Craig Interpolation and Proof Manipulation

Theory and Applications to Model Checking

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
Simone Fulvio Rollini

under the supervision of
Prof. Natasha Sharygina

February 2014

Dissertation Committee

Prof. Rolf Krause	Università della Svizzera Italiana, Switzerland
Prof. Evanthia Papadopoulou	Università della Svizzera Italiana, Switzerland
Prof. Roberto Sebastiani	Università di Trento, Italy
Prof. Ofer Strichman	Technion, Israel

Dissertation accepted on 06 February 2014

Research Advisor	PhD Program Director	PhD Program Director
Prof. Natasha Sharygina	**Prof. Igor Pivkin**	**Prof. Stefan Wolf**

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.

Simone Fulvio Rollini
Lugano, 06 February 2014

Abstract

Model checking is one of the most appreciated methods for automated formal verification of software and hardware systems. The main challenge in model checking, i.e. scalability to complex systems of extremely large size, has been successfully addressed by means of symbolic techniques, which rely on an efficient representation and manipulation of the systems based on first order logic.

Symbolic model checking has been considerably enhanced with the introduction in the last years of Craig interpolation as a means of overapproximation. Interpolants can be efficiently computed from proofs of unsatisfiability based on their structure. A number of algorithms are available to generate different interpolants from the same proof; additional interpolants can be obtained by turning a proof into another proof of unsatisfiability of the same formula using transformation techniques.

Interpolants are thus not unique, and it is fundamental to understand which interpolants have the highest quality, resulting in an optimal verification performance; it is an open problem to determine what features determine the quality of interpolants, and how they are connected with the effectiveness in verification.

The goal of this thesis is to identify aspects that make interpolants good, and to develop new techniques that guide the generation of interpolants in order to improve the performance of model checking approaches.

We contribute to the state-of-the-art by providing a characterization of quality in terms of semantic and syntactic features, such as logical strength, size, presence of quantifiers. We present a family of algorithms that generalize existing techniques and are able to produce interpolants of different structure and strength, with and without quantifiers, from the same proof. These algorithms are examined in the context of various model checking applications, where collections of interdependent interpolants satisfying particular properties are needed; we formally analyze the relationships among the properties and derive necessary and sufficient conditions on the applicability of the algorithms.

We introduce a framework for proof manipulation; it includes a method to overcome limitations in existing interpolation algorithms for first order theories, as well as a set of compression algorithms, which, reducing the size of proofs, consequently

reduce the size of interpolants generated from them.

Finally, we provide experimental evidence that size and logical strength have a significant impact on verification: small interpolants improve the verification performance, while stronger or weaker interpolants are beneficial in different model checking applications.

Interpolants are generated from proofs, as produced by logic solvers. As a secondary line of research, we developed and successfully tested a hybrid propositional satisfiability solver built on a model-based stochastic technique known as cross-entropy method.

Contents

Chapter 1

Introduction

In the last three decades there has been a considerable investment in the development of techniques and tools aimed at making the formal verification of hardware and software fully automated. Among various approaches, *model checking* is highly appreciated for its exhaustiveness and degree of automation: once a model (of hardware or software) and a specification of a property (desired behavior) are given, a tool can be run to explore all possible behaviors of the model, determining whether the property holds.

The main difficulty in model checking is that for complex systems the size of the model often becomes too large to be handled in a reasonable amount of time. A remarkable improvement can be obtained by means of *symbolic techniques*: the model is encoded together with the negation of the property in a logic formula, satisfiable if and only if the specification is not met by the model. If this is the case, a logic solver can produce a satisfying assignment, representing a violating behavior; otherwise, a *proof of unsatisfiability* can be built as a witness of the validity of the property in the model.

Symbolic model checking has been considerably enhanced with the introduction of *Craig interpolation*, as a means to perform a guided overapproximation of sets of states of the model. Interpolants can be efficiently computed from proofs of unsatisfiability; various algorithms are available to generate different interpolants from the same proof. Moreover, a given proof can be transformed into another proof of unsatisfiability of the same formula by means of *proof manipulation* techniques, consequently allowing additional interpolants.

Interpolants are thus not unique, and it is fundamental to understand which interpolants result in an optimal verification performance: it is an open problem to determine what features determine the quality of interpolants, and how they are connected with the effectiveness in the verification.

