Controlled and Effective Interpolation

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Leonardo de Sá Alt

under the supervision of
Natasha Sharygina

December 2016

Dissertation Committee

Fernando Pedone	Università della Svizzera italiana, Switzerland
Jan Kofroň	Charles University in Prague, Czech Republic
Philipp Rümmer	Uppsala University, Sweden
Robert Soulé	Università della Svizzera italiana, Switzerland

Dissertation accepted on 09 December 2016

Research Advisor	PhD Program Director
Natasha Sharygina	**Michael Bronstein / Walter Binder**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Leonardo de Sá Alt
Lugano, 09 December 2016

Abstract

Model checking is a well established technique to verify systems, exhaustively and automatically. The state space explosion, known as the main difficulty in model checking scalability, has been successfully approached by symbolic model checking which represents programs using logic, usually at the propositional or first order theories level.

Craig interpolation is one of the most successful abstraction techniques used in symbolic methods. Interpolants can be efficiently generated from proofs of unsatisfiability, and have been used as means of over-approximation to generate inductive invariants, refinement predicates, and function summaries.

However, interpolation is still not fully understood. For several theories it is only possible to generate one interpolant, giving the interpolation-based application no chance of further optimization via interpolation. For the theories that have interpolation systems that are able to generate different interpolants, it is not understood what makes one interpolant better than another, and how to generate the most suitable ones for a particular verification task.

The goal of this thesis is to address the problems of how to generate multiple interpolants for theories that still lack this flexibility in their interpolation algorithms, and how to aim at good interpolants.

This thesis extends the state-of-the-art by introducing novel interpolation frameworks for different theories. For propositional logic, this work provides a thorough theoretical analysis showing which properties are desirable in a labeling function for the Labeled Interpolation Systems framework (LIS). The Proof-Sensitive labeling function is presented, and we prove that it generates interpolants with the smallest number of Boolean connectives in the entire LIS framework. Two variants that aim at controlling the logical strength of propositional interpolants while maintaining a small size are given. The new interpolation algorithms are compared to previous ones from the literature in different model checking settings, showing that they consistently lead to a better overall verification performance.

The Equalities and Uninterpreted Functions (EUF)-interpolation system,

presented in this thesis, is a duality-based interpolation framework capable of generating multiple interpolants for a single proof of unsatisfiability, and provides control over the logical strength of the interpolants it generates using labeling functions. The labeling functions can be theoretically compared with respect to their strength, and we prove that two of them generate the interpolants with the smallest number of equalities. Our experiments follow the theory, showing that the generated interpolants indeed have different logical strength. We combine propositional and EUF interpolation in a model checking setting, and show that the strength of the interpolation algorithms for different theories has to be aligned in order to generate smaller interpolants.

This work also introduces the Linear Real Arithmetic (LRA)-interpolation system, an interpolation framework for LRA. The framework is able to generate infinitely many interpolants of different logical strength using the duality of interpolants. The strength of the LRA interpolants can be controlled by a normalized strength factor, which makes it straightforward for an interpolation-based application to choose the level of strength it wants for the interpolants. Our experiments with the LRA-interpolation system and a model checker show that it is very important for the application to be able to fine tune the strength of the LRA interpolants in order to achieve optimal performance.

The interpolation frameworks were implemented and form the interpolation module in OpenSMT2, an open source efficient SMT solver. OpenSMT2 has been integrated to the propositional interpolation-based model checkers FunFrog and eVolCheck, and to the first order interpolation-based model checker HiFrog. This thesis presents real life model checking experiments using the novel interpolation frameworks and the tools aforementioned, showing the viability and strengths of the techniques.

Acknowledgements

I start by stating that even though this thesis has my name as the author, it would not exist without many other people.

I am grateful for the reviewers of this thesis, Professor Fernando Pedone (Università della Svizzera italiana), Professor Robert Soulé (Università della Svizzera italiana), Professor Jan Kofroň (Charles University in Prague) and Professor Philipp Rümmer (Uppsala University), who took the time to thoroughly analyse this work and give valuable criticism. I especially thank Professor Rümmer for the insights on EUF when I felt stuck. In fact, it is interesting to notice how the smallest observation, insight, recommendation or criticism can easily turn into great guidance. I thank Professor Natasha Sharygina, my thesis supervisor, for sharing her experience, expertise, and guiding me through the last years, regarding a vast amount of topics. Doctor Antti Hyvärinen is also a big insight provider to this work, as a SAT/SMT mentor, reader of all my proofs (tough person to convince), climbing instructor, and friend.

I received financial support from the Swiss National Science Foundation, and I am grateful for that.

Universitá della Svizzera italiana is a great place to be, and I enjoyed both the place and the people there, of various research groups. My colleagues were supportive, helpful and good people to be around. I will not mention names because they are many, but I also thank my family, my friends and my girlfriend Sarah for their support during the completion of this work.

Finally, I thank Pizzeria da Franco for providing immediate happiness.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 SAT and SMT solving

Deciding the *satisfiability* (SAT) of a propositional formula is one of the most important problems in computer science. The problem consists in finding an assignment to the variables of the formula that make the formula true. The Cook-Levin theorem, which states that SAT is NP-Complete, led to the discovery that several other problems are also NP-Complete. This brought magnificent importance to the P $\stackrel{?}{=}$ NP problem, one of the most important open problems nowadays.

Algorithms that aim at solving SAT are known as SAT solvers. For the past 20 years, SAT solvers have improved drastically, and enabled its practical use in many different applications, such as software and hardware model checking, planning and integrated circuits McMillan [2003]; Biere et al. [1999]; Bjesse et al. [2001]; Rintanen et al. [2006]; Sheeran et al. [2000]; Chen and Keutzer [1999].

Even though applying SAT solving to various problems became a successful approach, some applications need a more expressive logic, such as first order logic. However, first order logic is undecidable. The notion of first order theory became then common: a decidable or semi-decidable fragment of first order logic related to a specific domain. Some examples of first order theories are linear arithmetic, equalities and arrays. This led to the creation of decision procedures for specific theories. The problem of deciding the satisfiability of a formula in the language of a specific theory became known as *Satisfiability Modulo Theories* (SMT). Nowadays, SMT solvers are very efficient and flexible tools that support many theories and their combination, which has enabled practical software model checking and has been responsible for the success or

optimization of many applications, such as hardware verification, verification of hybrid systems and compilers Ranise and Déharbe [2003]; Barrett et al. [2005]; Audemard et al. [2002a].

Research about SMT solvers has also led to the creation of smt-lib Barrett et al. [2015], an international initiative to aid SMT development containing a language to describe formulas in different theories and benchmarks for different theories and their combination. SMT solvers under current development include z3 de Moura and Bjørner [2008], CVC4 Barrett et al. [2011], MathSAT5 Cimatti et al. [2013b], Yices2 Dutertre [2014], and OpenSMT Hyvärinen et al. [2016]. OpenSMT is a compact and open source SMT solver that has as main goal to make SMT solvers easy to understand and extend. Its open source license and modular architecture allows new theories and algorithms to be quickly added to the framework. OpenSMT supports the theories of Equalities and Uninterpreted Functions (EUF) and Linear Real Arithmetic (LRA). The used solving algorithms for EUF and LRA are, respectively, *congruence closure* Nelson and Oppen [1980]; Nieuwenhuis and Oliveras [2005] and *general simplex* Dutertre and de Moura [2006], both used by most SMT solvers.

1.2 Symbolic Model Checking

Model checking Clarke and Emerson [1982]; Queille and Sifakis [1982] is a highly successful technique for verifying hardware and software systems in a fully automated manner. The technique is based on analysing the state space of the system and verifying that the behavior of the program does not violate its specifications. A system P is described as a set of states S, a set $I \subseteq S$ of initial states, and a transition relation $T \subseteq S \times S$ specifying how the system state evolves over time. A possibly infinite sequence of states $s_0, s_1, \ldots$ where $s_0 \in I$, and $(s_i, s_{i+1}) \in T$ for all i is called an *execution* of the system. *Safety properties* are a practically important class of specifications expressing that certain states describing errors in the system behavior should not be contained in any execution. Safety properties are particularly important in verification since they can be expressed as a reachability problem making the related computations efficient.

For most systems, the concrete state space S is too large to be expressed explicitly. *Symbolic model checking* addresses this problem by expressing the state over a set of variables and the transition relation as a formula over these variables. The problem of reachability can this way be reduced to determining the satisfiability of a logical formula, a task that has received a significant

amount of interest lately with the development of efficient SAT and SMT solvers (see, e.g., Cimatti et al. [2013a]; de Moura and Bjørner [2008], Bruttomesso, Pek, Sharygina and Tsitovich [2010]; Rollini et al. [2013]).

SAT-based bounded model checking Biere et al. [1999, 2003] (BMC) is a powerful approach to assure safety of software up to a fixed bound k. BMC works by 1) unwinding the symbolic transition relation up to k steps, 2) encoding negation of an assertion (that essentially expresses a safety property) in the state reached after k steps, and 3) passing the resulting formula to a SAT solver for determining the satisfiability. If the BMC formula is unsatisfiable, the program is safe. Otherwise, the program is unsafe, and each model of the formula corresponds to a counterexample. BMC has been successfully used in several contexts Audemard et al. [2002b]; Armando et al. [2006]; Junttila and Dubrovin [2008] and forms the basis of several techniques aiming at unbounded model checking with safe inductive invariant generation.

Even though symbolic encoding of the system can be logarithmic in the size of its search space S, the problem of determining satisfiability is often still overwhelming. To make computing more practical in such cases the system is *abstracted* by grouping several states together and expressing the transitions over these grouped states. There are several ways how abstraction can be implemented in practice. Most of them can be reduced to generating an abstract system $\hat{P}$ from a given system P in such a way that the set of executions of S is a sub-set of those of $\hat{S}$. Thus, any property that can be shown to hold for all executions of the abstraction $\hat{S}$ also holds for all executions of S.

In *abstract interpretation* the target is to compute inductive, but not necessarily safe, invariants for the program using an abstract simulation of the program Cousot and Cousot [1977]. The abstraction is obtained by substituting concrete program with *abstract domains* suitable for the properties of the analyzed program (for examples, see Cousot and Cousot [1977]; Clarisó and Cortadella [2004]; Halbwachs and Péron [2008]). While greatly simplifying the model-checking task, abstraction might allow property violations not present in the original software. Such *spurious counterexamples* can be used to refine the abstractions through methods such as *predicate abstraction* Graf and Saïdi [1997] and its refinements, *counter-example guided abstraction refinement* (CEGAR) Clarke et al. [2000], and *lazy abstraction* Henzinger et al. [2002]; McMillan [2006].

The capability of the refinement procedure to find a suitable abstraction level is a critical factor in determining the practical success of a model checker based on abstract interpretation and refinement. With the emergence of extended SAT/SMT solvers and symbolic encoding, approaches based on *Craig*

interpolation Craig [1957]; McMillan [2003] have become very important tools for computing abstractions. However, it has been widely acknowledged that interpolation has been used as a black box, limiting its flexibility and usability. Applications have no control whatsoever on the resulting interpolants. This thesis addresses the problem of the ability to control interpolants.

1.3 Craig Interpolation

An important component in many tasks related to symbolic model checking is to divide an unsatisfiable formula into two parts, A and B, and compute a *Craig interpolant* I, defined over symbols appearing both in A and B, such that A implies I and $I \wedge B$ is still unsatisfiable Craig [1957]. Assuming that $A \wedge B$ is a symbolic encoding of the system P, the interpolant I can be seen as an over-approximation of the part of the program described in A that is sufficiently detailed to guarantee unsatisfiability with the problem description in B. Let I' be another interpolant for A. If $I \rightarrow I'$, we say that I is *stronger* than I', which in turn is *weaker* than I. This type of problems shows up naturally in various verification approaches. For instance, if A describes a set of states and B encodes an example of error-free behavior, a suitable interpolant I can be used to construct a safe inductive invariant for the states in A McMillan [2003]. Similarly, if A consists of a description of a program function f and B consists of the rest of the program together with negation of an assertion, I can be interpreted as an over-approximation of the function f satisfying the assertion Sery et al. [2012c]. Other applications of interpolation in model checking include the refinement phase to synthesize new predicates Henzinger et al. [2004], or to approximate the transition relation Jhala and McMillan [2005]. Model checkers supporting interpolation include CPAChecker Beyer and Keremoglu [2011], CBMC Clarke et al. [2004], FunFrog Sery et al. [2012a], and eVolCheck Fedyukovich et al. [2013], just to name a few.

Depending on the first-order theory in which A and B is defined, interpolants may be effectively computed from a proof of unsatisfiability of $A \wedge B$. For purely propositional formulas, the methods described in Pudlák [1997] (P) Krajícek [1997] and McMillan [2005] (M_s) can be used to traverse the proof obtained from a propositional satisfiability solver and compute interpolants. Another method, presented in D'Silva et al. [2010], is the *labeled interpolation system* (LIS), a powerful and generic framework able to compute propositional interpolants of different strength. It also introduces M_w, the dual of M_s, and generalizes the main previous algorithms P and M_s. When the theory of A and

B is more expressive than propositional logic, it still is often possible to resort to an SMT solver to compute an interpolant. In this case the resolution proof of unsatisfiability will contain original clauses from $A \wedge B$ plus theory-lemmas, clauses produced by a theory solver representing tautologies. Given this proof, it is possible to extract an interpolant using the method described in Yorsh and Musuvathi [2005a], provided that the theory-solver is able to compute an interpolant from a conjunction of literals in the theory.

In order to use interpolants in a straightforward manner, it is necessary that they are *quantifier-free*. It can be shown that every recursively enumerable theory that eliminates quantifiers is *quantifier-free interpolating*, and every quantifier-free interpolating theory eliminates quantifiers Kapur et al. [2006]. Examples of such theories are *equality with uninterpreted functions* (EUF) McMillan [2005]; Fuchs et al. [2009], *difference logic* (DL) Cimatti et al. [2008], *linear real arithmetic* (LRA) McMillan [2005]; Cimatti et al. [2008], and the *bit-vector* (BV) theories. The case for theory of *linear integer arithmetic* (LIA) is, however, more complicated. For example the Presburger arithmetic (in its original formulation) does not admit quantifier-free interpolants; however the addition of stride predicates to the language makes the theory quantifier-free interpolating Pugh [1991]; Brillout et al. [2011b]; Jain et al. [2008]. Similarly also *theory of arrays* ($\mathcal{A}$) needs to be extended to obtain quantifier-free interpolants Bruttomesso et al. [2011].

Interpolants computed from proofs are known to be often highly redundant Cabodi et al. [2015a], necessitating different approaches for optimizing them. One way of compacting propositional interpolants is through applying transformations to the resolution refutation. For example, Rollini et al. [2013, 2012] compare the effect of such compaction on interpolation algorithms on three widely used interpolation algorithms M_s, P and M_w D'Silva et al. [2010] in connection with function-summarization-based model checking Fedyukovich et al. [2013]; Sery et al. [2012a]. A similar approach is studied in D'Silva et al. [2010] combined with an analysis on the strength of the resulting interpolant. Different size-based reductions are further discussed in Cabodi et al. [2015a]; Fontaine et al. [2011a]. While often successful, these approaches might produce a considerable overhead in large problems. An interesting analysis in Bloem et al. [2014] concentrates on the effect of identifying subsumptions in the resolution proofs. A significant reduction in the size of the interpolant can be obtained by considering only CNF-shaped interpolants Vizel et al. [2013]. A light-weight interpolant compaction can be performed by specializing through simplifying the interpolant with a truth assignment Jancik et al. [2014].

In many verification approaches using counterexamples for refinement, it

is possible to abstract an interpolant obtained from a refuted counterexample. For instance, Rümmer and Subotić [2013], and Alberti et al. [2012] present a framework for generalizing interpolants based on templates. A related approach for generalizing interpolants in unbounded model-checking through abstraction is presented in Cabodi et al. [2006] using incremental SAT solving. It is also possible to produce interpolants without the proof Chockler et al. [2013], and there is a renewed interest in interpolation techniques used in connection with modern ways of organizing the high-level model-checking algorithm McMillan [2014]; Cabodi et al. [2014].

Open problems. While Craig Interpolation has an established track record as an over-approximation tool in symbolic model checking, its behavior is still not fully understood. Moreover, the model checking applications have no control over the interpolants, their suitability, size, and strength. This thesis aims to solve the following open problems:

- Why aren't the current interpolation algorithms able to generate good interpolants regardless the interpolation problem?
- How can one generate interpolants that increase the performance of the overall verification process?
- How can the applications have more control over the interpolants it needs?

Our goal in this dissertation is to answer those questions in a sound way, and provide a framework that allows interpolation based model checking to move forward.

1.4 Contributions

This dissertation presents contributions both on the theoretical and practical sides. On the theoretical side we thoroughly study and propose different interpolation algorithms in various logics. For propositional logic, this work presents new algorithms that are able to generate small interpolants and control the strength of interpolants at the same time. For first order theories, we introduce the interpolation system template, and instantiate it for the theories of equalities and uninterpreted functions and linear real arithmetic, providing a new way of creating different interpolants that can be partially ordered with

respect to logical strength for these two theories. We also show how the size of different interpolation algorithms resulting from these systems vary.

Our practical contributions include implementation of all the techniques mentioned above and evaluation of the approach when used by model checkers.

The contributions are summarized as follows:

- Experimental and theoretical study on the quality of previous propositional interpolation algorithms.
  We study previous propositional interpolation algorithms experimentally searching for the best one, and give theoretical reasons why those algorithms fail to generate small interpolants in general.

- Flexible propositional interpolation to generate small interpolants.
  Following the previous theoretical analysis, we overcome the efficiency problems by developing the Proof-Sensitive (PS) interpolation algorithm for propositional logic, which aims at constructing small interpolants.

- Adjustment of the strength of propositional flexible interpolants.
  Different applications may have specific strength requirements, so we also developed extensions of the flexible algorithm PS, which control the strength of interpolants as needed.

- Interpolation System Template (IST).
  Many interpolation algorithms rely on a *dag-like* proof of unsatisfiability. We created an interpolation system template that can be instantiated to first order theories and uses labeling functions and the duality of interpolants to control their strength.

- *EUF* Interpolation System.
  We instantiated the IST for the theory of Equalities and Uninterpreted Functions. The *EUF* Interpolation Systems is capable of generating interpolants of different strength in a controlled way.

- Analysis of labeling functions for the *EUF* Interpolation Systems.
  We theoretically analyzed how different labeling functions can have an impact on the size of interpolants.

- *LRA* Interpolation System.
  The *LRA* Interpolation System is able to generate infinitely many interpolants of different strength in a controlled way.

- All the new techniques above are implemented in OpenSMT2, an efficient SMT-solver.
  With all the techniques combined, interpolation for propositional logic, *EUF* and *LRA* now can be tailored to meet the needs of the model checker.

- Development of the model checker HiFrog.
  HiFrog encodes C programs into the first-order theories EUF and LRA, and uses interpolants to represent function summaries.

- Real life study, combining a model checker and the developed tool.
  The real life context is model checking with function summaries. We compare variants of the tool HiFrog that use propositional logic, *EUF* or *LRA* and make use of the interpolants provided by OpenSMT2.

1.4.1 Analyzing Labeling Functions for Propositional Logic

Craig interpolation is a successful technique in symbolic model checking. Since its introduction, several interpolation-based applications have benefited from this novel technique. However, it became evident that generating an interpolant is not enough: we need control. Before the Labeled Interpolation Systems, interpolation procedures cared about generating an interpolant, any interpolant. Even though the LIS framework gives flexibility on the generation of interpolants for propositional logic, it was not clear which labeling function would behave best in a model checking scenario.

Research contribution. We contribute to propositional interpolation by conducting an experimental and theoretical study on labeling functions. We compared the labeling functions M_s, P, prior to LIS, and M_w, a natural consequence of LIS, in two model checking settings: incremental checking with FunFrog Sery et al. [2012a] and upgrade checking with eVolCheck Fedyukovich et al. [2013]. Both tools are interpolation-based bounded model checkers that use interpolants as over-approximations of function summaries.

For FunFrog, we show that stronger interpolants behave best, whereas weaker interpolants are good for eVolCheck.

The labeling functions M_s, P and M_w are the most simple ones within LIS, and do not aim at any optimization. The second part of our contribution is a thorough theoretical analysis of LIS labeling functions. We show that *uniform* labeling functions yield smaller interpolants, and that it is very hard to remove

all the occurrences of a specific variable from the interpolant, since the proof has to be p-annihilable, which is very rare in practice.

The results of our work have been published in Rollini et al. [2013], Alt et al. [2016], and Rollini et al. [2015], and are discussed in Chapter 4.

1.4.2 Flexible and Controlled Propositional Interpolants

Most interpolation algorithms rely on a proof of unsatisfiability of $A \wedge B$, such that A is the part to be over-approximated. Propositional logic is the language of SAT solvers and central for SMT solving, therefore is essential for symbolic model checking. Thus, interpolation for propositional logic is a topic that must be heavily studied by different techniques, if the community wants to keep optimizing program verification.

Several different approaches try to improve the performance and usability of interpolation-based applications by providing *better* interpolants. Of course the idea of what a *good* interpolant varies, yielding for instance, algorithms that care about the strength of interpolants D'Silva et al. [2010]; Rollini et al. [2013], the size and structure of interpolants Rollini et al. [2012]; Vizel et al. [2013]; Cabodi et al. [2006, 2015b], or the semantics of variables Rümmer and Subotić [2013]; Jancik et al. [2014].

Since the suitability of interpolants depends ultimately on the needs of the interpolation-based application, each of these topics has to be extended in order to give to the application more control on the interpolants that are generated by the interpolation algorithms. This is still an issue even though several interpolation approaches exist for propositional logic.

Research contribution. We have created the *Proof-Sensitive* labeling function that is proven to generate small interpolants in an efficient way. The strength of interpolants is also a very important feature, so we have developed a weaker and a stronger labeling functions that are variants of the Proof-Sensitive. The goal of these two labeling functions is to provide strength control while still aiming at small interpolants. We show experimentally that the new Proof-Sensitive labeling functions are able to increase the performance of model checkers by generating interpolants that are small and more suitable to the applications, compared to prior work.

The results of our work have been published in Alt et al. [2016] and Hyvärinen et al. [2015], and are discussed in Chapter 4.

1.4.3 Controlling EUF Interpolants

The theory of Equalities and Uninterpreted Functions is essential to program verification for two reasons. The first reason is that in many cases it is enough to assume that a given function returns the same values when invoked with same arguments to prove that a program is safe. Furthermore, proving program equivalence Godlin and Strichman [2013] and modeling memory Stump et al. [2001] benefit from EUF abstraction. The second reason is the importance of EUF to theory combination. Since equality is usually the only common operator between theories, when theories are combined EUF plays a central and essential role.

Even though the EUF theory is so important for program verification, the current interpolation algorithms McMillan [2005]; Fuchs et al. [2009] for EUF are used as black-boxes, without providing any means for the interpolation-based application to tune its interpolants. This may lead to bottlenecks in program verification to the point that a technique becomes impractical.

Research contribution. We contribute by providing the EUF-interpolation system, the first interpolation framework for EUF that is able to control the strength of the generated interpolants. The strength control is done via labeling functions that can have their strength compared a priori. This is particularly useful for the applications for which the necessary strength of interpolants is known. As a theoretical study about the size of the interpolants generated by different EUF labeling functions, we prove that the strongest and the weakest labeling functions in the framework are also the ones that generate the interpolants with the smallest number of equalities. We provide extensive experimentation in two different settings, experiments with a controlled set of experiments where we interpolate over several complex smt-libbenchmarks, and experiments with the interpolation-based model checker HiFrog. Our experiments show the viability of the method, and how valuable new interpolation algorithms are for an interpolation-based application.

The results of our work will appear in the FMCAD 2017 Alt, Asadi, Hyvärinen and Sharygina [2017] proceedings and are discussed in Chapter 5.

1.4.4 Controlling LRA Interpolants

The theory of Linear Real Arithmetic is one of the oldest and most important first order fragments used for program verification. Almost every real life system needs reasoning over numbers, and if reasoning over integers is not

necessary, the LRA theory provides a very efficient and practical solving framework that enables the verification of programs that would be infeasible with the theory of Linear Integer Arithmetic.

Despite the success and usability of the LRA theory in contemporary SMT solvers, the control that a verification application has over LRA interpolation is still an issue. Different LRA interpolation algorithms have focused on the simplicity Albarghouthi and McMillan [2013]; Scholl et al. [2014], efficiency Rybalchenko and Sofronie-Stokkermans [2007], or a simple proof system McMillan [2005]; Pudlák [1997], but none of them allow the application to interfere in the interpolation process in order to obtain interpolants most suitable for verification tasks.

Research contribution. This thesis contributes by introducing the LRA-interpolation system, an LRA interpolation framework that is able of not only producing multiple interpolants for a single refutation proof, but an infinite amount of them. The infinitely many generated interpolants can be sorted by strength using a *strength factor* that is normalized (between 0 and 1) and provides a very easy way to control interpolants. We show that different strength factors can be used in different parts of the proof, allowing the interpolation-based application to fine-tune different parts of the system. Our experiments show the viability and usability of the LRA-interpolation system in a controlled setting consisting of smt-libbenchmarks and when integrated with a real life model checker.

The results of our work will appear in the HVC 2017 Alt, Hyvärinen and Sharygina [2017] proceedings and are discussed in Chapter 6.

1.4.5 Interpolating OpenSMT2

Interpolants are mostly computed from a proof of unsatisfiability. Therefore, interpolation procedures require a tight cooperation with SAT and SMT solvers, and in some cases, among each other. Besides, interpolation procedures are non-trivial and must be implemented efficiently to ensure good performance for the interpolation-based application.

Research contribution. We have implemented the interpolation modules of OpenSMT2, containing all the theoretical contributions listed above.

The implemented propositional interpolation algorithm is LIS, with built-in support to the labeling functions M_s, P, M_w and D_{min} from the literature, and the labeling functions PS, PS_w and PS_s presented in this thesis. Our

implementation is integrated with the SMT solver and theory interpolation algorithms which compute partial interpolants for some specific nodes of the SAT refutation proof.

Our implementation of the EUF-interpolation system introduced in this work relies on an extra congruence graph data structure built while solving an EUF problem. It has built-in support to the EUF interpolation algorithms Itp_s, Itp_w and Itp_r, described in Chapter 5.

The LRA-interpolation system is implemented based on the explanation set derived from the simplex algorithm. No extra data structure is needed, since the explanation set has to be built for convergence of the SMT solver. It supports custom strength factors from 0 to 1, and has the built-in options of using Itp_s and Itp_w.

The results of these implementation efforts together with the experimentation evaluation of the tools FunFrog, eVolCheck and HiFrog have been published in Hyvärinen et al. [2016]; Alt et al. [2016]; Hyvärinen et al. [2015].

1.4.6 HiFrog

Interpolants are used in several different ways in symbolic model checking. One of the successful applications of interpolants is in function summarization. In this approach, when an assertion (i.e., the safety property of the program) is proven safe, summaries of the involved functions are computed and stored for further reuse while proving either a sequence of assertions in a single program, as in incremental checking, or the same assertion in different versions of the code, as in upgrade checking. Tools based on this approach exist, for instance FunFrog and eVolCheck. The issue with these two tools (and other tools based on bit-blasting) is that they rely on propositional logic to encode C programs. This process can easily blow up and generate huge formulas, which might make verification impossible.

Research contribution. To improve efficiency of interpolation-based incremental checking, we have developed HiFrog, a summarization-based bounded model checker that uses interpolants to represent function summaries. HiFrog supports the first order theories EUF and LRA, which represent different levels of abstraction and lead to different performances. HiFrog relies on the interpolation module from OpenSMT2 to create tailored interpolants, and extends this support to the user.

The results of our work on HiFrog have been published in Alt, Asadi, Chockler, Even Mendoza, Fedyukovich, Hyvärinen and Sharygina [2017].

1.5 Organization of the thesis

Chapter 2 describes two interpolation-based model checking algorithms. A brief explanation of how each technique works and uses interpolants is given, justifying the need for better interpolants. Chapter 3 presents a thorough description of all the logical concepts need in this thesis. That includes terminology and concepts related to propositional logic, and the first order theories EUF and LRA. The following chapters present the novelties of this work, in an order that follows Section 1.4. Chapter 4 presents our theoretical studies on previous interpolation algorithms for propositional logic and novel labeling function that guarantee small interpolants. Chapter 5.1 introduces the Interpolation System Template (IST), showing how labeling functions and the duality of interpolants are used to control interpolant strength. Chapter 5 instantiate the IST into the EUF-Interpolation System, which allows the generation of a multitude of new interpolants with strength guarantees. An analysis of labeling functions is also given, showing how to generate interpolants with a small number of equalities. Chapter 6 presents the LRA-Interpolation System, which is able to generate infinitely many interpolants with strength guarantees. Chapter 7 describes the architecture and implementation of the tools developed along this thesis, and summarizes the experimental results. Finally, Chapter 8 gives final remarks and concludes this thesis.

Chapter 2

Interpolation-based Model Checking

2.1 Interpolation as Means of Over-Approximation

The first use of Craig interpolants in model checking was the interpolation and SAT-based model checking algorithm from McMillan [2003]. The basic idea of the algorithm is to use bounded model checking as a subroutine, and find a closed safe set of states to prove the safety property or a counterexample that proves that the safety property is unsafe. This section describes this algorithm in a high level.

We would first like to clarify the difference between the two meanings that bounded model checking might have in the context of verification. The most common and practical use of the term bounded model checking is to characterize model checking algorithms that, given a bound k, unroll program loops and recursive calls up to k, and verify safety properties with the unrolled code. The second use is to refer to a similar approach, but related specifically to transition systems, in the sense that the bound k is used as the number of steps from the initial states. In this case, a bounded model checking (BMC) problem consists of a set of initial state, a transition function, a set of error states, and the bound k. The bounded model checker has then as a goal to determine if an error state can be reached from some initial state after the application of at most k steps. In this section we refer to the latter definition of bounded model checking.

Let S be a transition system representing a program to be verified with respect to a safety property. Let I be the set of initial states, T the transition function, E the set of error states, and k an integer. A BMC problem that

verifies the safety of S up to a bound k can be represented as the formula

$$BMC(S,k) = I \wedge \left(\bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{s_e \in E} s_e \right). \tag{2.1}$$

If $BMC(S,k)$ is satisfiable (SAT), the states represent an assignment for the variables that is a counterexample for S. On the other hand, if $BMC(S,k)$ is unsatisfiable (UNSAT), the program is safe up to k and interpolants can be used to search for an inductive invariant. Suppose $BMC(S,k)$ is UNSAT. Then, it can be partitioned such that $A_1 = I \wedge T(s_0, s_1)$ and $B_1 = \left(\bigwedge_{1 \leq i < k} T(s_i, s_{i+1})\right) \wedge \left(\bigvee_{s_e \in E} s_e\right)$, and an interpolant P_1 for this partitioning is computed. The interpolant P_1 is an over-approximation of the states that are reachable from the set of initial states in one transition step. It is also unsatisfiable when conjoined with B_1, meaning that no assignment satisfying P_1 can reach E in $k-1$ steps. Let R be an over-approximation of the reachable states, initialized such that $R = I$. If $P_1 \rightarrow R$, R is an inductive invariant and S is safe for the general unbounded case. Otherwise, we update R such that $R = R \vee P_1$, compute an interpolant P_2 for the partitioning $A_2 = I \wedge T(s_0, s_1) \wedge T(s_1, s_2)$ and $B_2 = \left(\bigwedge_{2 \leq i < k} T(s_i, s_{i+1})\right) \wedge \left(\bigvee_{s_e \in E} s_e\right)$, and check if $P_2 \rightarrow R$ to see if R is an inductive invariant. This process repeats until (i) an inductive invariant is found; (ii) a counterexample is found; or (iii) no inductive invariant or counterexample is found after the generation of $k+1$ interpolants. If (iii) happens, k needs to be increased and the process continued.

The work in McMillan [2003] also proves that by increasing k, eventually either a counterexample or an inductive invariant is found.

2.2 Function Summarization for Bounded Model Checking

In Sery et al. [2012c,b], a set of techniques to make BMC incremental by allowing the handling different assertions and different versions of a software source code was introduced. The key concept behind this approach is *function summarization*, a technique to create and use over-approximation of the function behavior via Craig Interpolation. Assume that a safety property has been shown to hold for a BMC unwinding of a given program by showing the corresponding formula unsatisfiable. The constructed interpolants are then stored in a persistent storage and are available for the use by the model checker on demand.

This technique handles the *unwound static single assignment* (USSA) approximation of the program, where loops and recursive function calls are unwound up to a fixed limit, and each variable is only assigned at most once. The benefit of the USSA-approximation is that it allows encoding into a suitable first order theory, as well as propositional logic, and can be passed to an SMT solver supporting the theory, or a SAT solver. The constructed function summaries can be used in two applications: A1 incremental substitution while checking the same program with respect to different assertions; and A2 upgrade validation by localized checking of summaries with respect to the old assertions.

