptyset$ **do**
  Pick first clause $C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t}) \in \mathcal{HC}$ in topological order
  nextClauses ← $\{c \in \mathcal{HC} \mid$ head symbol of c is $p\}$
  remaining ← *remaining* \ *nextClauses*
  for $i \leftarrow 1, \ldots,$ *arity*(p) **do**
    Create fresh variable x_i
    quantifiers ← *quantifiers* . $\forall x_i$
  guard ← *false*
  for $(f, p(\bar{s})) \in$ *requiredChecks* **do** ▷ Checks with symbol p
    guard ← *guard* $\vee$ ($f \wedge \bar{s} = \langle x_1, \ldots, x_n \rangle$)
  matrix ← *matrix* $\wedge$ (*guard* $\rightarrow$ ENCODEBODIES(*nextClauses*, $\langle x_1, \ldots, x_n \rangle$))
 return *quantifiers* . *matrix*

function ENCODEBODIES(*clauses*, $\bar{s}$)
  result ← *false*
  for $C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t}) \in$ *clauses* **do**
    quantifiers ← *quantifiers* . $\exists fv(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t}))$
    for $i \leftarrow 1, \ldots, n$ **do**
      Create fresh Boolean flag f_i
      quantifiers ← *quantifiers* . $\exists f_i$
      checksRequired ← *checksRequired* $\cup \{(f_i, p_i(\bar{t}_i))\}$
    disjunct ← $\bar{t} = \bar{s} \wedge C \wedge f_1 \wedge \cdots \wedge f_n$
    result ← *result* $\vee$ *disjunct*
 return *result*

We illustrate the succinct encoding using an example. Consider the clauses

(C1) r(X,Y) ← Y = X + 1
(C2) r(X,Y) ← Y = X + 2
(C3) s(X,Z) ← r(X, Y) ∧ r(Y, Z)
(C4) false ← s(X, Z) ∧ X >= 0 ∧ Z <= 0

The formula resulting from the succinct encoding is:

∃ x0, x1, f1 . ∀ x3, x4.
∃ x5, x6, x7, f2, f3 . ∀ x10, x11.
∃ x12, x13, x14, x15.
(C4) x1 >= 0 ∧ 0 >= x0 ∧ f1 ∧
  ((f1 ∧ x1 = x3 ∧ x0 = x4) →
(C3) (x7 = x3 ∧ x6 = x4 ∧ f2 ∧ f3)) ∧
  (((f2 ∧ x7 = x10 ∧ x5 = x11) ∨
    (f3 ∧ x5 = x10 ∧ x6 = x11)) →

$$(C1) \qquad ((x13 = x10 \wedge x12 = x11 \wedge x12 = x13 + 1) \vee$$
$$(C2) \qquad (x15 = x10 \wedge x14 = x11 \wedge x14 = x15 + 2)))$$

□

In terms of program analysis the Theorem 5.6.2 states that the Boolean programs with no recursive calls are PSPACE-complete. The result for Boolean programs is proved in [GY13] using reduction to the reachability analysis problem in transition systems over finite vectors of Boolean variables. The programs that use more expressive Constraint languages than Booleans lead to a significant increase in the complexity of solving Horn clauses. We will next consider the programs over Presburger arithmetic.

Theorem 5.6.4. *Semantic solvability of recursion-free Horn clauses over the constraint language of quantifier-free Presburger arithmetic is co-NEXPTIME-complete.*

Proof. Looking for upper bounds, it is easy to see that solvability of Horn clauses is in co-NEXPTIME for any constraint language with satisfiability problem in NP (for instance, quantifier-free Presburger arithmetic). This is because the size of the expansion $exp(\mathscr{HC})$ is at most exponential in the size of $\mathscr{HC}$. We can show the lower bound by simulating exponential-time-bounded Turing machines (possibly non-deterministic, with binary tape) to recursion-free Horn clauses over quantifier-free Presburger arithmetic.

Following [HMU01], with simplifications, A Turing machine $M = (Q, \delta, q_0, F)$ is defined by

- a finite non-empty set Q of states,
- an initial state $q_0 \in Q$,
- a final state $f \in Q$,
- a transition relation $\delta \subseteq ((Q \setminus \{f\}) \times \{0,1\}) \times (Q \times \{0,1\} \times \{L,R\})$.

Wlog, we assume that $Q = \{0,1,\ldots,f\} \subseteq \mathbb{Z}$ and $q_0 = 0$.

We define a relation symbol $step(q,l,r,q',l',r')$ to represent single execution steps of the machine. The parameters l,r,l',r' represent the tape, which is encoded as non-negative integers; the bits in the binary representation of the integers are the contents of the tape cells. l is the tape left of the head, r the tape right of the head. The least-significant bit of r is the tape cell at the head position. l',r' are the corresponding post-state variables after one execution step.

A tuple $(q,b,q',b',R) \in \delta$ (moving the tape to the right) is represented by a clause

$$step(q,\ x,\ b+2y,\ q',\ b'+2x,\ y)$$

where x, y are the implicitly universally quantified variables of the clause, and q, b, q', b' concrete numeric constants. Figure 5.6 depicts how the l and r variables are changed when the Turing machine moves to right.

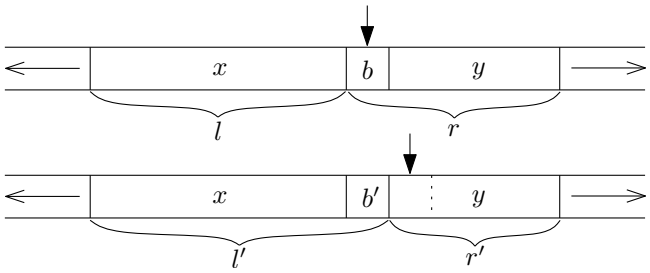


Figure 5.6: Moving the head of Turing machine to right and replacing the previous value

Similarly, a tuple $(q, b, q', b', L) \in \delta$ is encoded as

$$0 \leq x \leq 1 \rightarrow \mathit{step}(q,\ x+2y,\ b+2z,\ q',\ y,\ x+2b'+4z)$$

To represent termination, we add a clause $\mathit{step}(f, x, y, f, x, y)$, implying that the machine will stay in the final state f forever.

We then introduce n further clauses to model an execution sequence of length 2^n:

$$\begin{aligned}
\mathit{step}(x,y,z,x',y',z') \wedge \mathit{step}(x',y',z',x'',y'',z'') &\rightarrow \mathit{step}^1(x,y,z,x'',y'',z'') \\
\mathit{step}^1(x,y,z,x',y',z') \wedge \mathit{step}^1(x',y',z',x'',y'',z'') &\rightarrow \mathit{step}^2(x,y,z,x'',y'',z'') \\
&\cdots \\
\mathit{step}^{n-1}(x,y,z,x',y',z') \wedge \mathit{step}^{n-1}(x',y',z',x'',y'',z'') &\rightarrow \mathit{step}^n(x,y,z,x'',y'',z'')
\end{aligned}$$

The final clauses expresses that the Turing machine does not terminate within 2^n steps, when started with the initial tape t: $\mathit{step}^n(0,0,t,f,x,y) \rightarrow \mathit{false}$.

Clearly, the expansion $\mathit{exp}(\mathscr{HC})$ of the resulting set $\mathscr{HC}$ of Horn clauses is unsatisfiable (i.e., $\mathscr{HC}$ can be solved) if and only if no execution of the Turing machine, starting with the initial tape t, terminates within 2^n steps.

□

5.6.1 The Complexity of Different Classes of Horn Clauses

The lower bounds in Theorem 5.6.2 and Theorem 5.6.4 hinge on the fact that sets of Horn clauses can contain shared relation symbols in bodies. Neither result holds if we restrict attention to body-disjoint Horn clauses, which correspond to disjunctive interpolation as introduced in Sect. 5.4. Since the expansion $\mathit{exp}(\mathscr{HC})$ of body-disjoint Horn clauses is linear

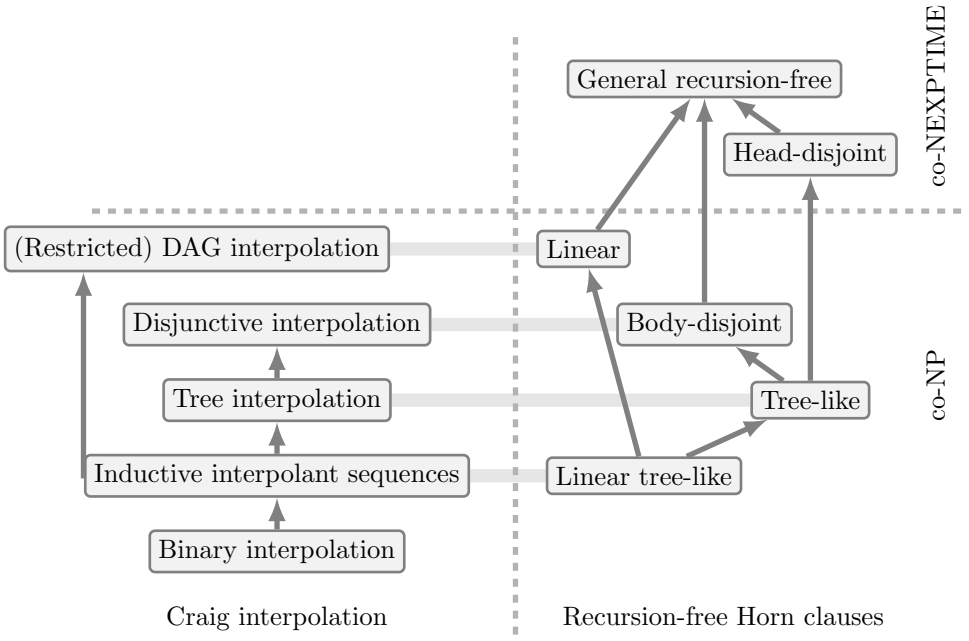


Figure 5.7: Relationship between different forms of Craig interpolation, and different fragments of recursion-free Horn clauses. An arrow from A to B expresses that problem A is (strictly) subsumed by B. The complexity classes "co-NP" and "co-NEXPTIME" refer to the problem of checking solvability of Horn clauses over quantifier-free Presburger arithmetic.

in the size of the set of Horn clauses, also solvability can be checked efficiently:

Theorem 5.6.5. *Semantic solvability of a set of body-disjoint Horn clauses, and equivalently the existence of a solution for a disjunctive interpolation problem, is in co-NP when working over the constraint languages of Booleans and quantifier-free Presburger arithmetic.*

Proof. A body-disjoint set of clauses $\mathscr{HC}$ is solvable if and only if the expansion $exp(\mathscr{HC})$ is unsatisfiable. Thanks to the body-disjointness property the expansion is polynomial. The expansion $exp(\mathscr{HC})$ is a disjunction of Presburger arithmetic formulae each of which linear in the size of the problem and in NP. So the whole disjunction $exp(\mathscr{HC})$ is in NP and the set $\mathscr{HC}$ belongs to co-NP. □

Body-disjoint Horn clauses are still expressive: they can directly encode acyclic control-flow graphs, as well as acyclic unfolding of many simple recursion patterns.

Theorem 5.6.6. *Semantic solvability of recursion-free linear Horn clauses over the constraint language of quantifier-free Presburger arithmetic is in co-NP.*

Proof. A set $\mathscr{HC}$ of recursion-free linear Horn clauses is solvable if and only if the expansion $exp(\mathscr{HC})$ is unsatisfiable. For linear clauses, $exp(\mathscr{HC})$ is a disjunction of (possibly)

exponentially many formulae, each of which is linear in the size of $exp(\mathscr{HC})$. Consequently, satisfiability of $exp(\mathscr{HC})$ is in NP, and unsatisfiability in co-NP. □

Corollary 5.6.7. *Semantic solvability of recursion-free head-disjoint Horn clauses over the constraint language of quantifier-free Presburger arithmetic is co-NEXPTIME-hard.*

The proof given in Theorem 5.6.4 for co-NEXPTIME-hardness of recursion-free Horn clauses over quantifier-free Presburger arithmetic can be adapted to only require head-disjoint clauses. This is because a single execution step of a non-deterministic Turing machine can be expressed as quantifier-free Presburger formula. □

Figure 5.7 summarizes the complexity results of this section. It depicts the correspondence between different classes of recursion-free Horn clauses and the interpolation problem.

5.7 From Recursion-free Horn Clauses to Well-founded Clauses

It is natural to ask whether the considerations of the last sections also apply to clauses that are not Horn clauses (i.e., clauses that can contain multiple positive literals), provided the clauses are "recursion-free." Is it possible, like for Horn clauses, to compute solutions of recursion-free clauses in general by means of computing Craig interpolants?

To investigate the situation for clauses that are not Horn, we first have to generalise the concept of clauses being recursion-free: the definition provided in Sect. 5.2, formulated with the help of the dependence relation $\rightarrow_{\mathscr{HC}}$, only applies to Horn clauses. For non-Horn clauses, we instead choose to reason about the absence of infinite propositional resolution derivations. Because the proposed algorithms for solving recursion-free sets of Horn clauses all make use of *exhaustive expansion* or *inlining*, i.e., the construction of all derivations for a given set of clauses, the requirement that no infinite derivations exist is fundamental.

Somewhat surprisingly, we observe that all sets of clauses without infinite derivations have the shape of Horn clauses, up to renaming of relation symbols. This means that procedures handling Horn clauses cover all situations in which we can hope to compute solutions with the help of Craig interpolation.