This thesis focuses on the notion of *quality of interpolants*: the goal is to identify aspects that make interpolants good, and to develop new techniques that guide the generation of interpolants in order to improve the performance of model checking approaches.

We contribute to the state-of-the-art in a number of ways. First, we provide a characterization of quality from both a semantic and a syntactic point of view, taking into account features as logical strength, size of formulae, and presence of quantifiers.

Second, we present a family of algorithms that generalize existing techniques and are able to produce interpolants of different structure and strength, with and without quantifiers, from the same proof; they apply to arbitrary inference systems, in the context of both propositional logic and first order theories.

Besides dealing with the generation of individual interpolants, we also consider multiple interdependent interpolants; several well-known model checking approaches rely in fact on collections of interpolants, which have to satisfy particular properties. We formally analyze the relationships among the properties and outline a hierarchy; we derive necessary and sufficient conditions on the applicability of interpolation algorithms for propositional logic and first order theories, so that the properties are satisfied, making verification possible.

We introduce a framework for proof manipulation; it includes a method to overcome limitations in existing interpolation algorithms for first order theories, as well as a set of compression algorithms, which, reducing the size of proofs, consequently reduce the size of interpolants generated from them.

Our theoretical contributions are matched by experimental evidence that size and logical strength have a significant impact on verification: small interpolants improve the verification performance, while stronger or weaker interpolants are beneficial in different model checking applications. This experimentation could be carried out thanks to the development of a new tool, PeRIPLO, that offers routines for interpolation and proof manipulation with applications to model checking.

Interpolants are generated from proofs, which in turn are produced by logic solvers. As a secondary line of research, we have investigated deterministic and stochastic algorithms for propositional satisfiability, creating and successfully putting to the test a hybrid solver built on a model-based stochastic technique known as cross-entropy method.

In the rest of the chapter we discuss in detail each of the contributions mentioned above, accompanying them with the relevant background while outlining the structure of the thesis.

1.1 Symbolic Model Checking

Model checking [CE81, QS82] is one of the most successful techniques[1] effectively used at an industrial scale for the formal verification of hardware designs and software components. The fundamental idea is to analyze a system by means of its description in terms of *states*, a snapshot of the system at a particular instant during its execution, *transitions*, describing how the system states evolve over time, and a *property* to be verified.

A system can be explicitly represented by means of a directed graph, where nodes correspond to states and two nodes are connected if there is a transition from the state represented by the first node to that represented by the second node. In this case, verification of a property with respect to some initial configuration of the system reduces (in simple cases) to *reachability analysis*, i.e. checking whether there exists a path in the graph connecting an initial state with a state where the property does not hold.

Given the complexity of systems nowadays in use, the number of states to be traversed (or even represented) during a model checking run may easily go beyond the resources of the available computers; this problem is known as *state space explosion*. An important turning point in this direction was set with the introduction of *symbolic methods* [BCM+92, McM92] which enabled verification of models of previously unmanageable sizes.

In symbolic model checking states and transitions are encoded into some decidable fragment of first order logic; the effort of exploring the state space eventually depends on the efficiency of the encoding and of the solving procedures for formulae in that fragment. State formulae F are defined over a first order signature Σ, and represent the set of states corresponding to their models; transition formulae are defined over $\Sigma \cup \Sigma'$, where Σ and Σ' are associated with current and next state: their models represent pairs of states such that the first can reach the second via a transition. The union of all the transitions defines the transition relation T. A transition system (S, T) is made by a state formula S, characterizing the initial states of the system, and a transition relation T. The *image* F' of a state formula F w.r.t. T is a state formula representing the set of states reachable from the states of F in one step (i.e., applying T once). A state formula F_2 is said to be reachable from a state formula F_1 if it is possible to reach some of the states of F_2 starting from the states of F_1, computing the image a finite number of times.

A property to be verified is usually encoded as a state formula P: if $\neg P$ is not reachable from S, then no state in the system actually violates the property. A simple

[1] The inventors of model checking E.M.Clarke, E.A.Emerson and J.Sifakis have been the recipients of the ACM Turing Award in 2007.

way of verifying a property of this kind is to perform *symbolic reachability analysis*: starting from S, T is iteratively applied until either a state formula is found inconsistent with P, or all the system states have been explored. The original approach of [BCM+92, McM92] is based on *binary decision diagrams* [Bry86] (BDDs) to store states as propositional formulae and to compute the image of sets of states; the main drawback is that exact image computation is computationally expensive, since it involves the use of quantifier elimination procedures.