The approach of A1 considers given assertions one at a time. It computes function summaries with respect to the first assertion (if possible) and attempts to substitute summaries for the function while checking consecutive assertions. If a check fails after such substitution (i.e. the solver returns SAT), *refinement* is needed: the summaries are discarded and the check is repeated with concrete function bodies. The approach of A2, in contrast, considers a new version of the program, previously proven safe. The idea is to check whether the updated code is still over-approximated by the old summaries, and if needed, also to perform *refinement*.

The next paragraphs discuss the details of the model checking algorithm that uses function summaries. In particular, Alg. 1 outlines the method for constructing function summaries (non-empty if the program is *SAFE*) in SMT-based BMC.

BMC formula construction. The USSA approximation is encoded into a BMC formula as follows:

$$\texttt{CreateFormula}(\hat{f}) \triangleq \phi_{\hat{f}} \wedge \bigwedge_{\hat{g} \in \hat{F}: nested(\hat{f},\hat{g})} \texttt{CreateFormula}(\hat{g})$$

For a function call $\hat{f} \in \hat{F}$, the formula is constructed recursively, by conjoining the subformula $\phi_{\hat{f}}$ corresponding to the body of the function with a separate subformula for every nested function call. The logical formula $\phi_{\hat{f}}$ is constructed from the USSA form of the body of the function f using the theory T.

Summarization. If the BMC formula is unsatisfiable, i.e., the program is safe, the algorithm proceeds with interpolation. The function summaries are constructed as interpolants from a proof of unsatisfiability of the BMC formula. In order to generate the interpolant, for each function call $\hat{f}$ the BMC formula is

split into two parts. First, $\phi_{\hat{f}}^{subtree}$ corresponds to the subformulas representing the function call $\hat{f}$ and all the nested functions. Second, $\phi_{\hat{f}}^{env}$ corresponds to the context of the call $\hat{f}$, i.e., to the rest of the encoded program and possible erroneous behaviors *error*.

$$\phi_{\hat{f}}^{subtree} \triangleq \bigwedge_{\hat{g}\in\hat{F}:subtree(\hat{f},\hat{g})} \phi_{\hat{g}} \qquad \phi_{\hat{f}}^{env} \triangleq error \wedge \bigwedge_{\hat{h}\in\hat{F}:\neg subtree(\hat{f},\hat{h})} \phi_{\hat{h}}$$

Therefore, for each function call $\hat{f}$, the `Interpolate` method splits the BMC formula into $A \equiv \phi_{\hat{f}}^{subtree}$ and $B \equiv \phi_{\hat{f}}^{env}$ and generates an interpolant $I_{\hat{f}}$ for (A, B). We refer to $I_{\hat{f}}$ as a *summary* of function call $\hat{f}$.

Algorithm 1 Function summarization in SMT-based BMC

Input: USSA P_ν with function calls $\hat{F}$ and main function f_{main}
Ouput: Verification result: $\{SAFE, UNSAFE\}$, summaries $\{I_{\hat{f}}\}$
Data: BMC formula ϕ

1: $\phi \leftarrow$ `CreateFormula`$(\hat{f}_{main})$
2: $result, proof \leftarrow$ `Solve`(ϕ) ▷ run SAT/SMT-solver
3: **if** $result =$ SAT **then**
4: **return** *UNSAFE*, $\varnothing$
5: **end if**
6: **for** each $\hat{f} \in \hat{F}$ **do** ▷ extract summaries
7: $I_{\hat{f}} \leftarrow$ `Interpolate`$(\hat{f})$
8: **end for**
9: **return** *SAFE*, $\{I_{\hat{f}}\}$

Various verification techniques implement the idea of function summarization in different forms McMillan [2006, 2010]; Albarghouthi et al. [2012]. This thesis uses three tools that implement Craig interpolation-based function summaries as described above. The tools FunFrog, eVolCheck, and HiFrog are used in model checking experiments with the interpolation techniques presented in Chapters 4, 5 and 6. Depending on the specific technique using interpolants as function summaries, the interpolation algorithms that are enabled need to be restricted. For instance, the tool eVolCheck requires tree interpolation, as provided in Rollini et al. [2013].

Chapter 3

Technical Background

This chapter describes all the technical background necessary for the understanding of the contributions presented in this thesis.

We introduce first propositional logic, the foundation of computer science. Besides its many related and important topics outside verification, propositional logic was responsible for the beginning of symbolic model checking, and much of the success of formal verification is due to the developments in SAT solvers.

We then move on to introduce two of the most important first order theories in SMT solving, the quantifier-free fragments of Equalities and Uninterpreted Functions (EUF) and Linear Real Arithmetic (LRA). These two theories give different levels of abstraction, and are essential to represent programs. They are also part of the reason for the success of SMT-based software model checking.

3.1 Propositional Logic Preliminaries

Given a finite set of propositional variables, a *literal* is a variable p or its negation $\neg p$. A *clause* is a finite set of literals and a formula ϕ in *conjunctive normal form* (CNF) is a set of clauses. We also refer to a clause as the disjunction of its literals and a CNF formula as the conjunction of its clauses. A variable p occurs in the clause C, denoted by the pair (p, C), if either $p \in C$ or $\neg p \in C$. The set $var(\phi)$ consists of the variables that occur in the clauses of ϕ. We assume that double negations are removed, i.e., $\neg\neg p$ is rewritten as p. A *truth assignment* σ assigns a Boolean value to each variable p. A clause C is satisfied if $p \in C$ and $\sigma(p)$ is true, or $\neg p \in C$ and $\sigma(p)$ is false. The propositional satisfiability problem (SAT) is the problem of determining whether there is a truth assignment satisfying each clause of a CNF formula ϕ. The special con-

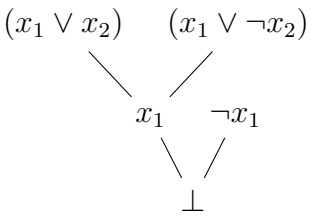
stants $\top$ and $\bot$ denote the empty conjunction and the empty disjunction. The former is satisfied by all truth assignments and the latter is satisfied by none. A formula ϕ *implies* a formula ϕ', denoted $\phi \to \phi'$, if every truth assignment satisfying ϕ satisfies ϕ'. The *size* of a propositional formula is the number of logical connectives it contains. For instance the unsatisfiable CNF formula

$$\phi = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1) \tag{3.1}$$

of size 14 consists of 4 variables and 5 clauses. The occurrences of the variable x_4 are $(x_4, \neg x_2 \vee x_4)$ and $(x_4, \neg x_2 \vee \neg x_3 \vee \neg x_4)$.

For two clauses C^+, C^- such that $p \in C^+$, $\neg p \in C^-$, and for no other variable q both $q \in C^- \cup C^+$ and $\neg q \in C^- \cup C^+$, a *resolution step* is a triple C^+, C^-, $(C^+ \cup C^-) \setminus \{p, \neg p\}$. The first two clauses are called the *antecedents*, the latter is the *resolvent* and p is the *pivot* of the resolution step. A *resolution refutation* R of an unsatisfiable formula ϕ is a directed acyclic graph where the nodes are clauses and the edges are directed from the antecedents to the resolvent. The nodes of a refutation R with no incoming edge are the clauses of ϕ, and the rest of the clauses are resolvents derived with a resolution step. The unique node with no outgoing edges is the empty clause. The *source clauses* of a refutation R are the clauses of ϕ from which there is a path to the empty clause. Example 1 shows a proof of unsatisfiability for a simple formula in CNF using subsequent uses of the resolution rule.

Example 1. *Let ψ be a propositional CNF consisting of the conjunction of the following set of clauses: $\{(x_1 \vee x_2), (x_1 \vee \neg x_2), (\neg x_1)\}$. The tree below presents the proof that ψ is unsatisfiable. The leaves are the original clauses from ψ, and the edges are the application of a resolution rule to infer a new clause. When the contradiction is reached ($\bot$) we have a proof of unsatisfiability.*



The *labeled interpolation system* D'Silva et al. [2010] (LIS) is a framework that, given propositional formulas A, B, a refutation R of $A \wedge B$ and a *labeling function* L, computes an interpolant I for A based on R. The refutation together with the partitioning A, B is called an *interpolation instance* (R, A, B).

The labeling function L assigns a label from the set $\{a, b, ab\}$ to every variable occurrence (p, C) in the clauses of the refutation R. A variable is *shared* if it occurs both in A and B; otherwise it is *local*. For all variable occurrences (p, C) in R, $L(p, C) = a$ if p is local to A and $L(p, C) = b$ if p is local to B. For occurrences of shared variables in the source clauses the label may be chosen freely. The label of a variable occurrence in a resolvent C is determined by the label of the variable in its antecedents. For a variable occurring in both its antecedents with different labels, the label of the new occurrence is ab, and in all other cases the label is equivalent to the label in its antecedent or both antecedents.

An interpolation algorithm based on LIS computes an interpolant with a dynamic algorithm by annotating each clause of R with a *partial interpolant* starting from the source clauses. The partial interpolant of a source clause C is

$$I(C) = \begin{cases} \bigvee\{l \mid l \in C \text{ and } L(var(l), C) = b\} & \text{if } C \in A, \text{ and} \\ \bigwedge\{\neg l \mid l \in C \text{ and } L(var(l), C) = a\} & \text{if } C \in B, \end{cases} \tag{3.2}$$

The partial interpolant of a resolvent clause C with pivot p and antecedents C^+ and C^-, where $p \in C^+$ and $\neg p \in C^-$, is

$$I(C) = \begin{cases} I(C^+) \vee I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = a, \\ I(C^+) \wedge I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = b, \text{ and} \\ (I(C^+) \vee p) \wedge (I(C^-) \vee \neg p) & \text{otherwise.} \end{cases} \tag{3.3}$$

The LIS framework also provides a convenient tool for analyzing whether the interpolants generated by one interpolation algorithm always imply the interpolants generated by another algorithm. If we order the three labels so that $b \leq ab \leq a$, it was shown D'Silva et al. [2010] that given two labeling functions L and L' resulting in the interpolants I_L and $I_{L'}$ in LIS and having the property that $L(p, C) \leq L'(p, C)$ for all occurrences (p, C), it is true that $I_L \rightarrow I_{L'}$. In this case we say that the interpolation algorithm obtained from LIS using the labeling L' is *weaker* than the interpolation algorithm that uses the labeling L (*stronger*).

The interpolation algorithms M_s McMillan [2003] and P Pudlák [1997], considered pioneers, can be obtained as special cases of LIS by providing a labeling function returning b and ab, respectively. By using the labels, LIS provides a lattice that partially orders labeling functions, with M_s being the strongest labeling function. A labeling function that always returns a is considered dual to M_s, and is named M_w D'Silva et al. [2010], since it is the weakest labeling function in LIS.

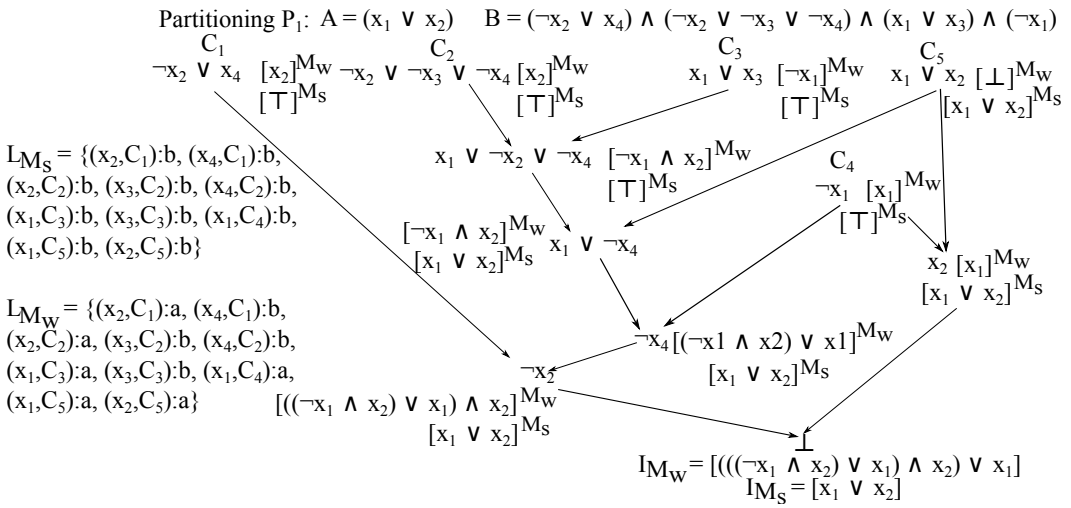


Figure 3.1. Different interpolants obtained from the refutation using the partitioning P_1.

We define here two concepts that will be useful in Chapter 4: the class of *uniform* labeling functions, and the *internal size* of an interpolant.

Definition 1. *A labeling function is* uniform *if for all pairs of clauses $C, D \in R$ containing the variable p, $L(p, C) = L(p, D)$, and no occurrence is labeled ab. Any interpolation algorithm with uniform labeling function is also called uniform.*

An example of non-uniform labeling function is D_{min}, presented in D'Silva [2010]. D_{min} labels occurrences of shared variables by copying its class, is proven to produce interpolants with the least number of distinct variables.

Definition 2. *The* internal size *$IntSize(I)$ of an interpolant I is the number of connectives in I excluding the connectives contributed by the partial interpolants associated with the source clauses.*

The following example illustrates the concepts discussed in this section by showing how LIS can be used to compute interpolants with two different uniform algorithms M_s and M_w.

Example 2. *Consider the unsatisfiable formula $\phi = A \wedge B$ where ϕ is from Eq. (3.1) and $A = (x_1 \vee x_2)$ and $B = (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1)$. Fig. 3.1 shows a resolution refutation for ϕ and the partial interpolants computed by the interpolation algorithms M_s and M_w. Each*

clause in the refutation is associated with a partial interpolant ψ generated by labeling L_{M_s} (denoted by $[\psi]^{\mathrm{M}_s}$) and a partial interpolant ψ' generated by labeling L_{M_w} (denoted by $[\psi']^{\mathrm{M}_w}$). The generated interpolants are $Itp_{\mathrm{M}_s} = x_1 \vee x_2$ and $Itp_{\mathrm{M}_w} = (((\neg x_1 \wedge x_2) \vee x_1) \wedge x_2) \vee x_1$. Now consider a different partitioning $\phi' = A' \wedge B'$ for the same formula where the partitions have been swapped, that is, $A' = B$ and $B' = A$. Using the same refutation, we get the interpolants $Itp'_{\mathrm{M}_s} = (((x_1 \vee \neg x_2) \wedge \neg x_1) \vee \neg x_2) \wedge \neg x_1 = \neg Itp_{\mathrm{M}_w}$ and $Itp'_{\mathrm{M}_w} = \neg(x_1 \vee x_2) = \neg Itp_{\mathrm{M}_s}$ The internal size of Itp_{M_s} is 0, whereas the internal size of Itp_{M_w} is 4.

3.2 Integration of Propositional and Theory Interpolation.

Before we introduce the preliminaries for the first order theories studied in this thesis, we describe how theory interpolants are used in an SMT solver.

An SMT solver takes as input a propositional formula where some atoms are interpreted over a theory, in our case equalities over uninterpreted functions and inequalities over rational numbers. If a satisfying truth assignment for the propositional structure is found, a theory solver is queried to determine the consistency of its equalities. In case of inconsistency the theory solver adds a reason-entailing clause to the propositional structure. The process ends when either a theory-consistent truth assignment is found or the propositional structure becomes unsatisfiable.

The SMT framework provides a natural integration for the theory and propositional interpolants. The clauses provided by the theory solver are annotated with their theory interpolant and are used as partial interpolants as leaves of the refutation proof. The propositional interpolation algorithm then proceeds as normal and uses the theory interpolants instead of computing propositional interpolants for theory clauses. The propositional interpolation algorithms control the strength of the resulting interpolant by choosing the partition for the shared variables through labeling functions Alt et al. [2016]. The theory interpolation systems presented in this thesis also use labeling functions to generate interpolants of different strength. If this is the case for the framework in use, the labeling given by the propositional interpolation algorithm has to be followed by the theory interpolation algorithm to preserve interpolant soundness.

3.3 EUF Preliminaries

The theory of equalities and uninterpreted functions extends propositional logic by adding equality ($=$) and disequality ($\neq$) to the logical symbols, and allowing functions and predicates as non-logical symbols. It has the following axioms:

$$x = x \tag{3.4}$$

$$x = y \Rightarrow y = x \tag{3.5}$$

$$(x = y \wedge y = z) \Rightarrow x = z \tag{3.6}$$

$$(x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \tag{3.7}$$

Most EUF solvers rely on the *congruence closure* algorithm Nelson and Oppen [1980]; Nieuwenhuis and Oliveras [2005] to decide the satisfiability of a set of equalities and disequalities. The algorithm, described in Alg. 2, has as input a finite set Eq of equalities, and the set T of subterm-closed terms over which Eq is defined. The main idea of the algorithm is to compute *equivalence classes*, that is, sets of terms that are equivalent. If in the end of the computation of the equivalent classes, there are two nodes u and v such that they belong to the same equivalent class and $(u \neq v)$ is an element of the set of problem disequalities, the problem is unsatisfiable. If no such pair exists, the problem is satisfiable. During the execution the algorithm builds an undirected *congruence graph* G using the set T as nodes, that represents the equivalence classes. If two nodes are connected, they are equivalent. We write $(x \sim y)$ if there is a path in G connecting x and y.

Theorem 1 (c.f. Nelson and Oppen [1980]; Nieuwenhuis and Oliveras [2005])**.** *Let $S \supseteq Eq$ be a set containing EUF equalities and disequalities over the terms T. The set S is satisfiable if and only if the congruence graph G constructed by* CongruenceClosure*(T, Eq) has no path $(x \sim y)$ such that $(x \neq y) \in S$.*

During the creation of G, an edge (x, y) is added only if $(x \sim y)$ does not hold, which ensures that G is acyclic. Therefore, for any pair of terms x and y such that $(x \sim y)$ holds in G, there is a unique *path* $\overline{xy}$ connecting these terms. Empty paths are represented by $\overline{xx}$. Example 3 shows in detail how Alg. 2 works.

Algorithm 2 Congruence closure

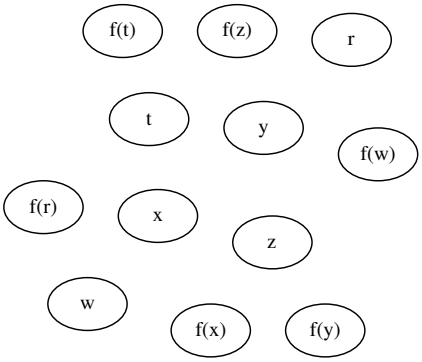
```
1: procedure CongruenceClosure(T, Eq)
2:     Initialize E ← ∅ and G ← (T, E)
3:     repeat pick x, y ∈ T such that (x ≁ y)
4:         if (a) (x = y) ∈ Eq or
5:         (b) x is f(x_1, ..., x_k), y is f(y_1, ..., y_k), and
6:         (x_1 ∼ y_1), ..., (x_k ∼ y_k) then
7:                 E ← E ∪ {(x, y)}
8:         end if
9:     until E does not grow
10: end procedure
```

Example 3. *Let* $Eq = \{x = f(z), y = f(w), z = w\}$. *We have that* $T = \{x, y, z, w, f(x), f(y), f(z), f(w)\}$ *is the set of nodes of the congruence graph, which in the beginning of the algorithm contains no edges. Fig. 3.2 shows a graphical representation of the steps of the algorithm. We can pick the pairs* (x, y), (x, z), (x, w), $(x, f(x))$ *and* $(x, f(y))$, *but none of the conditions from the algorithm is satisfied for those pairs. Let us then pick* $(x, f(z))$. *Condition (a) of the algorithm is satisfied, so this pair is added as an edge in the congruence graph (Fig. 3.2b). Let us now pick* $(y, f(w))$. *Condition (a) is also satisfied for this pair, therefore it becomes an edge (Fig. 3.2c). Now the next and only pair we can pick that changes the graph is* $(z = w)$. *Condition (a) is satisfied and it becomes an edge (Fig. 3.2d). This edge enables the use of pair* $(f(z), f(w))$, *which satisfies condition (b), resulting in a new edge. Now no pair can be picked, and the algorithm terminates (Fig. 3.2e). We can see that Eq is satisfiable and the algorithm proved that the set* $\{x, f(z), f(w), y\}$ *is one of the equivalence classes. Notice that if* $(x \neq y) \in Eq$, *it would be unsatisfiable, since the algorithm proved that* $(x = y)$ *is true, which would contradict an original disequality.*

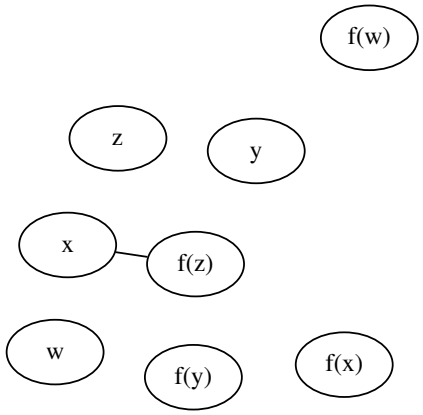
For an arbitrary path π, we use the notation $[\![\pi]\!]$ to represent the equality of the terms that π connects. If, for example, π connects x and y, then $[\![\pi]\!] := (x = y)$. We also extend this notation over sets of paths P so that $[\![P]\!] := \bigwedge_{\sigma \in P} [\![\sigma]\!]$.

An edge may be added to a congruence graph G because of two different reasons in Alg. 2 at line 7. Edges added because of Condition (a) are called *basic*, while edges added because of Condition (b) are called *derived*. Let e be a derived edge $(f(x_1, \ldots, x_k), f(y_1, \ldots, y_k))$. The k *parent paths* of e are $\overline{x_1 y_1}, \ldots, \overline{x_k y_k}$.

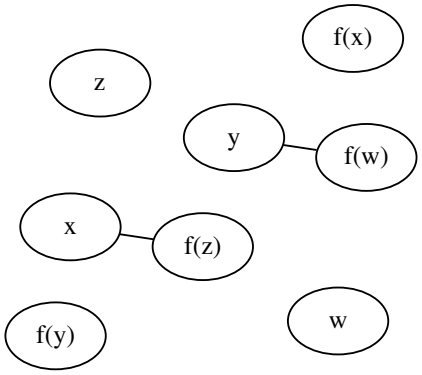
In order to generate an interpolant, the formula, or in this case, the set



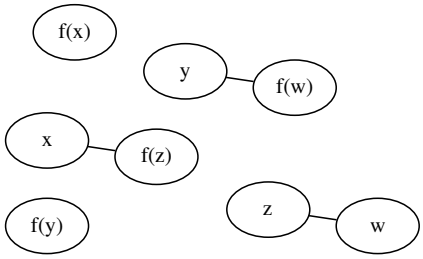
(a) Initial state of the algorithm, with an empty set of edges.



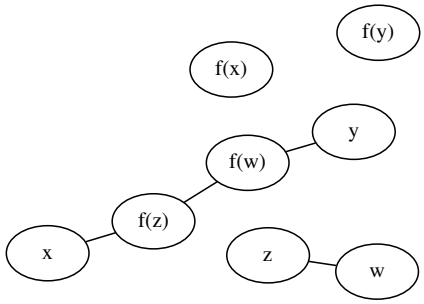
(b) Introduction of the edge $(x, f(z))$



(c) Introduction of the edge $(y, f(w))$



(d) Introduction of the edge (z, w)



(e) Final congruence graph

Figure 3.2. Computation of the congruence graph

of equalities, must be split into two partitions: $A \cup B$. Terms, equalities and formulas are *a-colorable* if all their non-logical symbols occur in A, and *b-colorable* if all their non-logical symbols occur in B. They are *colorable* if they are a or b-colorable, and *ab-colorable* if both. An edge (x, y) of a congruence graph has the same color as the equality $(x = y)$. A path in a congruence graph is colorable if all its edges are colorable, and a congruence graph is colorable if all its edges are colorable.

While it is possible to construct a non-colorable congruence graph, the following lemma and its constructive proof in Fuchs et al. [2009] state that we may assume without loss of generality that congruence graphs are colorable.

Lemma 1 (c.f. Fuchs et al. [2009])**.** *If x and y are colorable terms and if $A, B \models (x = y)$, then there exist a term set T and a colorable congruence graph over the equalities contained in $A \cup B$ and T in which $(x \sim y)$.*

Let A and B be two sets of equalities and disequalities. A *coloring* of a congruence graph $G = (E, T)$ created by a run of the congruence closure algorithm over the equalities and terms of $A \cup B$ is a function $C : E \to \{a, b\}$, that is, C assigns a color a or b to each edge, considering two restrictions: *(i)* basic edges e must be colored with a if $e \in A$ and with b if $e \in B$; and *(ii)* if an edge has color c, both its endpoints must be c-colorable. ab-colorable derived edges can be colored arbitrarily.

We denote a congruence graph G colored with a function C by G^C. A path is called an *a-path* if all its edges are colored a, and a *b-path* if all its edges are colored b. A *factor* of a path in G^C is a maximal subpath such that all its edges have the same color. Notice that every path is uniquely represented as a concatenation of the consecutive factors of opposite colors.

Example 4. *Let $A := \{(v_1 = f(y_1)), (f(y_2) = v_2), (y_1 = t_1), (t_2 = y_2), (s_1 = f(r_1)), (f(r_2) = s_2), (r_1 = u_1), (u_2 = r_2)\}$ and $B := \{(x_1 = v_1), (v_2 = x_2), (t_1 = f(z_1)), (f(z_2) = t_2), (z_1 = s_1), (r_1 = r_2), (s_2 = z_2), (u_1 = u_2), (x_1 \neq x_2)\}$. Fig. 3.3 shows a colored congruence graph G^C built while proving the unsatisfiability of A and B with Alg. 2. The congruence graph G^C demonstrates the joint unsatisfiability of A and B, since it proves $(x_1 = x_2)$ and $(x_1 \neq x_2)$ is an original term. Edges are represented by rectangles, and dotted arrows point to the parents of derived edges. Black rectangles and circles represent a-colorable nodes (terms) and a-colored edges, white edges and circles represent b-colorable nodes and b-colored edges by, and half filled circles represent ab-colorable nodes. In the first (top) path of G^C, we can see that basic edges (original equalities from $A \cup B$) are used to prove $(r_1 = r_2)$. This fact is used to infer $(f(r_1) = f(r_2))$,*

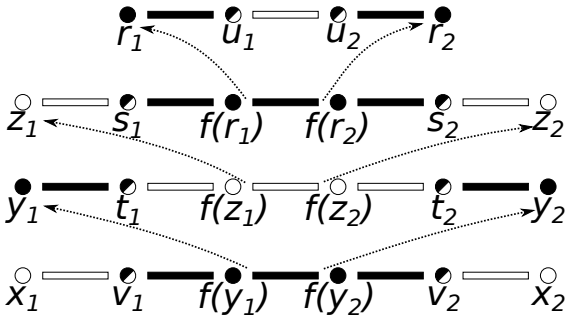


Figure 3.3. Congruence graph G^C that proves the unsatisfiability of $A \cup B$

which is in turn used as a derived edge in the path below, proving $(z_1 = z_2)$*. The equality* $(f(z_1) = f(z_2))$ *is then inferred and used to prove* $(y_1 = y_2)$ *in the path below. In the last (bottom) path of* G^C*, the derived edge representing* $(f(y_1) = f(y_2))$ *is created and finally* $(x_1 = x_2)$ *is proved.*

We discuss interpolant generation from congruence graphs in Chapter 5.

3.4 LRA Preliminaries

This section introduces the quantifier-free theory of Linear Real Arithmetic and the most common decision procedure for conjunctions of inequalities. We summarize Kroening and Strichman [2008] and Dutertre and de Moura [2006] with respect to the basics to understand the theory of LRA and the algorithm *general simplex*, used by most SMT solvers to decide satisfiability of LRA formulas.

Definition 3 (c.f. Kroening and Strichman [2008])**.** *The syntax of a formula in* linear arithmetic *is defined by the following rules:*

$$\begin{aligned} formula: &\ formula \ \wedge \ formula \mid (formula) \mid atom \\ atom: &\ sum\ op\ sum \\ sum: &\ term \mid sum \ + \ term \\ op: &= \ \mid \ < \ \mid \ \leq \\ term: &\ identifier \mid constant \mid constant\ identifier \end{aligned}$$

Even though the syntax may seem restrictive, some transformations can be done to achieve the common use of arithmetic. For example, subtraction is not used in Def. 3, but $x_1 - x_2$ can be read as $x_1 + (-1x_2)$. Also the relation

operators $>$ and $\geq$ can be replaced by $<$ and $\leq$ by negating the coefficients. This definition of linear arithmetic is valid for multiple domains, but in this thesis we are interested in the domain of rational numbers.

The *simplex* method is a widely used algorithm that aims at maximizing an objective function if a set of constraints is satisfiable. Here we interested in the decision problem of whether a set of constraints is satisfiable, rather than the maximization problem. For that, we use the *general simplex* algorithm. The general simplex algorithm allows two types of constraints as input: (i) equalities of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0, \tag{3.8}$$

and (ii) bounds for the variables:

$$l_i \leq x_i \leq u_i, \tag{3.9}$$

where l_i and u_i are real constants.

This representation of an input formula is called the *general form*. Representing input in this way does not affect the power of the algorithm, since it is possible to transform any linear constraint $L \bowtie R$ with $\bowtie \in \{=, \leq, \geq\}$ into the form described above in the following manner. Let m be the number of constraints. For each i-th constraint, where $1 \leq i \leq m$:
(i) Move all addends in R to the left-hand side obtaining $L' \bowtie b$, where b is a constant.
(ii) Introduce a new variable s_i, and add the constraints $L' - s_i = 0$ and $s_i \bowtie b$. If $\bowtie$ is $=$, replace $s_i = b$ by $s_i \leq b$ and $s_i \geq b$.

Example 5. *Consider the following conjunction of constraints:*

$$\begin{aligned} x_1 + x_2 &\geq 3 \\ x_1 - 2x_2 &\leq 0 \end{aligned}$$

The constraints are rewritten into general form:

$$\begin{aligned} x_1 + x_2 - s_1 &= 0 \\ x_1 - 2x_2 - s_2 &= 0 \\ s_1 &\geq 3 \\ s_2 &\leq 0 \end{aligned}$$

The m new variables s_1 and s_2 are called *additional variables*. The n variables that occur in the original constraints are called *problem variables*.

It is common to represent the input constraints as a m-by-$(n+m)$ matrix A, where each row represents a constraint, each column represents a variable, and the cells are the coefficients. The n problem variables and m additional variables are represented by the elements of a vector x of length $n+m$. Using this notation, the problem is equivalent to deciding whether there is a vector x such that

$$Ax = 0 \text{ and } \bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i, \tag{3.10}$$

where l_i and u_i are constants that bound the additional variables s_i.

Example 6. *Using the variable ordering x_1, x_2, s_1, s_2, the matrix representation for the constraints given in Example 5 is*

$$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 1 & -2 & 0 & -1 \end{pmatrix} \tag{3.11}$$

Notice that the part of the matrix corresponding to the additional variables is an m-by-m diagonal matrix, where the coefficients are -1. This is a direct consequence of applying the general form transformations on the original constraints. The matrix A changes throughout the computation of the algorithm, but the number of columns of this kind is always the same. The set of m variables corresponding to these columns are called *basic variables* and denoted by $\mathcal{B}$. The nonbasic variables are denoted by $\mathcal{N}$.

It is also convenient to use a *tableau* to manipulate matrix A. This tableau is matrix A without the diagonal matrix with -1 coefficients. The tableau is an m-by-n matrix, where the columns represent the nonbasic variables, and each row now represents the basic variable that has the -1 entry at that row in the diagonal sub-matrix, since this basic variable depends on the nonbasic ones.