Since constraints and relation symbol arguments are irrelevant for this observation, the following results are entirely formulated on the level of propositional logic:

- a propositional *literal* is either a Boolean variable p, q, r (positive literals), or the negation $\neg p, \neg q, \neg r$ of a Boolean variable (negative literals).
- a propositional *clause* is a disjunction $p \vee \neg q \vee p$ of literals. The multiplicity of a literal is important, i.e., clauses could alternatively be represented as multi-sets of literals.
- a *Horn clause* is a clause that contains at most one positive literal.

- given a set $\mathscr{HC}$ of Horn clauses, we define the dependence relation $\rightarrow_{\mathscr{HC}}$ on Boolean variables by setting $p \rightarrow_{\mathscr{HC}} q$ if and only if there is a clause in $\mathscr{HC}$ in which p occurs positively, and q negatively (like in Sect. 5.2). The set $\mathscr{HC}$ is called *recursion-free* if $\rightarrow_{\mathscr{HC}}$ is acyclic.

We can now generalise the notion of a set of clauses being "recursion-free" to non-Horn clauses:

Definition 5.7.1. *A set $\mathscr{C}$ of propositional clauses has the* termination property *if no infinite sequence $c_0, c_1, c_2, c_3, \ldots$ of clauses exists, such that*

- *$c_0 \in \mathscr{C}$ is an input clause, and*
- *for each $i \geq 1$, the clause c_i is derived by means of binary resolution from c_{i-1} and an input clause, using the rule*

$$\frac{C \vee p \qquad D \vee \neg p}{C \vee D}.$$

Lemma 5.7.1. *A finite set $\mathscr{HC}$ of Horn clauses has the termination property if and only if it is recursion-free.*

Proof. "$\Leftarrow$" The acyclic dependence relation $\rightarrow_{\mathscr{HC}}$ induces a strict well-founded order $<$ on Boolean variables: $q \rightarrow_{\mathscr{HC}} p$ implies $p < q$. The order $<$ induces a well-founded order $\ll$ on Horn clauses:

$$\begin{aligned} (p \vee C) \ll (q \vee D) &\Leftrightarrow p > q \text{ or } (p = q \text{ and } C <_{ms} D) \\ C \ll (q \vee D) &\Leftrightarrow \mathit{true} \\ C \ll D &\Leftrightarrow C <_{ms} D \end{aligned}$$

where C, D only contain negative literals, and $<_{ms}$ is the (well-founded) multi-set extension of $<$ [DM79].

It is easy to see that a clause $C \vee D$ derived from two Horn clauses $C \vee p$ and $D \vee \neg p$ using the resolution rule is again Horn, and $(C \vee D) \ll (C \vee p)$ and $(C \vee D) \ll (D \vee \neg p)$. The well-foundedness of $\ll$ implies that any sequence of clauses as in Def. 5.7.1 is finite.

"$\Rightarrow$" If the dependence relation $\rightarrow_{\mathscr{HC}}$ has a cycle, we can directly construct a non-terminating sequence $c_0, c_1, c_2, \ldots$ of clauses. □

Definition 5.7.2 (Renamable-Horn [Lew78])**.** *If A is a set of Boolean variables, and $\mathscr{C}$ is a set of clauses, then $r_A(\mathscr{C})$ is the result of replacing in $\mathscr{C}$ every literal whose Boolean variable is in A with its complement. $\mathscr{C}$ is called* renamable-Horn *if there is some set A of Boolean variables such that $r_A(\mathscr{C})$ is Horn.*

Example The set $\mathscr{C} = \{(x \vee \neg y \vee \neg z), (y \vee z), (\neg x)\}$ is renamable-Horn with respect to the renaming set $A = \{x, y\}$. The set $\mathscr{C} = \{(x \vee y \vee z), (\neg x \vee \neg y \vee \neg z)\}$ is not renamable-Horn. □

Checking the existence of the renaming set A can be done in linear time [Lew78].

Theorem 5.7.2. *If a finite set $\mathscr{C}$ of clauses has the termination property, then it is renamable-Horn.*

Proof. Suppose $\mathscr{C}$ is formulated over the (finite) set $p_1, p_2, \ldots, p_n$ of Boolean variables. We construct a graph (V, E), with $V = \{p_1, p_2, \ldots, p_n, \neg p_1, \neg p_2, \ldots, \neg p_n\}$ being the set of all possible literals, and $(l, l') \in E$ if and only if there is a clause $\neg l \vee l' \vee C \in \mathscr{C}$ (that means, a clause containing the literal l', and the literal l with reversed sign).[3]

The graph (V, E) is acyclic. To see this, suppose there is a cycle $l_1, l_2, \ldots, l_m, l_{m+1} = l_1$ in (V, E). Then there are clauses $c_1, c_2, \ldots, c_m \in \mathscr{C}$ such that each c_i contains the literals $\neg l_i$ and l_{i+1}. We can then construct an infinite sequence $d_0 = c_1, d_1, d_2, \ldots$ of clauses, where each d_i (for $i > 1$) is obtained by resolving d_{i-1} with $c_{(i \bmod m)+1}$, contradicting the assumption that $\mathscr{C}$ has the termination property.

Since (V, E) is acyclic, there is a strict total order $<$ on V that is consistent with E, i.e., $(l, l') \in E$ implies $l < l'$.

Claim: if $p < \neg p$ for every Boolean variable $p \in \{p_1, p_2, \ldots, p_n\}$, then $\mathscr{C}$ is Horn.

Proof of the claim: suppose a non-Horn clause $p_i \vee p_j \vee C \in \mathscr{C}$ exists (with $i \neq j$). Then $(\neg p_i, p_j) \in E$ and $(\neg p_j, p_i) \in E$, and therefore $\neg p_i < p_j$ and $\neg p_j < p_i$. Then also $\neg p_i < p_i$ or $\neg p_j < p_j$, contradicting the assumption that $p < \neg p$ for every Boolean variable p.

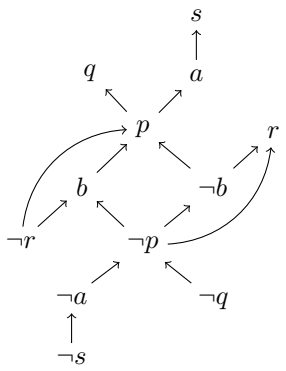
In general, choose $A = \{p_i \mid i \in \{1, \ldots, n\}, \neg p_i < p_i\}$, and consider the set $r_A(\mathscr{C})$ of clauses. The set $r_A(\mathscr{C})$ is Horn, since changing the sign of a Boolean variable $p \in A$ has the effect of swapping the nodes $p, \neg p$ in the graph (V, E). Therefore, the new graph (V, E') has to be compatible with a strict total order $<$ such that $p < \neg p$ for every Boolean variable p, satisfying the assumption of the claim above. □

Example We consider the following set of clauses:

$$\mathscr{C} = \{\neg a \vee s,\ a \vee \neg p,\ p \vee \neg b,\ b \vee p \vee r,\ \neg p \vee q\}$$

By constructing all possible derivations, it can be shown that the set has the termination property. The graph (V, E), as constructed in the proof, is:

[3] This graph could equivalently be defined as the implication graph of the 2-sat problem introduced in [Lew78], as a way of characterising whether a set of clauses is Horn.



A strict total order that is compatible with the graph is:

$$\neg s < \neg q < \neg r < \neg a < \neg p < b < \neg b < r < p < q < a < s$$

From the order we can read off that we need to rename the variables $A = \{s, q, r, a, p\}$ in order to obtain a set of Horn clauses:

$$r_A(\mathscr{C}) = \{a \vee \neg s,\ \neg a \vee p,\ \neg p \vee \neg b,\ b \vee \neg p \vee \neg r,\ p \vee \neg q\}$$

□

5.8 Model Checking with Recursive Horn Clauses

Whereas *recursion-free* Horn clauses generalise the concept of Craig interpolation, solving *recursive* Horn clauses corresponds to the verification of general programs with loops, recursion, or concurrency features [GLPR12]. Procedures to solve recursion-free Horn clauses can serve as a building block within model checking algorithms for recursive Horn clauses [GLPR12], and are used to construct or refine abstractions by analysing spurious counterexamples. In particular, our disjunctive interpolation can be used for this purpose, and offers a high degree of flexibility due to the possibility to analyse counterexamples combining multiple execution traces. We illustrate the use of disjunctive interpolation within a predicate abstraction-based algorithm for solving Horn clauses. Our model checking algorithm is similar in spirit to the procedure in [GLPR12], and is explained in Sect. 5.8.1.

And/or trees of clauses. For sake of presentation, in our algorithm we represent counterexamples (i.e., recursion-free sets of Horn clauses) in the form of and/or trees labelled with clauses. Such trees are defined by the following grammar:

$$\mathit{AOTree} ::= \mathit{And}(h, \mathit{AOTree}, \ldots, \mathit{AOTree}) \mid \mathit{Or}(\mathit{AOTree}, \ldots, \mathit{AOTree})$$

where h ranges over (possibly recursive) Horn clauses. We only consider well-formed trees, in which the children of every *And*-node have head symbols that are consistent with the body literals of the clause stored in the node, and the sub-trees of an *Or*-node all have the same head symbol. And/or trees are turned into body-disjoint recursion-free sets of clauses by renaming relation symbols appropriately.

Example The clauses in Figure 5.2 on page 43 can be represented by the following and/or tree (referring to clauses in Fig. 5.1).

$$And\Big((4),\ Or\big(And((1)),\ And((2),\ And((1))),\ And((3), And((1)))\big)\Big)$$

□

Solving and/or dags. Counterexamples extracted from model checking problems often assume the form of and/or *dags*, rather than and/or *trees*. Since and/or-dags correspond to Horn clauses that are not body-disjoint, the complexity-theoretic results of the last section imply that it is in general impossible to avoid the expansion of and/or-dags to and/or-trees; there are, however, various effective techniques to speed-up handling of and/or-dags (related to the techniques in [LQL12]). We highlight two of the techniques used in the interpolation engine Princess [BKRW11].

1) counterexample-guided expansion expands and/or-dags lazily, until an unsatisfiable fragment of the fully expanded tree has been found; such a fragment is sufficient to compute a solution. Counterexamples are useful in two ways: they can determine which or-branch of an and/or-dag is still satisfiable and has to be expanded further, but also whether it is necessary to create further copies of a shared subtree.

2) and/or dag restructuring factors out common sub-dags underneath an *Or*-node, making the and/or-dag more tree-like.

5.8.1 A Predicate Abstraction-based Model Checking Algorithm

Our model checking algorithm is in Fig. 5.8, and similar in spirit as the procedure in [GLPR12]; it has been implemented in the model checker Eldarica.[4] Solutions for Horn clauses are constructed in disjunctive normal form by building an abstract reachability graph over a set of given predicates. When a counterexample is detected (a clause with consistent body literals and head *false*), a theorem prover is used to verify that the counterexample is genuine; spurious counterexamples are eliminated by generating additional predicates by means of disjunctive interpolation.

In Fig. 5.8, $\Pi : \mathscr{R} \to \mathscr{P}_{\text{fin}}(Constr)$ denotes a mapping from relation symbols to the current (finite) set of predicates used to approximate the relation symbol. Given a (possibly recursive)

[4] http://lara.epfl.ch/w/eldarica

set $\mathscr{HC}$ of Horn clauses, we define an *abstract reachability graph* (ARG) as a hyper-graph (S, E). A hypergraph is a generalization of a graph in which an edge can connect any number of vertices.

- $S \subseteq \{(p, Q) \mid p \in \mathscr{R}, Q \subseteq \Pi(p)\}$ is the set of nodes, each of which is a pair consisting of a relation symbol and a set of predicates.
- $E \subseteq S^* \times \mathscr{HC} \times S$ is the hyper-edge relation, with each edge being labelled with a clause. An edge $E(\langle s_1, \ldots, s_n \rangle, h, s)$, with $h = (C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H) \in \mathscr{HC}$, implies that
  - $s_i = (p_i, Q_i)$ and $B_i = p_i(\bar{t}_i)$ for all $i = 1, \ldots, n$, and
  - $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$, where we write $Q_i[\bar{t}_i]$ for the conjunction of the predicates Q_i instantiated for the argument terms t_i.

An ARG (S, E) is called *closed* if the edge relation represents all Horn clauses in $\mathscr{HC}$. This means, for every clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow H) \in \mathscr{HC}$ and every sequence $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$ of nodes one of the following properties holds:

- $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \mathit{false}$, or
- there is an edge $E(\langle (p_1, Q_1), \ldots, (p_n, Q_n) \rangle, C, s)$ such that $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$.

Lemma 5.8.1. *A set $\mathscr{HC}$ of Horn clauses has a closed ARG (S, E) if and only if $\mathscr{HC}$ is syntactically solvable.*

Proof. "$\Rightarrow$": Define each relation symbol p as the disjunction $\bigvee_{(p,Q) \in S} \bigwedge Q$. Since S is closed under the edge relation, this yields a solution for the set $\mathscr{HC}$ of Horn clauses.

"$\Leftarrow$": Suppose $\mathscr{HC}$ is syntactically solvable, with each relation symbol p being mapped to the constraint C_p. We define the predicate abstraction $\Pi(p) = \{C_p\}$, and construct the ARG with nodes $S = \{(p, C_p)\}$, and the maximum edge relation E, which is closed. □

The function EXTRACTCEX extracts an and/or-tree representing a set of counterexamples, which can be turned into a recursion-free body-disjoint set of Horn clauses, and solved as described in Sect. 5.4.1. In general, the tree contains both conjunctions (from clauses with multiple body literals) and disjunctions, generated when following multiple hyper-edges (the case $|T| > 1$). Disjunctions make it possible to eliminate multiple counterexamples simultaneously. The algorithm is parametric in the precise strategy used to compute counterexamples (represented as non-deterministic choice in the pseudo code). The strategies we evaluated in the experiments (shown in the next section) are the following.