Bounded model checking [BCC+99, BCC+03] (BMC) introduces a method which relies on algorithms for the propositional satisfiability decision problem (known as SAT solvers) to check properties. The idea is to fix a bound k and to construct a propositional formula of the form $S \wedge T^0 \wedge \cdots \wedge T^{k-1} \wedge (\neg P^0 \vee \cdots \vee \neg P^k)$, that is satisfiable if and only if the property P is violated within k steps. k is iteratively increased until either a bug is found or an upper bound on the diameter of (the graph representation of) the system is reached. BMC is a well established technique, successfully employed in several contexts (e.g. [ACKS02, AMP09, JD08]), and it depends on efficient and robust tools (see next section), whose performance has been continuously improving along the years.

The use of *abstraction* [CC77] is another important paradigm used for mitigating the state explosion problem. The idea is to group several states into one abstract state, and then to model check the set of abstract states and the resulting abstract transition relation, which is less expensive than checking the original system. A successful technique in this direction is predicate abstraction [GS97], where states are partitioned with respect to a set of predicates. The use of abstraction may introduce *spurious* behaviors in the system, which have to be removed by means of a *refinement* phase, where new predicates are synthesized. The information provided by the spurious behavior can actually be used to perform the refinement, following an iterative abstract-check-refine approach; this idea is at the base of the well-known *counterexample-guided abstraction refinement* framework (CEGAR [CGJ+00]). The *lazy abstraction* approach, introduced in [HJMS02], improves on [CGJ+00] by allowing abstraction of the model on demand, so that different parts may show different degrees of precision, depending on the property to be verified.

1.2 SAT and SMT Solving

Satisfiability (or SAT), one of the central decision problems of informatics, consists in determining whether for a given propositional formula there exists an assignment of truth values to its variables such that the formula evaluates to true. Solving pro-

cedures for the satisfiability problem are known as *SAT solvers*.

SAT solvers have undergone a constant improvement in the last decade, and have become established tools across a variety of fields, software and hardware verification, planning, and scheduling [BCC+99, BCCZ99, BLM01, RN10, GSMT98]. However, reasoning using pure propositional logic is often not enough; in fact, applications in the aforementioned areas may require checking satisfiability in more expressive logics such as *first order theories*, where the semantics of some functions and predicates symbols is fixed a priori. This is needed, for example, when dealing with linear arithmetic, arrays, bit-vectors, pointers.

A successful solution consists in adopting reasoning methods which are specific to the theory under account, which give rise to decision procedures for the satisfiability of quantifier-free formulae in that theory. The work of Nelson and Oppen [NO79, NO80, Opp80] and Shostak [Sho84] in this direction has had a fundamental impact on verification, contributing to the realization of efficient *theory solvers* to check consistency of formulae in first order theories and combinations of them [KS08].

In the last decade new techniques for integrating the propositional reasoning of SAT solvers with theory-specific decision procedures have been applied in different communities and domains, from resource planning and temporal reasoning to formal verification of several kinds of designs, including real time and hybrid systems [PICW04, RD03, BFG+05, ACKS02].

The problem of deciding the satisfiability of a first order formula with respect to some decidable first order theory has become known as *satisfiability modulo theories (SMT)*, whereas the various solving procedures are called *SMT solvers* in analogy with SAT solvers. Most of these procedures are built on top of SAT techniques based on variants of *DPLL* (Davis-Putnam-Loveland-Logemann), a deterministic backtracking algorithm for deciding the satisfiability of propositional formulae [DLL62, DP60]. In particular, the *lazy* approach to SMT [Seb07], adopted by the majority of the state-of-the-art tools, is based on the integration of a SAT solver and one or more theory solvers, respectively handling the propositional and the theory-specific components of reasoning.