Example 7. *Continuing Example 6, the tableau and the bounds so far are:*

	x_1	x_2	
s_1	*1*	*1*	$3 \leq s_1$
s_2	*1*	*-2*	$s_2 \leq 0$

The tableau is just a different way to represent matrix A, since $Ax = 0$ can be rewritten into

$$\bigwedge_{x_i \in \mathcal{B}} \left(x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \right). \tag{3.12}$$

The algorithm uses additionally an assignment $\alpha : \mathcal{B} \cup \mathcal{N} \to \mathbb{Q}$. The initialization process of the algorithm works as follows: (i) the set of additional

Algorithm 3 General Simplex

```
1: procedure GeneralSimplex(Set of constraints S)
2:     Transform the linear system into the general form.
3:     B ← set of additional variables s_1, ..., s_m.
4:     Construct the tableau.
5:     Decide a fixed order for the variables.
6:     while True do
7:         if every basic variable satisfies its bounds then
8:             return SAT
9:         else
10:            x_i ← first basic variable in the order that does not satisfy its
   bounds.
11:            x_j ← first suitable nonbasic variable in the order.
12:            if x_j could not be set then
13:                return UNSAT
14:            end if
15:        end if
16:        Apply the Pivot operation on x_i and x_j.
17:    end while
18: end procedure
```

variables is assigned to the set of basic variables $\mathcal{B}$; (ii) the set of problem variables is assigned to the set of nonbasic variables $\mathcal{N}$; (iii) $\alpha(x_i) = 0$ and $\alpha(s_j) = 0$, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

Algorithm 3 summarizes all the necessary steps for the decision. The main loop of the algorithm checks if all the bounds are satisfied. If that is the case, the problem is satisfiable and the algorithm terminates, and α represents an assignment satisfying the constraints.

Suppose now that basic variable x_i violates one if its bounds (according to α), and therefore its value needs to be adjusted. From Eq. 3.12, we know that

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j. \tag{3.13}$$

Assume, without loss of generality, that x_i violated its upper bound. We can reduce the value of x_i by decreasing the value of a nonbasic variable x_j such that $a_{ij} > 0$ and $\alpha(x_j) > l_j$, or by increasing the value of a nonbasic variable x_j such that $a_{ij} < 0$ and $\alpha(x_j) < u_j$. A nonbasic variable that satisfies at least one of these conditions is called *suitable*. If no suitable variable exists at this

point of the algorithm, the set of constraints is unsatisfiable and the algorithm terminates.

After finding a suitable x_j, we have to adjust its value so that x_i satisfies its bounds l_i and u_i. Let θ be this adjustment value:

$$\theta = \frac{u_i - \alpha(x_i)}{a_{ij}}. \tag{3.14}$$

By adjusting x_j, x_i now satisfies its bounds, but it may be the case that x_j does not satisfy its bounds anymore. Therefore, we need to swap x_i and x_j in the tableau. As a result, x_i becomes nonbasic and x_j becomes basic. This transformation is called the *pivot operation*. Notice that an analogous process can be applied if x_i violates its lower bound instead of its upper bound (substituting u_i by l_i).

Definition 4 (c.f. Kroening and Strichman [2008])**.** *Given two variables x_i and x_j, the coefficient a_{ij} is called the* pivot element*. The column of x_j is called the* pivot column*. The row i is called the* pivot row*.*

The pivot element must be nonzero in order to apply the swapping.

The pivot operation is done by following the steps: (i) solve row i for x_j; and (ii) for all rows $l \neq i$, eliminate x_j by using the equality for x_j obtained from row i.

Example 8. *We move on from Example 7. As part of the initialization of the algorithm, we have that $\alpha(x_i) = 0$ and $\alpha(s_i) = 0$ for every problem and additional variable. Recall the tableau and the bounds:*

	x_1	x_2	
s_1	*1*	*1*	$3 \leq s_1$
s_2	*1*	*-2*	$s_2 \leq 0$

The lower bound of s_1 is 3, which is violated since $\alpha(s_1) = 0$. The next nonbasic variable in the ordering is x_1, which has a positive coefficient and no upper bound, therefore it is suitable for pivoting. Variable s_1 has to be increased by 3, which means that also x_1 has to be increased by 3 ($\theta = 3$). This changes the assignment such that $\alpha = \{x_1 \mapsto 3, x_2 \mapsto 0, s_1 \mapsto 3, s_2 \mapsto 3\}$. As described before, step (i) of pivoting is solving s_1's row for x_1:

$$s_1 = x_1 + x_2 \Leftrightarrow x_1 = s_1 - x_2. \tag{3.15}$$

Now we can use this equality to replace x_1 in the other row:

$$s_2 = (s_1 - x_2) - 2x_2 \Leftrightarrow s_2 = s_1 - 3x_2. \tag{3.16}$$

The result of the pivot operation is:

	s_1	x_2
x_1	*1*	*-1*
s_2	*1*	*-3*

Now the upper bound of s_2 is violated. The next suitable variable for pivoting is x_2. Variable s_2 needs to be reduced by 3 to meet its requirements, which means that x_2 needs to be increased by 1 ($\theta = 1$). This changes the assignment such that $\alpha = \{x_1 \mapsto 2, x_2 \mapsto 1, s_1 \mapsto 3, s_2 \mapsto 0\}$.

Solving row 2 for x_2:

$$s_2 = s_1 - 3x_2 \Leftrightarrow x_2 = \frac{s_1 - s_2}{3}. \tag{3.17}$$

Substituting in row 1:

$$x_1 = s_1 - \frac{s_1 - s_2}{3} \Leftrightarrow x_1 = \frac{2s_1 - s_2}{3}. \tag{3.18}$$

The result of the pivot operation is:

	s_1	s_2
x_1	$\frac{2}{3}$	$\frac{1}{3}$
x_2	$\frac{1}{3}$	$-\frac{1}{3}$

Since all the basic variables satisfy their bounds, the problem is satisfiable with α containing a satisfying assignment.

Example 9. *We now modify the set of constraints from Example 5 such that it is unsatisfiable. We add the constraint $x_2 \leq 0$. The new set of constraints is:*

$$\begin{aligned} x_1 + x_2 &\geq 3 \\ x_1 - 2x_2 &\leq 0 \\ x_2 &\leq 0 \end{aligned}$$

The constraints are rewritten into the general form:

$$\begin{aligned} x_1 + x_2 - s_1 &= 0 \\ x_1 - 2x_2 - s_2 &= 0 \\ x_2 &\leq 0 \\ s_1 &\geq 3 \\ s_2 &\leq 0 \end{aligned}$$

In this run, the algorithm general simplex starts in the same way as in Example 8, fixing the assignment for s_1. Recall from Example 8 that the assignment at this point was $\alpha = \{x_1 \mapsto 3, x_2 \mapsto 0, s_1 \mapsto 3, s_2 \mapsto 3\}$ and the tableau was

	s_1	x_2
x_1	*1*	*-1*
s_2	*1*	*-3*

The upper bound of s_2 was violated. To fix the violation, we chose x_2 since s_1 was not suitable. In the current run x_2 is also not suitable because of its upper bound. Therefore the problem is unsatisfiable.

When the general simplex algorithm is used as the LRA solver in an SMT solver, it needs to create an explanation for the unsatisfiability of the set of constraints, which works as a proof of unsatisfiability. Let x_i be the variable that violated its bounds and could not be fixed, l_i its lower bound, and u_i its upper bound. Let $\mathcal{N}^+ = \{x_j \in \mathcal{N} \mid a_{ij} > 0\}$ and $\mathcal{N}^- = \{x_j \in \mathcal{N} \mid a_{ij} < 0\}$. Eq. 3.19 and Eq. 3.20 show how the explanation Γ is constructed, respectively, as Γ^l for a lower bound violation and as Γ^u for an upper bound violation, according to Dutertre and de Moura [2006].

$$\Gamma^l = \{x_j \leq u_j \mid x_j \in \mathcal{N}^+\} \cup \{x_j \geq l_j \mid x_j \in \mathcal{N}^-\} \cup \{x_i \geq l_i\}. \tag{3.19}$$

$$\Gamma^u = \{x_j \geq l_j \mid x_j \in \mathcal{N}^+\} \cup \{x_j \leq u_j \mid x_j \in \mathcal{N}^-\} \cup \{x_i \leq u_i\}. \tag{3.20}$$

In the case of Example 9, we have an upper bound violation for variable s_2, so we use Eq. 3.20. We have that $\mathcal{N}^+ = \{s_1\}$ and $\mathcal{N}^- = \{x_2\}$, so $\Gamma = \{s_1 \geq 3\} \cup \{x_2 \leq 0\} \cup \{s_2 \leq 0\}$. Since $s_1 = x_1 + x_2$ and $s_2 = x_1 - 2x_2$, $\Gamma = \{x_1 + x_2 \geq 3\} \cup \{x_2 \leq 0\} \cup \{x_1 - 2x_2 \leq 0\}$.

3.4.1 LRA Interpolation

The explanation is a minimal subset of the original constraints, and can be used to create a proof of unsatisfiability which leads to an interpolant, as follows.

A simple approach to create LRA interpolants is presented in McMillan [2005]. The idea is to create a tree that represents the proof of unsatisfiability of the set of constraints and annotate the nodes with partial interpolants, similar to what is done in propositional logic. A *term* in this system is a linear combination $c_0 + c_1v_1 + \ldots + c_nv_n$, where $v_1 \ldots v_n$ are distinct variables

and $c_0 \dots c_n$ are constants. This proof system accepts constraints of the form $0 \leq term$. Let x be a term $c_0 + c_1 v_1 + \ldots + c_n v_n$ and c a constant. The notation cx is equivalent to the term $c * c_0 + c * c_1 v_1 + \ldots + c * c_n v_n$. The rules of the proof system from McMillan [2005] are presented in Table 3.1, where x and y are terms:

Table 3.1. LRA proof system from McMillan [2005].

$$\text{Hyp}\quad \frac{}{\phi}\ \phi \in \Gamma \qquad \text{Comb}\quad \frac{0 \leq x \quad 0 \leq y}{0 \leq x + y}$$

$$\text{Taut}\quad \frac{}{0 \leq c}\ c \text{ is a constant, } c \geq 0 \qquad \text{Mult}\quad \frac{0 \leq c \quad 0 \leq x}{0 \leq cx}\ c \text{ is a constant, } c > 0$$

Example 10. *Using the rules above, we can create a proof tree for Example 9. First we have to transform our atoms into the form accepted by the proof system:*

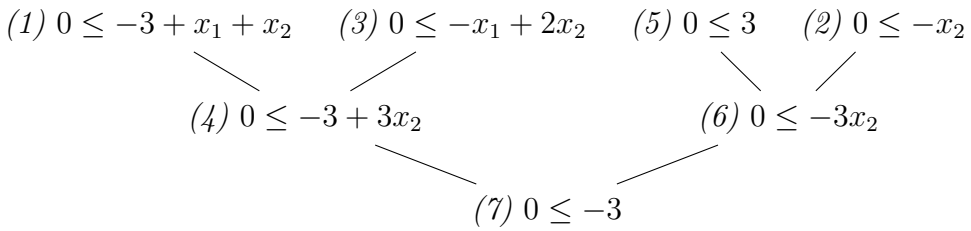
$$\begin{aligned} &(1)\ 0 \leq -3 + x_1 + x_2 \\ &(2)\ 0 \leq -x_2 \\ &(3)\ 0 \leq -x_1 + 2x_2 \end{aligned}$$

Applying the Hyp *rule to all the constraints is straightforward. We can then apply rule* Comb *to combine (1) and (3) and infer (4):*

$$\frac{0 \leq -3 + x_1 + x_2 \quad 0 \leq -x_1 + 2x_2}{(4)\ 0 \leq -3 + 3x_2}$$

We can apply rule Taut *to use the tautology (5)* $0 \leq 3$ *and then* Mult *to multiply (2) by (3) to infer (6)* $0 \leq -3x_2$*. Now we can apply* Comb *to combine (4) and (5) to finally infer (7)* $0 \leq -3$*, a contradiction.*

The tree representing this proof is the following:



The main idea in the interpolation procedure presented in McMillan [2005] is to use the contribution from the constraints from A to the sum that leads to the contradiction, where the contribution from A is effectively an interpolant. This contribution fulfills the requirements of an interpolant, because (i) it is clearly implied by A; (ii) when summed with B leads to the contradiction; (iii) when all the constraints from A are summed the local symbols from A are removed, therefore only common symbols from A and B remain.

The interpolation rules from McMillan [2005] are given in Table 3.2, where x, y, x' and y' are terms, and $[\phi]$ is the annotated term such that $0 \leq \phi$ is the partial interpolant for that node:

Table 3.2. Interpolation system from McMillan [2005].

$$\text{Hyp-A}\quad \frac{}{0 \leq x[x]}\ (0 \leq x) \in A \qquad \text{Hyp-B}\quad \frac{}{0 \leq x[0]}\ (0 \leq x) \in B$$

$$\text{Comb}\quad \frac{0 \leq x[x'] \quad 0 \leq y[y']}{0 \leq x + y[x' + y']} \qquad \text{Mult}\quad \frac{0 \leq c \quad 0 \leq x[x']}{0 \leq cx[cx']}$$

Example 11. *Suppose* $A = \{0 \leq -3 + x_1 + x_2, 0 \leq -x_1 + 2x_2\}$ *and* $B = \{0 \leq -x_2\}$. *We have shown in the previous examples that the conjunction of these sets of constraints is unsatisfiable, and how the rules of the proof system from McMillan [2005] are used to create a proof of unsatisfiability. Now we use the extended interpolation rules to generate an interpolant for* A.

Using the Hyp-A and Hyp-B rules we can infer the following partial interpolants:

$$\frac{}{(1)\ 0 \leq -3 + x_1 + x_2[-3 + x_1 + x_2]}\ (0 \leq -3 + x_1 + x_2) \in A$$

$$\frac{}{(3)\ 0 \leq -x_1 + 2x_2[-x_1 + 2x_2]}\ (0 \leq -x_1 + 2x_2) \in A$$

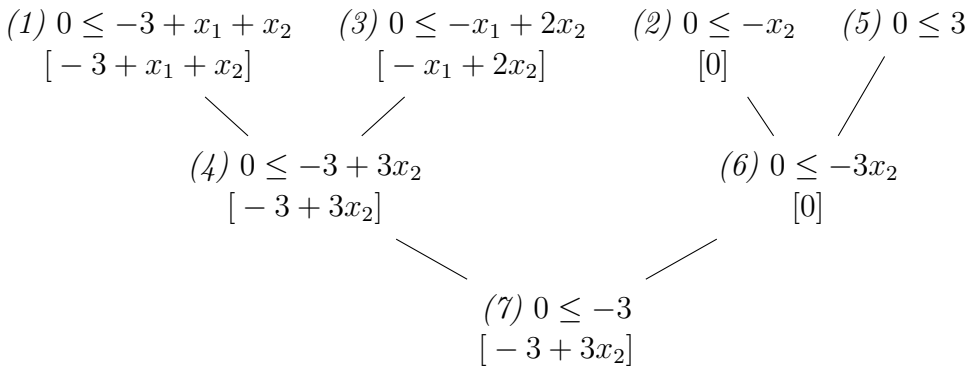
$$\frac{}{(2)\ 0 \leq -x_2[0]}\ (0 \leq -x_2) \in B$$

Following the proof from Example 10, we apply the Comb *rule on (3) and (1) to infer (4):*

$$\frac{0 \leq -3 + x_1 + x_2 \quad 0 \leq -x_1 + 2x_2}{(4)\ 0 \leq -3 + 3x_2[-3 + 3x_2]}$$

We now apply the Taut *rule to use the tautology (5)* $0 \leq 3$ *and then* Mult *rule on (5) and (2) to infer (5)* $0 \leq -3x_2[0]$. *By applying* Comb *on (4) and (5) we infer (6)* $0 \leq -3[-3 + 3x_2]$.

We have the following annotated proof of unsatisfiability:



The final interpolant for A *is* $0 \leq -3 + 3x_2$. *Notice that it contains only symbols that are common between* A *and* B. *The proof of unsatisfiability shows that it is the result of the sum of the constraints from* A, *which means it is implied by* A. *The proof also shows that when conjoined with* B, *the interpolant leads to a contradiction.*

We use this interpolation system and the duality of interpolants to create a new interpolation system for LRA, described in Chapter 6.

Chapter 4

Flexible and Controlled Propositional Interpolants

In SAT-based model checking, a widely used workflow for obtaining an interpolant for a propositional formula A is to compute a proof of unsatisfiability for the formula $\phi = A \wedge B$, use a variety of standard techniques for compressing the proof (see, e.g., Rollini et al. [2013]), construct the interpolant from the compressed proof, and finally simplify the interpolant Cabodi et al. [2015a]. LIS is a commonly used, flexible framework for computing the interpolant from a given proof that generalizes several interpolation algorithms parameterized by a labeling function. Given a labeling function and a proof, LIS uniquely determines the interpolant. However, the LIS framework allows significant flexibility in constructing interpolants from a proof through the choice of the labeling function.

Arguably, the suitability of an interpolant depends ultimately on the application Rollini et al. [2013], but there is a wide consensus that small interpolants lead to better overall performance in model checking Bloem et al. [2014]; Vizel et al. [2013]; Rollini et al. [2013]. However, generating small interpolants for a given partitioning is a non-trivial task. This chapter provides a thorough and rigorous analysis on how labeling functions in the LIS framework affect the size of propositional interpolants. Based on the analysis, the *proof-sensitive interpolation algorithm* PS is presented. PS produces small interpolants by adapting itself to the proof of unsatisfiability. We prove under reasonable assumptions that the resulting interpolant is always smaller than those generated by any other LIS-based algorithms, including the widely used algorithms M_s (McMillan McMillan [2003]), P (Pudlák Pudlák [1997]), and M_w (dual to M_s D'Silva et al. [2010]).

In some applications it is important to give guarantees on the logical strength of the interpolants. Since the LIS framework allows us to argue about the resulting interpolants by their logical strength D'Silva et al. [2010], we know that for a fixed problem $A \wedge B$ and a fixed proof of unsatisfiability, an interpolant constructed with M_s implies one constructed with P which in turn implies one constructed with M_w. While PS is designed to control the interpolant size, we additionally define two variants controlling the interpolant strength: the strong and the weak proof-sensitive algorithms computing, respectively, interpolants that imply the ones constructed by P and that are implied by the ones constructed by P.

We implemented the new algorithms in OpenSMT2, and confirm the practical significance of the algorithms with experiments presented in this chapter.

4.1 Labeling Functions for LIS

This section studies the algorithms based on the labeled interpolation system in (i) an experimental and (ii) an analytic setting. Our main objective is to provide motivation and a basis for developing and understanding labeling functions that construct interpolants having desirable properties. In particular, we will concentrate on three syntactic properties of the interpolants: the number of distinct variables; the number of literal occurrences; and the internal size of the interpolant. In most of the discussion in this section we will ignore the two classical optimizations on structural sharing and constraint simplification. While both are critically important for practicality of interpolation, our experimentation shows that they mostly have similar effect on all the interpolation algorithms we studied, and therefore they can be considered orthogonally (see Sec. 4.2.4). The exception is that *the non-uniform labeling functions allow a more efficient optimization compared to the uniform labeling functions* through constraint simplification. We recall from Def. 1 that a uniform labeling function consistently leads all the occurrences of a variable in a formula with the same label.

4.1.1 Analysing Labeling Functions Experimentally

We start this section by presenting an experimental work that motivated the following detailed theoretical analysis on labeling functions. We recall Example 2 from Chapter 3.1, where we prove the unsatisfiability of the formula ϕ:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1). \quad (4.1)$$

Please see Fig. 3.1 for the refutation proof. If we partition the formula ϕ such that $A = (x_1 \vee x_2)$ and $B = (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1)$, and interpolate using M_s and M_w, we have that the generated interpolants are, respectively, $Itp_{\mathrm{M}_s} = x_1 \vee x_2$ and $Itp_{\mathrm{M}_w} = (((\neg x_1 \wedge x_2) \vee x_1) \wedge x_2) \vee x_1$. If we change the partitioning such that $A' = B$ and $B' = A$, and interpolate over A' using the same refutation proof, we have that $Itp'_{\mathrm{M}_s} = (((x_1 \vee \neg x_2) \wedge \neg x_1) \vee \neg x_2) \wedge \neg x_1$ and $Itp'_{\mathrm{M}_w} = \neg(x_1 \vee x_2)$. It is straightforward to see that neither M_s or M_w was able to generate small interpolants after a simple change of partitioning.

Generating small interpolants is not a trivial task. Attempts on it include, for example, proof compression and interpolant simplification, heavy procedures that may lead to large overheads. Crafting labeling functions from LIS is a light way to generate interpolants with a specific property, as suggested by D'Silva et al. [2010] in the interpolation system D_{min}. In our research investigation regarding the problem of interpolants not being effective for program verification, we experimented with D_{min} and the abilities of LIS. D_{min} labels the literals by copying its class (a if it is in a clause from A or b otherwise) and is proven to generate interpolants with the least number of distinct variables.

Table 4.1 shows results of our preliminary experiments with the tool FunFrog using the interpolation algorithms M_s, P, M_w and D_{min}. FunFrog was used to verify safety properties in C programs that contain a complex function call tree structure, with assertions on various levels of the tree. Since FunFrog uses interpolation to create, store and reuse summaries for the program functions, the choice of interpolation algorithm may impact the number of refinements used in the process of verifying a certain property, which finally impacts the overall verification performance. Row *Time(s)* shows the total verification time that FunFrog spent to verify all the benchmarks using each labeling function, where the following row represents how much worse FunFrog was using each labeling function compared to the best of them (0). The row *Avg size* represents the average number of Boolean connectives in the interpolants generated using each of the labeling functions, where the following row shows how much larger were the interpolants generated using each labeling function compared to the one that led to the smallest average (0). Surprisingly, D_{min} behaved poorly and led to the worst verification time in FunFrog.

Based on this preliminary evidence, we conducted a thorough experimental analysis on the properties of labeling functions M_s, P and M_w. We used

the three labeling functions as the interpolation algorithm in two interpolation-based applications: (i) FunFrog, an incremental model checker, and (ii) eVolCheck, an upgrade checker. Both approaches use Craig interpolants to compute function summaries in the same manner as presented in Chapter 2.2, such that they are stored and reused. For eVolCheck, only M_s and P can be used, since M_w does not guarantee soundness of tree interpolants Rollini et al. [2012], a necessary property for upgrade checking. We used the model checkers to verify a set of C benchmarks characterized by a non-trivial call tree structure. These benchmarks contain assertions in different levels of the call-tree, which makes them particularly suitable for summarization-based verification. Tables 4.2 and 4.3 show, respectively, the performance results for FunFrog and eVolCheck, where *#Refinements* represents the number of function summaries that had to be discarded and recomputed, Avg$|I|$ is the average number of Boolean connectives in the interpolants, and *Time(s)* is the total time spent to verify all the benchmarks.

From Tables 4.2 and 4.3 we can observe the following. The strength of the interpolants have an impact on the convergence of the model checker. We can see that stronger interpolants lead to a smaller number of refinements for FunFrog, and weaker interpolants lead to a smaller number of refinements for eVolCheck. The other information we can extract from the experiments is that interpolants with a small number of Boolean connectives lead to a smaller verification time, as we can see from both tools. However, it is not clear how different labeling functions behave with respect to the size of the interpolants they generate.

The next section presents a theoretical analysis on the properties of labeling functions and how to derive small interpolants from LIS and refutation proofs simply by using a different labeling function.

	FunFrog			
	M_s	P	M_w	D_{min}
Time (s)	2333	3047	3207	3811
increase %	0	23	27	38
Avg size	48101	79089	86831	119306
increase %	0	39	44	59

Table 4.1. FunFrog experiments results using previous interpolation systems

Table 4.2. Performance results of FunFrog when using various labeling functions.

	M_s	P	M_w
#Refinements	290	298	308
Avg $\|I\|$	38886.62	39372.07	72994.08
Time(s)	4568.08	4929.93	6805.81

Table 4.3. Performance results of eVolCheck when using various labeling functions.

	M_s	P
#Refinements	65	63
Avg $\|I\|$	334554.64	377903.11
Time(s)	4322.57	4402.00

4.1.2 Analysing Labeling Functions Theoretically

An interesting special case in LIS-based interpolation algorithms is when the labeling can be used to reduce the number of distinct variables in the final interpolant. To make this explicit we define the concepts of a p-pure resolution step and a p-annihilable interpolation instance.

Definition 5. *Given an interpolation instance (R, A, B), a variable $p \in var(A) \cup var(B)$ and a labeling function L, a resolution step in R is* p-pure *if at most one of the antecedents contain p, or both antecedents C, D contain p but $L(p, C) = L(p, D) = a$ or $L(p, C) = L(p, D) = b$. An interpolation instance (R, A, B) is* p-annihilable *if there is a non-uniform labeling function L such that $L(p, C) = a$ if $C \in A$, $L(p, C) = b$ if $C \in B$, and all the resolution steps are p-pure.*

The following theorem shows the value of p-annihilable interpolation instances in constructing small interpolants.

Theorem 2. *Let (R, A, B) be an interpolation instance, $p \in var(A) \cap var(B)$, and I an interpolant obtained from (R, A, B) by means of a LIS-based algorithm. If $p \notin var(I)$, then (R, A, B) is p-annihilable.*

Proof. Assume that (R, A, B) is not p-annihilable, $p \in var(A) \cap var(B)$, but there is a labeling L which results in a LIS-based interpolation algorithm that constructs an interpolant not containing p. The labeling function cannot have $L(p, C) = b$ if $C \in A$ or $L(p, C) = a$ if $C \in B$ because p would appear in

the partial interpolants associated with the sources by Eq. (3.2). No clause C in R can have $L(p, C) = ab$ since all literals in the refutation need to be used as a pivot on the path to the empty clause, and having an occurrence of p labeled ab in an antecedent clause would result in introducing the literal p to the partial interpolant associated with the resolvent by Eq. (3.3) when used as a pivot. Every resolution step in the refutation R needs to be p-pure, since if the antecedents contain occurrences (p, C) and (p, D) such that $L(p, C) \neq L(p, D)$ either the label of the occurrence of p in the resolvent clause will be ab, violating the condition that no clause can have $L(p, C) = ab$ above, or, if p is pivot on the resolution step, the variable is immediately inserted to the partial interpolant by Eq. (3.3). □

While it is relatively easy to artificially construct an interpolation instance that is p-annihilable, they seem to be rare in practice (see Section 4.2.4). Hence, while instances that are p-annihilable would result in small interpolants, it has little practical significance at least in the benchmarks available to us. However, we have the following practically useful result which shows the benefits of labeling functions producing p-pure resolution steps in computing interpolants with low number of connectives.

Theorem 3. *Let (R, A, B) be an interpolation instance. Given a labeling function L such that the resolution steps in R are p-pure for all $p \in var(A \wedge B)$, and a labeling function L' such that at least one resolution step in R is not p-pure for some $p \in var(A \wedge B)$, we have $IntSize(ItpL) \leq IntSize(ItpL')$.*

Proof. For a given refutation R, the number of partial interpolants will be the same for any LIS-based interpolation algorithm. By Eq. (3.3) each resolution step will introduce one connective if both occurrences in the antecedents are labeled a or b and three connectives otherwise. The latter can only occur if the labeling algorithm results in a resolution step that is not p-pure for some p. □

Clearly, p-pure steps are guaranteed with uniform labeling functions. Therefore we have the following corollary:

Corollary 3.1. *Uniform labeling functions result in interpolants with smaller internal size compared to non-uniform labeling functions.*

4.1.3 Proof-Sensitive Interpolation

The main result of this chapter is the development of a labeling function that is uniform, therefore producing small interpolants by Corollary 3.1, and results in

the smallest number of variable occurrences among all uniform labeling functions. This *proof-sensitive labeling function* works by considering the refutation R when assigning labels to the occurrences of the shared variables.

Definition 6. *Let R be a resolution refutation for $A \wedge B$ where A and B consist of the source clauses, $f_A(p) = |\{(p, C) \mid C \in A\}|$ be the number of times the variable p occurs in A, and $f_B(p) = |\{(p, C) \mid C \in B\}|$ the number the variable p occurs in B. The proof-sensitive labeling function L_{PS} is defined as*

$$L_{\mathrm{PS}}(p, C) = \begin{cases} a & \text{if } f_A(p) \geq f_B(p) \\ b & \text{if } f_A(p) < f_B(p). \end{cases} \tag{4.2}$$

Note that since L_{PS} is uniform, it is independent of the clause C. Let Sh_A be the set of the shared variables occurring at least as often in clauses of A as in B and Sh_B the set of shared variables occurring more often in B than in A:

$$\begin{aligned} Sh_A &= \{p \in var(A) \cap var(B) \mid f_A(p) \geq f_B(p)\} \quad \text{and} \\ Sh_B &= \{p \in var(A) \cap var(B) \mid f_A(p) < f_B(p)\} \end{aligned} \tag{4.3}$$

Theorem 4 states the optimality with respect to variable occurrences of the algorithm PS among uniform labeling functions.

Theorem 4. *For a fixed interpolation instance (R, A, B), the interpolation algorithm* PS *will introduce the smallest number of variable occurrences in the partial interpolants associated with the source clauses of R among all uniform interpolation algorithms.*

Proof. The interpolation algorithm PS is a uniform algorithm labeling shared variables either as a or b. Hence, the shared variables labeled a will appear in the partial interpolants of the source clauses from B of R, and the shared variables labeled b will appear in the partial interpolants of the source clauses from A of R. The sum of the number of variable occurrences in the partial interpolants associated with the source clauses by PS is

$$n_{\mathrm{PS}} = \sum_{v \in Sh_B} f_A(v) + \sum_{v \in Sh_A} f_B(v).$$

We will show that swapping uniformly the label of any of the shared variables will result in an increase in the number of variable occurrences in the partial interpolants associated with the source clauses of R compared to n_{PS}. Let v be a variable in Sh_A. By (4.2) and (4.3), the label of v in PS will be a. Switching the label to b results in the size $n' = n_{\mathrm{PS}} - f_B(v) + f_A(v)$. Since v was

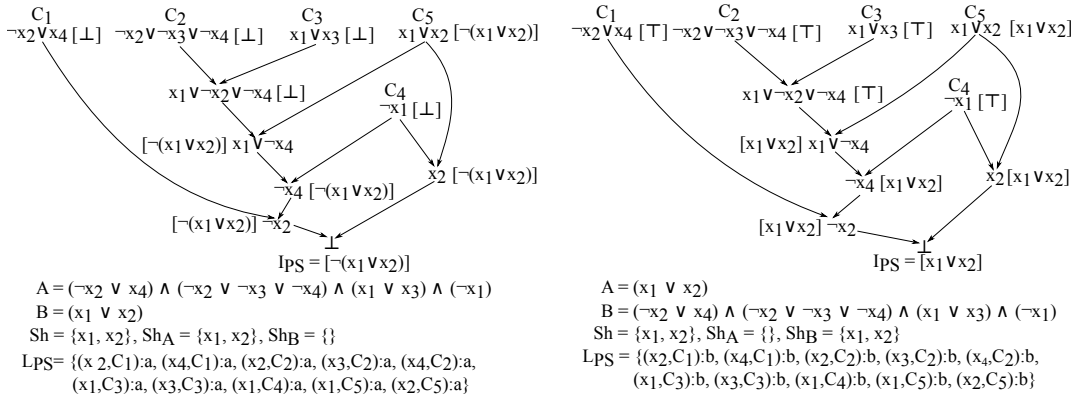


Figure 4.1. Interpolants obtained by PS.

in Sh_A we know that $f_A(v) \geq f_B(v)$ by (4.3), and therefore $-f_B(v)+f_A(v) \geq 0$ and $n' \geq n_{\mathrm{PS}}$. An (almost) symmetrical argument shows that swapping the label for a variable $v \in Sh_B$ to a results in $n' > n_{\mathrm{PS}}$. Hence, swapping uniformly the labeling of PS for any shared variable will result in an interpolant having at least as many variable occurrences in the leaves. Assuming no simplifications, the result holds for the final interpolant. □

Example 12. *Fig. 4.1 shows the interpolants that* PS *would deliver if applied to the same refutation R of ϕ and partitionings $A \wedge B$ and $A' \wedge B'$ given in Example 2. Notice that* PS *adapts the labeling to the best one depending on the refutation and partitions, and gives small interpolants for both cases.*

From Theorems 2, 3 and 4 we immediately have the following:

Corollary 4.1. *For not p-annihilable interpolation instances, for any variable p, the proof-sensitive labeling function will result in interpolants that have the smallest internal size, the least number of distinct variables compared to any other labeling function from LIS, since only p-annihilable interpolation instances are capable of variable elimination, and least variable occurrences in the source partial interpolants.*

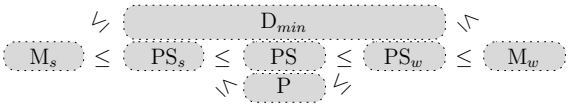
Because of the way L_{PS} labels the variable occurrences, we cannot beforehand determine the strength of PS relative to, e.g., the algorithms M_s, P, and M_w. Although it is often not necessary that interpolants have a particular strength, in some applications this has an impact on performance or even soundness Rollini et al. [2013]. To be able to apply the idea in applications requiring specific interpolant strength, for example tree interpolation, a weak

and a strong version of the proof-sensitive interpolation algorithm are also introduced, PS_w and PS_s. The corresponding labeling functions L_{PS_w} and L_{PS_s} are defined as

$$L_{\mathrm{PS}_w}(p, C) = \begin{cases} a & \text{if } p \text{ is not shared and } C \in A \text{ or } p \in Sh_A \\ b & \text{if } p \text{ is not shared and } C \in B \\ ab & \text{if } p \in Sh_B \end{cases} \tag{4.4}$$

$$L_{\mathrm{PS}_s}(p, C) = \begin{cases} a & \text{if } p \text{ is not shared and } C \in A \\ b & \text{if } p \text{ is not shared and } C \in B, \text{ or } p \in Sh_B \\ ab & \text{if } p \in Sh_A \end{cases} \tag{4.5}$$

Finally, it is fairly straightforward to see based on the definition of the labeling functions that the strength of the interpolants is partially ordered as shown in the diagram below.