$S := \emptyset,\ E := \emptyset,\ \Pi := \{p \mapsto \emptyset \mid p \in \mathscr{R}\}$ ▷ Empty graph, no predicates
function CONSTRUCTARG
  while *true* **do**
    pick clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow H) \in \mathscr{HC}$
      and nodes $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$
      such that $\neg\exists s.\ (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, s) \in E$
      and $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \not\models \mathit{false}$
    if no such clauses and nodes exist **then return** $\mathscr{HC}$ is solvable
    if $H = \mathit{false}$ **then** ▷ Refinement needed
      $\mathit{tree} := \mathit{And}(h, \text{EXTRACTCEX}(p_1, Q_1), \ldots, \text{EXTRACTCEX}(p_n, Q_n))$
      if *tree* is unsatisfiable **then**
        extract disjunctive interpolant from *tree*, add predicates to Π
        delete part of (S, E) used to construct *tree*
      else return $\mathscr{HC}$ is unsolvable, with counterexample trace *tree*
    else ▷ Add edge to ARG
      then $H = p(\bar{t})$
      $Q := \{\phi \in \Pi(p) \mid \{C\} \cup Q_1 \cup \ldots \cup Q_n \models \phi\}$
      $e := (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, (p, Q))$
      $S := S \cup \{(p, Q)\},\ E := E \cup \{e\}$

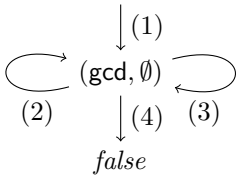
function EXTRACTCEX(*root* : *S*) ▷ Extract disjunctive interpolation problem
  pick $\emptyset \neq T \subseteq E$ with $\forall e \in T.\ e = (\_, \_, \mathit{root})$
  return $\mathit{Or}\{\mathit{And}(h, \text{EXTRACTCEX}(s_1), \ldots, \text{EXTRACTCEX}(s_n)) \mid (\langle s_1, \ldots, s_n\rangle, h, \mathit{root}) \in T\}$

Figure 5.8: Algorithm for construction of abstract reachability graphs.

TI extraction of a single counterexamples with minimal depth
(which means that disjunctive interpolation reduces to **T**ree **I**nterpolation), and

DI simultaneous extraction of all counterexamples with minimal depth
(so that genuine **D**isjunctive **I**nterpolation is used).

Example We consider the Horn clauses given in Figure 5.1 in page 43. Starting with an empty predicate map Π, the function CONSTRUCTARG will construct the reachability graph shown on the right (edges are labelled with the clauses from Fig. 5.1). Since *false* is reachable, function EXTRACTCEX will be called to extract a counterexample; possible results of executing EXTRACTCEX include: □



$$\mathit{tree}_1 = \mathit{And}\big((4),\ \mathit{And}((1))\big),$$
$$\mathit{tree}_2 = \mathit{And}\big((4),\ \mathit{Or}(\mathit{And}((1)),\ \mathit{And}((2), \mathit{And}((1))),\ \mathit{And}((3), \mathit{And}((1))))\big)$$

The counterexample tree_2 corresponds to the clauses shown in Fig. 5.2. Elimination of this counterexample with the help of disjunctive interpolation yields the predicates discussed in Example 5.4 in page 56, which are sufficient to construct a closed ARG.

5.8.2 Global Model Checking with Disjunctive Interpolation

We have also implemented a simpler "global" algorithm that approximates each relation symbol globally with a single conjunction of inferred predicates. In the same spirit of the previous algorithm the global algorithm also uses a Cartesian abstraction by treating the predicates of each state separately. The two algorithms behave similarly in our experience, with the global one occasionally slower, but conceptually simpler. The global approach requires more complex predicates to be discovered through interpolation and therefore relies more heavily on interpolation. What allowed us to use a simpler algorithm is precisely the more general form of the interpolation. This shows another advantage of more expressive interpolation: the simplicity of verification algorithms we can build on top of it.

We use the following notation.

- $R = \{r_0, \ldots, r_n\}$: Names of relation variables
- $P = \{p_0, \ldots, p_m\}$: Set of all predicates on program variables
- $\Pi : R \to 2^P$: Mapping from relation variables to the set of predicates that can be used to approximate it
- $\alpha : R \to 2^P$: Mapping from relation variables to the predicates that currently approximate it
- Φ set of Horn constraints

An Abstract Reachability Graph (ARG) $G = \langle S, e, w \rangle$ is an AND/OR graph

- $S = I \cup L$ is a set of nodes partitioned to abstraction and concrete nodes. The abstraction nodes $I \subseteq R \times 2^P$ are labeled with a relation variable (a name from R) and a set of predicates that hold at that particular point. The concrete nodes are labeled with an interpreted relation and their outdegree is 0.
- $e \subseteq (S \times 2^S)$ is the transition relation
- $w \subseteq S \times S$ is the subsumption relation

The algorithm ConstructGlobalARG in Figure 5.9 picks an unsatisfied constraint from the given set of constraints. It then tries to satisfy the picked up constraint by removing the predicates that are assigned to the right hand side of the constraint until it satisfies. The algorithm adds the constraint to the reachability graph by attaching the conjuncts in the body of the constraint to the literal in the head.

The refinement procedure Refinement returns a mapping from relation variables to a set of predicates.

```
d = ⟨S = ∅, e = ∅, w = ∅⟩                                   ▷ Empty graph
procedure AddToGraph(head, body)                            ▷ Adding edges to reachability graph
    for c ← conjunct(body) do
        n ← (if c ∈ R then (c, α(c)) else c)                ▷ Make a vertex for the literal
        S ← S ∪ n
        e ← e ∪ (head, n)                                   ▷ Edge from head to new child
    S ← S ∪ {head}

function ConstructGlobalARG(Φ, Π)                           ▷ Construction of the reachability graph
    α ← Π
    while ∃(A → r) ∈ Φ. ⊭ α[A → r] do                       ▷ Existence of unsatisfied rule
        for p ← α(r). ⊭ (α[A] → p) do
            α(r) ← α(r) \ {p}                               ▷ Remove predicate p that does not hold
        parent ← (r, α(r))
        if ∃(r, ψ) ∈ S.(ψ ⊨ α(r)) then                      ▷ Add subsumption edge
            w ← w ∪ {((r, ψ), parent)}
        AddToGraph(parent, A)
    if ∃(A → false ∈ Φ). ⊭ α(A) → false then                ▷ The rule corresponding to error
        AddToGraph((r_error, true), A)
    else
        declare SAFE
    return d

function Solve(Φ)
    ∀r ∈ R.Π(r) ← false                                      ▷ Add the predicate false everywhere
    T ← ConstructGlobalARG(Φ)
    preds ← refinement(T)                                   ▷ Refining the graph
    while preds ≠ ∅ do
        Π ← Π ∪ preds
        T ← ConstructGlobalARG(Φ)
        preds ← refinement(T)
    declare UNSAFE
```

Figure 5.9: Pseudo-code for the global algorithm ARG

5.9 Experimental Evaluation

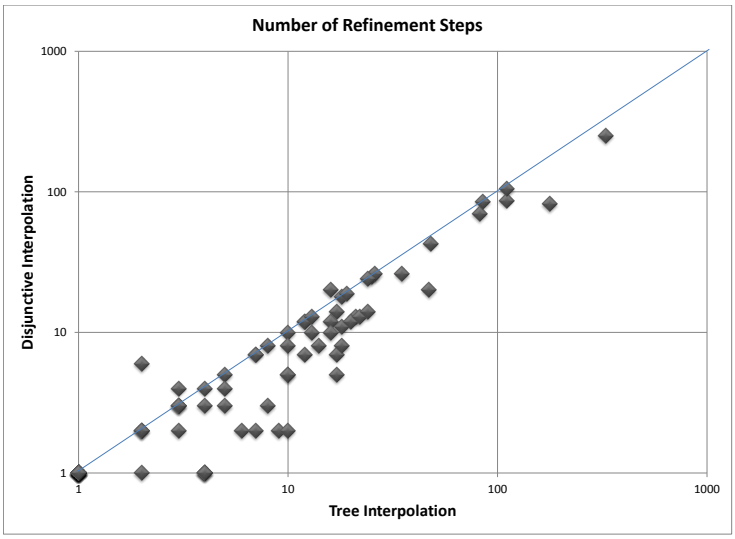
We have evaluated our algorithm on a set of benchmarks in integer linear arithmetic. Figure 5.10 provides the running time of Eldarica on the benchmarks with respect to an Intel ®Xeon ®2.66GHz machine with 16 GB of RAM. The benchmarks are from different resources. The benchmarks (a), (b), (c), (d), (f) and (I) are from the NTS library [HKG+12] translated into Horn clauses [5]. These are recursive algorithms, benchmarks extracted from programs with singly-linked lists, benchmarks from the NECLA static analysis suite, verification conditions for programs with arrays, C programs with asynchronous procedure calls translated using

[5] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/

Model	Time [s] TI	DI	HSF	Z3
(a) Recursive Models				
addition (C)	0.52	0.22	0.27	0.04
binarysearch (C)	0.38	0.26	0.22	0.02
identity (C)	0.39	0.46	0.23	0.03
mccarthy91 (C)	1.68	1.25	0.24	0.03
mccarthy92 (C)	0.27	0.25	-	0.05
merge (C)	0.32	0.83	0.33	0.04
palindrome (C)	0.07	0.08	0.24	0.04
parity (C)	0.09	0.11	-	-
remainder (C)	0.05	0.06	-	-
triple (C)	0.24	0.21	-	0.03
(b) Examples from L2CA [BBH+06]				
bubblesort (E)	2.64	1.76	0.85	0.26
insdel (E)	0.04	0.05	0.29	0.02
insertsort (E)	0.4	0.4	0.41	0.08
listcounter (E)	0.07	0.07	0.29	0.02
listreversal (C)	0.5	0.5	0.46	0.14
listreversal (E)	0.18	0.17	0.34	0.05
mergesort (E)	0.92	0.92	0.68	0.16
selectionsort (E)	1.15	1.15	0.68	0.15
(c) NECLA benchmarks				
inf1 (E)	0.17	0.12	0.38	0.02
inf4 (E)	0.37	0.46	0.34	0.05
inf6 (C)	0.31	0.32	0.31	0.04
inf8 (C)	0.52	0.57	0.61	0.06
(d) Verification conditions for array programs [BHI+09]				
rotation_vc.1 (C)	0.37	0.36	0.39	0.13
rotation_vc.2 (C)	0.53	0.49	0.59	0.18
rotation_vc.3 (C)	0.32	0.33	0.33	0.02
rotation_vc.1 (E)	0.21	0.22	0.37	0.07
split_vc.1 (C)	1.15	1.09	1	0.36
split_vc.2 (C)	-	-	0.68	0.18
split_vc.3 (C)	0.61	0.59	0.46	0.03
split_vc.1 (E)	1.08	1.08	0.99	0.2
(e) Benchmarks from HSF [GLPR12]				
amebsa (C)	59.41	59.67	-	3.98
amotsa (C)	0.53	0.51	-	0.23
choldc (C)	-	-	33.72	5.78
crank (C)	-	-	-	0.75
cyclic (C)	7.5	7.68	11.26	6.76
lop (C)	5.24	2.61	-	9.68
pzextr (C)	2.2	2.17	-	0.28
qrdcmp (C)	18.21	18.93	-	0.31
qrsolv (C)	0.88	0.86	15.64	-
rsolv (C)	82.02	80.46	2.08	0.82
tridag (C)	3.56	5.32	-	0.16

Model	Time [s] TI	DI	HSF	Z3
(f) Examples from [GM12]				
f_rec (E)	5.03	8.38	2.24	1.09
h1 (E)	0.19	0.15	0.35	0.09
h1h2 (E)	0.35	0.46	0.45	0.1
nch (C)	1.71	4.63	12.36	-
plb_simple (C)	0.33	0.33	0.42	0.06
plb_simple (E)	0.66	1.09	0.53	0.26
server.manual	0.02	0.02	0.22	0.02
simple (E)	0.19	0.15	0.33	0.07
test0 (C)	0.46	0.5	0.42	0.14
test0 (E)	0.19	0.16	0.35	0.08
test1 (C)	0.82	1.24	0.53	0.14
test1 (E)	1.11	0.81	0.57	0.19
test2_1 (E)	0.85	0.94	0.48	0.28
test2_2 (E)	0.5	0.55	0.5	0.16
test2 (C)	0.92	0.57	0.55	0.28
test4 (C)	0.76	0.7	0.61	0.16
test4 (E)	0.38	0.38	0.47	0.13
test_recursion (E)	229.5	266.16	20.06	7.55
wrpc	0.05	0.04	0.44	0.02
wrpc.manual	0.68	0.51	0.23	0.15
(g) Control Flow and Integer Variables [Bey13]				
test_locks_10.c (C)	130.11	53.82	1171.02	0.98
test_locks_11.c (C)	475.84	149.96	-	1.24
test_locks_12.c (C)	-	518.32	-	1.54
test_locks_5.c (C)	5.32	2.93	5.73	0.22
test_locks_6.c (C)	4.05	5.82	17.15	0.31
test_locks_7.c (C)	6.33	8.22	50.08	0.43
test_locks_8.c (C)	14.28	16.58	145.09	0.58
test_locks_9.c (C)	40.3	26.8	406.17	0.76
(h) Benchmarks from [Kin]				
ch-triangle-location-nr.1 (E)	0.54	2.6	0.43	0.1
delauny-edge-flipping.7 (E)	0.09	0.09	0.26	0.01
fortune-full.10 (C)	27.57	26.45	-	19.63
fortune-full-nonrobust.17 (E)	0.57	0.75	6.69	0.37
giftwrapping.25 (E)	0.2	0.2	0.69	0.28
graham.27 (C)	15.29	15.25	816.52	15.91
graham-scan-full.31 (C)	18.01	50.14	788	7.39
incremental-2lists.37 (E)	25.6	3.59	132.9	4.52
incremental.35 (C)	51.68	36.22	-	13.35
point-location-nr.49 (E)	0.15	0.16	0.36	0.13
slow-hull.55 (E)	0.18	0.19	0.41	0.21
(i) VHDL models from [SV07]				
asfifoFE (C)	488.26	991.32	-	77.46
asfifoStatus (C)	10.6	10.41	2.78	2.7
counter (C)	0.1	0.13	0.25	0.04
register (C)	0.05	0.05	0.23	0.03

Figure 5.10: Benchmarks for model checking Horn clauses. The letter after the model name distinguishes **C**orrect from models with a reachable **E**rror state. "-" indicates timeout (limit is half an hour) or error in the tool. **Disclaimer:** The table is the author's assessment with the most recent versions of the tools provided by the developers. The intention is by no means to assert the superiority of any particular tool over another. The main purpose here is to give an insight to the emerging trend of Horn clause verification.