1.2.1 Stochastic Techniques for Satisfiability

In the SAT framework several algorithms for satisfiability have been proposed, both incomplete (unable to prove unsatisfiability of a formula) and complete. Most of the state-of-the-art algorithms fall into two main classes: the ones (complete) which implement the deterministic DPLL algorithm [DP60, DLL62]; the others (incomplete) based on stochastic local search (SLS) [HS05]. The two classes demonstrate bet-

ter performance on different kinds of problems, and can be considered in this sense complementary. There have been several works in the last years aimed at combining DPLL and SLS in an effective manner, trying to exploit the benefits of both to obtain a competitive complete hybrid solver [SKM97].

These kinds of approaches have proved quite successful, but open problems still exist in terms of developing effective combinations of the particular deterministic and stochastic techniques, integrating global and local strategies of exploration.

Research Contribution: A DPLL-Stochastic Algorithm for Satisfiability. We address these challenges by contributing to the development of a new complete DPLL-stochastic algorithm. We provide a solution to the satisfiability problem built on a model-based stochastic technique known as *cross-entropy method*, an approach to rare event simulation and to combinatorial optimization [RK04].

In our solution, for a given SAT instance, a cross-entropy based algorithm is used to search for a satisfying assignment by applying an iterative procedure that optimizes an objective function correlated with the likelihood of satisfaction. The algorithm is intended as a preprocessing step to SAT solving, where cross-entropy identifies the areas of the space of all possible assignments that are more likely to contain a satisfying assignment; this information is in turn given to a DPLL SAT solver to suggest variables assignments during the search. We implemented the new algorithm in a tool named CROiSSANT and tested it on different sets of benchmarks; experimental results show that the approach is effective in guiding the search at solving time.

The results of our work have been published in [CIM+13] and are discussed in §2.2.

1.3 Interpolation-Based Model Checking

Craig's interpolation theorem [Cra57a, Cra57b] is a classical result in first order logic, and it has had a strong impact on the computer science community thanks to its application to areas such as hardware/software specification [DGS93], knowledge bases [AM05] and combination of theory solvers [NO79, GNZ05].

Formally, given an unsatisfiable conjunction of formulae $A \wedge B$, a Craig interpolant I is a formula that is implied by A, is unsatisfiable in conjunction with B, and is defined on the common language of A and B. An interpolant can be seen as an overapproximation of A that is still unsatisfiable with B.

In the ambit of model checking, interpolation has been gaining considerable attention since the work carried out by McMillan in [McM03, McM04a], where he

showed that it can be applied as an effective overapproximation technique. Interpolants are, for example, used with bounded model checking in [McM03]. In [JM05] interpolants are a means to approximate the transition relation of a system. Craig interpolation is employed for synthesizing predicates in [HJMM04] and [JM06]. In [McM06], interpolation is applied within the lazy abstraction framework of [HJMS02]. Interpolants can also be employed as summaries that abstract the behavior of function calls, as shown in [SFS11] and [SFS12b].

1.3.1 Interpolants Quality: Strength and Structure

Several state-of-the art approaches exist to obtain interpolants in an automated manner. The most common techniques generate an interpolant for $A \wedge B$ from a proof of unsatisfiability of the conjunction; benefits are that the generation has a complexity linear in the proof size, and interpolants contain only information relevant to the unsatisfiability of $A \wedge B$.

The concrete algorithms (or *interpolation systems*) for obtaining an interpolant depend on the language on which A and B are defined. Interpolation in propositional logic is addressed in [McM03, Pud97, Kra97, Hua95], later generalized by [DKPW10] and in turn by [Wei12]. A number of works discuss interpolation in first order theories, proposing systems able to generate interpolants for conjunctions of constraints [FGG+09, CGS08, BCF+07, GD07, BCF+09, Pug91, BKRW11, JCG08, KMZ06] or for formulae with arbitrary propositional structure [McM04a, YM05, CGS10], both in individual theories and combinations. The interpolation systems of [KV09, HKV12] can be used with arbitrary first order inference systems, but are restricted to a class of proofs with specific structural properties [KV09, JM06].

Interpolation has become a well-established technique in the area of model checking, and multiple interpolants can be generated from the same proofs; yet, it is not clear what characteristics make some interpolants better than others in the particular applications. One promising feature which has been examined in the literature is *logical strength* [JM05, DKPW10] (a formula F_1 is stronger than F_2 if $F_1 \models F_2$): since interpolants are inherently overapproximations, a stronger or weaker interpolant is expected to influence the speed of convergence of model checking algorithms, driving the verification process as a finer or coarser approximation. Besides semantic features, *structural characteristics* are also likely to affect the verification performance. Interpolants can be for example quantified or quantifier-free, the latter preferred for their applicability to model checking; interpolants can have a more or less rich content in terms of predicates; interpolants can correspond to smaller or larger formulae. Evidence in favor of generating and using small interpolants is in

particular provided by [CLV13] in the context of hardware model checking.