4.2 Experimental Evaluation

We compared the seven labeling functions for propositional interpolation described in this chapter in the context of three different model-checking tasks: (*i*) incremental software model checking with function summarization using FunFrog; (*ii*) checking software upgrades with function summarization using eVolCheck; and (*iii*) pre-image over-approximation for hardware model checking with PdTRAV Cabodi et al. [2006]. The wide range of experiments permits the study of the general applicability of the new techniques. In experiments (*i*) and (*ii*) the new algorithms are implemented within the verification process allowing us to evaluate their effect on the full verification run. Experiment (*iii*) focuses on the size of the interpolant, treating the application as a black box. Unlike in the theory presented in Section 4.1, all experiments use both structural sharing and constraint simplification, since the improvements given by these practical techniques are important. Experiments (*i*) and (*ii*) use a large set of benchmarks each containing a different call-tree structure and assertions distributed on different levels of the tree. For (*iii*), the benchmarks consisted of a set of 100 interpolation problems constructed by PdTRAV. All experiments use OpenSMT2 both as the interpolation engine and as the SAT solver.

Different verification tasks may require different kinds of interpolants. For example, Rollini et al. [2013] reports that the FunFrog approach works best with strong interpolants, whereas the eVolCheck techniques rely on weaker interpolants that have the tree-interpolation property. As shown in Rollini et al. [2012], only interpolation algorithms stronger than or equisatisfiable to P are guaranteed to generate sound tree interpolants. The tree interpolation strength requirement is restricted to this technique, and does not apply to other similar interpolation problems, such as sequence interpolants. Therefore, we evaluated only M_s, P and PS_s for (*ii*), and M_s, P, M_w, PS, PS_w and PS_s for (*i*) and (*iii*). D_{min} was evaluated against the other algorithms for (*i*), but couldn't be evaluated for (*ii*) because it does not preserve the tree interpolation property. For (*iii*), D_{min} was not evaluated due to its poor performance in (*i*).

In the experiments (*i*) and (*ii*), the overall verification time of the tools and average size of interpolants were analysed. For (*iii*) only the size was analysed. In all the experiments the size of an interpolant is the number of connectives in its DAG representation.

4.2.1 Interpolants as Function Summaries

FunFrog and eVolCheck are SAT-based model checkers for C implement the approach presented in Chapter 2.2, using interpolants to represent summaries of program functions. FunFrog is based on the incremental checking approach, whereas eVolCheck relies on the upgrade checking technique.

Experiments with FunFrog. The set of benchmarks consists of 23 C programs with different number of assertions. Each benchmark contains a complex function call tree with assertions on different levels of the tree. FunFrog verified the assertions one-by-one incrementally traversing the program call tree. The main goal of ordering the checks this way is to maximize the reuse of function summaries and thus to test how the labeling functions affect the overall verification performance. To illustrate our setting, consider a program with the chain of nested function calls

$$main()\{f()\{g()\{h()\{\}assert_g\}assert_f\}assert_{main}\},$$

where $assert_F$ represents an assertion in the body of function F. In a successful scenario, (a) $assert_g$ is detected to hold and a summary I^h for function h is created; (b) $assert_f$ is efficiently verified by exploiting I^h, and I^g is then built over I^h; and (c) finally $assert_{main}$ is checked against I^g. In this set of benchmarks, generating good interpolants for the first assertions can strongly

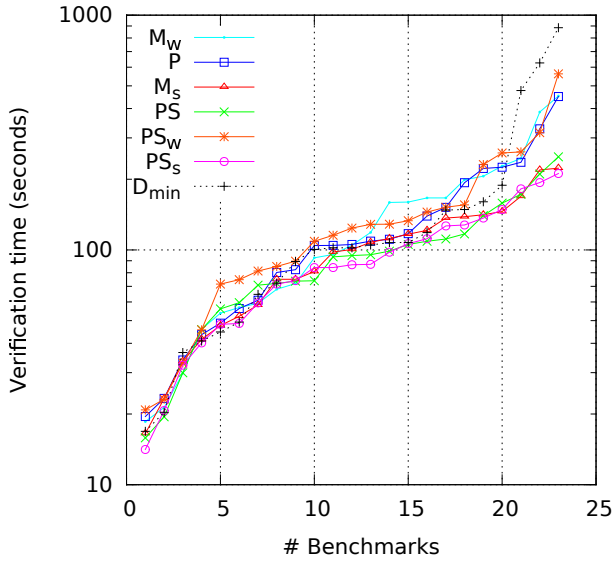


Figure 4.2. Overall verification time of FunFrog using different interpolation algorithms.

impact the chain verification of the subsequent assertions, given its complex characteristic.

Fig. 4.2 shows FunFrog's performance with each interpolation algorithm. Each curve represents an interpolation algorithm, and each point on the curve represents one benchmark run using the corresponding interpolation algorithm, with its verification time on the vertical axis. The benchmarks are sorted by their run time. The PS and PS_s curves are mostly lower than those of the other interpolation algorithms, suggesting they perform better. Table 4.4 shows the sum of FunFrog verification time for all benchmarks and the average size of all interpolants generated for all benchmarks for each interpolation algorithm. We also report the relative time and size increase in percents. Both PS and PS_s are indeed competitive for FunFrog, delivering interpolants smaller than the other interpolation algorithms. Table 4.5 shows FunFrog verification time for each benchmark (row) and labeling function, together with the average size of the interpolants used to prove that benchmark. In each row, the smallest verification time and interpolant size are highlighted. The labeling functions PS_s and PS have the most wins, 7 and 5 respectively. Notice that in most of the cases the labeling function that had the smallest size of interpolants had either the smallest verification time or a verification time close to the smallest. The latter happened in several benchmarks for PS.

Table 4.4. Sum of overall verification time and average interpolants size for FunFrog using the applicable labeling functions.

	FunFrog						
	M_s	P	M_w	PS	PS_w	PS_s	D_{min}
Time (s)	2333	3047	3207	2272	3345	**2193**	3811
increase %	6	39	46	3	52	0	74
Avg size	48101	79089	86831	43781	95423	**40172**	119306
increase %	20	97	116	9	137	0	197

Experiments with eVolCheck. The benchmarks consist of the ones used in the FunFrog experiments and their upgrades. We only experiment with M_s, P and PS_s since eVolCheck requires algorithms at least as strong as P. Fig. 4.3 demonstrates that PS_s, represented by the lower curve, outperforms the other algorithms also for this task. Table 4.6 shows the total time eVolCheck requires to check the upgraded versions of all benchmarks and average interpolant size for each of the three interpolation algorithms. Also for upgrade checking, the interpolation algorithm PS_s results in smaller interpolants and lower run times compared to the other studied interpolation algorithms. Table 4.7 shows detailed information for each benchmark. Each row shows the verification time for eVolCheck using each of the labeling functions, and the average size of the interpolants used to prove that benchmark, with the best of each highlighted. The performance of PS_s is even better in this experiment, being the winner in 16 out of 23 benchmarks. Even when PS_s does not lead to the best verification time, it is close to the best.

4.2.2 Over-approximating pre-image for Hardware Model Checking

PdTRAV Cabodi et al. [2006] implements several verification techniques including a classical approach of unbounded model checking for hardware designs **?**. Given a design and a property, the approach encodes the existence of a counterexample of a fixed length k into a SAT formula and checks its satisfiability. If the formula is unsatisfiable, proving that no counterexample of length k exists, Craig interpolation is used to over-approximate the set of reachable states. If the interpolation finds a fixpoint, the method terminates reporting safety. Otherwise, k is incremented and the process is restarted.

Table 4.5. Detailed information about the benchmarks ran with FunFrog.

	M_s		P		M_w		PS		PS_w		PS_s		D_{min}	
	Time(s)	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**
B1	74.9	37971	139.3	63254	206.1	187719	**70.8**	**30707**	133.1	62945.7	71.5	31016	102	53444
B2	138.5	122576	104.3	44989	104.8	40816	117	39132	562.1	548079	**86.8**	**35578**	100.5	46755
B3	16.5	9907	19.5	9584	18.5	8546	15.8	**6734**	20.8	15082	**14.1**	7212	16.8	7204
B4	23.2	6624	23.3	6527	21.3	4493	**19.4**	**3232**	23.1	4837	20.6	4477	20.3	3858
B5	47.6	5212	56.3	9102	46	4797	45.8	**4738**	45.6	4812	48.2	5093	**44.7**	4815
B6	111.8	16533	193.4	91678	95.7	15749	95.4	**14819**	**74.7**	14943	86.5	22315	89.1	15745
B7	223.1	144065	222.4	178879	245.4	**92290**	210.5	121826	258.7	95589	**193.6**	93154	883.8	674142
B8	107.7	30219	**104.5**	**23108**	199	113230	108.8	25691	108.8	26788	106.3	25811	107.4	26571
B9	121.1	62334	**80.1**	**27965**	159.3	109700	94.6	42685.625	152	101364	97.7	28572	160.4	130782
B10	141.2	69821	105.5	86536	166.4	56268	**105.4**	82694.5	128.5	84897	126.5	**56207**	107.6	83719
B11	**78.7**	**26970**	82.7	30391	131	90565	91.3	43326	175.4	83770	107.6	46880	183.9	82335
B12	169.1	86978	236.3	235021	**166.5**	160913	172.5	**76335**	315.8	343754	211.6	160716	626.3	874305
B13	97.9	56230	225	104427	386.8	282949	98.8	**39545**	115.5	43162	**84.1**	40424	188.5	86785.625
B14	52.1	16928	48.7	11805	53.7	12549	73.9	49699	261.5	312589	**48.6**	**11229**	49.2	11851
B15	58.2	34815	61	23060	59.2	29235	**56.1**	**21756**	84.9	57362	59.4	29967	64.6	24567
B16	74.9	23852	109	104525	71.9	**19554**	72	19632	**71.5**	19585	74	20350	72.1	19657
B17	100.5	54864	110	43687	**92.6**	43923	93.4	**37804**	128.6	137225	111.5	63365	104.9	46823
B18	**145.7**	53491	151.8	49373	159.8	60094	158.3	**47654**	155.6	48620	148.6	48272	148.7	50842
B19	136.6	91684	450.1	370456	227.7	288637	139.9	**80770**	145	82786	**127.7**	81274	147.8	81937
B20	33.5	7602	33.9	6806	**29.8**	6079	29.9	**6021**	32.9	7884	32.3	6330	36.5	11843
B21	41.6	9696	43.5	10378	56.8	22996	59.6	23954	81.3	39746	**40.1**	**7921**	40.9	8342
B22	80.9	13120	82.4	16125	**67.9**	**10799**	73.6	14428	89.5	31998	84.1	17383	102.1	46530
B23	116.7	41730	117.2	40410	118.6	39881	**111.3**	**39804**	123.8	47511	136.	41431	118.5	41096

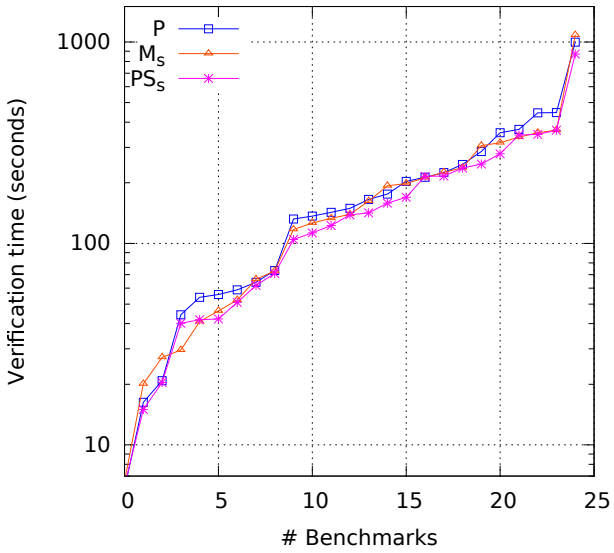


Figure 4.3. Overall verification time of eVolCheck using different interpolation algorithms.

Table 4.6. Sum of overall verification time and average interpolants size for eVolCheck using the applicable labeling functions.

	eVolCheck		
	M_s	PS_s	P
Time (s)	4867	**4422**	5081
increase %	10	0	16
Avg size	246883	**196716**	259078
increase %	26	0	32

Table 4.7. Detailed information for the benchmarks run with eVolCheck

	M		**P**		**PSs**	
	Time(s)	**Size**	**Time(s)**	**Size**	**Time(s)**	**Size**
B1	117.3	181954	165.5	319241	**104.6**	**155759**
B2	27.2	82183	20.8	41203	**20.4**	**34608**
B3	66.4	89123	64.2	85081	**61.9**	**64025**
B4	46.2	37877	58.8	66023	**41.8**	**22568**
B5	237.3	32822	246.8	48865	**236.6**	**32385**
B6	73.1	180422	73.4	155281	**70.9**	**141749**
B7	6.6	817	**6.0**	**619**	6.3	745
B8	**364.8**	498117	446.2	682687	365.1	**454416**
B9	139.5	136516	149.3	192747	**112.9**	**118989**
B10	193.2	169636	203.3	156927	**169.4**	**107000**
B11	**212.0**	398903	224.6	298583	216.1	**254480**
B12	**337.1**	332379	354.3	**307082**	347.6	326373
B13	354.5	353376	444.7	525151	**345.5**	**300324**
B14	316.3	683982	368.2	842993	**278.8**	**589807**
B15	133.3	**157633**	**132.2**	163467	138.5	194351
B16	305.6	493254	285.8	237692	**247.7**	**196568**
B17	**29.5**	**799**	54.0	180849	40.1	29723
B18	225.5	210548	**213.1**	**165406**	215.2	167407
B19	20.1	70870	16.3	39637	**15.0**	**32528**
B20	52.4	**56261**	55.8	66972	**51.0**	65388
B21	1082.2	1148482	998.8	1099093	**871.8**	**846222**
B22	198.2	287381	175.7	212919	**158.5**	**194488**
B23	126.3	167410	136.7	207696	**122.8**	**155301**

Table 4.8. Average size and increase relative to the winner for interpolants generated when interpolating over A (top) and B (bottom) in $A \wedge B$ with PdTRAV.

	PdTRAV					
	M_s	P	M_w	PS	PS_w	PS_s
Avg size	683233	724844	753633	**683215**	722605	685455
increase %	0.003	6	10	0	6	0.3
Avg size	699880	694372	649149	**649013**	650973	692434
increase %	8	7	0.02	0	0.3	7

Experiments. For this experiment, the benchmarks consist of interpolation instances generated by PdTRAV. We compare the effect of applying different interpolation algorithms on the individual steps of the verification procedure.

Table 4.8 (top) shows the average size of the interpolants generated for all the benchmarks using each interpolation algorithm, and the relative size compared to the smallest interpolant. Also for these approaches the best results are obtained from M_s, PS and PS_s, with PS being the overall winner. We note that M_s performs better than M_w likely due to the structure of the interpolation instances in these benchmarks: the partition B in $A \wedge B$ is substantially larger than the partition A. This structure favors algorithms that label many literals as b, since the partial interpolants associated with the clauses in B will be empty while the number of partial interpolants associated with the partition A will be small. To further study this phenomenon we interchanged the partitions, interpolating this time over B in $A \wedge B$ for the same benchmarks resulting in problems where the A part is large. Table 4.8 (bottom) shows the average size of the interpolants generated for these benchmarks and the relative size difference compared to the winner. Here M_w and PS_w perform well, while PS remains the overall winner.

PdTRAV experiments confirm in addition that PS is very capable in adapting to the problem, giving best results in both cases while the others work well in only one or the other.

We conclude that the experimental results are compatible with the analysis in Sec. 4.1. In the FunFrog and eVolCheck experiments, PS_s outperformed the other interpolation systems with respect to verification time and interpolant size.

4.2.3 Strength of PS

Section 4.1 states that we cannot know for sure the strength of the PS algorithm, because it uses both a and b labels. In order to have an idea of what kind of interpolants Itp_{PS} would deliver with respect to strength, the interpolants used in approaches i) and ii) were also stored and compared to Itp_P.

In such a comparison, four cases may arise:

1. $Itp_{PS} \rightarrow Itp_P$. Meaning that Itp_{PS} is stronger than Itp_P. This case did not happen for any benchmark.

2. $Itp_P \rightarrow Itp_{PS}$. Meaning that Itp_{PS} is weaker than Itp_P. This case happened in 35 benchmarks.

3. $Itp_{PS} \leftrightarrow Itp_P$. Meaning that Itp_{PS} is as strong as Itp_P (and also as weak as). This case happened in 50 benchmarks.

4. No implication. Nothing can be said about Itp_{PS}'s strength. This case happened in 219 benchmarks.

We observed that out of 50 proofs from approach *i)* and 254 proofs from approach *ii)*, in most of them Itp_{PS} was equivalent or weaker than Itp_P. So even though it was shown in Section 4.1 that we cannot know precisely the strength of Itp_{PS}, this experimentation gives us evidence that we can expect Itp_{PS} to be a slightly weaker than Itp_P.

4.2.4 Effects of Simplification

It is interesting to note that in our experiments the algorithm PS was not always the best, and the non-uniform interpolation algorithm PS_s sometimes produced the smallest interpolant, seemingly contradicting Corollary 4.1. A possible reason for this anomaly could be in the small difference in how constraint simplification interacts with the interpolant structure. Assume, in Eq. (3.3), that $I(C^+)$ or $I(C^-)$ is either constant true or false. As a result in the first and the second case respectively, the resolvent interpolant size decreases by one in Eq. (3.3). However in the third case, potentially activated only for non-uniform algorithms, the simplification if one of the antecedents' partial interpolants is false decreases the interpolant size by two, resulting in partial interpolants with smaller internal size. Therefore, in some cases, the good simplification behavior of non-uniform algorithms such as PS_s seems to result in slightly smaller interpolants compared to PS. We believe that this is also the reason why P behaves better than M_s and M_w in some cases.

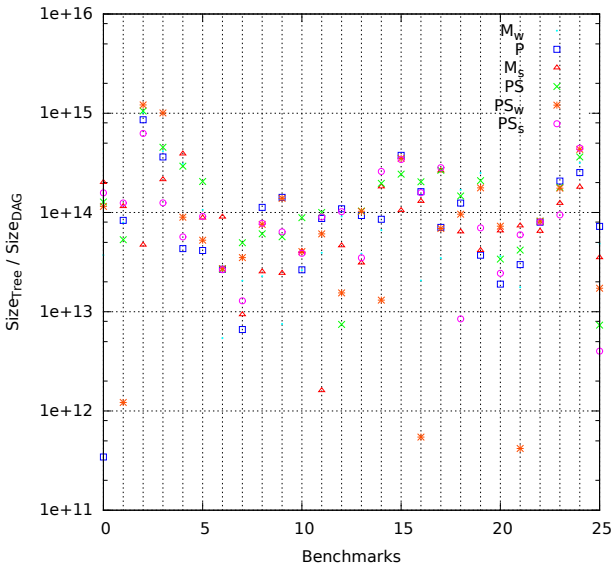


Figure 4.4. Relation $Size_{Tree}/Size_{DAG}$ on FunFrog benchmarks for different interpolation algorithms

We also observed that in only five of the benchmarks a labeling function led to interpolants with less distinct variables, the difference between the largest and the smallest number of distinct variables being never over 3%, suggesting that p-annihilable interpolation instances are rare. Finally, we measured the effect of structural sharing. To investigate the effect of structural sharing on simplifications, we analysed two parameters: the number of connectives in an interpolant on its pure tree representation ($Size_{Tree}$), and the number of connectives in an interpolant on its DAG representation ($Size_{DAG}$), which is the result of the application of structural sharing. Thus, we believe that the ratio $Size_{Tree}/Size_{DAG}$ is a good way to measure the amount of simplifications due to structural sharing.

Fig. 4.4 shows the results of this analysis on FunFrog benchmarks. Each vertical line represents a benchmark, and each point on this line represents the ratio $Size_{Tree}/Size_{DAG}$ of the interpolant generated by each of the interpolation algorithms for the first assertion of that benchmark. The reason why only the first assertion is considered is that from the second assertion on, summaries (that is, interpolants) are used instead of the original code, and therefore it is not guaranteed that the refutations will be the same when different interpolation algorithms are applied.

It is noticeable that the existence of more/less simplifications is not related

to the interpolation algorithms, since all of them have cases where many/few simplifications happen. Therefore, there is no difference between any of the algorithms with respect to structural sharing.

4.3 Related work

This section describes interpolation algorithms related to propositional logic. For work related to other theories please see Sections 5.1.2, 5.4, and 6.3.

Different aspects of interpolation for propositional logic have been studied since its introduction in model checking in McMillan [2003]. One of the most important recent results is D'Silva [2010], where LIS (Labeled Interpolation Systems) is introduced. LIS is a framework capable of generating multiple interpolants for a single proof of unsatisfiability. This is done by the application of different *labeling functions* which can also control the strength of the generated interpolants. Two specific labeling functions from LIS are also able to generalize the previous interpolation algorithms from Pudlák [1997] and McMillan [2005]. The authors also present two new labeling functions: D_{min}, a labeling function that generates interpolants with the smallest amount of distinct variables in the framework, and a labeling function that behaves as the dual of McMillan [2005]. The Proof-Sensitive interpolation algorithms described in this chapter are built on top of LIS.

A comparison between the interpolation algorithms from Pudlák [1997], McMillan [2005] and its dual when used by two different model checkers was done in Rollini et al. [2013]. The two applications used interpolants to compute function summaries for incremental verification and upgrade checking. It was shown that for one of the applications a logically stronger interpolation algorithm led to less refinements, whereas for the other application this was achieved by a logically weaker interpolation algorithm. This led to the conclusion that the suitability of interpolants depends on the application.

Even though it is hard to say whether an interpolant is good or not based only on its logical strength, there is a consensus that the size of the interpolant might have a direct impact on the performance of the interpolation-based application. There are different techniques that try to reduce the size of interpolants.

Interpolants can be compacted through applying transformations to the refutation proof. The work presented in Rollini et al. [2012] applies several proof compression techniques in order to reduce the size of the interpolants generated using that proof. The approach works well, as shown in Rollini et al. [2013], but compressing a proof requires heavy computation, and this might be

an impractical overhead in many cases.

It is also possible to apply post-interpolation reduction. In Cabodi et al. [2015a] circuit reduction techniques such as ODC, BDD-based sweeping and Constant Propagation are used to reduce the size of propositional interpolants. Also this technique, even though often successful, is computationally expensive and may create an overhead.

The Proof-Sensitive interpolation algorithms presented in this chapter are a lightweight alternative to reduce the size of interpolants, since they require little extra computation and are proved to generate small interpolants.

A significant reduction in the size of the interpolant can be obtained by considering only CNF-shaped interpolants Vizel et al. [2013]. The main idea of this technique is to first compute an approximation for the interpolant. Then, as a second stage, inductive reasoning is applied to transform the interpolant approximation into a sound interpolant. However, the strength of these interpolants is not as easily controllable as in the LIS interpolants, making the technique harder to apply in certain model checking approaches.

The extension of LIS presented in Jancik et al. [2014] enables new optimization methods for interpolation-based model checkers. By specifying a truth assignment, the application can make a single SAT query and generate interpolants for different interpolation problems for the same proof, reducing the number of SAT queries and increasing its performance. This work has an orthogonal goal compared to the Proof-Sensitive interpolation algorithms, and can be used together.

In Rümmer and Subotić [2013] a semantic framework to explore different interpolants is presented, with the goal of finding suitable interpolants that ensure fast convergence for model checkers. The semantic framework is based on the notion of interpolation abstraction. The authors show that by using interpolation abstraction with interpolant templates they were able to increase convergence of interpolation-based applications. Even though the goal of most work related to interpolation is to find more suitable interpolants, this technique is orthogonal to the Proof-Sensitive interpolation algorithms in the sense that they use different methods to achieve that goal, which leads to different results.

A related approach for generalizing interpolants in unbounded model-checking through abstraction is presented in Cabodi et al. [2006] using incremental SAT solving. While this direction is orthogonal to ours, we believe that the ideas presented here and addressing the interpolation back-end would be useful in connection with the generalization phase.

4.4 Summary and Future Work

Even though reducing the size of propositional interpolants has been studied using different approaches, a lightweight technique that is able to generate small interpolants without leading to a considerable computation overhead was missing.

This chapter presented a theoretical study on properties of labeling functions from the LIS framework. More specifically, the main results are the following theorems.

(i) If an interpolation instance is not p-annihilable (see Def. 5), which in our experimentation turns out almost always to be the case, then all LIS interpolants constructed from the refutation have the same number of distinct variables (Theorem 2);

(ii) For a given interpolation instance, the interpolants I_n obtained with any non-uniform labeling function and I_u obtained with any uniform labeling function satisfy $\mathit{IntSize}(I_u) \leq \mathit{IntSize}(I_n)$. (Theorem 3); and

(iii) Among uniform labeling functions, the *proof-sensitive* labeling function (see Def. 6) results in the least number of variable occurrences in the partial interpolants associated with the source clauses (Theorem 4).

The proof-sensitive interpolant strength can only be given the trivial guarantees: it is stronger than I_{M_w} and weaker than I_{M_s}. At the expense of the minimality in the sense of Corollary 3.1, we introduce in Eq. (4.4) and Eq. (4.5) the weak and strong versions of the proof-sensitive labeling functions.

The proof-sensitive interpolation algorithms are part of the LIS framework. This might be seen as a limitation of the approach, but the LIS framework has shown to be very general in the sense that it is able to generate a multitude of interpolants of different strength, and also generalizes previous interpolation algorithms for propositional logic.

Our experiments in two different settings, incremental software model checking and interpolation problems from hardware verification, show that the proof-sensitive algorithms consistently lead to smaller interpolants. For the model checkers this meant performance improvement in the total verification time.

Even though LIS and interpolation for propositional logic have been studied with respect to strength and size of interpolants, there are still unsolved questions. For instance, it would be interesting to be able to know what makes *p-annihilable* proofs rare, and how to make them common.

4.4.1 Related Publications

The results described in this chapter have been published in the following papers:

- Hyvärinen, A. E. J., Alt, L. and Sharygina, N. [2015]. Flexible interpolation for efficient model checking, Mathematical and Engineering Methods in Computer Science - 10th International Doctoral Workshop, MEMICS 2015, Telč, Czech Republic, October 23-25, 2015, Revised Selected Papers, pp. 11–22.

- Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2016]. A proof- sensitive approach for small propositional interpolants, in A. Gurfinkel and S. A. Seshia (eds), Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18- 19, 2015. Revised Selected Papers, Springer International Publishing, Cham, pp. 1–18.

- Rollini, S. F., Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2013]. Periplo: A framework for producing effective interpolants in sat- based software verification, in K. McMillan, A. Middeldorp and A. Voronkov (eds), Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 683–693.

Chapter 5

Controlling EUF Interpolants

5.1 Generalizing Interpolation Systems

Satisfiability modulo theories (SMT) is a modeling approach based on expressing a decision problem as a propositional function where some Boolean variables have a special interpretation as equalities and inequalities over a theory τ. SMT has an increasingly important role in model checking, and as a result there is a similar need to construct not only propositional but also theory interpolants fit for a particular application.

Several existing Craig interpolation algorithms (e.g., D'Silva et al. [2010]; McMillan [2005]; Fuchs et al. [2009]; Gao and Zufferey [2016]) rely on traversing a directed, rooted, acyclic graph that represents a proof of unsatisfiability. We call such interpolation algorithms *dag-like*. The nodes of the graph are the logical concepts used in the proof systems, such as clauses for propositional logic, and equalities in the theory of equalities. This chapter presents an approach for controlling the logical strength of theory interpolants for a class of interpolation algorithms based on constructing partial interpolants on a dag-like structure. Our idea is based on the observation that a *dual interpolant* for A can be obtained by the negation of the over-approximation of B. The idea is presented as an *interpolation system template* which is instantiated to an interpolation algorithm by a theory τ and a *labeling function* determining which partial interpolants will be computed as the duals.

We call *τ-interpolation system* the system resulting from instantiating the interpolation system template for the theory τ, and *(τ, L)-interpolation algorithm* the τ-interpolation system with the labeling function L. If the strength relationship between an interpolant and its dual can be established for the τ-interpolation system, the function L controls the strength of the interpolants

produced by the (τ, L)-interpolation algorithm.

5.1.1 Duality-based Interpolation and Strength

Let T be the graph proving the unsatisfiability of $A \wedge B$. The interpolation procedure annotates each node n of T with a *partial interpolant* I_n such that

(a) $A \models I_n$;

(b) $B, I_n \models n$;

(c) The non-logical symbols in I_n are common to A and B.

A dag-like interpolation algorithm consists of two procedures: *(i)* Itp_L^A for computing partial interpolants for leaf nodes; and *(ii)* Itp_C^A for computing a partial interpolant for an inner node as a combination of its children. Fig. 5.1 (*bottom left*) illustrates this schematically.

The interpolation system template allows constructing several strength-controlled interpolants through the notion of *duality*.

Let Itp^A be an interpolant for A computed with a given algorithm. Its *dual* is the negation of the interpolant Itp^B computed for B by the same algorithm. Since Itp^A is an interpolant, it satisfies the following criteria: *(i)* $A \rightarrow Itp^A$; *(ii)* $Itp^A \rightarrow \neg B$; and *(iii)* $var(Itp^A) \subseteq var(A) \cap var(B)$. Similarly, Itp^B satisfies *(iv)* $B \rightarrow Itp^B$; *(v)* $Itp^B \rightarrow \neg A$; and *(vi)* $var(Itp^B) \subseteq var(A) \cap var(B)$. From *(v)* we can see that $A \rightarrow \neg Itp^B$, and from *(iv)* that $\neg Itp^B \rightarrow \neg B$. Since *(iii)* and *(vi)* are equal conditions, also $\neg Itp^B$ is an interpolant for A.

We generalize this idea to partial interpolants by specifying Itp_L^B and Itp_C^B for computing the dual partial interpolants for leaf and non-leaf nodes respectively. The control over strength of the interpolants then follows if the strength relation of a partial interpolant and its dual can be established for both Itp_L^A and Itp_C^A:

Theorem template. Let τ be a theory using a proof system that yields a refutation proof that has a dag-like structure. Let Itp_L^A and Itp_L^B be, respectively, an algorithm to compute partial interpolants for leaf nodes and its dual. Let Itp_C^A and Itp_C^B be, respectively, an algorithm to compute partial interpolants for non-leaf nodes and its dual. If (i) for every leaf node n, $Itp_L^A(n) \rightarrow Itp_L^B(n)$ and (ii) for every non-leaf node m, $Itp_C^A(m) \rightarrow Itp_C^B(m)$, then the strength of the final interpolant can be controlled by a labeling function $L : N \rightarrow \{s, w\}$, such that a dual interpolant is used if the node's label is w, and the conventional

Algorithm 4 Interpolation System Template

```
1: procedure ComputeInterpolant(node)
2:     if node is a leaf then
3:         if L(node) = s then
4:             return Itp_L^A(c)
5:         else
6:             return Itp_L^B(c)
7:         end if
8:     end if
9:     Children's partial interpolants C ← {}
10:    for each child c of node do
11:        C ← C ∪ ComputeInterpolant(c)
12:    end for
13:    if L(node) = s then
14:        return Itp_C^A(C)
15:    else
16:        return Itp_C^B(C)
17:    end if
18: end procedure
```

interpolant is used if the label is s. In particular, given a strength operator $\sqsupseteq$ such that $s \sqsupseteq s$, $s \sqsupseteq w$ and $w \sqsupseteq w$, we also have that a labeling functions L results in a *stronger* interpolant than a labeling function L' (*weaker*) if for every node n, $L(n) \sqsupseteq L'(n)$.

To instantiate the interpolation system template for a theory τ, conditions (i) and (ii) above need to be shown for τ and its proof system. In Sec. 5.2 we establish this relation for EUF, and in Chapter 6 we use the notion of duality of interpolants to create the LRA-interpolation system.