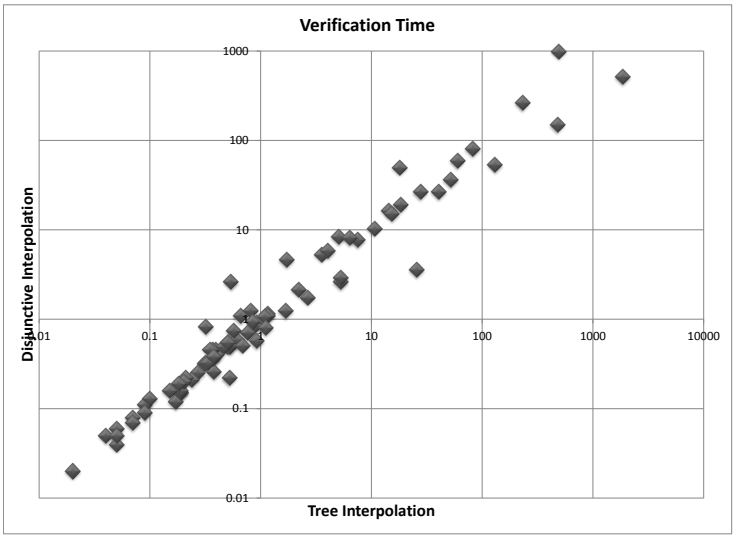


Figure 5.11: Comparison of the number of required refinement steps, and the runtime (in seconds), for the case of single counterexamples (**TI**) and simultaneous extraction of all minimal-depth counterexamples (**DI**). The diagrams use a logarithmic scale to better visualize the results.

the approach of [GM12] and VHDL models of circuits. The benchmarks (e) are taken from the HSF library of benchmarks [GLPR12]. The benchmarks (g) are extracted from the control flow and integer variables program of the International Competition on Software Verifica-

tion [Bey13]. The tool FLATA-C [6] extracted the benchmarks of (g) from the original programs. The benchmarks (h) are Horn clauses from the same resource [Bey13].

The experiments show comparable verification times and performance for tree interpolation and disjunctive interpolation runs, tending towards better times for the latter. The total time spent in verification for all the benchmarks of Figure 5.10 putting all together was around 60 minutes for **TI** and around 40 minutes for **DI**. Studying the results more closely, we observed that **DI** consistently led to a smaller number of abstraction refinement steps (the scatter plot in Fig. 5.11); this indicates that **DI** is indeed able to eliminate multiple counterexamples simultaneously, and to rapidly generate predicates that are useful for abstraction. The experiments also showed that there is a trade-off between the time spent generating predicates, and the quality of the predicates. In **TI**, on average 4% of the verification is used for predicate generation (interpolation), while with **DI** 24% is used; in some of the benchmarks from [GM12], this led to the phenomenon that **DI** was slower than **TI**, despite fewer refinement steps. This may also become better as we make further improvements to our prototype implementation of disjunctive interpolation.

We also compared our results to the performance of HSF [7] and Z3 [8], two of the state-of-the-art verification engines capable of solving Horn clauses. Z3 was faster on average.

5.10 Towards a Library of Interpolation Benchmarks

In order to support the development of interpolation engines, Horn solvers, and verification systems, we have started to collect relevant benchmarks of recursion-free Horn clauses, categorised according to the classes determined in the previous sections.[9] The benchmarks have been extracted from runs of the model checker Eldarica which processes systems of (usually recursive) Horn clauses by iteratively solving recursion-free unwindings. In this way a set of recursion-free systems of Horn clauses can be syntesized for each recursive verification problem. For each recursive verification problem, in this way a set of recursion-free systems of Horn clauses (of varying size) can be synthesised. The benchmarks can be used to evaluate both Horn solvers and interpolation engines, according to the correspondence in Fig. 5.7.

At the moment, our benchmarks are extracted from the verification problems in [RHK13], and formulated over the constraint language of linear integer arithmetic. In the future, it is planned to also include other constraint languages, including rational arithmetic and the theory of arrays. The benchmarks are stored in SMT-LIB 2 format [BST10]. All of the benchmarks can be solved by Eldarica, and by the Horn solving engine in Z3 [HB12].

The current number of available benchmarks is provided in the table below. In order to

[6] http://www-verimag.imag.fr/FLATA-C.html
[7] http://www7.in.tum.de/tools/hsf/
[8] http://z3.codeplex.com/
[9] http://lara.epfl.ch/w/horn-nonrec-benchmarks

evaluate the effectiveness of ordinary SMT quantifier handling on the Horn benchmarks, we also ran Z3 [dMB08] (without the Horn engine) on the benchmarks. The results show that two engines that have knowledge of Horn clauses, Eldarica and Z3-Horn, solve all of the benchmarks (with Z3's well-engineered engine faster than Eldarica). In contrast, when Z3 is used without Horn extension, as a prover for quantified formulas, the default quantifier instantiation strategy proves to be too weak to solve all benchmarks.

Class	**#Benchmarks**	**Average Time**		**%Solved Z3**
		Eldarica	Z3	(without Horn)
General recursion-free	541	0.6	0.1	80%
Head-disjoint	991	0.1	0.1	85%
Linear	971	0.7	0.1	32%
Linear tree-like	1993	0.4	0.1	55%

Figure 5.12: Average time in seconds of solving each category by Eldarica and Z3. The last column shows the percentage handled by Z3 without the Horn engine. Time-out was set to 60 seconds.

6 Compositional Verification of Timed Systems Using Horn Clauses

> I have made this letter longer than usual, only because I have not had the time to make it shorter.
>
> Blaise Pascal

In formal verification of real-time systems not only the computation matters but the clock time and the exact moment of computing a result is crucial. Taking the timing characteristic of systems into account makes complications in proving the correctness of real-time systems. With the emerging and the growth in the complexity of embedded systems the verification of time properties of embedded software is becoming more vital. In many domains time automata [AD94] are the predominant way in describing timed system. The theory of timed automata is a well established theory for modeling and verifying real-time systems, with many applications both in an industrial and academic context. In the model of timed automata there are a set of automata running in parallel with each other. Two automata can synchronize at certain rendezvous points. Modeling systems using timed automata enables us to analyze a variety of relevant properties including schedulability, worst-case execution time of concurrent systems, interference, as well as functional properties. Although model checking of timed automata has been studied extensively during the last two decades, the scalability of these tools remains a concern, in particular when applied to problems of industrial size. When verifying networks of timed automata, the size of the combined state space can be a limiting factor. Many tools and model checking techniques for timed automata have been studied extensively during the last two decades. The UPPAAL [LPY97] tool is one of the most successful tools in both academy and research.

In this chapter we present an interpolation-based predicate abstraction framework which encodes timed automata as sets of Horn clauses, with the help of Owicki-Gries and Rely-Guarantee encoding schemes. For modeling a network of timed automata we used the well-known classical approaches of Owicki-Gries [OG76] along with Rely-Guarantee [Jon83]. The latter approach has the advantage to be compositional. Modeling parallel systems using

these two approaches were first described in [GLPR12]. The difference of the model in this chapter with the method of [GLPR12] lies in the presence of time and synchronization between threads.

We have extended the front-end of our verification engine Eldarica [HKG+12] to translate a system of timed automata in UPPAAL format into a set of Horn clauses. This is particularly interesting to study the difference of Owicki-Gries and Rely-Guarantee for timed systems in terms of performance and the size of the generated constraints. We show the feasibility of our approach through benchmarks and case studies and present a comparison between the different concurrency encodings.

6.1 The Theory of Timed Automata

Timed Automata. Let C be a set of (real-valued) clock variables, let X be a finite set of data variables, and let Σ be a finite set of actions. A *timed automaton* is a tuple $A = \langle N, l^0, E, Inv \rangle$, where N_i is a finite set of locations, $l^0 \in N$ is the initial location, $E \subseteq N \times \mathscr{G}(C) \times \Sigma \times 2^C \times Constr \times N$ is a finite set of transition edges, and $Inv : N \to \mathscr{G}(C)$ maps every location to an *invariant*. An edge $\langle l, g, a, r, \phi(X, X'), l' \rangle \in E$ represents discrete transitions of the automaton from location l to location l'. A transition is enabled if guard g is satisfied, performs the action a and the data transition $\phi(X, X')$, and resets the clocks in the set r to zero. Σ is defined as the set $\{local\} \cup \{a!, a? \mid a \in \mathscr{C}\}$, for a set $\mathscr{C}$ of communication channels (the label *local* is usually left out in diagrams). The set $\mathscr{G}(C)$ of clock guards consists of conjunctions of atomic guards, defined by $g ::= (x \bowtie n) \mid (x \bowtie y + n) \mid true \mid g \wedge g$, where $x, y \in C$, $n \in \mathbb{N}$, and $\bowtie_c \in \{<, \leq, =, \geq, >\}$.

A timed automaton is well-formed if the invariant $Inv(l_0)$ of the initial location is satisfied by mapping all clocks $c \in C$ to 0. (Note that the invariant $Inv(l)$ of every location l is automatically convex.)

Networks of Timed Automata. A network of timed automata is the parallel composition $A_1 || A_2 || \cdots || A_n$ of n timed automata, where each $A_i = \langle N_i, l_i^0, E_i, Inv_i \rangle$. A network of automata can either execute *local* transitions of any of the automata, or *synchronising* transitions of two (different) automata performing complementary actions $a!$ and $a?$.

Operational Semantics. The state of a network $A_1 || A_2 || \cdots || A_n$ of n automata is defined as a pair $s = (l, u, v)$ where $l = \langle l_1, \ldots, l_n \rangle$ is a vector of locations for each automaton, $u : C \to \mathbb{R}$ is an assignment of reals to clock variables, and $v : X \to U$ is an assignment of individuals to data variables. We use the notation $u + d$, for some $d \in \mathbb{R}_+$, to express the updated clock assignment $(u + d)(x) = u(x) + d$. Further, for a set of clocks $r \subseteq C$, the expression $[r \mapsto 0]u$ denotes the assignment that maps each clock in r to 0, and all other clocks to their previous value. We write $u \models g$ to express that a clock guard g holds for a clock assignment u, and

$v, v' \models \phi$ to express that a data transition $\phi = \phi(X, X')$ is satisfied by the valuations v, v'. The formula $Inv(l) = Inv_1(l_1) \wedge \cdots \wedge Inv_n(l_n)$ denotes the conjunction of the active invariants of all automata.

The initial state of the network of automata is defined as $s_{init} \stackrel{\text{def}}{=} (l^0, u^0, v^0)$, where l^0 is a vector of initial locations, u^0 is a function mapping all clocks to 0, and v^0 is a function mapping all data variables to some default value. State transitions for networks of timed automata can be categorized as follows:

- *Delay transitions:* $(l, u, v) \rightarrow (l, u + d, v)$,
  provided that $(u + d) \models Inv(l)$.
- *Local transitions*: $(l, u, v) \rightarrow (l[l'_i/i], [r \mapsto 0]u, v')$,
  if there is a transition $\langle l_i, g, local, r, \phi, l'_i \rangle \in E_i$ such that
  $u \models g$, $v, v' \models \phi$, and $([r \mapsto 0]u) \models Inv(l[l'_i/i])$.
- *Synchronising transitions*: $(l, u, v) \rightarrow (l[l'_i/i, l'_j/j], [r \cup r' \mapsto 0]u, v'')$,
  if $i \neq j$, and there are transitions
  $\langle l_i, g, a!, r, \phi, l'_i \rangle \in E_i$ and $\langle l_j, g', a?, r', \phi', l'_j \rangle \in E_j$ such that
  $u \models g$, $([r \mapsto 0]u) \models g'$, $([r \cup r' \mapsto 0]u) \models Inv(l[l'_i/i, l'_j/j])$, and
  there is v' such that $v, v' \models \phi$ and $v', v'' \models \phi'$.

Safety. We are concerned with the verification of *safety properties,* which assert that some undesired behaviour of a system never occurs. For sake of brevity, we assume that safety properties have been translated upfront to *unreachability*, i.e., to the claim that certain *error states* in a system are not reachable. A safety specification for a network $A_1||A_2||\cdots||A_n$ of automata is therefore a pair $\langle i, l_{err} \rangle$, and states that location $l_{err} \in N_i$ of automaton A_i is unreachable.

6.2 Reasoning about Concurrent Programs

A Hoare triple has the form $\{P\}S\{Q\}$ denoting that whenever the precondition P holds before the initiation of the program S, if the program executes and terminates the postcondition Q will be true. Owicki and Gries [OG76] generalised the Hoare triples to reason about concurrent programs. In this approach a standard Hoare proof is carried out for each thread. The parallel execution rule requires that the proof of each thread does not interfere with the proof of the others.

$$\frac{\{P_1\}S_1\{Q_1\} \qquad \{P_2\}S_2\{Q_2\} \qquad \begin{array}{l} S_1 \text{ does not interfere with the proof of } S_2 \\ S_2 \text{ does not interfere with the proof of } S_1 \end{array}}{\{P_1 \wedge P_2\}S_1||S_2\{Q_1 \wedge Q_2\}}\ \text{OWICKI-GRIES}$$

Assume that the thread S_1 has the proof outline $\{P_1 = \alpha_1\}c_1\{\alpha_2\}c_2 \cdots \{\alpha_n\}c_n\{\alpha_{n+1} = Q_1\}$ and the thread S_2 has the proof outline $\{P_2 = \beta_1\}c'_1\{\beta_2\}c'_2 \cdots \{\beta_m\}c'_m\{\beta_{m+1} = Q_2\}$ in which the statements c_i $(1 \leq i \leq n)$ and c'_j $(1 \leq j \leq m)$ are all atomic commands. Proving the interference freedom of S_1 from the execution of S_2 requires proving the following $n \times m$ formulae.

$$\forall \alpha_i \in \{\alpha_1, \cdots, \alpha_n\} \forall c'_j \in \{c'_1, \cdots, c'_m\} \qquad \{\alpha_i \wedge \beta_j\}\ c'_j\ \{\alpha_i\} \tag{6.1}$$

The approach of Owicki-Gries is not scalable and does not have compositionality. Jones [Jon83] introduced the rely-guarantee method, a compositional version of the Owicki-Gries system. The program specification in the Rely-Guarantee approach is of the form $\{P, R\}S\{G, Q\}$ where P is the precondition, R is the rely relation which includes any environment transition, G is the guarantee relation which includes any transition of S and Q is the postcondition. The parallel rule of Rely-Guarantee does not impose the heavy computation of the Owicki-Gries approach.

$$\frac{\{P_1, R_1\}S_1\{G_1, Q_1\} \qquad \{P_2, R_2\}S_2\{G_2, Q_2\}}{\{P1 \wedge P_2, R_1 \vee R_2\}S_1||S_2\{G_1 \wedge G_2, Q_1 \wedge Q_2\}}\text{RELY-GUARANTEE}$$

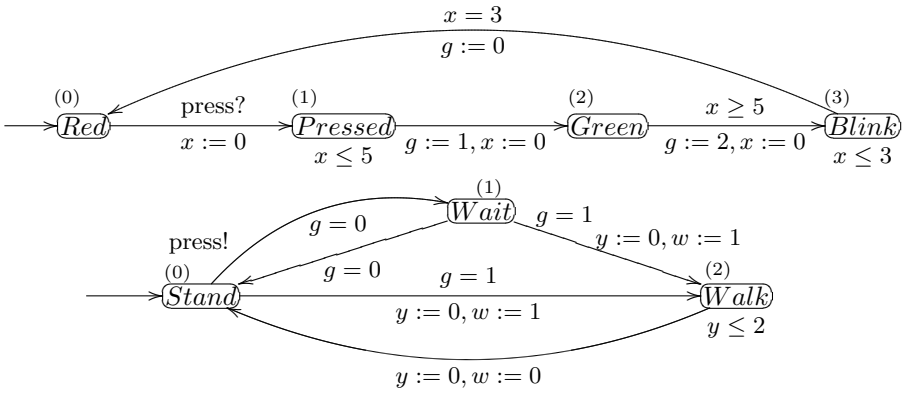
In the proof approach using Rely-Guarantee we give a proof of the post-condition and guarantee conditions of each program assuming that the rely condition holds. Then we prove that the guarantee condition of every other thread implies the rely condition of a program.

6.3 Motivating Example

The first automaton in Figure 6.1 models a crosswalk light, the second one models a pedestrian. The light automaton is initially *Red* and is waiting for the button to be pressed. After the pedestrian presses the button the light sets the local clock x to 0 and goes to the *Pressed* state. It takes the light 5 time units to change from *Pressed* to *Green*. The light resets the local clock to 0 and assigns the global variable g to 1 when moving to *Green*. When the total time of being *Green* is greater than or equal to 5 the light goes to *Blink* and during the transition it sets the local clock x to 0 and g to 2. Blinking lasts for 3 time units and then the light loops back to its initial state.

The pedestrian initially presses the button, or, if the light is already green ($g = 1$) he goes to *Walk*. After pressing the button the pedestrian waits in *Wait* until either the light goes green (moving to *Walk* when $g = 1$) or he gives up (moving back to *Stand* when $g = 0$). The walking takes less than or equal to 2 time units and the pedestrian returns back to the initial state or enters the error state if the light is not green anymore after the two 2 time units.

Figure 6.1 shows the local clauses for the automata. We assign a relation symbol to each automaton in the system, L for light and P for pedestrian. The first parameter of the relation symbols is a global clock c that we use to put all the clocks in the system in sync. All the global and local clocks of the automata are measured with respect to c. If the automaton L resets its clock x to 0 we represent it with $c - x = 0$. After the first parameter c we place the clocks



$L(c,y,x,w,g,t_0,t_1)$	$\leftarrow$	$y=c, x=c, w=0, g=0, t_0=0, t_1=0$
$L(c,y,x,w,g_2,0,t_1)$	$\leftarrow$	$L(c,y,x,w,g_1,3,t_1), c-x=3, g_2=0$
$L(c,y,x_2,w,g_2,2,t_1)$	$\leftarrow$	$L(c,y,x_1,w,g_1,1,t_1), c-x_2=0, g_2=1$
$L(c,y,x_2,w,g_2,3,t_1)$	$\leftarrow$	$L(c,y,x_2,w,g_1,2,t_1), c-x_1\geq 5, c-x_2\leq 3, c-x_2=0, g_2=2$
$P(c,y,x,w,g,t_0,t_1)$	$\leftarrow$	$y=c, x=c, w=0, g=0, t_0=0, t_1=0$
$P(c,y_2,x,w_2,g,t_0,0)$	$\leftarrow$	$P(c,y_1,x,w_1,g,t_0,2), c-y_2=0, w_2=0$
$P(c,y,x,w,g,t_0,0)$	$\leftarrow$	$P(c,y,x,w,g,t_0,1), g=0$
$P(c,y_2,x,w_2,g,t_0,2)$	$\leftarrow$	$P(c,y_1,x,w_1,g,t_0,1), g=1, c-y_2<2, c-y_2=0, w_2=1$
$P(c,y_2,x,w_2,g,t_0,2)$	$\leftarrow$	$P(c,y_1,x,w_1,g,t_0,0), g=1, c-y_2<2, c-y_2=0, w_2=1$
$P(c,y_2,x,w_2,g,t_0,1)$	$\leftarrow$	$P(c,y_1,x,w_1,g,t_0,0), g=0$
$false \leftarrow L(c,y,x,w,g,t_0,t_1), P(c,y,x,w,g,t_0,t_1), g=0, w=1$		

Figure 6.1: Pedestrian Crossing Light and the local clauses of each automaton

and variables as parameters. In order to ensure completeness we share the local states of the automata globally, so the parameters of the relation symbols are all the same. The last parameters t_0 and t_1 are place holders showing the current state of automata. Initially all the clocks are assigned to c, the variables to 0, the place holders to the initial states. The local clauses track the changes of the variables and clocks throughout the non-synchronizing transitions. Figure 6.2 represent the encoding of timed transitions into Horn clauses.

In order to prove safety we have to verify it is impossible to have $g=0$ and $w=1$ at the same time. For this purpose we add the clause with *false* in head and the erroneous state in body to the system to ensure the incorrect behavior is excluded from the solution. By augmenting the

$L(c_2,y,x,w,g,1,t_1)$	$\leftarrow$	$L(c_1,y,x,w,g,1,t_1), c_2\geq c_1, c_2-x\leq 5$
$L(c_2,y,x,w,g,3,t_1)$	$\leftarrow$	$L(c_1,y,x,w,g,3,t_1), c_2\geq c_1, c_2-x\leq 3$
$L(c_2,y,x,w,g,t_0,t_1)$	$\leftarrow$	$L(c_1,y,x,w,g,t_0,t_1), c_2\geq c_1, t_0\neq 1, t_0\neq 3$
$P(c_2,y,x,w,g,t_0,2)$	$\leftarrow$	$P(c_1,y,x,w,g,t_0,2), c_2\geq c_1, c_2-y\leq 2$
$P(c_2,y,x,w,g,t_0,t_1)$	$\leftarrow$	$P(c_1,y,x,w,g,t_0,t_1), c_2\geq c_1, t_1\neq 2$

Figure 6.2: Encoding of timed transitions in Figure 6.1

clauses of any of the Owicki-Gries or Rely-Guarantee approaches we obtain the solution to the system.

$P(c, y, x, w, g, t_0, t_1) \leftarrow \neg(w = 1, g \neq 1)$

The assertion here does not depend on the behavior of Light.

$L(c, y, x, w, g, t_0, t_1) \leftarrow true.$

6.4 Modeling Local Transitions

In the next sections, we assume that a network $A_1||A_2||\cdots||A_n$ of automata has been given, together with a safety specification $\langle i, l_{err}\rangle$. We will introduce two methods to verify the specification $\langle i, l_{err}\rangle$ by means of encoding into a system of Horn clauses. The two approaches differ in the way concurrency is encoded: Sect. 6.5.1 gives an encoding on the basis of the *Owicki-Gries* methodology, while Sect. 6.5.2 leverages the *Rely-Guarantee* approach. Both methods share the way in which local transitions of automata are handled, which is introduced in the next paragraphs.

The Horn clauses are formulated over a constraint language that combines three domains: (i) the theory of (rational/real) *difference-bound constraints,* which is used to encode clocks C, transition guards, and invariants, (ii) the finite sets N_i of *control locations* of the individual automata, and (iii) the language used for data variables X and transition constraints. We defer details how these theories are handled to Sect. 6.6, and for the time being assume a constraint language that is rich enough to capture (i)–(iii); in particular, we assume that every control location $l \in N_i$ constitutes a term in the language.

As a further simplifying assumption, we do not consider clocks shared between multiple automata. That means, we assume that the set C of clocks is partitioned into n disjoint sets, $C = C_1 \uplus C_2 \uplus \cdots \uplus C_n$, in such a way that each automaton A_i only refers to clocks in the set C_i.[1]

Relation symbols. We associate a relation symbol $\mathsf{P}_i(\mathsf{c}, \bar{\mathsf{u}}, \bar{\mathsf{x}}, \bar{\mathsf{l}})$ with each automaton A_i in a network; the argument vectors of the relation symbol represent:

- the global clock c, representing absolute time during system execution;
- the clock variables $\bar{\mathsf{u}}$, with $|\bar{\mathsf{u}}| = |C|$;
- the data variables $\bar{\mathsf{v}}$, with $|\bar{\mathsf{v}}| = |X|$;
- the location variables $\bar{\mathsf{l}}$, with $|\bar{\mathsf{l}}| = n$.