Algorithm 4 describes on a high level the *interpolation system template* for constructing interpolants for a dag-like interpolation algorithms as the partial interpolant of the root node based on the dual interpolants. The template can be instantiated to a (τ, L)-interpolation algorithm by specifying the procedures Itp_L^A, Itp_L^B, Itp_C^A, and Itp_C^B for the theory τ, and by providing the labeling function L for the nodes.

Sec. 5.2 describes the EUF-interpolation system, an instance of the interpolation system template.

5.1.2 Related Work

This section describes parametric interpolation systems that generalize interpolation procedures for first order theories. For work related to specific theories please see Sections 4.3, 5.4, and 6.3.

The work presented in Weissenbacher [2012] presents a parametric interpolation system that extends D'Silva et al. [2010] (LIS), supporting hyper-resolution and providing more flexibility than LIS, in the sense that the strength of the interpolants it generates can be controlled with more freedom than their previous work in D'Silva et al. [2010]. Hyper-resolution is a compact inference system that avoids intermediate clauses, and therefore does not need to generate the entire refutation proof. Interpolation for hyper-resolution is done introducing extra interpolation rules that generalize the rules from LIS. When the presented algorithm is applied to proofs of unsatisfiability that are local, it is able to generalize certain interpolation systems for first order logic.

The technique given in Kovács et al. [2013] goes even further and generalizes Weissenbacher [2012] by providing the *parametric interpolation framework*, which generalizes inference systems and interpolation systems for arbitrary first order theories that are based on recursive procedures that generate interpolants based on refutation proofs. This is achieved by providing their own inference system that produces *RB*-derivations and using their specific set of interpolation rules. The parametric interpolation framework is able to handle both quantified and quantifier-free instances, and is able to compute interpolants of different shape and strength. Unlike the work from Weissenbacher [2012], the framework presented in Kovács et al. [2013] can be applied to non-local proofs.

Even though both techniques from Weissenbacher [2012] and Kovács et al. [2013] do support strength control, the basic difference between them and the interpolation system template presented in this chapter is the conceptual approach and focus. The techniques presented in Weissenbacher [2012] and Kovács et al. [2013] are theoretical frameworks that have as a goal to generalize previous Boolean interpolation systems and first order theories that satisfy locality. The aim of the interpolation system template is to formalize the approach of duality-based interpolation, allowing the applications to control the strength of interpolants for a specific theory if certain requirements are met for the interpolation algorithms of that specific theory.

The technique presented in Totla and Wies [2016] gives a generic framework to create new interpolation procedures for theories without support by reducing them to theories that do have interpolation procedures. The main idea is to formalize an application-specific theory as an extension of a *base theory*, by

adding symbols and universally quantified axioms. The generalization may lead to interpolation systems that are not complete, and the authors present a model-theoretic requirement that allows the identification of theories for which the instantiation is complete.

The interpolation system template and the work from Totla and Wies [2016] also differ in their main focus. The goal of Totla and Wies [2016] is to allow interpolation for theories for which direct interpolation algorithms is too hard, whereas the focus of the interpolation system template is to provide a common way to control the strength of interpolants.

An important skill in constructing mathematical proofs is to identify the aspects of the problem that are relevant. When applied to formal reasoning about the correctness of a software this means ignoring the parts of the system that play no role in its correctness. One such approach that seems to work well in automated software verification is to employ the theory of equality of uninterpreted functions (EUF): in some cases it suffices to assume that a given function returns the same value when invoked with the same arguments. This technique is particularly useful, for example, when modeling memory or arrays Stump et al. [2001], and proving program equivalence Godlin and Strichman [2013].

Generalizing a formula over the states reachable by a program is a natural subtask when summarizing the behavior of a procedure Sery et al. [2012a], or computing a fixed-point of a transition function McMillan [2003]; Bradley [2011]. These techniques are now popular in software model-checking Beyer and Keremoglu [2011]; Gurfinkel et al. [2015]; Cimatti and Griggio [2012], resulting in a growing interest in interpolation over uninterpreted functions.

The EUF theory is also very important for theory combination. The equality operator is usually the common symbol between different theories, and EUF plays a central and essential role when theories are combined. Theory combination is a non-trivial task that attracts a lot of research, since new algorithms for solving and interpolating are necessary.

The existing interpolation systems for EUF are treated as black-boxes, and only generate one interpolant out of a refutation proof or give very little room for flexibility when generating interpolants. This limits the applicability of interpolants, in the sense that it might affect the performance of interpolation-based applications in a way that makes it impractical. Therefore it is necessary to allow the application to choose what kind of interpolants should be generated, such that optimal performance is achieved.

This chapter presents the EUF-interpolation system, a duality based interpolation system that aims at giving more control over the strength and size of

EUF interpolants. We show that the EUF-interpolation system is an instance (EUF, L) of the interpolation system template presented in Chapter 5.1, based on the *EUF* interpolation algorithm presented in McMillan [2005]; Fuchs et al. [2009]. The EUF-interpolation system is able to generate several different interpolants for the same interpolation problem, and allows strength control by means of labeling functions. To simplify the analytical discussion we describe the approach in a notation slightly different from Fuchs et al. [2009] that, with a specific labeling function L (see Example 13) constructs interpolants syntactically equivalent to Fuchs et al. [2009]. Once we have presented the EUF-interpolation system, we proceed to showing the strength result.

Section 5.3 presents experiments involving the EUF-interpolation system and various labeling functions in a controlled setting and in a model checker.

5.2 The EUF-Interpolation System

An EUF interpolation problem is a 5-Tuple $P = (A, B, G^C, \pi, L)$ where A and B are two sets of equalities and disequalities such that $A \cup B$ is unsatisfiable, G^C is a congruence graph with coloring C, π is a path $\overline{xy}$ in G, such that the disequality $(x \neq y)$ exists in $A \cup B$, and L is a labeling function. Our goal is to compute an interpolant for A. The two constant labeling functions $L_s(\pi) := s$ and $L_w(\pi) := w$ will be useful in the analysis. We omit A, B, G^C and L when they are clear from the context, referring to the interpolation problem as $Itp(\pi)$.

The interpolation algorithms in McMillan [2005] and Fuchs et al. [2009] essentially compute an interpolant by collecting the A-factors that prove $(x = y)$ in G^C. To maintain the unsatisfiability with the B part of the problem, the A factors will then be implied by their *B-premise set*. A *premise set* for a color is the set of equalities of the opposite color justifying the existence of a parent edge. As stated in Chapter 5.1, it is also possible to compute a dual interpolant for A as the negation of an interpolant for B. More technically, the B-premise set $\mathcal{B}$ for a path π is

$$\mathcal{B}(\pi) := \begin{cases} \bigcup\{\mathcal{B}(\sigma) | \sigma \text{ is a factor of } \pi\}, \text{if } \pi \text{ has} \geq 2 \text{ factors;} \\ \{\pi\}, \text{if } \pi \text{ is a } B\text{-path; and} \\ \bigcup\{\mathcal{B}(\sigma) | \sigma \text{ is a parent of an edge of } \pi\}, \\ \quad \text{if } \pi \text{ is an } A\text{-path.} \end{cases} \tag{5.1}$$

To compute the dual interpolant we will need similarly to collect the B-factors that prove $(x = y)$ in G^C, implied by their A-premise set. The A-premise set

$\mathcal{A}$ for a path π is defined as

$$\mathcal{A}(\pi) := \begin{cases} \bigcup\{\mathcal{A}(\sigma) | \sigma \text{ is a factor of } \pi\}, \text{if } \pi \text{ has} \geq 2 \text{ factors;} \\ \{\pi\}, \text{if } \pi \text{ is an } A\text{-path; and} \\ \bigcup\{\mathcal{A}(\sigma) | \sigma \text{ is a parent of an edge of } \pi\}, \\ \quad \text{if } \pi \text{ is a } B\text{-path.} \end{cases} \tag{5.2}$$

We extend the notation of $\mathcal{A}$ and $\mathcal{B}$ over a set S of paths with a 'syntactic sugar' $\mathcal{A}(S) := \bigcup_{\sigma \in S} \mathcal{A}(\sigma)$ and $\mathcal{B}(S) := \bigcup_{\sigma \in S} \mathcal{B}(\sigma)$. For any two operators $\mathcal{O}, \mathcal{O}'$ whose range contains this domain we define the composite operator $\mathcal{O}\mathcal{O}'(\sigma) := \mathcal{O}(\mathcal{O}'(\sigma))$; and define recursively $\mathcal{O}^0(\sigma) := \sigma$, and $\mathcal{O}^n := \mathcal{O}(\mathcal{O}^{n-1})$.

The functions J_A and J_B give, respectively, the contribution of an individual A-factor and an individual B-factor to the interpolants.

$$J_A(\pi) := [\![\mathcal{B}(\pi)]\!] \rightarrow [\![\pi]\!] \tag{5.3}$$

$$J_B(\pi) := [\![\mathcal{A}(\pi)]\!] \rightarrow [\![\pi]\!] \tag{5.4}$$

Let S be a set of paths. The notation $S|_c$ represents the subset of S containing the paths σ such that $L(\sigma) = c$. The number of paths in an interpolation problem is exponential in the size of the problem, resulting in potential challenges in defining the labeling function. However this is not an issue since the interpolation algorithm presented in this section almost always handles only factors and therefore it suffices to label the linear number of factors. The only path that is not necessarily a factor that needs to be labeled is the path in the congruence graph that contradicts an original disequality. The (EUF, L)-algorithm for computing the EUF interpolant over A for a path $\overline{xy}$, $Itp(P)$, where $P = (A, B, G^C, \pi, L)$, is defined using four sub-procedures I_A, I'_A, I_B, and I'_B that are invoked depending on which partition the conflict $x \neq y$ lies and what color the path has as:

$$Itp(P) := \begin{cases} I_A(\overline{xy}) & \text{if } (x \neq y) \in B \text{ and } L(\overline{xy}) = s, \\ I'_A(\overline{xy}) & \text{if } (x \neq y) \in A \text{ and } L(\overline{xy}) = s, \\ \neg I_B(\overline{xy}) & \text{if } (x \neq y) \in A \text{ and } L(\overline{xy}) = w, \text{ and} \\ \neg I'_B(\overline{xy}) & \text{if } (x \neq y) \in B \text{ and } L(\overline{xy}) = w. \end{cases} \tag{5.5}$$

The sub-procedures for I_A and I_B are defined as

$$I_A(\pi) := (\bigwedge_{\sigma \in \mathcal{A}(\pi)} J_A(\sigma)) \wedge (\bigwedge_{\sigma \in \mathcal{B}\mathcal{A}(\pi)|_s} I_A(\sigma)) \wedge (\bigwedge_{\sigma \in \mathcal{B}\mathcal{A}(\pi)|_w} \neg I'_B(\sigma)) \tag{5.6}$$

$$I_B(\pi) := (\bigwedge_{\sigma \in \mathcal{B}(\pi)} J_B(\sigma)) \wedge (\bigwedge_{\sigma \in \mathcal{AB}(\pi)|_s} I_B(\sigma)) \wedge (\bigwedge_{\sigma \in \mathcal{AB}(\pi)|_w} \neg I'_A(\sigma)). \tag{5.7}$$

For the cases where the conflict $x \neq y \in A$ and $L(\overline{xy}) = s$ and $x \neq y \in B$ and $L(\overline{xy}) = w$ the path $\overline{xy} = \pi$ needs to be decomposed for computing the partial interpolant as $\pi_1\theta_B\pi_2$ or $\pi_1\theta_A\pi_2$, where θ_C is the longest subpath of π with c-colorable endpoints. Hence, I'_A and I'_B are

$$I'_A(\pi) := I_A(\theta_B) \wedge (\bigwedge_{\sigma \in \mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)} I_A(\sigma)) \wedge ([\![\mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)]\!] \rightarrow \neg[\![\theta_B]\!]). \tag{5.8}$$

$$I'_B(\pi) := I_B(\theta_A) \wedge (\bigwedge_{\sigma \in \mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)} I_B(\sigma)) \wedge ([\![\mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)]\!] \rightarrow \neg[\![\theta_A]\!]). \tag{5.9}$$

Theorem 5. *Given two sets of equalities and disequalities A and B such that $A \cup B$ is unsatisfiable, a colored congruence graph G^C containing a path $\pi := \overline{xy}$ such that $(x \neq y) \in A \cup B$, and a labeling function L, Eq. (5.5) computes a valid interpolant for A using L over G^C.*

Proof. Depending on whether A or B contains π and on the labeling function, Eq. (5.5) may use Eq. (5.8), Eq. (5.9), Eq. (5.6) or Eq. (5.7). Therefore we have to analyze the four equations as separate cases. We show that each case leads to a sound interpolant, and therefore the theorem holds.

(i) $I'_A(\pi)$

Eq. (5.8) behaves as the algorithm presented in Fuchs et al. [2009], and computes interpolants for A when the disequality $(x \neq y)$ is in A.

(ii) $\neg I'_B(\pi)$

Eq. (5.9) is the dual of Eq. (5.8), and clearly computes interpolants for B when the disequality is in B. We can transform it into an interpolant for A by negating it, as it is done in Eq. (5.5).

(iii) $I_A(\pi)$

Eq. (5.6) behaves similarly to the interpolation procedure presented in Fuchs et al. [2009], with the addition of dual interpolants and labeling functions. First J_I computes the individual contribution of the A-factors that prove π in G^C ($\mathcal{A}(\pi)$) to the interpolant, and then conjoins it with the interpolants of their B-premise sets ($\mathcal{BA}(\pi)$).

(iv) $\neg I_B(\pi)$

Eq. (5.7) is the dual of Eq. (5.6) and computes an interpolant for B, which can be transformed into an interpolant for A by negating it.

□

The EUF-interpolation system of Eq. (5.5) is an instance of the interpolation system template. The congruence graph represents the dag-like structure using factors as dag nodes and a labeling function as presented in Chapter 5.1. Using Eq. (5.6) and Eq. (5.7), we have that $Itp_L^A := J_A$ and $Itp_L^B := J_B$, since leaf nodes represent factors in a congruence graph that have empty premise sets. Finally we have that $Itp_C^A := I_A$ (or I'_A if A contains the disequality that the congruence graph contradicts) and $Itp_C^B := I_B$ (or I'_B if B contains the disequality that the congruence graph contradicts).

Notice that in Eq. 5.5, Eq. 5.6 and Eq. 5.8, if L_s is used as the labeling function, only I_A and I'_A are used. This makes the algorithm collect only A-factors and their premises for the interpolant, which is the same approach as in Fuchs et al. [2009]. This shows that the interpolation algorithm from Fuchs et al. [2009] is an instance of the EUF-interpolation system represented by the labeling function L_s. The following example shows how Eq. (5.5) can be used to compute the interpolants from Fuchs et al. [2009] using L_s.

Example 13. *Let* $A := \{(x_1 = f(x_2)), (f(x_3) = x_4), (x_4 = f(x_5)), (f(x_6) = x_7)\}$ *and* $B := \{(x_2 = x_3), (x_5 = x_6), (x_1 \neq x_7)\}$. *Fig. 5.1 shows a possible congruence graph* G^C *that proves the joint unsatisfiability of* A *and* B *(by proving* $(x_1 = x_7)$ *such that* $(x_1 \neq x_7) \in A \cup B$*) and its tree representation, with each node annotated by its partial interpolant. Notice that the labeling function* L_s *used in* G^C *labels all the factors as* s*. From Eq. (5.5) we have that* $Itp(\overline{x_1x_7}) = I_A(\overline{x_1x_7})$*, because* $L_s(\overline{x_1x_7}) = s$ *and* $(x_1 \neq x_7) \in B$*. The call to* $I_A(\overline{x_1x_7})$ *is represented by the root node in the tree in Fig. 5.1. First we compute* $\mathcal{A}(\overline{x_1x_7}) = \{\overline{x_1x_7}\}$ *and* $\mathcal{BA}(\overline{x_1x_7}) = \{\overline{x_2x_3}, \overline{x_5x_6}\}$*. Then from Eq. (5.6) we have that* $I_A(\overline{x_1x_7}) = J_A(\overline{x_1x_7}) \wedge I_A(\overline{x_2x_3}) \wedge I_A(\overline{x_5x_6})$*. The calls to* $I_A(\overline{x_2x_3})$ *and* $I_A(\overline{x_5x_6})$ *are represented by the edges from the root to the leaf nodes in the tree in Fig. 5.1. We then proceed computing* $\mathcal{A}(\overline{x_2x_3}) = \emptyset$ *and* $\mathcal{BA}(\overline{x_2x_3}) = \emptyset$ *which lead to* $I_A(\overline{x_2x_3}) = \top$*; and* $\mathcal{A}(\overline{x_5x_6}) = \emptyset$ *and* $\mathcal{BA}(\overline{x_5x_6}) = \emptyset$ *which lead to* $I_A(\overline{x_5x_6}) = \top$ *(the partial interpolants of the leaf nodes). Finally we have that* $I_A(\overline{x_1x_7}) = ((x_2 = x_3) \wedge (x_5 = x_6)) \rightarrow (x_1 = x_7)$ *is the partial interpolant of the root node, representing the final interpolant for* A*.*

5.2.1 The Strength

Let $P = (A, B, G^C, \pi, L_s)$ and $P' = (A, B, G^C, \pi, L_w)$ be two interpolation problems differing only in the labeling function. We will show in Theorem 6 that $Itp(P) \rightarrow Itp(P')$, and then in Example 14 that there are cases where the strength relation is strict in the sense that there are models that satisfy

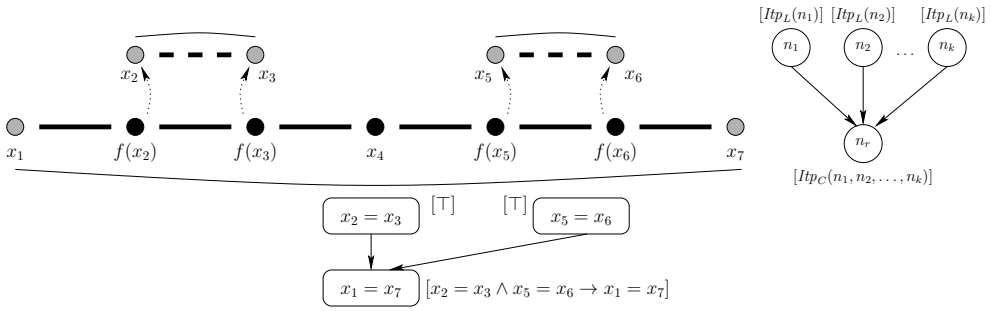


Figure 5.1. Computing partial interpolants for the EUF-interpolation system. The bottom left dag illustrates the computation in terms of the dag-like interpolation algorithm.

$Itp(P')$ but do not satisfy $Itp(P)$. Theorem 6 needs Lemma 4 which in turn is a generalization of Lemma 2. We then show our main result on EUF in Theorem 7 that the new interpolation procedure for EUF presented here meets all the requirements described in Chapter 5.1, that is, we provide a way to compare the strength of interpolants based on the labeling functions used.

Definition 7. *Let G be a congruence graph, and σ an arbitrary factor from G. We say that σ is* relevant *to ω if either $J_A(\sigma)$ or $J_B(\sigma)$ is called during the computation of $I_A(\omega)$ or $I_B(\omega)$.*

Lemma 2. *Let G^C be a congruence graph with coloring C, and ω a factor from G. Then $I_A(\omega) \wedge I_B(\omega) \rightarrow [\![\omega]\!]$.*

Proof. Let R^ω be the set of factors relevant (Def. 7) to ω in a congruence graph, R^ω_A the subset of R^ω containing only A-factors and R^ω_B the subset of R^ω containing only B-factors.

From Eq. (5.6) we can clearly see that

$$R^\omega_A := \mathcal{A}(\omega) \cup \mathcal{A}(\mathcal{BA}(\omega)) \cup \ldots \cup \mathcal{A}((\mathcal{BA})^k(\omega)) \cup \ldots, \tag{5.10}$$

and from Eq. (5.7) that

$$R^\omega_B := \mathcal{B}(\omega) \cup \mathcal{B}(\mathcal{AB}(\omega)) \cup \ldots \cup \mathcal{B}((\mathcal{AB})^k(\omega)) \cup \ldots. \tag{5.11}$$

Because congruence graphs are acyclic and finite, for any ω there exists an integer n such that $\mathcal{A}((\mathcal{BA})^n(\omega)) = \emptyset$ and $\mathcal{B}((\mathcal{AB})^n(\omega)) = \emptyset$, which allows us to rewrite the previous equations as

$$R^\omega_A := \mathcal{A}(\omega) \cup \mathcal{A}(\mathcal{BA}(\omega)) \cup \ldots \cup \mathcal{A}((\mathcal{BA})^n(\omega)), \tag{5.12}$$

$$R_B^\omega := \mathcal{B}(\omega) \cup \mathcal{B}(\mathcal{AB}(\omega)) \cup \ldots \cup \mathcal{B}((\mathcal{AB})^n(\omega)). \tag{5.13}$$

If ω is an A-factor, we have that $\mathcal{A}(\omega) = \{\omega\}$ and therefore we can write $\mathcal{B}(\omega) \equiv \mathcal{BA}(\omega)$. Using that we can infer that $\mathcal{A}((\mathcal{BA})^{n+1}(\omega)) = \emptyset$ (by the definition of n), and we can change Eq. (5.13) to

$$R_B^\omega := \mathcal{BA}(\omega) \cup (\mathcal{BA})^2(\omega)) \cup \ldots \cup (\mathcal{BA})^{n+1}(\omega). \tag{5.14}$$

We follow the proof assuming that ω is an A-factor, using Eq. (5.12) and Eq. (5.14) to represent R_A^ω and R_B^ω respectively, and then argue that the proof is symmetrical for the case where ω is a B-factor.

From Eq. (5.12) and Eq. (5.14), we can then see that

$$R^\omega := \mathcal{A}(\omega) \cup \mathcal{BA}(\omega) \cup \mathcal{A}(\mathcal{BA}(\omega)) \cup \ldots \cup \mathcal{A}((\mathcal{BA})^n(\omega)) \cup (\mathcal{BA})^{n+1}(\omega) \tag{5.15}$$

and therefore

$$I_A(\omega) \wedge I_B(\omega) = (\bigwedge_{\sigma \in R_A^\omega} J_A(\sigma)) \wedge (\bigwedge_{\sigma \in R_B^\omega} J_B(\sigma)). \tag{5.16}$$

When J_A and J_B are computed over a set that has an empty premise set, the result is not a conjunction of implications, but a conjunction of equalities. In this case, $J_B((\mathcal{BA})^{n+1}(\omega)) = [\![(\mathcal{BA})^{n+1}(\omega)]\!]$ because $\mathcal{A}((\mathcal{BA})^{n+1}(\omega)) = \emptyset$. Thus, after applying the functions J_A and J_B we have that

$$\begin{aligned} I_A(\omega) \wedge I_B(\omega) = \; & \left(\textstyle\bigwedge_{i=0}^{n} \bigwedge_{\sigma \in \mathcal{A}((\mathcal{BA})^i(\omega))} ([\![\mathcal{B}(\sigma)]\!] \rightarrow [\![\sigma]\!])\right) \\ & \wedge \left(\textstyle\bigwedge_{i=0}^{n} \bigwedge_{\sigma \in (\mathcal{BA})^i(\omega)} ([\![\mathcal{A}(\sigma)]\!] \rightarrow [\![\sigma]\!])\right) \\ & \wedge [\![(\mathcal{BA})^{n+1}(\omega)]\!]. \end{aligned} \tag{5.17}$$

We know that formulas of the form $((a_1 \rightarrow b_1) \wedge \ldots \wedge (a_n \rightarrow b_n)) \rightarrow ((\bigwedge_{i \in 1..n} a_i) \rightarrow (\bigwedge_{i \in 1..n} b_i))$ are tautologies. Using that and Eq. (5.17), we can then show that

$$\begin{aligned} I_A(\omega) \wedge I_B(\omega) \; & \rightarrow (\textstyle\bigwedge_{i=0}^{n} ([\![(\mathcal{BA})^{i+1}(\omega)]\!] \rightarrow [\![\mathcal{A}((\mathcal{BA})^i(\omega))]\!])) \\ & \wedge (\textstyle\bigwedge_{i=0}^{n} ([\![\mathcal{A}((\mathcal{BA})^i(\omega))]\!] \rightarrow [\![(\mathcal{BA})^i(\omega)]\!])) \\ & \wedge [\![(\mathcal{BA})^{n+1}(\omega)]\!] \end{aligned} \tag{5.18}$$

Because $[\![(\mathcal{BA})^{n+1}(\omega)]\!]$ has no implicant, it has to be true in the formula, starting a chain that satisfies the antecedents of all the implications in Eq. (5.18), since $(\mathcal{BA})^{n+1}(\omega)$ is the premise set of $\mathcal{A}((\mathcal{BA})^n(\omega))$, which is the premise set of $(\mathcal{BA})^n(\omega)$ and so on. Because of that, we can simplify the formula to

$$I_A(\omega) \wedge I_B(\omega) \rightarrow \left(\bigwedge_{i=0}^{n} [\![\mathcal{A}((\mathcal{BA})^i(\omega))]\!]\right) \wedge \left(\bigwedge_{i=0}^{n+1} [\![(\mathcal{BA})^i(\omega)]\!]\right). \tag{5.19}$$

Therefore, we have that

$$\forall r \in R^{\omega}.(I_A(\omega) \wedge I_B(\omega)) \rightarrow [\![r]\!]. \tag{5.20}$$

For the case where ω is a B-factor, we have that $\mathcal{B}(\omega) = \{\omega\}$, which implies that $\mathcal{AB}(\omega) \equiv \mathcal{A}(\omega)$. Using that, we can infer that $\mathcal{B}((\mathcal{AB})^{n+1}(\omega)) = \emptyset$ and we can also change Eq. (5.12) to

$$R_A^{\omega} := \mathcal{AB}(\omega) \cup (\mathcal{AB})^2(\omega) \cup \ldots \cup (\mathcal{AB})^n(\omega). \tag{5.21}$$

Using Eq. (5.21) and Eq. (5.13) to represent R_A^{ω} and R_B^{ω} respectively, the same reasoning is followed and we can show that Eq. (5.20) holds also if ω is a B-factor.

If ω is an A-factor, then $\mathcal{A}(\omega) = \{\omega\}$ and J_A is called for ω in the first iteration of Eq. (5.6). On the other hand, if ω is a B-factor, then $\mathcal{B}(\omega) = \{\omega\}$ and J_B is called for ω in the first iteration of Eq. (5.7). This shows that ω is a relevant factor and by Eq. (5.20) we conclude the proof. □

Lemma 3. *Let π be an arbitrary path in the congruence graph, and $\phi(\pi)$ the set of all factors in π. Then $I_A(\pi) = \bigwedge_{\sigma \in \phi(\pi)} I_A(\sigma)$ and $I_B(\pi) = \bigwedge_{\sigma \in \phi(\pi)} I_B(\sigma)$.*

Proof. By the definition of $\mathcal{A}$ in Eq. (5.2), we know that $\mathcal{A}(\pi) = \bigcup_{\sigma \in \phi(\pi)} \mathcal{A}(\sigma)$. By the definition of I_A, we can see that J_A is computed individually for each element of $\mathcal{A}(\pi)$ in $I_A(\pi)$, and I_A is called recursively for the B-premise sets of each individual element of $\mathcal{A}(\pi)$. Therefore, $\bigwedge_{\sigma \in \phi(\pi)} I_A(\sigma) = \bigwedge_{\sigma \in \mathcal{A}(\pi)} I_A(\sigma)$, which has the same effect of $I_A(\pi)$. The result is analogous for $I_B(\pi)$. □

Lemma 4. *Lemma 2 holds when ω is a path containing multiple factors.*

Proof. Let ω be a path built by multiple factors and $\phi(\omega)$ the set containing these factors. By Lemma 3 we know that $I_A(\omega) = \bigwedge_{\sigma \in \phi(\omega)} I_A(\sigma)$ and $I_B(\omega) = \bigwedge_{\sigma \in \phi(\omega)} I_B(\sigma)$; by Lemma 2 we know that $\forall \sigma \in \phi(\omega).(I_A(\sigma) \wedge I_B(\sigma)) \rightarrow [\![\sigma]\!]$. Because the elements of $\phi(\omega)$ are factors linking nodes to prove ω, we know that $(\bigwedge_{\sigma \in \phi(\omega)} [\![\sigma]\!]) \rightarrow [\![\omega]\!]$. Therefore we have that $(I_A(\omega) \wedge I_B(\omega)) \rightarrow [\![\omega]\!]$. □

Theorem 6. *For fixed A, B, G^C, and $\overline{xy}$, for the corresponding interpolants defined in Eq. (5.5) it holds that $Itp(A, B, G^C, \overline{xy}, L_s) \rightarrow Itp(A, B, G^C, \overline{xy}, L_w)$.*

Proof. We only consider the case where $(x \neq y) \in B$ and note that the case where $(x \neq y) \in A$ is completely symmetrical. Let $\pi := \overline{xy}$ and $\psi = I_A(\pi) \wedge I'_B(\pi)$. By Eq. (5.9) we have that

$$\psi = I_B(\theta_A) \wedge \bigwedge_{\sigma \in \mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)} I_B(\sigma) \wedge ([\![\mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)]\!] \rightarrow \neg [\![\theta_A]\!]) \wedge I_A(\pi), \tag{5.22}$$

where π is decomposed as $\pi_1\theta_A\pi_2$, and θ_A is the largest subpath of π with A-colorable endpoints. In order to show that $I_A(\pi) \rightarrow \neg I'_B(\pi)$, we prove that $\psi \rightarrow \bot$, which leads to the theorem. In $I'_B(\pi)$, π is split into $\pi_1\theta_A\pi_2$. From the definition of θ_A, we know that π_1 and π_2 are B-factors. Therefore, using Lemma 3, we can say that $I_A(\pi) = I_A(\pi_1) \wedge I_A(\theta_A) \wedge I_A(\pi_2)$. Because π_1 and π_2 are subpaths of π, we know that $\mathcal{A}(\pi_1), \mathcal{A}(\pi_2) \subseteq \mathcal{A}(\pi)$. Using Lemma 3, we have that $(I_A(\pi_1) \wedge I_A(\pi_2)) = \bigwedge_{\sigma\in\mathcal{A}(\pi_1)} I_A(\sigma) \wedge \bigwedge_{\sigma\in\mathcal{A}(\pi_2)} I_A(\sigma)$.

We can now see that both $\bigwedge_{\sigma\in\mathcal{A}(\pi_1)\cup\mathcal{A}(\pi_2)} I_B(\sigma)$ and $\bigwedge_{\sigma\in\mathcal{A}(\pi_1)\cup\mathcal{A}(\pi_2)} I_A(\sigma)$ are contained in ψ. From Lemma 2 we know that *(i)* $\psi \rightarrow [\![\mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)]\!]$. $I_A(\theta_A)$ and $I_B(\theta_A)$ are also contained in ψ, therefore from Lemma 2 we have that *(ii)* $\psi \rightarrow [\![\theta_A]\!]$. From *(i)* and *(ii)* we see that $\psi \rightarrow ([\![\theta_A]\!] \wedge \neg[\![\theta_A]\!])$. □

Theorem 6 proves one of the requirements of Theorem 5.1.1, which is that $Itp^A_L \rightarrow Itp^B_L$, that is, the interpolation algorithm that generates conventional interpolants implies the interpolation algorithm that generates dual interpolants. As described earlier in this section, these two algorithms are instances of the EUF-interpolation system when using, respectively, labeling functions L_s and L_w.

To show that the implication is not trivial in general, we show by example that three different labeling functions being applied to the congruence graph from Example 4 result in three interpolants which do not share models pairwise.

Example 14. *Consider again the sets A and B and the congruence graph G^C from Example 4 and Fig. 3.3. Let L_c a custom labeling function mapping the paths to labels as $\{\overline{x_1x_2} : s, \overline{x_1v_1} : s, \overline{v_1v_2} : s, \overline{v_2x_2} : s, \overline{y_1t_1} : w, \overline{t_1t_2} : w, \overline{t_2y_2} : w, \overline{z_1s_1} : w, \overline{s_1s_2} : w, \overline{s_2z_2} : w, \overline{r_1u_1} : w, \overline{u_1u_2} : w, \overline{u_2r_2} : w\}$. We recall that the labeling function only needs to be defined on the factors and the path that contradicts the original disequality in $A \cup B$, in this case $\overline{x_1x_2}$. The labels are shown over curves representing which path is being labeled. The custom labeling function L_c represents the intent of generating stronger partial interpolants closer to $(x_1 = x_2)$, and weaker partial interpolants in the inner explanations. Let Itp_s and Itp_w be, respectively, the interpolants generated by Eq. (5.5) using L_s and L_w. The computed interpolants are $Itp_s = ((t_1 = t_2) \rightarrow (v_1 = v_2)) \wedge ((u_1 = u_2) \rightarrow (s_1 = s_2))$ and $Itp_w = \neg((u_1 = u_2) \wedge ((s_1 = s_2) \rightarrow (t_1 = t_2)) \wedge \neg(v_1 = v_2))$.*

To show how the EUF-interpolation system uses the duality of interpolants, Fig. 5.2 shows the tree of factors that represent the congruence graph used in this example, labeled by L_c. Notice that at every node, the label controls if an interpolant for A or B (dual) should be derived for that node. Let Itp_c be the interpolant generated using L_c. Because the disequality $(x_1 \neq x_2)$ is in B and

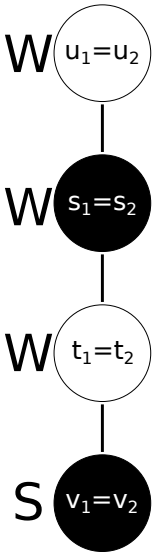


Figure 5.2. Tree of factors that represents the congruence graph from Example 14.

the root node, that represents the equality $(v_1 = v_2)$*, has label* s*, we have that* $Itp_c = I_A(\overline{v_1 v_2})$*. The function* I_A*, at this point, computes* $J_I(\overline{v_1 v_2})$ *and moves to the dual interpolation algorithm for the node that represents* $(t_1 = t_2)$*, since its label is* w*, by computing* $\neg I'_B(\overline{t_1 t_2})$*. The labels of the following nodes are also* w*, so the dual interpolation algorithm is used again. Therefore, we have that* $Itp_c = ((t_1 = t_2) \rightarrow (v_1 = v_2)) \wedge \neg(((s_1 = s_2) \rightarrow (t_1 = t_2)) \wedge (u_1 = u_2) \wedge \neg(t_1 = t_2))$*.*

We also have that $Itp_s \rightarrow Itp_c \rightarrow Itp_w$*, and none of them is equivalent to another.*

Finally our main result is presented providing a way to partially order interpolants into a lattice based on their strength only by looking at the labeling function used. As a result it follows that the constant labeling functions L_s and L_w give, respectively, the strongest and the weakest interpolants within this framework.

Theorem 7. *Let* $\sqsupseteq$ *be a label strength operator such that* $s \sqsupseteq s$*,* $w \sqsupseteq w$ *and* $s \sqsupseteq w$*. Let* $P = (A, B, G^C, \overline{xy}, L)$ *and* $P' = (A, B, G^C, \overline{xy}, L')$ *be two interpolation problems where* L *and* L' *are two labeling functions such that* $L(\sigma) \sqsupseteq L'(\sigma)$ *for all the factors* σ *of* G^C*. Then* $Itp(P) \rightarrow Itp(P')$*.*

Proof. If $(x \neq y) \in B$, we can either use I_A or $\neg I_B$ to create an interpolant for A. On the other hand, if $(x \neq y) \in A$, we can use either I'_A or $\neg I_B$. Since only

Eq. (5.6) and Eq. (5.7) use labeling functions, we analyze only those equations in this proof.

From Eq (5.6) we can see that when a factor σ has label a a weakening step is applied, using $\neg I'_B(\sigma)$ instead of $I_A(\sigma)$. Let Itp be the interpolant generated without weakening, and Itp' the interpolant generated having applied the weakening step. We know that $I_A(\sigma) \subseteq Itp$ and $\neg I'_B(\sigma) \subseteq Itp'$. From Theorem 6 we know that $I_A(\sigma) \rightarrow \neg I'_B(\sigma)$, therefore we have that $Itp \rightarrow Itp'$.

Following the same reasoning, from Eq. (5.7) we can see that when a factor σ has label b a strengthening step is applied, using $\neg I'_A(\sigma)$ instead of $I_B(\sigma)$. Let Itp be the interpolant generated without strengthening, and Itp' the interpolant generated having applied this strengthening step. We know that $\neg I_B(\sigma) \subseteq Itp$ and $\neg\neg I'_A(\sigma) \subseteq Itp'$. From Theorem 6 we have that $I'_A(\sigma) \rightarrow \neg I_B(\sigma)$. Therefore we have that $Itp' \rightarrow Itp$. □

Theorem 7 shows that by using labeling functions s and w and Theorem 6, the second and final requirement of Theorem 5.1.1 is met, and therefore the strength of EUF interpolants can be controlled by the labeling functions.

5.2.2 Labeling Functions

The EUF-interpolation system presented above introduces a way of computing interpolants of different strength by labeling the factors of a congruence graph as s or w, depending on the required strength. Each labeling function leads to a potential new interpolant, and creating meaningful labeling functions is a very hard task on its own. In this paper we restrict ourselves to the study of the two extreme labeling functions with respect to strength, L_s and L_w, and prove the following important result related to the size of interpolants.

Theorem 8. *Let $P = (A, B, G^C, \pi, L)$. If $\pi \in B$, $L = L_s$ leads to the interpolant $Itp(P)$ that contains the smallest number of equalities, and if $\pi \in A$, $L = L_w$ leads to the interpolant $Itp(P)$ that contains the smallest number of equalities.*

Proof. Consider the labeling function L_s and the computation of $I_A(\pi)$. The usage of this labeling function makes the formula be entirely computed by I_A, never using I_B. Now let $I_A(\delta)$ be some arbitrary subcomputation of $I_A(\pi)$. First, $\bigwedge_{\sigma \in \mathcal{A}(\delta)} J_A(\sigma)$ is computed. From Eq. 5.3 we know that this formula contains the equality $[\![\sigma_A]\!]$ for every $\sigma_A \in \mathcal{A}(\delta)$ and the equality $[\![\sigma_B]\!]$ for every $\sigma_B \in \mathcal{BA}(\delta)$. Suppose then that a factor γ from $\mathcal{BA}(\delta)$ has label w, which results in the computation of $\neg I'_B(\gamma)$. We know that γ is a B-factor because

it came from $\mathcal{BA}(\delta)$, therefore *(i)* $\theta_A = \gamma$; and *(ii)* $\mathcal{B}(\gamma) = \{\gamma\}$. From *(i)* we have that $I'_B(\gamma)$ computes $I_B(\gamma)$, and from *(ii)* we have that $\mathcal{B}(\gamma) = \{\gamma\}$ and $J_B(\gamma)$ is then computed. This reintroduces $[\![\gamma]\!]$ in the interpolant (as the implicated part of $J_B(\gamma)$), since $I_A(\delta)$ already introduced it in the implicant part of some implication in $\bigwedge_{\sigma\in\mathcal{A}(\delta)} J_A(\sigma)$. Notice that if γ had label s this equality would not be reintroduced. The reasoning is symmetrical for the case where L_w is used for the computation of I_B. Therefore we have that $I_A(\pi)$ and $I_B(\pi)$ contain exactly the same equalities, with the difference being the side of the implication (implicant or implicated) that an equality appears in (because of Eq. 5.3 and Eq. 5.4). We can also see that any other labeling function L will introduce at least one equality in the interpolant more than once (when it changes from I_A to $\neg I'_B$ or from I_B to $\neg I'_A$).

If $\pi \in B$, an interpolant can be computed by either $I_A(\pi)$ or $\neg I'_B(\pi)$. The interpolant $I_A(\pi)$ has less equalities because it does not introduce the conflict in the interpolant, as $\neg I'_B(\pi)$ does with the term $([\![\mathcal{A}(\pi_1) \cup \mathcal{A}(\pi_2)]\!] \rightarrow \neg[\![\theta_A]\!])$. Therefore, L_s leads to the interpolant with the least number of equalities.

If $\pi \in A$, an interpolant can be computed by either $\neg I_B(\pi)$ or $I'_A(\pi)$. Symmetrically, the interpolant $\neg I_B(\pi)$ has less equalities because it does not introduce the conflict in the interpolant, as $I'_A(\pi)$ does with the term $([\![\mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)]\!] \rightarrow \neg[\![\theta_B]\!])$. Therefore, L_w leads to the interpolant with the least number of equalities. □

5.3 Experimental Evaluation

We implemented and integrated the system into the existing propositional interpolation implementation in the OpenSMT2 solver. We report experiments in two different settings in the implementation: (i) interpolating over unsatisfiable QF_UF benchmarks from smt-lib; and (ii) running the approach integrated in HiFrog, an interpolation-based incremental model checker for C. The benchmarks and the software are available at `verify.inf.usi.ch/euf-interpolation`. Before describing the experiments we give a concise explanation on how EUF and propositional interpolation are integrated.

5.3.1 Interpolation over smt-libbenchmarks

The motivation for this experiment is to stress-test the implementation with respect to generating interpolants of different strength and size. The chosen benchmarks lead to complex congruence graphs, and therefore the interpolation

problems are non-trivial. Following Fuchs et al. [2009]; Cimatti et al. [2008], we randomly split the assertions in each benchmark to partitions A and B. The set consists of 106 QF_UF instances resulting in total over two and a half million EUF interpolants.

Logical strength. The theory interpolation algorithms use three labeling functions L_s, L_w (see Chapter. 5), and L_r, a labeling function that labels all components randomly as either s or w. The algorithms are called, respectively, Itp_s, Itp_w, and Itp_r. We use the proof-sensitive interpolation algorithm Alt et al. [2016] in the propositional structure. This results in three final interpolants I_s, I_w and I_r for each benchmark.

We computed the strength relationship for each theory partial interpolant as well as the final SMT interpolants. Even though the EUF interpolants are often simple, in 71% of them it was possible to generate at least two interpolants of different strength, and 5.73% resulted in all three having different strength.

After solving and interpolating, we ran extra experiments to check the strength relations of the final interpolants I_s, I_w and I_r. Since the final interpolants are much more complex, of the 106 benchmarks, 55 ran out of memory while computing the strength relations. For the remaining 51, all the three final interpolants were pairwise inequivalent, confirming that the framework is able to generate interpolants of different strength.

Interpolant size. Since the propositional and EUF interpolation algorithms are to a large degree independent, it is natural to ask what combination of the algorithms is most efficient. This experiment studies the question using the interpolant size as a measure of efficiency. We use several propositional interpolation algorithms from the literature to study the combinations, all instances of the labeled interpolation system D'Silva et al. [2010]; Alt et al. [2016] supported by OpenSMT2. Fig. 5.3 shows the algorithms ordered with respect to the logical strength of the interpolants they compute, which resulted from the theoretical analysis presented in Chapter 4. The algorithms M_s, P, and M_w use a fixed labeling for the shared variables whereas the algorithms $\mathrm{PS}, \mathrm{PS}_s$, and PS_w use the proof structure to optimize the labeling. The six propositional and three EUF interpolation algorithms result in 18 combinations. We measure the sizes of the final interpolants both in *(i)* the number of Boolean connectives (Fig. 5.4); and *(ii)* the number of EUF equalities (Fig. 5.5). Excluding the instances where we encountered memory outs we report the results on 82 of the original 106 benchmarks. For each benchmark, we computed the

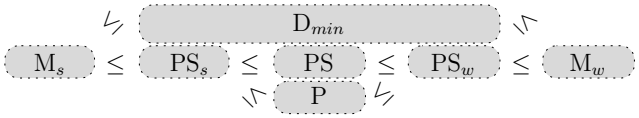


Figure 5.3. The relative strength of the propositional interpolation algorithms Alt et al. [2016]

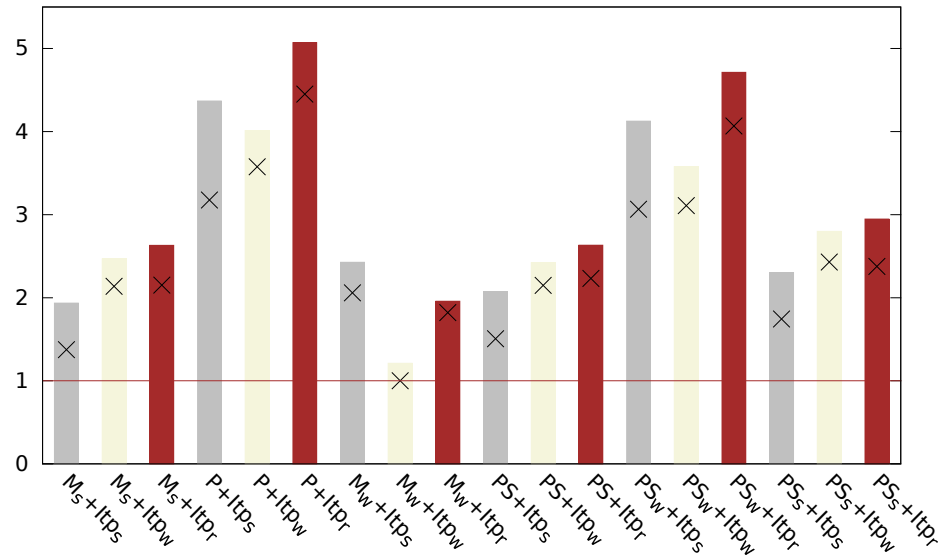


Figure 5.4. Comparison between interpolation combinations with respect to the number of Boolean connectives in the final interpolant

smallest number of Boolean connectives or equalities in the interpolant among all the configurations (*best*) and the ratio *combination/best* for each possible combination, which shows us how much worse each combination did compared to the best combination for that benchmark. Notice that the ratio of the best combination for a benchmark is one and therefore no ratio can be less than one. The bars present the average and the crosses the median of those ratios among all the benchmarks for each combination.

In Fig. 5.4 the combination $M_w + Itp_w$ gives the smallest number of Boolean connectives, and $M_s + Itp_s$ appears in the second place. The median of $M_w + Itp_w$ is 1, which means that it was responsible for the smallest number of connectives in at least half of the benchmarks, and its average of 1.2 shows that even when this was not the case, the combination was still close to the optimum. On the losing side, we make two observations. The EUF interpola-

tion algorithm Itp_r leads to a larger number of Boolean connectives, and the propositional interpolation algorithm P leads to larger interpolants.

Interestingly the combinations PS + Itp_s and PS_s + Itp_s have low medians and average, which are good, but not the best. This seemingly contradicts our earlier observation in Alt et al. [2016] that PS and PS_s consistently lead to small number of connectives in the interpolant. Based on the experiments the likely reason is the soundness restriction in integration (see Sec. 3.2). This restriction occurs when using a propositional and a theory interpolation system that use labeling functions to control strength. If the propositional interpolation algorithm labels a certain term aiming at a specific strength, the theory interpolation algorithm has to follow this label (see Sec. 3.2). If the strength aimed by the propositional interpolation algorithm is the opposite of the theory interpolation algorithm, redundant equalities are added by the EUF-interpolation system. We can see our hypothesis in the experiments, since the results get gradually worse when the propositional and the EUF interpolation algorithms disagree more on the labeling (strength), best being PS_s + Itp_s and the worst PS_w + Itp_s.

We can observe the same trend in Fig. 5.5 in the number of EUF equalities. A strong propositional interpolation algorithm (M_s, PS_s) combined with Itp_s leads to smaller interpolants compared to their combination with Itp_w; and a weak propositional interpolation algorithm (M_w, PS_w) combined with Itp_w leads to smaller interpolants compared to their combination with Itp_s. Also notice that PS, a propositional interpolation algorithm that tends to balance the distribution of variables Alt et al. [2016], leads to very similar results when combined with Itp_s and Itp_w.

5.3.2 Interpolation-Based Incremental Verification

We integrated the EUF-interpolation system with the incremental model checker HiFrog as part of OpenSMT2, and used it to verify a set of C benchmarks from SV-COMP (`https://sv-comp.sosy-lab.org/`) and other sources.

We use both purely propositional logic and QF_UF to model the programs. Table 5.1 shows the verification time for HiFrog with propositional logic in the column *Bool Ver. Time*; and QF_UF in the columns marked *EUF Verification Time*.

Unlike the bit-precise propositional model, the QF_UF model provides an over-approximation of the program behavior. If HiFrog reports that a safety property is true under QF_UF it is also true for the propositional model. However, if a property is reported false, it may indicate either a real or a spurious

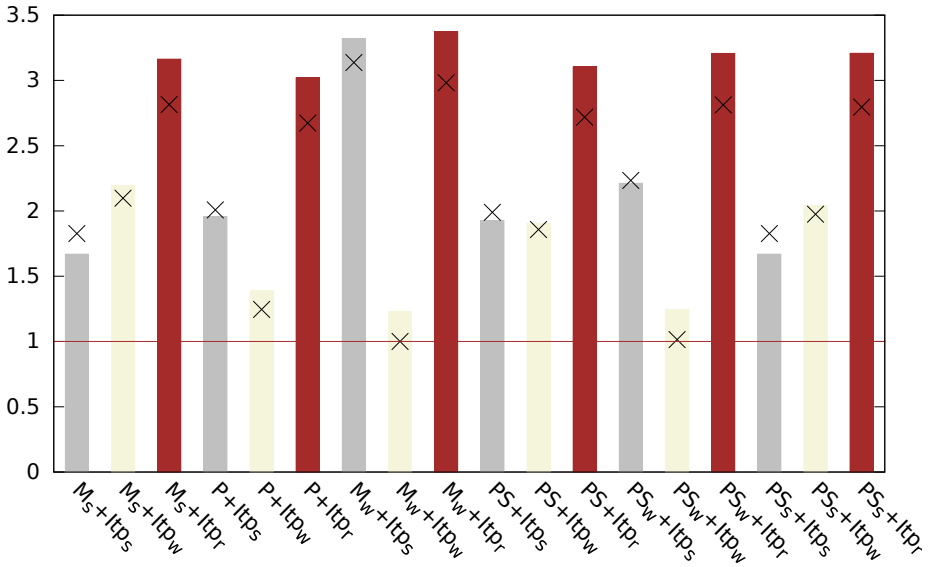


Figure 5.5. Comparison between interpolation combinations with respect to the number of equalities in the final interpolant

counterexample introduced by the EUF abstraction. In case of false properties the model checker can for instance consult the propositional encoding to get the correct result. We report run times for three variations of the model checker. Column *EUF only* reports the time used only by the EUF check. Column *+Spurious Overhead* reports the time when HiFrog is allowed to query the spuriousness of the counter-example from an oracle and only needs to consult the propositional encoding if the answer is yes. Column *+Full Overhead* reports the time when HiFrog needs to resort to the propositional encoding always in case of a failure to verify. Notably the use of EUF as an abstraction technique usually speeds up the solving even in the case of the full overhead.

We also report the effect of interpolation algorithm strength to the number of required refinements for the four combinations $\mathrm{M}_s + Itp_s$, $\mathrm{M}_s + Itp_w$, $\mathrm{M}_w + Itp_s$ and $\mathrm{M}_w + Itp_w$ in the last four columns. We note that the number of summary refinements varies sometimes considerably over the combinations, demonstrating the advantage of the flexibility our framework provides for the EUF-interpolation. Table 5.2 shows the comparison between combinations of interpolation algorithms with respect to the overall verification time spent by HiFrog. We can see that no specific combination is the overall winner, and that the EUF-interpolation system is necessary to achieve general optimal performance.

Table 5.1. Verification results of a set of C benchmarks.

	#Asrt	EUF Results			Bool Ver. Time (s)	EUF Verification Time (s)			Summary Refinements			
		Correct	SAT	Spurious SAT		EUF Only	+Spurious Overhead	+Full Overhead	M_s+ Itp_s	M_s+ Itp_w	M_w+ Itp_s	M_w+ Itp_w
floppy1	18	15	3	3	69.598	8.296	34.739	34.739	28672	27648	24320	32256
floppy2	21	18	3	3	192.079	46.65	122.544	122.544	37120	41216	37632	40704
kbfiltr1	10	10	0	0	4.123	1.304	1.304	1.304	4864	4864	4864	4864
diskperf1	14	11	3	3	193.695	29.731	67.78	67.78	45568	44544	47104	48384
floppy3	19	16	4	3	76.212	9.583	36.421	43.737	28928	34304	26624	29952
kbfiltr2	13	13	0	0	10.23	3.107	3.107	3.107	4864	4864	4864	4864
floppy4	22	19	4	3	207.261	52.564	127.869	144.073	41472	43008	40704	43776
kbfiltr3	14	14	1	0	18.664	5.649	5.649	14.563	10240	10496	10240	10496
tcas_asrt	162	149	145	13	86.032	16.753	21.648	99.95	59648	60160	59648	60160
cafe	115	100	100	15	19.164	4.273	5.77	14.665	6656	6656	6656	6656
s3	131	123	112	8	1.499	1.722	1.804	2.98	0	0	0	0
mem	149	146	52	3	44.627	59.867	59.94	78.466	23808	25088	23808	25088
ddv	152	56	105	96	260.283	11.609	122.03	122.938	7936	7936	7936	7936
token	54	54	20	0	962.277	150.977	150.977	998.568	15616	13568	15616	13568
disk	79	62	72	17	8194.963	241.268	638.209	8151.212	9472	38912	9472	38912

Table 5.2. Verification time of HiFrog using different combinations of interpolation algorithms.

	M_s +Itp_s	M_s +Itp_w	M_w +Itp_s	M_w +Itp_w
floppy1	37.314	36.255	**34.739**	39.186
floppy2	138.579	144.93	**122.544**	139.768
kbfiltr1	1.489	1.469	**1.304**	1.464
diskperf1	68.17	**58.597**	67.78	64.015
floppy3	43.783	47.744	**43.737**	44.988
kbfiltr2	**2.961**	3.082	3.107	3.001
floppy4	152.531	**142.452**	144.073	154.35
kbfiltr3	**14.626**	15.379	14.563	15.2
tcas_asrt	100.467	99.95	**99.869**	100.473
cafe	14.608	14.665	**14.582**	14.718
s3	2.98	2.98	**2.859**	2.903
mem	78.728	**78.466**	79.325	78.727
ddv	**122.487**	122.938	122.884	123.005
token	999.015	998.568	1000.373	**998.166**
disk	**8147.576**	8151.212	8150.354	8156.35
TOTAL	9925.314	9918.687	**9902.093**	9936.314

5.4 Related Work

This section describes interpolation algorithms for EUF and other similar theories, such as Arrays and theory combination. For work related to other theories please see Sections 4.3, 5.1.2, and 6.3.

The interpolation algorithms given for the theories of LRA and EUF in McMillan [2005] pioneer the field. First, proof systems for the individual theories of EUF and LRA are given. These proof systems are based on inference rules that resemble the resolution rule for propositional logic. The interpolation algorithms then extend each of those rules by annotating each formula (original or inferred) with a partial interpolant. The work presented also presents a proof system for the Boolean combination of EUF and LRA formulas, and extend those for interpolation.

The main idea of the interpolation procedures from McMillan [2005] is to collect the contribution from partition A to the proof of unsatisfiability. This leads to the fact that these interpolation procedures are able to generate only one interpolant for a given interpolation problem.

The approach presented in Fuchs et al. [2009] is a graph-based generalization of McMillan [2005]. Instead of using its own proof system, this approach is built on top of congruence graphs resulting from the congruence closure algorithm. The main idea is similar to McMillan [2005], in the sense that it traverses the congruence graph collecting the contribution to the proof that comes from partition A.

A fundamental difference to the algorithm presented in McMillan [2005] is the fact that the interpolation algorithm presented in Fuchs et al. [2009] notices equalities that contain only shared symbols and allows these equalities to be placed in A or B, without losing soundness. This flexibility can potentially lead to different interpolants.

By using duality of interpolants and labeling function, the EUF-interpolation system presented in this chapter generalizes the work from Fuchs et al. [2009], which can be instantiated in our system by one of the many possible labeling functions. The EUF-interpolation system also works on top of congruence graphs and shares the flexibility of choosing the partition of equalities that contain only shared symbols.

After presenting the literature work closely related to the EUF-interpolation system, we describe orthogonal work about interpolation for the theory of Arrays and theory combination.

The theory of Equalities and Uninterpreted Functions is often used by model checkers to abstract arrays, but in many cases that is not enough and the pro-

gram needs to be encoded using the theory of Arrays, therefore extending this need to interpolation algorithms for Arrays. In Bruttomesso et al. [2012], an interpolation procedure is given for an extension of the theory of Arrays. This extension contains the *diff* operator which tells if two arrays of different and is not considered in the classic theory of Arrays. They present an interpolating solver for the theory of Arrays, using *metarules* that prepare the proof of unsatisfiability for interpolation.

EUF is very important to theory combination, since equality is usually the common operator between different theories. When interpolating over a combination of theories is necessary the interpolation procedures have to be specialized. The work presented in Yorsh and Musuvathi [2005b] introduced the idea of equality-interpolating solvers, a necessary concept for the soundness of interpolants. Their technique uses interpolation procedures for individual theories as black-boxes, and interpolates over the equalities that are propagated between theories. In Goel et al. [2009], an extension of Fuchs et al. [2009] is given for combined theories, overcoming one possible issue of the equality propagation from Yorsh and Musuvathi [2005b]. This overhead is removed by transforming the proof of unsatisfiability from *almost-colorable* to *colorable*, and then constructing a sound interpolant from the colorable proof.

5.5 Summary and Future Work

Interpolation over EUF formulas so far has been done using algorithms that do not provide flexibility for the application. Seen as black-boxes, prior EUF interpolation algorithms generate only one interpolant out of a refutation proof, without offering any means of control.

This chapter presented a novel framework to generate interpolants for the quantifier-free theory of Equalities and Uninterpreted Functions. Our framework, the EUF-Interpolation System, is able to generate a multitude of interpolants of different strength. The strength of interpolants can be controlled by labeling functions. We show how two labeling functions can be compared with respect to the strength of the interpolants they generate.

We also prove that the strongest and weakest labeling functions of the framework generate the interpolants with the smallest number of equalities among all the possible labeling functions.

We report on experiments using the EUF-Interpolation System in combination with different propositional interpolation algorithms and in a model checker. The experiments show that it is important to match the strength

of the propositional algorithm with the EUF labeling function, and that the choice of labeling function can impact the performance of the model checker.

The EUF-Interpolation System allows new challenges to be studied, since for the first time EUF interpolation can be controlled. An interesting topic would be to analyze the effect of more complex labeling functions in the EUF-interpolation system, that generate interpolants that are more suitable to the needs of the interpolation-based application.

Another interesting topic is theory combination in SMT solving. Adapting the EUF-interpolation system for theory combination would be a non-trivial task that could lead to optimizations in interpolation for combined theories.

5.5.1 Related Publications

The results described in this chapter will appear in the FMCAD 2017 proceedings in the following paper Alt, Asadi, Hyvärinen and Sharygina [2017]:

- L. Alt, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. Duality-based Interpolation for Quantifier-Free Equalities and Uninterpreted Functions.

Chapter 6

Controlling LRA Interpolants

Software verification gained a boost with the introduction of SMT solvers. Encoding software with propositional logic became a burden that higher level theories try to soften by introducing different levels of abstraction, on arithmetic operations or function bodies, for example.

One of the most important theories to represent programs is the theory of Linear Arithmetic, since it is essential in the verification and synthesis of practical applications. Different decision and interpolation procedures exist for Linear Arithmetic in different domains, such as LRA (Linear Real Arithmetic) and LIA (Linear Integer Arithmetic). Even though reasoning over integers may be necessary for software verification in many cases, LRA is specially important since its decision procedure is much faster compared to LIA and reasoning over reals is enough in several situations, such as verification of device drivers and hybrid systems. Given the importance and the central role of interpolation in program verification, it is natural to desire LRA interpolants that meet the needs of the model checkers.

Different approaches to interpolate over LRA formulas exist, attending different needs: efficiency in generating interpolants, simplicity of the formulas, and integration with SMT solvers. Even though these different approaches exist, the flexibility of LRA interpolation systems is still an issue, since none of these techniques allow the interpolation-based application to interfere in the interpolant generation.

This chapter presents the LRA-interpolation system, a novel duality based interpolation framework that is able to generate infinitely many interpolants of different strength for a single interpolation problem. The framework is based on McMillan [2005] and extended with a parameter controlling the strength of the interpolants. In Section 6.1 we introduce the interpolation framework,

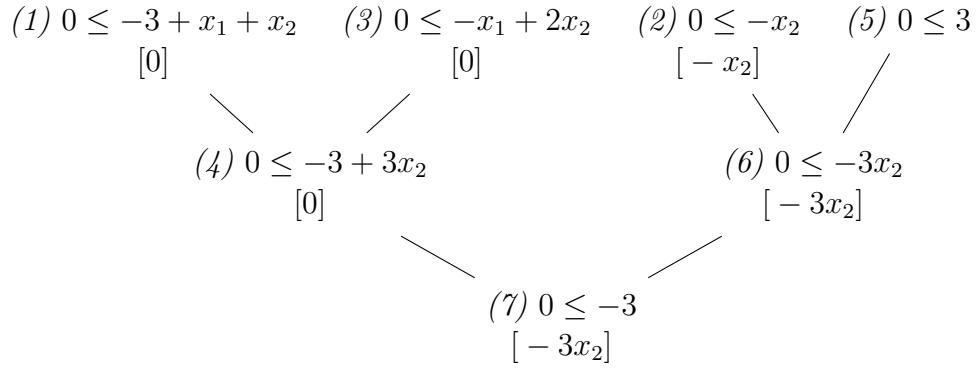
and in Section 6.1.1 we present the parameter that controls the strength of interpolants and prove its properties. To the best of our knowledge, this chapter presents the first flexible interpolation system for LRA capable of generating an infinite amount of interpolants for a given interpolation problem.

Section 6.2 presents experiments involving the LRA-interpolation system and various strength factors in a controlled setting and in a model checker.

6.1 LRA Interpolation System

An LRA interpolation problem is a 3-tuple $P = (A, B, R)$ where A and B are two sets of LRA constraints such that they are unsatisfiable when conjoined, and R is the proof of unsatisfiability for $A \cup B$. The main idea in this Chapter is to apply the duality of interpolants to the LRA interpolation algorithm presented in Chapter 3.4, and obtain new interpolation algorithm. This can be done by interpolating over B instead of A, and then negating the interpolant. Let Itp_M be the interpolation algorithm from McMillan [2005] and Itp_D its dual. Given an interpolation problem $P = (A, B, R)$, we define the interpolation algorithm Itp_D such that $Itp_D(A, B, R) = \neg Itp_M(B, A, R)$, following the duality of interpolants.

Example 15. *Recall Example 11, where we had* $A = \{0 \leq -3 + x_1 + x_2, 0 \leq -x_1 + 2x_2\}$ *and* $B = \{0 \leq -x_2\}$. *If we interpolate over* B *instead of* A *(by using* Itp_D*), we generate the following annotated proof of unsatisfiability:*



The dual interpolant for this problem is then $\neg(0 \leq -3x_2)$.

Formally, the dual interpolation algorithm works as shown in Table 6.1, where x, y, x' and y' are terms, and $[\phi]$ is the annotated term such that $0 \leq \phi$ is the partial interpolant for that node:

Table 6.1. Dual interpolation system.

$$\frac{}{0 \leq x[0]}\ (0 \leq x) \in A \qquad \frac{}{0 \leq x[x]}\ (0 \leq x) \in B$$

$$\frac{0 \leq x[x'] \quad 0 \leq y[y']}{0 \leq x + y[x' + y']} \qquad \frac{0 \leq c \quad 0 \leq x[x']}{0 \leq cx[cx']}$$

Let γ be the term annotated for the root of the proof tree (the contradiction). The inequality $0 \leq \gamma$ is an interpolant for B. By duality of interpolants, we can assert that $\neg(0 \leq \gamma)$ is an interpolant for A.