[1] Our encodings can be extended to the general case, at the cost of an increase in notational complexity.

a)

$$Local(A,\langle i,l_{err}\rangle) \stackrel{\text{def}}{=} \mathcal{HC}_{init} \cup \mathcal{HC}_{te} \cup \mathcal{HC}_{trans} \cup \mathcal{HC}_{err}$$

$$\mathcal{HC}_{init} \stackrel{\text{def}}{=} \{\mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \leftarrow \bar{\mathsf{u}} = \mathsf{c} \wedge \mathsf{l}_0 = l_0^0 \wedge \cdots \wedge \mathsf{l}_n = l_n^0\}_{1\le i\le n}$$

$$\mathcal{HC}_{te} \stackrel{\text{def}}{=} \{\mathsf{P}_i(\mathsf{c}',\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \mathsf{c}' \ge \mathsf{c} \wedge \mathsf{l}_i = m \wedge Inv_i(m)(\bar{\mathsf{u}})\}_{1\le i\le n,\, m\in L_i}$$

$$\mathcal{HC}_{trans} \stackrel{\text{def}}{=} \{\mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge\ \tau_i(e,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')\}_{\substack{1\le i\le n,\\ e=\langle l_1,g,a,r,\phi,l_2\rangle\in E_i}}$$

$$\mathcal{HC}_{err} \stackrel{\text{def}}{=} \{false \leftarrow \mathsf{P}_1(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \cdots \wedge \mathsf{P}_n(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \mathsf{l}_e = l_{err}\}$$

b)

$$OG(A,\langle i,l_{err}\rangle) \stackrel{\text{def}}{=} Local(A,\langle i,l_{err}\rangle) \cup \mathcal{HC}_{inter-loc} \cup \mathcal{HC}_{sync} \cup \mathcal{HC}_{inter-sync}$$

$$\mathcal{HC}_{inter-loc} \stackrel{\text{def}}{=} \{\mathsf{P}_j(\mathsf{c},\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}), \mathsf{P}_j(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \tau_i(e,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')\}_{\substack{1\le i,j\le n,\\ e=\langle l_1,g,a,r,\phi,l_2\rangle\in E_i}}$$

$$\mathcal{HC}_{sync} \stackrel{\text{def}}{=} \{(\bigcup_{1\le i,j\le n} Sync_{i,j}(e_i,e_j)) \cup (\bigcup_{1\le i,j\le n} Sync_{j,i}(e_j,e_i))\}_{\substack{i\ne j,\\ e_i=\langle l_{i_1},g_i,a!,r_i,\phi_i,l_{i_2}\rangle\in E_i,\\ e_j=\langle l_{j_1},g_j,a?,r_j,\phi_j,l_{j_2}\rangle\in E_j}}$$

$$Sync_{p,q}(e_p,e_q) \stackrel{\text{def}}{=} \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \mathsf{P}_j(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \tau_p(e_p,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \wedge \tau_q(e_q,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')$$

$$\mathcal{HC}_{inter-sync} \stackrel{\text{def}}{=} \{\mathsf{P}_k(\mathsf{c},\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge\ \mathsf{P}_j(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge\ \mathsf{P}_k(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \tau_i(e_i,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \wedge \tau_j(e_j,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')\}_{\substack{1\le i,j\le n,\\ e_i=\langle l_{i_1},g_i,a!,r_i,\phi_i,l_{i_2}\rangle\in E_i,\\ e_j=\langle l_{j_1},g_j,a?,r_j,\phi_j,l_{j_2}\rangle\in E_j}}$$

c)

$$RG(A,\langle i,l_{err}\rangle) \stackrel{\text{def}}{=} Local(A,\langle i,l_{err}\rangle) \cup \mathcal{HC}_{rely} \cup \mathcal{HC}_{guarantee}$$

$$\mathcal{HC}_{rely} \stackrel{\text{def}}{=} \{\mathsf{P}_i(\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge\ \mathsf{E}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \wedge \tau_i(e,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')\}_{\substack{1\le i\le n,\\ e=\langle l_1,g,a,r,\phi,l_2\rangle\in E_i}}$$

$$\mathcal{HC}_{guarantee} \stackrel{\text{def}}{=} \{\mathsf{E}_j(\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') \leftarrow \mathsf{P}_i(\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}}) \wedge \tau_i(e,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}')\}_{\substack{1\le i,j\le n,\\ e=\langle l_1,g,a,r,\phi,l_2\rangle\in E_i}}$$

Figure 6.3: $A = A_1||A_2||\cdots||A_n$ is a network of automata $A_i = \langle N_i, l_i^0, E_i, Inv_i\rangle$ with the safety specification $\langle i, l_{err}\rangle$. For a local transition $e = \langle l_1, g, a, r, \phi, l_2\rangle$ we define $\tau_i(e,\mathsf{c},\bar{\mathsf{u}},\bar{\mathsf{v}},\bar{\mathsf{l}},\mathsf{c}',\bar{\mathsf{u}}',\bar{\mathsf{v}}',\bar{\mathsf{l}}') = g(\mathsf{c}-\bar{\mathsf{u}}) \wedge \phi(\bar{\mathsf{v}},\bar{\mathsf{v}}') \wedge \mathsf{l}_i = l_1 \wedge \mathsf{l}_i' = l_2 \wedge \bigwedge_{j\ne i} \mathsf{l}_j' = \mathsf{l}_j \wedge \bigwedge_{a\in r} \mathsf{u}_a' = \mathsf{c} \wedge \bigwedge_{a\notin r} \mathsf{u}_a' = \mathsf{u}_a \wedge Inv_i(l_2)(\bar{\mathsf{u}})$. The figure depicts the Horn clauses for (a) local transitions, (b) Owicki-Gries and (c) Rely-Guarantee approach.

Following the operational semantics we encode different transitions of the network as Horn clauses. Figure 6.3 summarizes the Horn clause encoding for an automata.

There are three different types of local transition rules for an automaton: initialisation, time elapse and local transition. The set $\mathcal{HC}_{init}$ contains all the Horn clauses that initialise the system. $\mathcal{HC}_{te}$ contains the time elapse Horn clauses and $\mathcal{HC}_{trans}$ contains the Horn clauses describing local transitions and $\mathcal{HC}_{err}$ describes the error clause. In the initialisation encodings for each automaton we set all data variables to 0, clock variables to the global time c and all location variables to initial locations. The time elapse rule expresses that the

global time increases provided that the invariant for the automaton holds, all other variables (clock, data and location) stay the same. The local transition rule expresses that an automaton can transition from one state to another state provided that the transition guard g and the invariant in the next state hold $Inv_i(l')$. Finally, the error clause checks our property and asserts that if an automata is in an error state, this then implies *false*.

6.5 Interleaving and Concurrency Rules

6.5.1 Owicki-Gries Method

In [OG76] Owicki and Gries generalize the method of Hoare logic to reason about concurrent programs that communicate and synchronize on shared variables. In addition to local partial correctness of each automaton, the Owicki-Gries approach requires establishing the interference-freedom of proofs. To provide completeness the local variables are promoted to the global scope.

Figure 6.3(b) defines the set of Horn clauses required for interference-freedom and synchronization in a network of timed automata. In particular, $\mathscr{HC}_{inter-local}$ formalizes the encoding of the Owicki-Gries interference-freedom rule into Horn clauses.

In particular, $\mathscr{HC}_{inter-local}$ formalizes the encoding of the Owicki-Gries interference- freedom rule into Horn clauses for local transitions. Intuitively, this rule requires an automaton $P_j(c, \bar{u}, \bar{v}, \bar{l})$ to be *invariant* for all other local transitions τ_i of other automata in the network. In timed automata we have only binary hand shaking and the clauses $\mathscr{HC}_{sync}$ demonstrate the effect of a synchronization between two automata. The interference-free clauses $\mathscr{HC}_{inter-sync}$ are required for the synchronizing transitions. The additional clauses required by Owicki-Gries for the example in Section 6.3 are given in Figure 6.4.

$P(c,y,x,w,g_2,3,t_1)$	$\leftarrow$	$P(c,y,x,w,g_1,0,t_1), L(c,y,x,w_1,g_1,0,t_1), c-x=3, g_2=0$
$P(c,y,x_2,w,g_2,2,t_1)$	$\leftarrow$	$P(c,y,x_1,w,g_1,1,t_1), L(c,y,x_1,w_1,g_1,1,t_1), c-x_2=0, g_2=1$
$P(c,y,x_2,w,g_2,3,t_1)$	$\leftarrow$	$P(c,y,x_1,w,g_1,2,t_1), L(c,y,x_1,w,g_1,2,t_1),$ $c-x_1\geq 5, c-x_2\leq 3, c-x_2=0, g_2=2$
$L(c,y_2,x,w_2,g,t_0,0)$	$\leftarrow$	$L(c,y_1,x,w_1,g,t_0,2), P(c,y_1,x,w_1,g,t_0,2), c-y_2=0, w_2=0$
$L(c,y,x,w,g,t_0,0)$	$\leftarrow$	$L(c,y,x,w,g,t_0,1), P(c,y,x,w,g,t_0,1), g=0$
$L(c,y_2,x,w_2,g,t_0,2)$	$\leftarrow$	$L(c,y_1,x,w_1,g,t_0,1), P(c,y_1,x,w_1,g,t_0,1),$ $g=1, c-y_2<2, c-y_2=0, w_2=1$
$L(c,y_2,x,w_2,g,t_0,2)$	$\leftarrow$	$L(c,y_1,x,w_1,g,t_0,0), P(c,y_1,x,w_1,g,t_0,0),$ $g=1, c-y_2<2, c-y_2=0, (w_2==1)$
$P(c,y,x_2,w,g,1,1)$	$\leftarrow$	$P(c,y,x_1,w,g,0,0), L(c,y,x_1,w,g,0,0), g=0, c-x_2=0$
$L(c,y,x_2,w,g_2,1,1)$	$\leftarrow$	$P(c,y,x_1,w,g_1,0,0), L(c,y,x_1,w,g_1,0,0), g=0, c-x_2=0$

Figure 6.4: Owicki-Gries interference-freedom and synchronization clauses

6.5.2 Rely-Guarantee Method

$$
\begin{array}{ll}
L & (c, y_2, x_2, w_2, g_2, press_2, t_{02}, t_{12}) \leftarrow L(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), \\
 & EL(c, y_1, y_2, x_1, x_2, w_1, w_2, g_1, g_2, press_1, press_2, t_{01}, t_{02}, t_{11}, t_{12}), t_{01} = t_{02} \\
P & (c, y_2, x_2, w_2, g_2, press_2, t_{02}, t_{12}) \leftarrow P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), \\
 & EP(c, y_1, y_2, x_1, x_2, w_1, w_2, g_1, g_2, press_1, press_2, t_{01}, t_{02}, t_{11}, t_{12}), t_{11} = t_{12} \\
\hline
EP & (c, y_1, y_1, x_1, x_1, w_1, w_1, g_1, g_1, press_1, press_1, t_{01}, t_{02}, t_{11}, t_{11}) \leftarrow \\
 & L(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{01} = 3, t_{02} = 0, g_1 = 0, c - x_1 = 3 \\
EP & (c, y_1, y_1, x_1, x_2, w_1, w_1, g_1, g_1, press_1, press_1, t_{01}, t_{02}, t_{11}, t_{11}) \leftarrow \\
 & L(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{01} = 1, t_{02} = 2, g_1 = 1, c - x_2 = 0 \\
EP & (c, y_1, y_1, x_1, x_2, w_1, w_1, g_1, g_1, press_1, press_1, t_{01}, t_{02}, t_{11}, t_{11}) \leftarrow \\
 & L(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{01} = 2, t_{02} = 3, g_1 = 2, c - x_2 = 0, c - x_1 \geq 5, c - x_2 \leq 3 \\
EP & (c, y_1, y_1, x_1, x_2, w_1, w_1, g_1, g_1, press_1, press_2, t_{01}, t_{02}, t_{11}, t_{11}) \leftarrow \\
 & L(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{01} = 0, t_{02} = 1, c - x_2 = 0, c - x_2 \leq 5, \\
 & press_1 \neq 0, press_1 \neq 1, press_2 = 0 \\
EL & (c, y_1, y_2, x_1, x_1, w_1, w_2, g_1, g_1, press_1, press_1, t_{01}, t_{01}, t_{11}, t_{12}) \leftarrow \\
 & P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{11} = 2, t_{12} = 0, c - y_2 = 0, w_2 = 0 \\
EL & (c, y_1, y_1, x_1, x_1, w_1, w_1, g_1, g_1, press_1, press_1, t_{01}, t_{01}, t_{11}, t_{12}) \leftarrow \\
 & P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{11} = 1, t_{12} = 0, g_1 = 0 \\
EL & (c, y_1, y_1, x_1, x_1, w_1, w_1, g_1, g_1, press_1, press_2, t_{01}, t_{01}, t_{11}, t_{12}) \leftarrow \\
 & P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), press_1 = 0, press_2 = 2, t_{11} = 0, t_{12} = 1, g_1 = 0 \\
EL & (c, y_1, y_2, x_1, x_1, w_1, w_2, g_1, g_1, press_1, press_1, t_{01}, t_{01}, t_{11}, t_{12}) \leftarrow \\
 & P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{11} = 1, t_{12} = 2, c - y_2 = 0, w_2 = 1, g_1 = 1, c - y_2 < 2 \\
EL & (c, y_1, y_2, x_1, x_1, w_1, w_2, g_1, g_1, press_1, press_1, t_{01}, t_{01}, t_{11}, t_{12}) \leftarrow \\
 & P(c, y_1, x_1, w_1, g_1, press_1, t_{01}, t_{11}), t_{11} = 0, t_{12} = 2, c - y_2 = 0, w_2 = 1, g_1 = 1, c - y_2 < 2
\end{array}
$$

Figure 6.5: Horn clauses for the Rely-Guarantee approach

The Rely-Guarantee method [Jon83] is a compositional way for proving the correctness of parallel programs with shared variables. This approach constructs two sets of constraints for each automaton. The "rely" constraints express the assumption of an automaton about the shared data among the automata. The "guarantee" constraints are the changes that an automaton makes on the shared data. Rely-Guarentee provides a more modular way of proof comparing to Owicki-Gries.

Figure 6.3(c) defines the Horn clause encoding for the rely-guarentee encoding *RG*. In the clauses we represent the environment of an automaton A_i with the relation symbol $\mathsf{E}_i(\mathsf{c}, \bar{\mathsf{u}}, \bar{\mathsf{v}}, \bar{\mathsf{l}}, \mathsf{c}', \bar{\mathsf{u}}', \bar{\mathsf{v}}', \bar{\mathsf{l}}')$ which shows the relation between the the previous unprimed values of the variables and clocks with the new primed values.

The rely clause for an automaton A_i asserts that the automaton stays in the same location with the variable and clock updates from the environment. A guarantee clause captures the effect of each transition of the an automaton A_i in the environment of all other automata A_j.

To model synchronization in this approach we designate a variable to each channel in the system. Similar to other variables in the system, the channel variables are initialized to

0. The sender party shows the interest of synchronization by assigning the corresponding channel variable to its identity. The identities are all non-zero. The receiver party accepts to synchronize by re-assigning the channel variable to 0. This is a pre-processing step and as such there is no explicit synchronization clause required for the rely-guarentee encoding. Figure 6.5 represents the rely and guarantee clauses for the running example in Section 6.3.