Lemma 5. *Let $P = (A, B, R)$ be an interpolation problem. Let the inequality $c_1 \leq x$ be the interpolant generated by Itp_M, where c_1 is a constant and x is an LRA term. Then Itp_D generates an interpolant of the form $\neg(c_2 \leq -x)$, where c_2 is a constant.*

Proof. Let R be a proof tree that proves the unsatisfiability of $A \wedge B$, annotated with partial interpolants. We know that R is constructed by summing the inequalities from A and B, occasionally multiplied by a constant. The proof can be seen as a way to parenthesize these operations. Using associativity of sum, we can rearrange any arbitrary proof such that the contradiction is inferred via an application of the *Comb* rule on two inequalities (1) $0 \leq -c_1 + x$ and (2) $0 \leq -c_2 - x$ such that: (i) $0 \leq -c_1 + x$ is inferred only using inequalities from A; and (ii) $0 \leq -c_2 - x$ is inferred only using inequalities from B, where x is an LRA term and $-c_1 - c_2 > 0$. Because of (i), we have that the partial interpolants for (1) and (2) when computing Itp_M are, respectively, $[-c_1 + x]$ and 0; because of (ii), we have that the partial interpolants for (1) and (2) when computing Itp_D are, respectively, 0 and $[-c_2 - x]$. Therefore, we can see that the term annotated with the contradiction node is $[-c_1 + x]$ when computing Itp_M and $[-c_2 - x]$ when computing Itp_D. Because of that, we know that the interpolant Itp_M is $c_1 \leq x$ and the interpolant Itp_D is $\neg(c_2 \leq -x)$. □

Lemma 6. *Given an interpolation problem $P = (A, B, R)$, then $Itp_M(P) \rightarrow Itp_D(P)$.*

Proof. From Lemma 5 we know that we can rearrange any proof of unsatisfiability R such that the contradiction is inferred via an application of the *Comb* rule on two inequalities (1) $0 \leq -c_1 + x$ and (2) $0 \leq -c_2 - x$ such that: (i)

$0 \leq -c_1 + x$ is inferred only using inequalities from A; and (ii) $0 \leq -c_2 - x$ is inferred only using inequalities from B, where x is an LRA term and $-c_1 - c_2 > 0$. Let $P_D = (B, A, R)$ be the problem of interpolating over B instead of A. We know that $Itp_M(P) = c_1 \leq x$ and $Itp_M(P_D) = c_2 \leq -x$. We also know that applying the *Comb* rule on (1) and (2) leads to a contradiction, so $Itp_M(P)$ implies that $Itp_M(P_D)$ cannot be true. Since $Itp_D(P) = \neg Itp_M(P_D)$, the Lemma follows. □

Lemma 7. *Let $c_1 \leq x$ and $\neg(c_2 \leq -x)$ be the interpolants generated for the same interpolation problem by Itp_M and Itp_D, respectively. The interpolants generated by Itp_M and Itp_D represent lower bounds for x, where $-c_2 \leq c_1$.*

Proof. In $Itp_M(P)$, c_1 is clearly a lower bound for x. Transforming $Itp_D(P)$ we have that $\neg(c_2 \leq -x) \equiv \neg(-c_2 \geq x) \equiv -c_2 < x$. Lemma 6 shows that $Itp_M(P) \rightarrow Itp_D(P)$, therefore, because c_1 and $-c_2$ are lower bounds for x, it is true that $-c_2 \leq c_1$. □

6.1.1 The Strength Factor

Now that we have established the strength relation between Itp_M and Itp_D, we can introduce the LRA-interpolation system. Our idea is based on the fact that Itp_M and Itp_D represent lower bounds for the same term (Lemma 7), which means that any constant c in the interval $[-c_2, c_1]$ can substitute c_1 in $Itp_M = c_1 \leq x$, to create a new interpolant $Itp_c = c \leq x$.

Lemma 8. *Let $c_1 \leq x$ and $\neg(c_2 \leq -x)$ be the interpolants generated for the same interpolation problem P by Itp_M and Itp_D, respectively. Let c be a constant such that $-c_2 < c \leq c_1$. Then $Itp_c = c \leq x$ is an interpolant for P.*

Proof. Because $c \leq c_1$, we have that $Itp_M(P) \rightarrow Itp_c(P)$. Because $-c_2 < c$, we have that $Itp_c(P) \rightarrow Itp_D(P)$. Therefore Itp_c is an interpolant for P. □

Since the bounds c_1 and $-c_2$ change from problem to problem, it is easier to normalize this interval and apply a *strength factor*. Let P be an interpolation problem such that $Itp_M(P) = c_1 \leq x$ and $Itp_D(P) = -c_2 \leq -x$. Given a factor f such that $0 \leq f \leq 1$, we can create a new interpolant $Itp_f = c_f \leq x$, where $c_f = c_1 - (f * (c_1 - -c_2))$.

We extend the notion of LRA interpolation problem to include the strength factor: $P = (A, B, R, f)$.

Notice that if $-c_2 < c_1$ it is possible to generate infinitely many interpolants of different strength for a given interpolation problem.

Algorithm 5 LRA-interpolation system

```
1: procedure Itp(P = (A, B, R, f))
2:     Requires 0 ≤ f ≤ 1.
3:     if f = 1 then
4:         return Itp_D(P)
5:     end if
6:     if f = 0 then
7:         return Itp_M(P)
8:     end if
9:     I_M ← Itp_M(P)
10:    I_D ← Itp_D(P)
11:    bound_M ← bound(I_M)
12:    bound_D ← bound(I_D)
13:    δ ← bound_M − bound_D
14:    c ← bound_M − δ ∗ f
15:    x ← term(I_M)
16:    I_f ← (c ≤ x)
17:    return I_f
18: end procedure
```

Theorem 9. *Let f and f' be two different strength factors such that $f \leq f'$. We have that $Itp(A, B, R, f) \rightarrow Itp(A, B, R, f')$.*

Proof. Analogous to the proof of Lemma 8. □

Theorem 9 shows that the strength of the LRA interpolants can be controlled by the strength factor (hence the name). The interpolation algorithm Itp_M from McMillan [2005] is represented by the strength factor 0, and generates the strongest interpolants among the LRA-interpolation system. The dual interpolation algorithm Itp_D generates the weakest interpolants.

The main advantage of the LRA-interpolation system can be visualized when it is combined with propositional interpolation in an SMT solver.

Example 16. *Let $A = x \leq 1 \wedge y \leq 1$ and $B = (y \geq 4 \wedge x \geq 0 \wedge x \leq 3) \vee (x \geq 3 \wedge y \geq 0)$ be two Boolean formulas such that the atoms are LRA terms. Notice that we cannot decide satisfiability of $A \wedge B$ using simplex only. We can, for instance, use an SMT solver. The formula $A \wedge B$ is unsatisfiable, proven by two unsatisfiable queries to the LRA solver, where each query consists of a conjunction of LRA constraints and is solved using the simplex algorithm.*

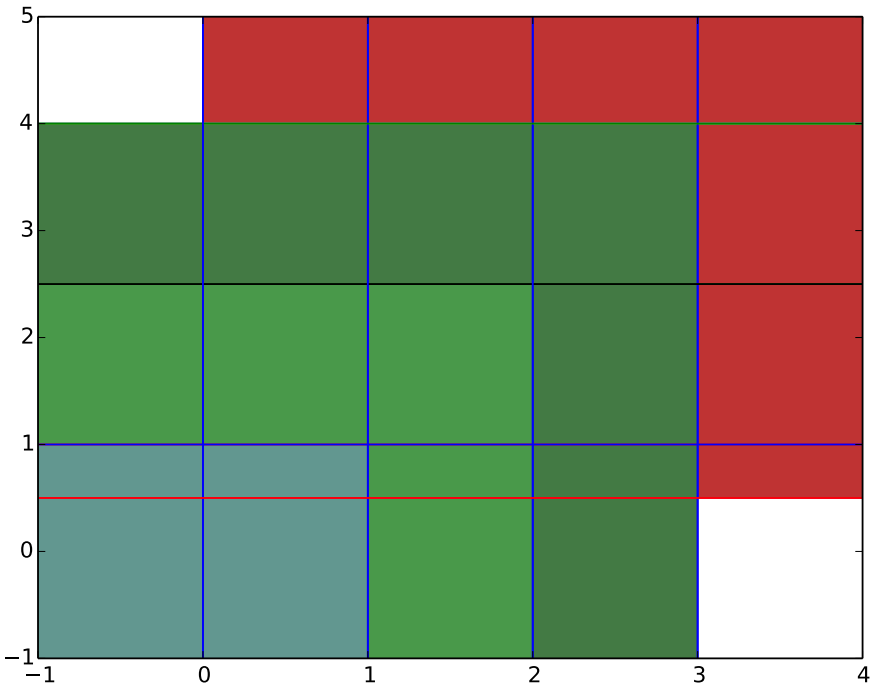


Figure 6.1. LRA problem and different interpolants.

The LRA interpolants that are generated by these queries are then used in propositional interpolation. Using a fixed propositional interpolation algorithm we get the following interpolants for A when changing the LRA interpolation algorithm:

$$\begin{aligned} Itp_M: &\quad I_M = x \leq 1 \wedge y \leq 1 \\ Itp_D: &\quad I_D = \neg x \geq 3 \wedge \neg y \geq 4 \\ Itp_{0.5}: &\quad I_{0.5} = x \leq 2 \wedge y \leq 2.5 \end{aligned}$$

Notice that naturally we have that $I_M \rightarrow I_{0.5} \rightarrow I_D$.

Fig. 6.1 shows the graphical representation of the problem. The blue region is A and the red region is B. We can see graphically that they are unsatisfiable when conjoined. The interpolant I_M *happens to be the same as A. Interpolants* $I_{0.5}$ *and* I_D *are represented, respectively, by the light and dark green areas of the graph.*

Lemma 9. *Since the strength of the LRA interpolants is given in the level of the theory interpolants, if a propositional interpolation algorithm from the Labeled Interpolation Systems is used, the strength of the final interpolant is maintained.*

Proof. Let f and f' be two strength factors such that Itp_f and $Itp_{f'}$ are the interpolation algorithms from the LRA-interpolation system derived by f and f', respectively. Let Itp be an arbitrary propositional interpolation algorithm from LIS (see Chapter 3.1). Let $P = (A, B, R)$ be an interpolation problem, I_f be the interpolant generated using Itp and Itp_f, and I'_f the interpolant generated using Itp and $Itp_{f'}$.

Eq. 3.2 and Eq. 3.3 show how an interpolation algorithm from LIS creates interpolants from the leaves to the root. Eq. 3.2 is only applied to Boolean clauses and not to theory clauses, so we can disregard it. Let n be a non-leaf node of R that has as children two theory leaves, t_1 and t_2. Let $x_1 = Itp_f(t_1)$, $x_2 = Itp_f(t_2)$, $y_1 = Itp_{f'}(t_1)$ and $y_2 = Itp_{f'}(t_2)$. We know that $x_1 \rightarrow y_1$ and $x_2 \rightarrow y_2$.

We now analyze the three possibilities to build the partial interpolant for n in Eq. 3.3:

- The first is a disjunction of the partial interpolants of t_1 and t_2. We know that $((x_1 \rightarrow y_1 \wedge x_2 \rightarrow y_2)) \rightarrow ((x_1 \vee x_2) \rightarrow (y_1 \vee y_2))$, so the strength is maintained.

- The second is a conjunction of the partial interpolants of t_1 and t_2. We know that $((x_1 \rightarrow y_1 \wedge x_2 \rightarrow y_2)) \rightarrow ((x_1 \wedge x_2) \rightarrow (y_1 \wedge y_2))$, so the strength is maintained.

- The third is a conjunction of two disjunctions, formed by the partial interpolants of t_1 and t_2 and the pivot of the resolution rule which is an arbitrary variable. It is also true that $((x_1 \rightarrow y_1 \wedge x_2 \rightarrow y_2)) \rightarrow (((x_1 \vee p) \wedge (x_2 \vee \neg p)) \rightarrow ((y_1 \vee p) \wedge (y_2 \vee \neg p)))$, so the strength is maintained.

The case where n has as children a theory leaf and a Boolean leaf clearly holds. Since the rest of R is annotated in the same way we have that the Lemma holds.

□

Lemma 9 states that the LRA-interpolation system allows strength control for the final interpolant (containing a Boolean combination of LRA atoms)

even when different strength factors are applied to different theory leaves in the refutation proof. This gives the application even more chances of generating an interpolant that suits its needs.

6.2 Experimental Evaluation

We implemented and integrated the LRA-interpolation system into the existing propositional interpolation in OpenSMT2.

LRA can be used in software model checking to abstract the heavy-weight bit-precise propositional encoding to gain speed-up due to the higher-level theory. In this case, if a model checker reports that a certain property is true when using LRA, it is also true for the propositional model. However, if a property is determined unsafe in LRA, the generated counterexample might be spurious and introduced by the LRA abstraction. We report the integration of the LRA-interpolation system in OpenSMT2 with the C model checker HiFrog, verifying a set of benchmarks that consists of C code both from the industry and from SV-COMP (`https://sv-comp.sosy-lab.org/`). HiFrog uses interpolants to create, store, and reuse function summaries to incrementally check different assertions in a program. We describe two different results: (i) comparison between different strength factors for the LRA-interpolation system and their effects on the model checker; (ii) comparison between propositional logic and LRA encoding within the model checker.

Tables 6.2 and 6.3 show, respectively, the verification time and number of function refinements in HiFrog for each benchmark using different combinations of propositional and LRA interpolation algorithms, where $\text{Itp}_{0.5}$ is the LRA strength factor 0.5, and the bold numbers are the smallest verification time or number of refinements for that benchmark. We chose the LRA strength factor of 0.5 to have the strongest and weakest interpolants in the experiment, as well as the middle factor, expected to have a mid-level of strength. In a model checking scenario where the suitable strength is not known, a simple approach would be to start with the strength factor 0.5, and tune the factor depending on whether the generate interpolants were too precise or too abstract for the model checking instance. Interestingly the per-instance winning algorithms are almost evenly distributed, making it hard to predict which algorithm provides the lowest run time on our benchmarks, the exception being the strong propositional algorithms M_s and PS_s, which score no wins. Inside each propositional algorithm Itp_s scores in total 18 wins compared to five wins of $\text{Itp}_{0.5}$ and 11 wins of Itp_w. However, for certain instances a given LRA al-

gorithm is consistently better: in particular for **kbfiltr1** $\text{Itp}_{0.5}$ almost always wins, and for **diskperf1** Itp_w always wins. Finally we note that there is little correlation between the number of refinements and the run times, suggesting that the run time invested in the solving phase may pay off in higher quality interpolants in applications where convergence is the dominating performance criterion as opposed to run time.

Similarly to EUF, LRA provides an abstraction to the bit-precise propositional encoding. If HiFrog reports that a certain property is true when using LRA, it is also true for the propositional model. Otherwise, if a property is said to be unsafe, it may be the case that the generated counterexample is spurious, introduced by the LRA abstraction. In this case, one approach that may be used by the model checker to decide the final answer is to use the precise encoding instead of the abstraction. We report run times for three settings of the model checker in Table 6.4. Column *LRA only* reports the time used only by the LRA check. Column *+Spurious Overhead* reports the time when HiFrog is allowed to query the spuriousness of the counter-example from an oracle, in case the LRA formula is SAT. In this case, HiFrog has to be executed again with the propositional encoding only if the answer is yes, that is, the counterexample is spurious. Column *+Full Overhead* reports the time when HiFrog cannot query an oracle regarding the spuriousness of a counterexample, and has to run again using the propositional encoding. Surprisingly many of the assertions can be checked only using LRA logic, as indicated by the big numbers in *correct*. However, in the current implementation, especially when resorting to the *+Full Overhead* mode, we see that the spuriousness checks result in a significant overhead. In the *+Spurious Overhead* mode where a more intelligent strategy for refinement is used, the use of LRA in encoding provides almost a three-fold speed-up for the solving.

In general the experiments implicate that modelling with LRA can provide big speed-ups with respect to propositional models and that the LRA interpolation algorithms are not forming a bottleneck for the solver performance.

6.3 Related work

This section describes interpolation algorithms for LRA and other similar theories, such as Linear Integer Arithmetic, Nonlinear Real Arithmetic and the combination of Presburger Arithmetic with Uninterpreted terms and Arrays. For work related to other theories please see Sections 4.3, 5.1.2, and 5.4.

The pioneering work in LRA interpolation is Pudlák [1997], which shows

Table 6.2. Verification time for HiFrog using different combinations of propositional and LRA interpolation algorithms.

	M_s +Itp_s	M_s +$Itp_{0.5}$	M_s +Itp_s	P +Itp_s	P +$Itp_{0.5}$	P +Itp_w	M_w +Itp_s	M_w +$Itp_{0.5}$	M_w +Itp_w
floppy1	28.242	33.824	28.154	28.017	34.123	28.076	**27.464**	34.135	27.905
tcas_asrt	66.648	65.97	65.982	66.595	66.188	66.599	65.431	66.981	67.278
kbfiltr1	5.234	5.424	5.282	5.376	5.153	5.384	5.296	5.161	5.45
diskperf1	603.58	561.444	476.029	586.589	547.595	**439.785**	608.748	607.272	475.023
cafe	4.861	4.785	4.858	4.949	4.821	4.807	4.753	**4.738**	4.798
s3	1.765	1.75	1.774	1.779	1.749	1.785	1.77	1.801	1.788
mem	105.547	135.58	106.736	104.484	135.294	106.198	106.097	137.287	106.24
ddv	12.74	12.963	12.997	12.737	12.807	13.085	12.488	12.856	12.725
disk	809.027	988.197	1203.24	**799.117**	997.529	1285.17	799.523	977.445	1254.65
TOTAL	1637.644	1809.937	1905.052	**1609.643**	1805.259	1950.889	1631.57	1847.676	1955.857
	PS +Itp_s	PS +$Itp_{0.5}$	PS +Itp_w	PS_w +Itp_s	PS_w +$Itp_{0.5}$	PS_w +Itp_w	PS_s +Itp_s	PS_s +$Itp_{0.5}$	PS_s Itp_w
floppy1	28.125	34.139	28.049	27.782	34.099	28.054	28.052	34.176	27.78
tcas_asrt	67.56	65.509	66.211	65.895	**65.387**	66.911	66.341	66.41	68.155
kbfiltr1	5.44	**5.098**	5.565	5.506	5.282	5.567	5.49	5.121	5.623
diskperf1	666.463	557.978	472.857	616.047	534.862	452.984	603.903	548.572	446.217
cafe	4.81	4.796	4.787	4.818	4.751	4.827	4.847	4.768	4.801
s3	1.779	1.79	**1.726**	1.755	1.744	1.752	1.758	1.773	1.772
mem	105.141	134.812	106.866	**103.7**	135.601	105.707	105.124	135.805	106.601
ddv	12.79	13.011	12.673	12.779	12.518	12.793	12.958	12.607	**12.467**
disk	803.73	1001.47	1238.24	826.235	998.468	1248.02	815.055	996.464	1293.55
TOTAL	1695.838	1818.603	1936.974	1664.517	1792.712	1926.615	1643.528	1805.696	1966.966

Table 6.3. Number of function refinements for HiFrog using different combinations of propositional and LRA interpolation algorithms.

	M_s +Itp_s	M_s +$Itp_{0.5}$	M_s +Itp_s	P +Itp_s	P +$Itp_{0.5}$	P +Itp_w	M_w +Itp_s	M_w +$Itp_{0.5}$	M_w +Itp_w
floppy1	27136	25088	24832	27136	25088	24832	27136	25088	**24576**
tcas_asrt	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**
kbfiltr1	**5120**	**5120**	5376	**5120**	**5120**	5376	**5120**	**5120**	5376
diskperf1	39936	39168	39168	39680	39168	39168	41472	**37376**	39168
cafe	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**
s3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
mem	25600	**25088**	25600	25600	**25088**	25600	25600	**25088**	25600
ddv	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**
disk	47616	**41472**	64000	47616	**41472**	64000	47616	**41472**	64000
TOTAL	213504	204032	227072	213248	204032	227072	215040	**202240**	226816
	PS +Itp_s	PS +$Itp_{0.5}$	PS +Itp_w	PS_w +Itp_s	PS_w +$Itp_{0.5}$	PS_w +Itp_w	PS_s +Itp_s	PS_s +$Itp_{0.5}$	PS_s Itp_w
floppy1	27136	25088	**24576**	27136	25088	24832	27136	25088	**24576**
tcas_asrt	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**	**53760**
kbfiltr1	**5120**	**5120**	5376	**5120**	**5120**	5376	**5120**	**5120**	5376
diskperf1	40192	39168	38144	39680	39168	39936	39168	39168	39168
cafe	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**	**6400**
s3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
mem	25600	**25088**	25600	25600	**25088**	25600	25600	**25088**	25600
ddv	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**	**7936**
disk	47616	**41472**	64000	47616	**41472**	64000	47616	**41472**	64000
TOTAL	213760	204032	225792	213248	204032	227840	212736	204032	226816

Table 6.4. Comparison between propositional and LRA encoding in HiFrog.

	#Assertions	LRA Results			Bool Verification Time (s)	LRA Verification Time (s)		
		Correct	SAT	Spurious SAT		LRA Only	+Spurious Overhead	+Full Overhead
floppy1	21	16	5	5	192	27.5	193	193
tcas_asrt	162	162	132	0	86	65.5	65.5	144
kbfiltr1	10	10	0	0	4.12	5.3	5.3	5.3
diskperf1	14	10	4	4	194	609	667	667
cafe	115	105	95	10	19.2	4.75	5.9	14.8
s3	131	126	109	5	1.5	1.77	1.82	3
mem	149	146	52	3	44.6	106	106	125
ddv	152	56	105	96	260	12.5	123	124
disk	79	62	72	17	8190	800	1200	8710

that an interpolant can be generated by first creating a proof of unsatisfiability for $A \wedge B$ such that a contradiction of the form $0 \leq c$ is derived, where $c < 0$. The interpolant is then obtained out of the proof by combining inequalities from A. This work has been used in McMillan [2005], where inference rules are given and extended to accommodate the interpolation algorithm from Pudlák [1997].

The work presented in Dutertre and de Moura [2006] aims at integrating SMT solving and LRA interpolation in a better way. This is done by extracting the proof of unsatisfiability from the simplex algorithm, which is the most used LRA decision algorithm by SMT solvers.

Our work uses the technique from Dutertre and de Moura [2006] and generalizes McMillan [2005], in the sense that the LRA-interpolation system is able to generate an infinite amount of interpolants for a single interpolation problem, whereas the interpolation algorithm from Pudlák [1997]; McMillan [2005] can only generate one.

The work presented in Albarghouthi and McMillan [2013] aims at constructing interpolants that are "beautiful". In practice, this is reflected in an attempt to construct convex interpolants for $A \wedge B$, where A and B are disjunctions of constraints. Their idea is to interpolate over subsets of A and B first, trying to converge to an interpolant that covers A and B without using A and B entirely, which can lead to interpolants that are more general and have a simpler geometrical shape.

The LRA-interpolation system is orthogonal to Albarghouthi and McMillan [2013], since it aims at creating interpolants of different strength. The techniques can also be used together, with the algorithm from Albarghouthi and McMillan [2013] using the LRA-interpolation system to generate interpolants for specific subsets of A and B.

Although it is common that interpolation procedures work on a proof of unsatisfiability, it is also possible to generate interpolants without a structure representing the proof of unsatisfiability. Rybalchenko and Sofronie-Stokkermans [2007] gives an interpolation procedure for LRA following this approach. Their idea is to first reduce the interpolation problem to constraint solving in linear arithmetic, and then, to apply a linear programming solver as a black box to generate the interpolant.

The goal of Rybalchenko and Sofronie-Stokkermans [2007] is to provide an efficient interpolation procedure for LRA. The main practical issue with their approach is that most of SMT solvers already use the simplex algorithm for LRA solving, which makes proof-based LRA interpolation very efficient afterwards. The algorithm from Rybalchenko and Sofronie-Stokkermans [2007]

is also not able to generate multiple interpolants for a given problem, and does not provide strength control as the LRA-interpolation system does.

The work presented in Scholl et al. [2014] extends Rybalchenko and Sofronie-Stokkermans [2007] in the sense that it also computes LRA interpolants by linear programming. Instead of computing interpolants for the single theory conflicts, it optimizes the computation by computing shared interpolants for a maximal number of conflicts, thus minimizing the number of linear equations in the interpolant. This work shares the disadvantages of Rybalchenko and Sofronie-Stokkermans [2007], and also does not provide control on the strength of the generated interpolants.

A basic difference between the LRA-interpolation system and other interpolation algorithms is the control over strength. The interpolation procedure for Nonlinear Real Arithmetic presented in Gao and Zufferey [2016] is similar to the LRA-interpolation system in the sense that it is able to provide interpolants of different strength, controlled by a labeling function. The interpolation system from Gao and Zufferey [2016] uses the proof of unsatisfiability generated by a decision procedure based on Interval Constraint Propagation. In a high level, their system is similar to Pudlák [1997].

After presenting the closer related work, we proceed to presenting interpolation procedures that support other theories, therefore being orthogonal to the LRA-interpolation system presented here. When reasoning over the integers is necessary, a specialized interpolation algorithm needs to be applied. In Brillout et al. [2011b], an interpolating sequent calculus is presented for the quantifier-free fragment of Presburger Arithmetic (PA). This proof system is then extended to support annotation with partial interpolants. This work introduces an interpolating cut-rule called *strengthen* which subsumes many cut-rules for linear integer programming. This cut-rule is able to handle mixed cuts, a known difficulty when interpolating over PA. The work in Brillout et al. [2011a] extends Brillout et al. [2011b] by introducing interpolation procedures for quantified Presburger Arithmetic and its combination with Uninterpreted Predicates, Uninterpreted Functions and Extensional Arrays. In another attempt at reducing the difficulty of interpolating over LIA, the work presented in Jain et al. [2009] aims at efficiently creating interpolants for subsets of Liner Integer Arithmetic, such as Diophantine and modular equations, and Diophantine disequations. Griggio et al. [2011] give an efficient interpolation procedure for an augmented version of LIA: the theory is extended by adding a ceiling function. This leads to a simplification in the interpolation procedure and in simpler interpolants.

6.4 Summary and Future Work

Several works have been trying to improve LRA interpolation, given the importance of the LRA theory for software verification. The current LRA interpolation algorithms do not provide flexibility or control over the interpolants it generates, which may lead to a drawback in the performance and usability of an interpolation-based application.

We introduced in this chapter the LRA-interpolation system, an interpolation framework for the theory of Linear Real Arithmetic that is able to generate an infinite amount of interpolants for a given interpolation problem. These interpolants have different strength which can be controlled by a strength factor.

This chapter presented also experiments both combining several propositional interpolation algorithms with the LRA-interpolation system and integrated with the interpolation-based model checker HiFrog.

The LRA-interpolation system opens new possibilities for LRA interpolation. Since it allows the application to define the strength of the interpolants, it would be interesting to see, for instance, what happens to interpolants when different factors are applied in different parts of the system that is being verified. This way the application can give a different focus to parts of the system that matter the most.

Another interesting task would be to combine the LRA-interpolation system with the technique from Albarghouthi and McMillan [2013] to help on convergence of simple interpolants. This could potentially give a great performance boost to systems that are based on LRA interpolation.

6.4.1 Related Publications

The results described in this chapter will appear in the HVC 2017 proceedings in the following paper Alt, Hyvärinen and Sharygina [2017]:

- L. Alt, A. E. J. Hyvärinen, and N. Sharygina. LRA Interpolants from No Man's Land.

Chapter 7

Implementation

7.1 OpenSMT2

We implemented the previous and novel interpolation algorithms for propositional logic from Chapter 4, the *EUF*-interpolation system from Chapter 5 and the *LRA*-interpolation system from Chapter 6 in OpenSMT2. This section describes the architecture and implementation details of OpenSMT2 which can be found at `http://verify.inf.usi.ch/opensmt`. OpenSMT2 was used in the evaluation experiments for Chapters 4, 5 and 6, and the results can be found in those chapters.

7.1.1 Basic functionalities

OpenSMT2 is build on the foundations of the previous OpenSMT generations, while providing more efficient data structures and native support to smt-libversion 2. Besides, OpenSMT2 offers a wide range of new features, such as parallel solving, incremental solving, interpolation for different theories, and SAT proof compression. Fig. 7.1 shows a high level overview of the architecture of OpenSMT2.

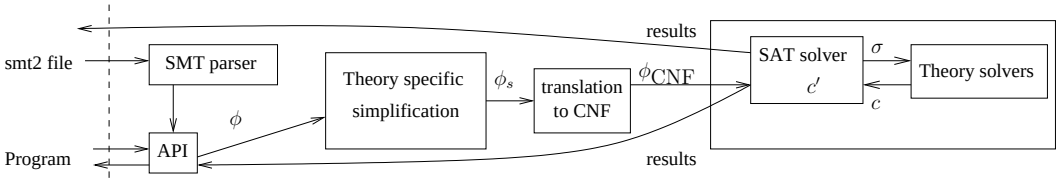


Figure 7.1. Overview of the architecture of OpenSMT2.

OpenSMT2 supports receiving a problem either from an smt-libfile or from an interface as a library. The problem is transformed into an SMT formula ϕ which is the simplified into a formula ϕ_s, using simplifications from both the Boolean and theory levels. The simplified formula ϕ_s is then converted into CNF resulting in the formula ϕ_{CNF}. The SAT solver then receives ϕ_{CNF} and starts the search. When there is a satisfying assignment found by the SAT solver, the corresponding theory solver has to certify that this is indeed a satisfying assignment. In case this assignment is actually unsatisfiable, the theory solver returns a new clause c and the problem is updated to $\phi_{CNF} := \phi_{CNF} \wedge c$. The search terminates when either the SAT solver says that ϕ_{CNF} is unsatisfiable or the theory solver accepts an assignment as SAT.

7.1.2 Modularity

OpenSMT2 is an open source SMT solver written in C++ that has as one of its goals modularity and easy approachability by newcomers. In an ideal case, one would be able to create a solver for a new theory without even touching the existing code, only by plugging in new files.

A theory τ in OpenSMT2 consists of three elements: a logic L_τ that represents the language of τ; a solver S_τ containing the solving algorithms; and a theory T_τ that connects the logic and the solver. There are four abstract classes that have to be derived when implementing a new solver: Theory, Logic, TSolver and TSolverHandler. Table 7.1 describes the necessary methods that have to be implemented for each class when supporting a new theory.

7.1.3 Interpolation Modules

Fig. 7.2 shows the overall architecture of OpenSMT2 from a new perspective, highlighting the interpolation modules. When the given problem is UNSAT, it is possible to retrieve interpolants. The application must tell OpenSMT2 what should be the partitioning (A, B) of the problem such that the interpolant is an over-approximation of A. If this information is not given OpenSMT2 assumes the first asserted formula as A and the rest as B.

It is also possible for the application to make extra requirements to the interpolation module.The application can request proof reduction and the interpolation algorithm for each theory.

The proof reduction module implements the following proof reduction techniques: (i) RecyclePivotsWithIntersection (RPI) from Bar-Ilan et al. [2009]; Fontaine et al. [2011b]; (ii) LowerUnits (LU) from Fontaine et al. [2011b]; (iii)

Table 7.1. Abstract methods that must be overridden to implement new theories.

Method	Description
	Theory
simplify	Entry point for theory specific simplifications.
	Logic
mkConst	Create logic-specific constants.
isUFEquality	Check whether a given equality is uninterpreted.
isTheoryEquality	Check whether a given equality is from a theory.
insertTerm	Insert a theory term.
retrieveSubstitutions	Get the substitutions based on the logic.
	TSolverHandler
assertLit_special	Assert literals in the simplification phase.
	TSolver
assertLit	Assert a theory literal.
pushBacktrackPoint, popBacktrackPoint	Incrementally add and remove asserted theory literals.
check	Check theory consistency of the asserted literals.
getValue	obtain a value of a theory term once a model has been found.
computeModel	compute a concrete model for the theory terms once the theory solver finds a model consistent.
getConflict	return a compact explanation of the theory-inconsistency in the form of theory literals.
getDeduction	get theory literals implied under the current assignment.
declareTerm	inform the theory solvers about a theory literal.

a structural hashing approach (SH) similar to Cotton [2010]; (iv) and the local rewritting rules from Rollini et al. [2011, 2014]; Bruttomesso, Rollini, Sharygina and Tsitovich [2010].

The LIS framework D'Silva et al. [2010] is implemented for propositional logic, with the built-in interpolation algorithms from the literature M_s McMillan [2005], P Pudlák [1997], M_w D'Silva et al. [2010], and the novel interpolation algorithms presented in Chapter 4 PS, PS_w and PS_s. If no interpolation algorithm is chosen, M_s is used. The application can also provide its own labeling function to be used by LIS.

The EUF-interpolation system from Chapter 5 is implemented for EUF, together with the build-in interpolation algorithms Itp_s, Itp_w and Itp_r from Chapter 5, where Itp_s is equivalent to the algorithm from Fuchs et al. [2009]. If no interpolation algorithm is chosen, Itp_s is used. The application can also provide its own labeling function for the congruence graph.

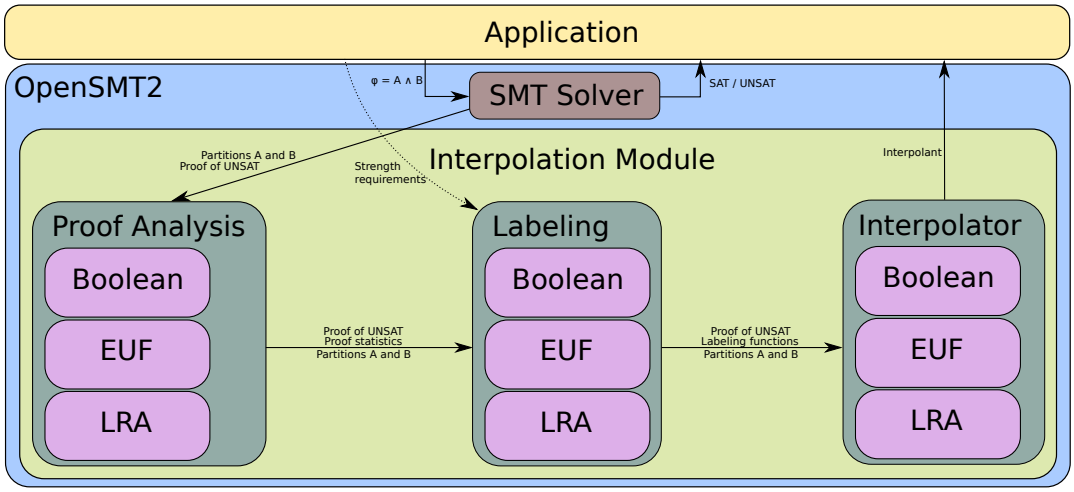


Figure 7.2. Overall verification/interpolation framework.

The LRA-interpolation system from Chapter 6 is implemented for LRA, and accepts the normalized strength factors from Chapter 6, where 0 is equivalent to the algorithm from McMillan [2005]. If no strength factor is given, 0 is chosen.

After this information is given, OpenSMT2 starts by explicitly building the SAT proof of unsatisfiability of the given problem. The Boolean interpolation module might need to collect statistics from the proof before the interpolation process starts, in case the Proof-Sensitive interpolation algorithms are used. After applying a topological sorting, the graph is traversed bottom up applying the interpolation rules from LIS using the specified propositional interpolation algorithm to annotate each node with its partial interpolant. For the theory leaves, the specific theory interpolation system is used with the specified labeling function (for EUF) or strength factor (for LRA). When the interpolant is constructed it is finally returned to the application.

7.2 HiFrog

The novel first order interpolation techniques presented in this thesis were integrated in a new model checker, HiFrog. HiFrog is an SMT-based bounded model checker for C that uses interpolants to create function summaries and reuse them in the incremental checking way presented in Chapter 2.2. Besides, it supports different levels of abstraction by using an SMT-solver and the theories EUF, LRA, and propositional logic. This section describes the architecture

and implementation details of HiFrog.

The basic verification flow applied by HiFrog is the following: the C program to be verified is given, along with previously computed or user defined function summaries. The assertions in the C program are processed incrementally, using the applicable function summaries. If the result of the verification of an assertion is SAT, refinement is done. Otherwise, the interpolator SMT solver provides interpolants for the UNSAT query, which then are stored as function summaries.

Fig. 7.3 gives an overview of the many components of HiFrog. The parser consists of the goto-cc symbolic compiler. It transforms the C code into a *goto-program* such that all loops and recursive calls are unwound up to a certain bound. The SMT encoder is called for each assertion, and starts by creating an SSA version of the unwound program. This guarantees that each function call has its own SSA representation. At this point, the SMT encoder applies the chosen logic (propositional, EUF or LRA) and a theory-specific formula is generated, along with the applicable function summaries in that theory. Function summaries can be either previously generated or user defined. The formula is then given to the SMT solver, which determines if that assertion is safe or not. If the SMT solver returns UNSAT, we have that the assertion is true, and function summaries are extracted. Otherwise, refinement is needed. If function summaries were used, it may be the case that they introduced spurious behaviors into the formula. In this case, the summaries involved in this assertion are replaced by the precise encoding of the functions they summarize, and the SMT solver is called again.

The different theories supported by HiFrog allow performance improvement and scalability, since encoding arithmetic operations, for instance, using bit-precise encoding can be expensive. Replacing bit-precision encoding with linear arithmetic or uninterpreted functions can significantly reduce verification time and still be effective, since an UNSAT result for the theory-abstracted formula also holds for the original problem. If the theory-abstracted formula leads to a SAT result, it may be because of spurious behaviors introduced by the abstraction, in which case the theory has to be refined to a more precise one.

7.3 Summary of the Experimental Evaluation

This thesis presented thorough experimental evaluation for the novel interpolation systems and algorithms that were provided for propositional logic in Chapter 4, EUF in Chapter 5, and LRA in Chapter 6. We have implemented all

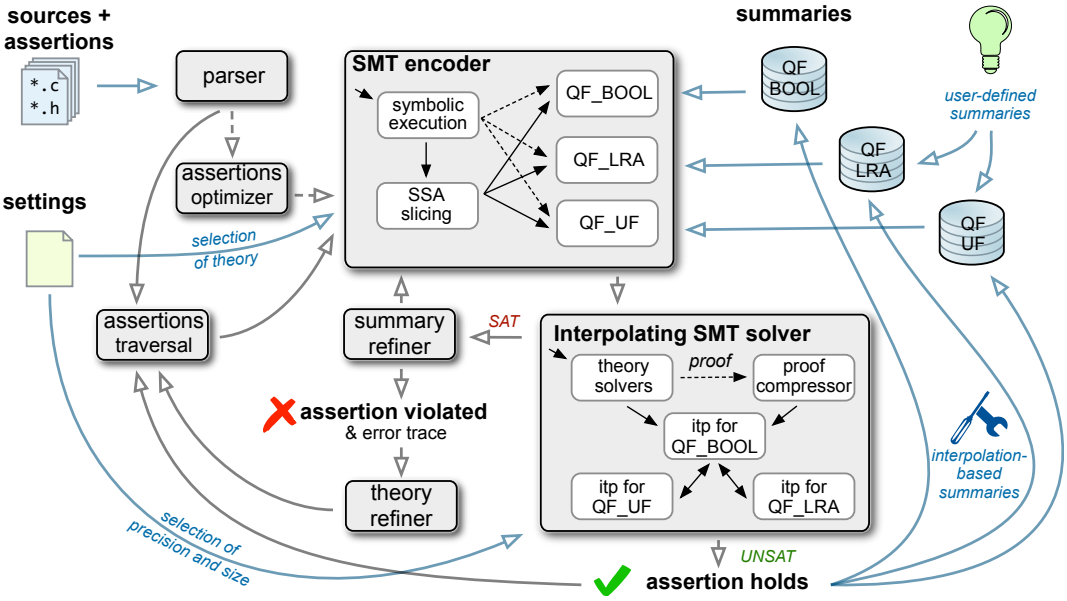


Figure 7.3. HiFrog's architecture overview.

the new algorithms in OpenSMT2, and integrated it with interpolation-based model checkers.

For propositional logic, we analyzed the effect of different labeling functions for the LIS interpolation framework in two model checking scenarios: (i) incremental checking with FunFrog; and (ii) upgrade checking with eVolCheck. Both scenarios had as a task the verification of a set of C benchmarks with a complex call-tree, and both tools are bounded model checkers for C that rely on propositional logic to encode the program, and use interpolants to create function summaries which are stored and reused.

From the initial comparison between the LIS labeling functions from the literature M_s, P, and M_w, we could see that the strength of propositional interpolants is very important for the application, and knowing what level of strength an application needs may increase convergence. In our case, we had that FunFrog is an application that works well with stronger interpolants, whereas weaker interpolants are good for eVolCheck. Considering the integration of propositional and theory interpolants, this difference for HiFrog which works better with stronger interpolants, for instance, becomes even more apparent. We also saw that creating small interpolants in a lightweight manner, avoiding the overhead of expensive procedures such as proof reduction and interpolant minimization, is important for the overall model checking performance. This became clear in the next set of experiments with propositional

interpolation, when we introduced the Proof-Sensitive labeling functions. The experiments confirmed the theory, showing that the labeling function PS is able to generate small interpolants. We also had that PS_s, combining the ability of generating interpolants of small size and guaranteeing strength, consistently led to a model checking performance that was better than the other labeling functions, both in FunFrog and eVolCheck.

For the experiments with the first order theories EUF and LRA we have integrated OpenSMT2 with HiFrog, relying, respectively, on the EUF-interpolation system and on the LRA-interpolation system, both implemented in the OpenSMT2 interpolation module.

In the experiments where HiFrog used EUF, we wanted to observe the viability of the EUF-interpolation system, that is, if the newly introduced interpolation algorithms Itp_w and Itp_r, derived from the labeling functions for the EUF-interpolation system L_s and L_r, would not lead to an overhead in EUF interpolation. Our experiments followed the theory, showing that Itp_s and Itp_w do generate the EUF interpolants with the smallest number of equalities. As a second goal, we observed the strength of the generated interpolants, and we have that indeed the interpolants had different strength, showing that the EUF-interpolation system is capable of generating interpolants that are different not only syntactically but also with respect to their logical strength. For these two first observations we used complex smt-libbenchmarks, partitioning the problem into A and B arbitrarily. Our last observation is the combination of the various propositional interpolation algorithms studied previously with Itp_s, Itp_w and Itp_r in a model checker, responsible for verifying a set of benchmarks from SV-COMP. To the best of our knowledge this is the first experiment combining propositional and theory interpolation algorithms. Our experiments show interesting results, such as the need to align the strength of the propositional and theory interpolation algorithms in order to generate smaller interpolants.

For the LRA experiments, we knew theoretically that the size of the interpolants generated by different LRA interpolation algorithms cannot be different. Our goal with this set of experiments was again to combine propositional interpolation algorithms with theory interpolation algorithms, in this case LRA, and analyze their effect in the performance of the model checker. These experiments also show very interesting results, such as the fact that the LRA strength factor that led to the smallest number of refinement steps in the model checker, that is, faster convergence, was not the strongest nor the weakest, but 0.5, the central strength factor. This shows that interpolants of very fine tuned strength are necessary to achieve optimal performance.

Chapter 8

Conclusions

Craig interpolants have been successfully used in symbolic model checking as means of over-approximation in different approaches. Interpolants can be computed from resolution proofs of unsatisfiability, and being able to generate multiple interpolants from a fixed proof is essential in the search for optimized interpolants.

This thesis addresses the problems of how to give control over the generation of interpolants to the application for different theories, and how to generate interpolants that increase the performance of the application. The main research contributions are summarized as follows.

Experimental and Theoretical Analysis of Labeling Functions for Propositional Logic. The first half of Chapter 4 starts by presenting preliminary experiments that motivated a theoretical study on the labeling functions. These experiments showed which general properties of interpolants are important, and that the labeling function from the literature should be improved. A thorough theoretical analysis then follows, showing which characteristics a labeling function needs in order to generate interpolants with a small number of Boolean connectives.

Proof-Sensitive Labeling Functions for Propositional Logic. Following on the theoretical analysis of labeling functions for propositional logic, Chapter 4 presents the Proof-Sensitive (PS) labeling function, proven to generate small interpolants in an efficient way. Two variants of PS, PS_w and PS_s also guarantee the strength of the created interpolants, which might be an important requirement in different interpolation-based applications. Our experiments show that the Proof-Sensitive labeling functions PS and PS_s out-

perform the others when used in two model checking scenarios, incremental checking and upgrade checking, by generating smaller interpolants.

Controlling EUF interpolants Chapter 5 presents the EUF-interpolation system, a duality-based interpolation framework capable of generating multiple interpolants for a single EUF proof of unsatisfiability. The framework allows control over the interpolants using labeling functions which can be compared with respect to strength. We also show that the strongest and weakest labeling functions within the EUF-interpolation system are also the ones that lead to the smallest number of equalities in an EUF interpolant. Our experiments with smt-libbenchmarks show that the framework indeed generates interpolants of different strength. We also integrated it with a model checker, combining propositional and EUF interpolation. We compared different combinations of interpolation algorithms, and showed that (i) it is important to align the strength of the interpolation algorithms for the different theories; and (ii) the strongest and the weakest EUF labeling functions do lead to a better performance in the model checker.

Controlling LRA interpolants Chapter 6 introduces the LRA-interpolation system, an interpolation framework capable of generating an infinite amount of LRA interpolants from the same proof of unsatisfiability. The main idea of the LRA-interpolation system is to compute a conventional interpolant, using an LRA interpolation algorithm from the literature, and its dual. We prove that these two interpolants are different bounds represented as inequalities, and that it is possible to derive an interval of real numbers leading to infinite possibilities for LRA interpolants. We show how the strength of such interpolants can be controlled using a normalized strength factor. We integrated the LRA-interpolation system with a model checker and our experiments show that it is very important to be able to fine tune the strength of interpolants to help convergence in the model checker.

Interpolating OpenSMT2. We have implemented all the novel interpolation frameworks for propositional logic, EUF and LRA in the SMT solver OpenSMT2, creating an interpolation module. OpenSMT2 is an efficient and open source SMT solver that has support to solving propositional logic, EUF and LRA, interpolation, and parallelism. Chapter 7.1 describes the architecture and implementation details of OpenSMT2. OpenSMT2 is available at `http://verify.inf.usi.ch/opensmt`.

HiFrog. Function summarization-based incremental model checkers suffer from the exponential explosion in the size of the formulas when translating programs into propositional logic. We attend this need by creating HiFrog, an incremental bounded model checker that uses interpolants as over-approximations of function summaries, and uses the theories EUF and LRA to encode C programs. Bit-precision is often not necessary to prove safety properties, and our experiments show that by using these first order theories, SMT-based verification can be much faster than only SAT-based verification. Chapter 7.2 describes the architecture and implementation details of HiFrog which can be found at `http://verify.inf.usi.ch/hifrog`.

8.1 Future work

The use of Craig interpolants in symbolic model checking still has a lot of potential to be further explored, especially in SMT. The EUF-interpolation system enables the possibility of fine tuning interpolation for theory combination which is commonly relied on because of the equality operator. Theory combination is a non-trivial topic that attracts the attention of many researchers, and improving interpolation in this topic is essential for optimal performance when interpolating over combination of theories.

The LRA-interpolation system, even though already able to generate an infinite amount of interpolants of different strength, may be improved. One possible approach is to use Farkas' lemma to generate even more interpolants that have a different geometrical shape. The strength might not be directly comparable to the ones currently generated by the LRA-interpolation system, but theoretical work on it could provide means of controlling the general strength of LRA interpolants. Combining the LRA-interpolation system with the Beautiful Interpolants Albarghouthi and McMillan [2013] technique also sounds promising, leading to an approach that could be able to generate simple interpolants that guarantee strength control.

In a more general discussion, the idea of duality-based interpolation can be further applied to different theories, leading to more theory interpolation algorithms that provide control over the created interpolants. It would be interesting to analyze this idea specially for the theories of Linear Integer Arithmetics and the Theory of Arrays.

Bibliography

Albarghouthi, A., Gurfinkel, A. and Chechik, M. [2012]. Whale: An interpolation-based algorithm for inter-procedural verification, *in* V. Kuncak and A. Rybalchenko (eds), *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 39–55.

Albarghouthi, A. and McMillan, K. L. [2013]. Beautiful interpolants, *in* N. Sharygina and H. Veith (eds), *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 313–329.

Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S. and Sharygina, N. [2012]. Lazy abstraction with interpolants for arrays, *in* N. Bjørner and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning: 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 46–61.

Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2017]. Hifrog: Smt-based function summarization for software verification, *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pp. 207–213.

Alt, L., Asadi, S., Hyvärinen, A. E. J. and Sharygina, N. [2017]. Duality-based interpolation for quantifier-free equalities and uninterpreted functions, *FMCAD 2017, to appear.*

Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2016]. A proof-sensitive approach for small propositional interpolants, *in* A. Gurfinkel and

S. A. Seshia (eds), *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18–19, 2015. Revised Selected Papers*, Springer International Publishing, Cham, pp. 1–18.

Alt, L., Hyvärinen, A. E. J. and Sharygina, N. [2017]. LRA interpolants from no man's land, *HVC 2017, to appear.*

Armando, A., Mantovani, J. and Platania, L. [2006]. Bounded model checking of software using smt solvers instead of sat solvers, *in* A. Valmari (ed.), *Model Checking Software: 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 146–162.

Audemard, G., Cimatti, A., Kornilowicz, A. and Sebastiani, R. [2002a]. Bounded model checking for timed systems, *in* D. A. Peled and M. Y. Vardi (eds), *Formal Techniques for Networked and Distributed Sytems — FORTE 2002: 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11–14, 2002 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 243–259.

Audemard, G., Cimatti, A., Kornilowicz, A. and Sebastiani, R. [2002b]. Bounded model checking for timed systems, *in* D. A. Peled and M. Y. Vardi (eds), *Formal Techniques for Networked and Distributed Sytems — FORTE 2002: 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11–14, 2002 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 243–259.

Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O. and Strichman, O. [2009]. Linear-time reductions of resolution proofs, *in* H. Chockler and A. J. Hu (eds), *Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 114–128.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. and Tinelli, C. [2011]. Cvc4, *in* G. Gopalakrishnan and S. Qadeer (eds), *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 171–177.

Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A. and Zuck, L. [2005]. Tvoc: A translation validator for optimizing compilers, *in* K. Etessami and S. K. Rajamani (eds), *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 291–295.

Barrett, C., Fontaine, P. and Tinelli, C. [2015]. The SMT-LIB Standard: Version 2.5, *Technical report*, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

Beyer, D. and Keremoglu, M. E. [2011]. Cpachecker: A tool for configurable software verification, *in* G. Gopalakrishnan and S. Qadeer (eds), *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 184–190.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M. and Zhu, Y. [1999]. Symbolic model checking using sat procedures instead of bdds, *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, ACM, New York, NY, USA, pp. 317–320.

Biere, A., Cimatti, A., Clarke, E., Strichman, O. and Zhu, Y. [2003]. Bounded model checking, *Advances in Computers* **58**: 118–149.

Bjesse, P., Leonard, T. and Mokkedem, A. [2001]. Finding bugs in an alpha microprocessor using satisfiability solvers, *in* G. Berry, H. Comon and A. Finkel (eds), *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 454–464.

Bloem, R., Malik, S., Schlaipfer, M. and Weissenbacher, G. [2014]. Reduction of resolution refutations and interpolants via subsumption, *in* E. Yahav (ed.), *Hardware and Software: Verification and Testing: 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, Springer International Publishing, Cham, pp. 188–203.

Bradley, A. R. [2011]. Sat-based model checking without unrolling, *in* R. Jhala and D. Schmidt (eds), *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 70–87.

Brillout, A., Kroening, D., Rümmer, P. and Wahl, T. [2011a]. Beyond quantifier-free interpolation in extensions of presburger arithmetic, *in* R. Jhala and D. Schmidt (eds), *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 88–102.

Brillout, A., Kroening, D., Rümmer, P. and Wahl, T. [2011b]. An interpolating sequent calculus for quantifier-free presburger arithmetic, *Journal of Automated Reasoning* **47**(4): 341–367.

Bruttomesso, R., Ghilardi, S. and Ranise, S. [2011]. Rewriting-based quantifier-free interpolation for a theory of arrays, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, Vol. 10 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 171–186.

Bruttomesso, R., Ghilardi, S. and Ranise, S. [2012]. Quantifier-free interpolation of a theory of arrays, *Logical Methods in Computer Science* **8**(2).

Bruttomesso, R., Pek, E., Sharygina, N. and Tsitovich, A. [2010]. The opensmt solver, *in* J. Esparza and R. Majumdar (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 150–153.

Bruttomesso, R., Rollini, S., Sharygina, N. and Tsitovich, A. [2010]. Flexible interpolation with local proof transformations, *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '10, IEEE Press, Piscataway, NJ, USA, pp. 770–777.

Cabodi, G., Loiacono, C. and Vendraminetto, D. [2015a]. Optimization techniques for craig interpolant compaction in unbounded model checking, *Formal Methods in System Design* **46**(2): 135–162.

Cabodi, G., Loiacono, C. and Vendraminetto, D. [2015b]. Optimization techniques for craig interpolant compaction in unbounded model checking, *Formal Methods in System Design* **46**(2): 135–162.

Cabodi, G., Murciano, M., Nocco, S. and Quer, S. [2006]. Stepping forward with interpolants in unbounded model checking, *Proceedings of the 2006*

IEEE/ACM International Conference on Computer-aided Design, ICCAD '06, ACM, New York, NY, USA, pp. 772–778.

Cabodi, G., Palena, M. and Pasini, P. [2014]. Interpolation with guided refinement: Revisiting incrementality in sat-based unbounded model checking, *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, FMCAD Inc, Austin, TX, pp. 12:43–12:50.

Chen, P. and Keutzer, K. [1999]. Towards true crosstalk noise analysis, *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '99, IEEE Press, Piscataway, NJ, USA, pp. 132–138.

Chockler, H., Ivrii, A. and Matsliah, A. [2013]. Computing interpolants without proofs, *in* A. Biere, A. Nahir and T. Vos (eds), *Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 72–85.

Cimatti, A. and Griggio, A. [2012]. Software model checking via ic3, *in* P. Madhusudan and S. A. Seshia (eds), *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 277–293.

Cimatti, A., Griggio, A., Schaafsma, B. J. and Sebastiani, R. [2013a]. The mathsat5 smt solver, *in* N. Piterman and S. A. Smolka (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 93–107.

Cimatti, A., Griggio, A., Schaafsma, B. and Sebastiani, R. [2013b]. The MathSAT5 SMT Solver, *in* N. Piterman and S. Smolka (eds), *Proceedings of TACAS*, Vol. 7795 of *LNCS*, Springer.

Cimatti, A., Griggio, A. and Sebastiani, R. [2008]. Efficient interpolant generation in satisfiability modulo theories, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 397–412.

Clarisó, R. and Cortadella, J. [2004]. The octahedron abstract domain, *in* R. Giacobazzi (ed.), *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 312–327.

Clarke, E., Kroening, D. and Lerda, F. [2004]. A tool for checking ansi-c programs, *in* K. Jensen and A. Podelski (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 168–176.

Clarke, E. M. and Emerson, E. A. [1982]. Design and synthesis of synchronization skeletons using branching time temporal logic, *in* D. Kozen (ed.), *Logics of Programs: Workshop, Yorktown Heights, New York, May 1981*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–71.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. [2000]. Counterexample-guided abstraction refinement, *in* E. A. Emerson and A. P. Sistla (eds), *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, Vol. 1855 of *LNCS*, Springer, pp. 154–169.

Cotton, S. [2010]. Two techniques for minimizing resolution proofs, *in* O. Strichman and S. Szeider (eds), *Theory and Applications of Satisfiability Testing – SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 306–312.

Cousot, P. and Cousot, R. [1977]. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, ACM, New York, NY, USA, pp. 238–252.

Craig, W. [1957]. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory, *The Journal of Symbolic Logic* **22**(3): 269–285.

de Moura, L. and Bjørner, N. [2008]. Z3: An efficient smt solver, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008,*

Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

D'Silva, V. [2010]. Propositional interpolation and abstract interpretation, *in* A. D. Gordon (ed.), *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 185–204.

D'Silva, V., Kroening, D., Purandare, M. and Weissenbacher, G. [2010]. Interpolant strength, *in* G. Barthe and M. Hermenegildo (eds), *Verification, Model Checking, and Abstract Interpretation: 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 129–145.

Dutertre, B. [2014]. Yices 2.2, *in* A. Biere and R. Bloem (eds), *Computer-Aided Verification (CAV'2014)*, Vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.

Dutertre, B. and de Moura, L. [2006]. A fast linear-arithmetic solver for dpll(t), *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 81–94.

Fedyukovich, G., Sery, O. and Sharygina, N. [2013]. evolcheck: Incremental upgrade checker for c, *in* N. Piterman and S. A. Smolka (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 292–307.

Fontaine, P., Merz, S. and Woltzenlogel Paleo, B. [2011a]. Compression of propositional resolution proofs via partial regularization, *in* N. Bjørner and V. Sofronie-Stokkermans (eds), *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 237–251.

Fontaine, P., Merz, S. and Woltzenlogel Paleo, B. [2011b]. Compression of propositional resolution proofs via partial regularization, *in* N. Bjørner and V. Sofronie-Stokkermans (eds), *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 237–251.

Fuchs, A., Goel, A., Grundy, J., Krstić, S. and Tinelli, C. [2009]. Ground interpolation for the theory of equality, *in* S. Kowalewski and A. Philippou (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 413–427.

Gao, S. and Zufferey, D. [2016]. Interpolants in nonlinear theories over the reals, *in* M. Chechik and J.-F. Raskin (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 625–641.

Godlin, B. and Strichman, O. [2013]. Regression verification: proving the equivalence of similar programs, *Softw. Test., Verif. Reliab.* **23**(3): 241–258.

Goel, A., Krstić, S. and Tinelli, C. [2009]. Ground interpolation for combined theories, *in* R. A. Schmidt (ed.), *Automated Deduction – CADE-22: 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 183–198.

Graf, S. and Saïdi, H. [1997]. Construction of abstract state graphs with PVS, *in* O. Grumberg (ed.), *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, Vol. 1254 of *LNCS*, Springer, pp. 72–83.

Griggio, A., Le, T. T. H. and Sebastiani, R. [2011]. Efficient interpolant generation in satisfiability modulo linear integer arithmetic, *in* P. A. Abdulla and K. R. M. Leino (eds), *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software,*

ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 143–157.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. [2015]. The seahorn verification framework, *in* D. Kroening and C. S. Păsăreanu (eds), *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, Springer International Publishing, Cham, pp. 343–361.

Halbwachs, N. and Péron, M. [2008]. Discovering properties about arrays in simple programs, *in* R. Gupta and S. P. Amarasinghe (eds), *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, ACM, pp. 339–348.

Henzinger, T. A., Jhala, R., Majumdar, R. and McMillan, K. L. [2004]. Abstractions from proofs, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, ACM, New York, NY, USA, pp. 232–244.

Henzinger, T. A., Jhala, R., Majumdar, R. and Sutre, G. [2002]. Lazy abstraction, *in* J. Launchbury and J. C. Mitchell (eds), *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, ACM, pp. 58–70.

Hyvärinen, A. E. J., Alt, L. and Sharygina, N. [2015]. Flexible interpolation for efficient model checking, *Mathematical and Engineering Methods in Computer Science - 10th International Doctoral Workshop, MEMICS 2015, Telč, Czech Republic, October 23-25, 2015, Revised Selected Papers*, pp. 11–22.

Hyvärinen, A. E. J., Marescotti, M., Alt, L. and Sharygina, N. [2016]. Opensmt2: An SMT solver for multi-core and cloud computing, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pp. 547–553.

Jain, H., Clarke, E. and Grumberg, O. [2008]. Efficient craig interpolation for linear diophantine (dis)equations and linear modular equations, *in* A. Gupta and S. Malik (eds), *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 254–267.

Jain, H., Clarke, E. M. and Grumberg, O. [2009]. Efficient craig interpolation for linear diophantine (dis)equations and linear modular equations, *Formal Methods in System Design* **35**(1): 6–39.

Jancik, P., Kofroň, J., Rollini, S. F. and Sharygina, N. [2014]. On interpolants and variable assignments, *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, FMCAD Inc, Austin, TX, pp. 22:123–22:130.

Jhala, R. and McMillan, K. L. [2005]. Interpolant-based transition relation approximation, *in* K. Etessami and S. K. Rajamani (eds), *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 39–51.

Junttila, T. and Dubrovin, J. [2008]. Encoding queues in satisfiability modulo theories based bounded model checking, *in* I. Cervesato, H. Veith and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning: 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–304.

Kapur, D., Majumdar, R. and Zarba, C. G. [2006]. Interpolation for data structures, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, pp. 105–116.

Kovács, L., Rollini, S. F. and Sharygina, N. [2013]. A parametric interpolation framework for first-order theories, *in* F. Castro, A. Gelbukh and M. González (eds), *Advances in Artificial Intelligence and Its Applications: 12th Mexican International Conference on Artificial Intelligence, MICAI 2013, Mexico City, Mexico, November 24-30, 2013, Proceedings, Part I*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 24–40.

Krajícek, J. [1997]. Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic, *J. Symb. Log.* **62**(2): 457–486.

Kroening, D. and Strichman, O. [2008]. *Linear Arithmetic*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 111–147.

McMillan, K. L. [2003]. Interpolation and sat-based model checking, *in* W. A. Hunt and F. Somenzi (eds), *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–13.

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

McMillan, K. L. [2006]. Lazy abstraction with interpolants, *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 123–136.

McMillan, K. L. [2010]. Lazy annotation for program testing and verification, *in* T. Touili, B. Cook and P. Jackson (eds), *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 104–118.

McMillan, K. L. [2014]. Lazy annotation revisited, *in* A. Biere and R. Bloem (eds), *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, Springer International Publishing, Cham, pp. 243–259.

Nelson, G. and Oppen, D. C. [1980]. Fast decision procedures based on congruence closure, *J. ACM* **27**(2): 356–364.

Nieuwenhuis, R. and Oliveras, A. [2005]. Proof-producing congruence closure, *in* J. Giesl (ed.), *Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 453–468.

Pudlák, P. [1997]. Lower bounds for resolution and cutting plane proofs and monotone computations, *Journal of Symbolic Logic* **62**(3): 981–998.

Pugh, W. [1991]. The omega test: A fast and practical integer programming algorithm for dependence analysis, *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, ACM, New York, NY, USA, pp. 4–13.

Queille, J. P. and Sifakis, J. [1982]. Specification and verification of concurrent systems in cesar, *in* M. Dezani-Ciancaglini and U. Montanari (eds), *International Symposium on Programming: 5th Colloquium Turin, April 6–8, 1982 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–351.

Ranise, S. and Déharbe, D. [2003]. Applying light-weight theorem proving to debugging and verifying pointer programs, *Electronic Notes in Theoretical Computer Science* **86**(1): 105 – 119.

Rintanen, J., Heljanko, K. and Niemelä, I. [2006]. Planning as satisfiability: parallel plans and algorithms for plan search, *Artificial Intelligence* **170**(12): 1031 – 1080.

Rollini, S. F., Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2013]. Periplo: A framework for producing effective interpolants in sat-based software verification, *in* K. McMillan, A. Middeldorp and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 683–693.

Rollini, S. F., Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2015]. *Optimizing Function Summaries Through Interpolation*, Springer International Publishing, Cham, pp. 73–82.

Rollini, S. F., Bruttomesso, R. and Sharygina, N. [2011]. An efficient and flexible approach to resolution proof reduction, *in* S. Barner, I. Harris, D. Kroening and O. Raz (eds), *Hardware and Software: Verification and Testing: 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 182–196.

Rollini, S. F., Bruttomesso, R., Sharygina, N. and Tsitovich, A. [2014]. Resolution proof transformation for compression and interpolation, *Formal Methods in System Design* **45**(1): 1–41.

Rollini, S. F., Sery, O. and Sharygina, N. [2012]. Leveraging interpolant strength in model checking, *in* P. Madhusudan and S. A. Seshia (eds), *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 193–209.

Rümmer, P. and Subotić, P. [2013]. Exploring interpolants, *Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design*, FMCAD '13, IEEE, pp. 69–76.

Rybalchenko, A. and Sofronie-Stokkermans, V. [2007]. Constraint solving for interpolation, *in* B. Cook and A. Podelski (eds), *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 346–362.

Scholl, C., Pigorsch, F., Disch, S. and Althaus, E. [2014]. Simple interpolants for linear arithmetic, *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 115:1–115:6.

Sery, O., Fedyukovich, G. and Sharygina, N. [2012a]. Funfrog: Bounded model checking with interpolation-based function summarization, *in* S. Chakraborty and M. Mukund (eds), *Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 203–207.

Sery, O., Fedyukovich, G. and Sharygina, N. [2012b]. Incremental upgrade checking by means of interpolation-based function summaries, *12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)*, IEEE, pp. 114 – 121.

Sery, O., Fedyukovich, G. and Sharygina, N. [2012c]. Interpolation-based function summaries in bounded model checking, *in* K. Eder, J. Lourenço and O. Shehory (eds), *Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 160–175.

Sheeran, M., Singh, S. and Stålmarck, G. [2000]. Checking safety properties using induction and a sat-solver, *in* W. A. Hunt and S. D. Johnson (eds), *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000 Austin, TX, USA, November 1–3, 2000 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 127–144.

Stump, A., Barrett, C. W., Dill, D. L. and Levitt, J. [2001]. A decision procedure for an extensional theory of arrays, *Proceedings of the 16th Annual*

IEEE Symposium on Logic in Computer Science, LICS '01, IEEE Computer Society, Washington, DC, USA, pp. 29–.

Totla, N. and Wies, T. [2016]. Complete instantiation-based interpolation, *Journal of Automated Reasoning* **57**(1): 37–65.

Vizel, Y., Ryvchin, V. and Nadel, A. [2013]. Efficient generation of small interpolants in cnf, *in* N. Sharygina and H. Veith (eds), *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 330–346.

Weissenbacher, G. [2012]. Interpolant strength revisited, *in* A. Cimatti and R. Sebastiani (eds), *Theory and Applications of Satisfiability Testing – SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 312–326.

Yorsh, G. and Musuvathi, M. [2005a]. A combination method for generating interpolants, *in* R. Nieuwenhuis (ed.), *Automated Deduction – CADE-20: 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 353–368.

Yorsh, G. and Musuvathi, M. [2005b]. A combination method for generating interpolants, *in* R. Nieuwenhuis (ed.), *Automated Deduction – CADE-20: 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 353–368.