6.5.3 Modeling Parameterized Systems

A main advantage of our technique over the previous finite-state checkers is the ability to model parameterized systems in which there can be an arbitrary number of instances for an automaton. For modeling parameterized systems each automaton takes a unique identifier i in addition to its normal arguments $\mathsf{P}_i(\mathsf{i}, \mathsf{c}, \bar{\mathsf{u}}, \bar{\mathsf{x}}, \bar{\mathsf{l}})$. To be able to store all the variables in the arguments of an automaton we sacrifice the completeness of our mapping by putting only the local and global variables in the relation symbol of each automaton. If the verification successfully proves the correctness of the approximated model we can assure that original system was correct. Using this approach we were able to verify a variation of the parameterized version of the Fischer protocol [Lam87].

6.6 Evaluation

Benchmark	Rely-Guarantee	♯ Clauses	Owicki-Gries	♯ Clauses
Counter	8.6	25	2.9	22
Light	17.1	31	5.9	34
Pedestrian	32.6	45	9.0	39
Peterson Algorithm	16.4	37	5.2	34
Simple Array	6.8	21	5.2	16
Train Gate	56.9	119	5.5	120
Fischer	29.6	57	11.7	54

Figure 6.6: Execution time for proving the correctness of non-paramatrized benchmarks

We evaluated our tool on the benchmarks from UPPAAL [2] as well as some new benchmarks available from the webpage of Eldarica [3]. Figure 6.6 shows the elapsed time in verification of the benchmarks (seconds) along with clause sizes per benchmark.

The number of clauses increase quadratically w.r.t the number of states in the rely-guarentee encoding, while clauses increase cubicly in the owicki-gries encoding.

We find the current results of this approach promising, because they show that even if we

[2] http://www.uppaal.org/benchmarks/
[3] http://lara.epfl.ch/w/eldarica

largely decouple semantic modeling using Horn clauses from the infinite-state verification algorithm, we obtain a useful tool, with the additional benefit of supporting parametrized systems with an unknown number of concurrent components. Tools tailored for timed systems [LPY97, Yov97, Wan04] implement many specialized techniques to make verification more efficient; in the future we plan to explore to which extent these techniques can be generalized into general-purpose strategies for solving recursive Horn clauses.

7 Related Work

> Good programmers know what to write. Great ones know what to rewrite and reuse.
>
> Eric S. Raymond

Predicate abstraction has proved to be a rich and fruitful direction in automated verification of detailed properties of infinite-state systems [GS97, HJMM04]. In this thesis we presented techniques to improve the applicability of predicate abstraction engines. For convenience and clarification the related work section is divided into subsections based on the different achievements and goals of the thesis.

7.1 Counterexample-Guided Accelerated Abstraction

The pioneering work in [BL99] is, to the best of our knowledge, the first to propose a solution to the divergence problem in predicate abstraction. More recently, sufficient conditions to enforce convergence of refinement in predicate abstraction are given in [BPR02], but it remains difficult to enforce them in practice. A promising direction for ensuring completeness with respect to a language of invariants is parameterizing the syntactic complexity of predicates discovered by an interpolating *split prover* [JM06]. Because it has the flavor of invariant enumeration, the feasibility of this approach in practice remains to be further understood.

To alleviate relatively weak guarantees of refinement in predicate abstraction in practice, researchers introduced *path invariants* [BHMR07] that rule out a family of counterexamples at once using constraint-based analysis. Our CEGAAR approach is similar in the spirit, but uses acceleration [BIK10, FL02, Boi99] instead of constraint-based analysis, and therefore has complementary strengths. Acceleration naturally generates precise *disjunctive invariants*, needed in many practical examples, while constraint-based invariant generation [BHMR07] resorts to an ad-hoc unfolding of the path program to generate disjunctive invariants. Acceleration

can also infer expressive predicates, in particular modulo constraints, which are relevant for purposes such as proving memory address alignment.

The idea of generalizing spurious error traces was introduced also in [HHP09], by extending an infeasible trace, labeled with interpolants, into a finite interpolant automaton. The method of [HHP09] exploits the fact that some interpolants obtained from the infeasibility proof happen to be inductive w.r.t. loops in the program. In our case, given a spurious trace that iterates through a program loop, we *compute* the needed inductive interpolants, combining interpolation with acceleration.

The CEGAAR algorithm was introduced in [HIK+12]. The method that is probably closest to CEGAAR is proposed in [CFLZ08]. In this work the authors define *inductive interpolants* and prove the existence of effectively computable inductive interpolants for a class of affine loops, called *poly-bounded*. The approach is, however, limited to programs with one poly-bounded affine loop, for which initial and error states are specified. We only consider loops that are more restricted than the poly-bounded ones, namely loops for which transitive closures are Presburger definable. On the other hand, our method is more general in that it does not restrict the number of loops occurring in the path program, and benefits from regarding both interpolation and transitive closure computation as black boxes. The ability to compute closed forms of certain loops is also exploited in algebraic approaches [BHHK10]. These approaches can also naturally be generalized to perform useful over-approximation [AAGP11] and under-approximation.

The article [KLW13] uses acceleration to find the summary of a loop. It then adds the under-approximation of the loop behavior as an auxiliary path to the loop structure. The acceleration is static and it does not dynamically accelerate loops on demand. Whereas our method works for Presuburger Integer arithmetic their approach supports assignments to arrays and arbitrary conditional branching by computing quantified conditionals.

7.2 Disjunctive Interpolants

There is a long line of research on Craig interpolation methods, and generalised forms of interpolation tailored to verification. For an overview of interpolation in the presence of theories, we refer the reader to [CGS10, BKRW11]. Binary Craig interpolation for implications $A \rightarrow C$ goes back to [Cra57], was used on conjunctions $A \wedge B$ in [McM03], and generalised to inductive sequences of interpolants in [HJMM04, McM06]. The concept of tree interpolation, strictly generalising inductive sequences of interpolants, is presented in the documentation of the interpolation engine iZ3 and in [MR13]; the computation of tree interpolants by computing a sequence of binary interpolants is also described in [HHP10]. In this thesis we presented a new form of interpolation, *disjunctive interpolation* [RHK13], which is strictly more general than sequences of interpolants and tree interpolants. Our implementation supports Presburger arithmetic, including divisibility constraints [BKRW11], which is rarely supported by existing tools, yet helpful in practice [HIK+12].

A further generalisation of inductive sequences of interpolants are restricted DAG interpolants [AGC12a], which also include disjunctiveness in the sense that multiple paths through a program can be handled simultaneously. Disjunctive interpolants are incomparable in power to restricted DAG interpolants, since the former does not handle interpolation problems in the form of DAGs, while the latter does not subsume tree interpolation. A combination of the two kinds of interpolants ("disjunctive DAG interpolation") is strictly more powerful (and harder) than disjunctive interpolation, see Sect. 5.6 for a complexity-theoretic analysis.

We discussed techniques and heuristics to practically handle shared sub-trees in disjunctive interpolation, extending the benefits of DAG interpolation to recursive programs. Inter-procedural software model checking with interpolants has been an active area of research. In the context of predicate abstraction, it has been discussed how well-scoped invariants can be inferred [HJMM04] in the presence of function calls. Based on the concept of Horn clauses, a predicate abstraction-based algorithm for bottom-up construction of function summaries was presented in [GLPR12]. Verification of programs with procedures is described in [HHP10] (using nested word automata) as well as in [AGC12b]. Function summaries generated using interpolants have also been used in bounded model checking [SFS11]. Researchers also showed how to lift these techniques to higher-order programs [JMR11, UTK13]. Several other tools handle procedures by increasingly inlining and performing under and/or over-approximation [SKK11, LQL12, TJ07], but without the use of interpolation techniques.

7.3 Horn Clauses

The use of Horn clauses as intermediate representation for verification was proposed in [GPR11a], with the verification of concurrent programs as main application. Encoding into Horn clauses is also used in logic programming community [PGS98]. The underlying procedure for solving sets of recursion-free Horn clauses, over the combined theory of linear *rational* arithmetic and uninterpreted functions, was presented in [GPR11b]. An algorithm to solve recursion-free systems of Horn constraints by repeated computation of binary interpolants was given in [Ter10], for the purpose of type inference. A range of further applications of Horn clauses, including inter-procedural model checking, was given in [GLPR12]. Horn clauses are also used as a format for verification problems supported by the SMT solver Z3 [HB12]. This thesis extends this direction by presenting general results about solvability and computational complexity, independent of any particular calculus. Our experiments are with linear *integer* arithmetic, arguably a more faithful model of discrete computation than rationals [HIK+12].

The use of Craig interpolation for solving Horn clauses is discussed in [MR13], concentrating on the case of tree interpolation. This thesis extends this work by giving a systematic study of the relationship between different forms of Craig interpolation and Horn clauses, as well as general results about solvability and computational complexity, independent of any particular calculus used to perform interpolation.

7.4 Verification of Timed Systems

There has been a vast amount of research in the field of verification of timed systems. Most notable approaches are timed automata [AD94], timed process algebra [NS91] and real-time logics [AH90]. The timed automata formalism has gained popularity due to its graphical nature with the ability of capturing real-time constraints by explicitly setting/resetting clock variables. A number of tools have been developed for verification of timed automata including Uppaal [LPY97], Kronos [Yov97], Rabbit [BLN03] and RED [Wan04]. These tools rely mostly on BDD-based representation of the finite-state model for explicitly searching the space. In contrast, our approach enables verification of variables with infinite ranges. Besides, one of the future works of this thesis is to include parameterized systems in the modeling thanks to the ability of representing the Integers without bounds.

The approach in [JSV04] encodes timed automata into a constraint logic program. The main focus is to check if the system modeled using timed automata is symmetric or not. The key advantage of proving such assertion is to use the result in the efficient proof of other assertions. Unlike our work they do not have a special encoding for concurrency.

The Owicki-Gries and Rely-Guarantee approaches for verifying the safety of concurrent programs using Horn clauses are described in [GLPR12]. The main difference between [GLPR12] and our work is the presence of time and synchronization between threads. There is also a line of research on bounded model checking of timed automata. In [NMA+02] the authors have encoded timed automata into formulae in difference logic, a propositional logic enriched with timing constraints. They have used bounded model checking to verify the safety of the system. The author of [Sor02] has used the bounded model checking approach to verify networks of timed systems. The given network of timed automata is propositionally encoded into a satisfiability problem and a SAT solver checks the result. The bounded model checking is effective for finding real counter-examples up to a maximum depth. Here in this thesis the focus is to prove the model by predicate abstraction.

The formal specification languages are also extended to capture precisely the timed behavior of systems. In [Hen91] the author has extended the language of temporal logic for specifying timed reactive systems. Accordingly the model checking algorithm is updated to take timed temporal logic formulae. In this thesis we simply model safety properties by making transitions to an error state whenever a critical requirement of the system is violated. Consideration of liveness properties in timed systems deserves to be considered as future work.

8 Conclusion

> When I am working on a problem, I never think about beauty but when I have finished, if the solution is not beautiful, I know it is wrong.
>
> Buckminster Fuller

From a broad perspective the subject of this dissertation was about improvements in one of the cutting-edge software verification techniques, namely counter-example guided predicate abstraction. To overcome the identified weaknesses the thesis proposed novel techniques. We presented CEGAAR, a new automated verification algorithm for integer programs. The algorithm combines interpolation-based abstraction refinement and acceleration of loops. The experimental results show that CEGAAR handles robustly a number of examples that cannot be handled by predicate abstraction or acceleration alone. Because many classes of systems translate into integer programs, our advance contributes to automated verification of infinite-state systems in general.

We exploited the language of Horn clauses as an intermediate language for representation of software programs. As a new form of Craig interpolation we introduced disjunctive interpolation tailored to model checkers based on Horn clauses. Disjunctive interpolation can be identified as solving body-disjoint systems of recursion-free Horn clauses, and subsumes a number of previous forms of interpolation, including tree interpolation. We believe that the flexibility of disjunctive interpolation is highly beneficial for building interpolation-based model checkers.

We classified different interpolation problems with the corresponding recursion-free Horn constraints. We gave algorithms to solve each class of recursion-free Horn clauses using reduction to an appropriate interpolation problem. We discussed the computational complexities of solving classes of recursion-free Horn clauses. In order to allow comparison among different tools we presented a library of publicly available benchmarks of recursion-free Horn clauses.

In this thesis we presented a compositional approach to verify timed systems. Using the two classical approaches of Owicki-Gries and Rely-Guarantee we mapped a concurrent timed automata model to the language of Horn clauses. Then we applied our predicate abstraction framework to verify the generated models. As a proof of concept of the techniques, this thesis presented the predicate abstraction framework Eldarica aiming at the domain of Presubrger arithmetic.

8.1 Future Directions

This thesis finishes by giving some possible ideas to pursue for future work.

Generalized form of Horn Clauses. In Section 5.7 we first raised the question to generalize the recursion-free Horn clauses to the case of recursion-free arbitrary clauses. We did not take the notion of subsumption between clauses, or loops in derivations into account there. This means that a set of clauses might give rise to infinite derivations even if the set of derived clause is finite. It is conceivable that notions of subsumption, or more generally the application of terminating saturation strategies [FLHT01], can be used to identify more general fragments of clauses for which syntactic solutions can effectively be computed. We might ask even for extensions of recursive Horn clauses. In the recent work of [BPR13] the existentially quantified Horn clauses are studied to proving the temporal properties of programs. Investigating more extensions of Horn clauses remain as an important landmark for future work.

Parameterized Concurrent System. In a parameterized system there is an arbitrary number of processes. This is in contrast to the methodology that we presented in the Chapter 6. A promising research direction is to consider the mapping of parameterized systems into Horn clauses. A relatively straightforward way is to incorporate arrays into the system and represent the infiniteness of the number of processes using arrays. This is of particular interest since most of the available verification and analysis tools are explicit state model checkers tailored for finite state systems with bounded number of automata and variables with limited size.

Better computation of disjunctive interpolants. We expect further performance improvements from better implementation of disjunctive interpolation and better techniques to select sets of counterexample paths given to interpolation. Enhancement to the Eldarica framework is one of the future aims of this thesis.

Bibliography

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AGC12a] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In *SAS*, pages 300–316, 2012.

[AGC12b] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

[AH90] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *LICS*, pages 390–401, 1990.

[ALGC12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of sat solvers. In *SPIN*, pages 146–162, 2006.

[BBH+06] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, 1999.

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

[Bey13] Dirk Beyer. Second competition on software verification - (summary of sv-comp 2013). In *TACAS*, pages 594–609, 2013.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic bound computation for loops. In *LPAR (Dakar)*, pages 103–118, 2010.

[BHI+09] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konecný, and Tomás Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *STTT*, 9(5-6):505–525, 2007.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[BIK10] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.

[BKRW11] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *J. Autom. Reasoning*, 47(4):341–367, 2011.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, 1999.

[BLN03] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for bdd-based verification of real-time systems. In *CAV*, pages 122–125, 2003.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[BMR12] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, 2012.

[Boi99] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*, volume PhD Thesis, Vol. 189. Collection des Publications de l'Université de Liège, 1999.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS*, pages 268–283, 2001.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, pages 158–172, 2002.

[BPR13] Tewodros Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, 2013.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. Technical report, 2010.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[CFLZ08] Nicolas Caniart, Emmanuel Fleury, Jérôme Leroux, and Marc Zeitoun. Accelerating interpolation-based model-checking. In *TACAS*, pages 428–442, 2008.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[CGS10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7, 2010.

[Cha09] Robert N. Charette. This car runs on code. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code, February 2009.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.

[Cra57] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[Der] Nachum Dershowitz. Software horror stories. http://www.cs.tau.ac.il/~nachumd/horror.html.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[FL02] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS*, pages 145–156, 2002.

[FLHT01] Christian G. Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In *Handbook of Automated Reasoning*, pages 1791–1849. 2001.

[Gan] Pierre Ganty. Personal Communication.

[GGL+12] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, pages 549–551, 2012.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[GM12] Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6, 2012.

[GPR11a] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[GPR11b] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.

[GS64] Seymour Ginsburg and Edwin H. Spanier. Bounded Algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

[GS66] Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[GY13] Patrice Godefroid and Mihalis Yannakakis. Analysis of boolean programs. In *TACAS*, pages 214–229, 2013.

[HB12] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[Hen91] Thomas A. Henzinger. *Temporal Specification and Verification of Real-Time Systems*, volume PhD Thesis, No. STAN-CS-91-1380. Stanford University - Computer Science Department, 1991.

[HHP09] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *SAS*, pages 69–85, 2009.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL*, pages 471–482, 2010.

[HIK+12] Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *ATVA*, pages 187–202, 2012.

[HIRV07] Peter Habermehl, Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Proving termination of tree manipulating programs. In *ATVA*, pages 145–161, 2007.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[HKG+12] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, pages 247–251, 2012.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa03] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[JMR11] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[JSV04] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP proof method for timed automata. In *RTSS*, pages 175–186, 2004.

[Kin] Zachary Kincaid. LIA horn benchmarks. https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Zachary/.

[KLR10] Daniel Kroening, Jérôme Leroux, and Philipp Rümmer. Interpolating quantifier-free Presburger arithmetic. In *LPAR (Yogyakarta)*, pages 489–503, 2010.

[KLW13] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, LNCS. Springer, 2013.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[Lew78] Harry R. Lewis. Renaming a set of clauses as a Horn set. *J. ACM*, 25(1):134–135, 1978.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *CAV*, pages 427–443, 2012.

[Man74] Zohar Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974.

[MCF+97] Zohar Manna, Michael Colón, Bernd Finkbeiner, Henny Sipma, and Tomás E. Uribe. Abstraction and modular verification of infinite-state reactive systems. In *Requirements Targeting Software and Systems Engineering*, pages 273–292, 1997.

[McM] Kenneth L. McMillan. iZ3 documentation. http://research.microsoft.com/en-us/um/redmond/projects/z3/iz3documentation.html.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.

[McM05a] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.

[McM05b] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[Min67] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[MLNH07] Mario Méndez-Lojo, Jorge A. Navas, and Manuel V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.

[Mon] David Monniaux. Personal Communication.

[MR13] Kenneth L. McMillan and Andrey Rybalchenko. Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, January 2013.

[NMA+02] Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In *FTRTFT*, pages 225–244, 2002.

[NS91] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In *CAV*, pages 376–398, 1991.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[PGS98] Julio C. Peralta, John P. Gallagher, and Hüseyin Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, pages 246–261, 1998.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems transactions of the american mathematical society. *Commun. ACM*, 74(2):358–366, 1953.

[RSS07] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, pages 346–362, 2007.

[Rub11] Andy Rubin. Google mobile boss andy rubin on apple, microsoft and tablets. http://allthingsd.com/20111019/andy-rubin-asiad/, October 2011.

[Rüm08] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, pages 274–289, 2008.

[SFS11] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conference*, pages 160–175, 2011.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973.

[Sor02] Maria Sorea. Bounded model checking for timed automata. *Electr. Notes Theor. Comput. Sci.*, 68(5):116–134, 2002.

[SV07] Ales Smrcka and Tomás Vojnar. Verifying parametrised hardware designs via counter automata. In *Haifa Verification Conference*, pages 51–68, 2007.

[Ter10] Tachio Terauchi. Dependent types from counterexamples. In *POPL*, pages 119–130, 2010.

[TJ07] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.

[Tur49] Alan M. Turing. Checking a large routine. pages 67–69, 1949.

[UTK13] Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, pages 75–86, 2013.

[Wan04] Farn Wang. Efficient verification of timed automata with bdd-like data structures. *STTT*, 6(1):77–97, 2004.

[WDD+12] Virginie Wiels, Rémi Delmas, David Doose, Pierre-Loïc Garoche, Jacques Cazin, and Guy Durrieu. Formal verification of critical aerospace software. *Aerospace Lab Journal*, (4), 2012.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

Curriculum Vitae

Hossein Hojjat

Laboratory for Automated Reasoning and Analysis
School of Computer & Communications Sciences
Swiss Federal Institute of Technology (EPFL) hossein.hojjat@epfl.ch

Education

- **EPFL** Lausanne, Switzerland
  PhD Student *Sep. 2008 - Jun. 2013 (expected)*
  - PhD Candidate under supervision of Prof. Viktor Kuncak
  - Member of ProgLab.Net project funded by Microsoft Research
  - Relevant courses (grades are out of 6): Advanced algorithms (Amin Shokrollahi-6), Logic and Automata Theory (Radu Iosif, Barbara Jobstmann-6), Problem solving in computer science (Tom Henzinger-5), Advanced topics in software analysis and verification (Viktor Kuncak-5.5), Distributed algorithms (Rachid Guerraoui-5)
- **University of Tehran** Tehran, Iran
  Msc., Software Engineering (Grades: 18.99 / 20) *Sep. 2005 - Nov. 2007*
  - Enrolled as a top student without passing the entrance examinations
  - Thesis title: "Formal verification of the object-based systems using process algebra" under supervision of Marjan Sirjani and MohammadReza Mousavi
  - Defended with honors (19.8 / 20)
  - Relevant courses: Network Security, Performance Evaluation of Computer Systems, Software Architecture, Verification of Concurrent Systems
- **University of Tehran** Tehran, Iran
  Bs. Software Engineering (Grades 17.69 / 20) *Sep. 2001 - Sep. 2005*
  - Graduated with Honors, second position
  - Relevant courses: Software Engineering, Internet Engineering, Advanced Algorithm Design,Digital Logic Circuits, Compiler Design, Advanced Software Engineering

Work Experience

- **IPM School of Computer Science** Tehran, Iran
  Researcher *Sep. 2005 - Sep. 2008*
  - Project title: Verification of network protocols
  - In the organization committee of summer and winter schools
    * Process theory - summer 2007
    * Foundations and Trends in Computer Science - winter 2008

  Formal Methods Laboratory Tehran, Iran
  Researcher and Programmer *Sep. 2005 - Sep. 2008*
  - Formal verification of SystemC designs

Technische Universiteit Eindhoven Eindhoven, The Netherlands
Research Visitor *Nov. 2007 - Jan. 2008*
– Process algebraic verification of hardware systems

Niroo Research Center Tehran, Iran
IT consultant *Jun. 2005 - Sep. 2005*
– Application of CRM software in a power distribution company

Reviewer & Organizing

- Member of the organizing committee in the FSEN conferences: FSEN'05 ,FSEN'07, FSEN'11 and FSEN'13
  – International Symposium of Software Engineering
- Reviewer for the conferences
  – ACSD'13,VSTTE'12,SAS'11,ESOP'11

Publications

Conference Papers

- Philipp Rümmer, Hossein Hojjat, Viktor Kuncak, Classifying and Solving Horn Clauses for Verification (VSTTE'13)
- Philipp Rümmer, Hossein Hojjat, Viktor Kuncak, Disjunctive Interpolants for Horn-Clause Verification (CAV'13)
- Hossein Hojjat, Radu Iosif, Filip Konečný, Viktor Kuncak and Philipp Rümmer: Accelerating Interpolants, Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)
- Hossein Hojjat, Filip Konečný, Florent Garnier, Radu Iosif, Viktor Kuncak and Philipp Rümmer: Verification Toolkit for Numerical Transition Systems (tool paper), Proceedings of the 18th International Symposium on Formal Methods (FM'12)
- Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat and Farhad Arbab: Analysis of Reo Circuits using Symbolic Execution, Proceedings of the 8th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'09)
- Hossein Hojjat, Mohammad Reza Mousavi Mousavi, Marjan Sirjani: Process Algebraic Verification of SystemC Codes, Proceedings of the 8th International Conference on Application of Concurrency to System Design (ACSD'08)
- Hossein Hojjat, Mohammad Reza Mousavi, Marjan Sirjani: A Framework for Performance Evaluation and Verification in Stochastic Process Algebras, Proceedings of the 22nd ACM Symposium on Applied Computing, Software Verification Track (SV'08)
- Hossein Hojjat, Marjan Sirjani, SMR Mousavi and Jan Friso Groote: Sarir: A $\mathcal{R}$ebeca to mCRL2 Translator, Proceedings of the 7th IEEE International Conference on Application of Concurrency to System Design (ACSD'07)

- Fahimeh Raja , Hadi Amiri , Samira Tasharofi and Hossein Hojjat and Farhad Oroumchian : Evaluation of part of speech tagging on Persian text, The Second Workshop on Computational Approaches to Arabic Script-based Languages (CAASL2'07)
- Hossein Hojjat, Hootan Nakhost, Marjan Sirjani: Formal Verification of the IEEE 802.1D Spanning Tree Protocol Using Extended Rebeca. Electr. Notes Theor. Comput. Sci. 159: 139-154 (2006)

Journal Papers

- Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat and Farhad Arbab: Symbolic Execution of Reo Circuits using Constraint Automata. Science of Computer Programming, Elsevier, v. 77, n. 7-8, pp. 848-869, 2012.
- Hossein Hojjat, Mohammad Reza Mousavi, Marjan Sirjani: Formal Analysis of SystemC Designs in Process Algebra. Fundam. Inform. v. 107, n. 1, pp. 19-42, 2011.
- Hossein Hojjat, Hootan Nakhost, Marjan Sirjani: Integrating Module Checking and Deduction in a Formal Proof for the Perlman Spanning Tree Protocol (STP), J.UCS Journal of Universal Computer Science, v. 13, n. 13, pp. 2076-2104, 2007.

Technical Reports

- Hossein Hojjat, Mohammad Reza Mousavi Mousavi, Marjan Sirjani: Application of process algebraic verification and reduction techniques to SystemC designs, Computer Science Report No. 08-15, Technische Universiteit Eindhoven.

Software Development

- Main developer of Eldarica, a predicate abstraction engine.
  - `http://lara.epfl.ch/w/eldarica`
- SystemC to mCRL2 Toolkit
  - `http://www.win.tue.nl/~mousavi/sysc08`
- Sarir: Rebeca to mcrl2 translator
  - `http://ece.ut.ac.ir/FML/sarir.htm`
- Stochastic Process Algebras to mcrl2 translator
  - `http://www.win.tue.nl/~mousavi/spa`

Teaching Assistance

Undergraduate

- Introduction to Computer Programming, (Fattane Taghiyareh) : Fall 2002
- Languages and Automata Theory, (Ali Mahjur) : Fall 2003, Spring 2004.
- Artificial Intelligence, (Hesham Faili) : Spring 2004, Fall 2004.

- Advanced Computer Programming, (Hossein Sheikh Attar) : Spring 2004
- Programming Languages Design, (Marjan Sirjani) : Fall 2005
- Informatique III (SSV), (Sebastian Gerlach) : Fall 2009
- Compiler Construction , (Viktor Kuncak) : Fall 2010
- Informatique Théorique Avancée, (Gregory Theoduloz) : Spring 2011

Graduate

- Modeling and Verification of Concurrent Systems, (Marjan Sirjani) : Spring 2006, Spring 2007.
- Synthesis, Analysis, and Verification, (Viktor Kuncak) : Spring 2010