Research Contribution: Systematic Generation of Interpolants of Different Strength and Structure. We contribute to a theoretical formalization of a generic interpolation approach, by characterizing the notion of interpolants quality in terms of structure and logical strength. We identify a class of recursive interpolation procedures, and, based on that, we introduce a new parametric interpolation framework for arbitrary theories and inference systems, which is able to compute interpolants of different structure and strength, with or without quantifiers, from the same proof. The framework subsumes as special cases the interpolation systems of [KV09, Wei12], and consequently those of [McM03, Pud97, Kra97, Hua95, DKPW10].

The results of our work have been published in [KRS13] and are discussed in §3.5.

Research Contribution: Impact of Interpolants Size and Strength in Model Checking. We put into practice our theoretical framework to address the key problem of generating effective interpolants in verification, by examining the impact of size and logical strength in the context of software SAT-based bounded model checking. Two case studies are taken into account, namely two bounded model checking applications which use interpolation to generate function summaries: (i) verification of a C program incrementally with respect to a number of different properties [SFS11], and (ii) incremental verification of different versions of a C program with respect to a fixed set of properties [SFS12b]. The PeRIPLO framework (one of the contributions of this thesis, see §1.4) is integrated with the model checkers FunFrog [SFS12a] and eVolCheck [FSS13], which respectively implement (i) and (ii), to drive interpolation by manipulating the the proofs from which the interpolants are computed and by generating interpolants of different strength.

We provide solid experimental evidence that compact interpolants improve the verification performance in the two applications. We also carry out a first systematic evaluation of the impact of strength in a specific verification domain, showing that different applications can benefit from interpolants of different strength: specifically, stronger and weaker interpolants are respectively desirable in (i) and (ii).

The results of our work have been published in [RAF+13] and are discussed in §3.6.

1.3.2 Interpolation Properties in Model Checking

In many verification tasks, a single interpolant, i.e., a single subdivision of constraints into two groups A and B, is not sufficient. For example, in the context of

counterexample-guided abstraction refinement [CGJ+00], a spurious behavior gives rise to a *sequence* of interpolants, which is then used to refine the abstraction. Properties such as path interpolation, simultaneous abstraction, tree interpolation are needed in common model checking approaches, among which predicate abstraction [JM06], lazy abstraction with interpolants [McM06], interpolation-based function summarization [SFS11, SFS12b].

These properties are not satisfied by arbitrary collections of interpolants and must be established for each verification technique. On one side, it is important to understand whether and how the properties are related to each other; on the other side, in order to make verification possible, it is necessary to identify the constraints that the satisfaction of a certain property puts on the applicability of the interpolation systems.

Research Contribution: Formal Analysis of Interpolation Properties. We identify and uniformly present the most common properties imposed on interpolation by existing verification approaches; we systematically analyze the relationships among the properties and show that they form a hierarchy. In doing so, we generalize the traditional setting consisting of a single interpolation system to allow for families of interpolation systems, thus giving the flexibility to choose different systems for computing different interpolants in a collection. The generality of these results stems from the fact that they are independent of the particular system adopted to generate the interpolants.

The results of our work have been published in [GRS13] and are discussed in Chapter 4.

Research Contribution: Interpolation Properties and Interpolant Strength. Besides studying interpolation properties from a higher perspective, we examine the relationships among them in concrete interpolation systems, building on the work of [DKPW10]. The propositional *labeled interpolation systems* of [DKPW10] are a suitable choice for an investigation of interpolant strength, since they consist of a parametric framework which allows to systematically generate interpolants of different strength from the same proof.

We formally prove both sufficient and necessary conditions for a family of labeled interpolation systems and for a single system to enjoy each interpolation property. These results allow for the systematic study of how interpolants strength affects state-of-the-art model checking techniques, while preserving their soundness.

The results of our work have been published in [RSS12, GRS13] and are discussed in Chapter 4.

Research Contribution: Extension of the Labeled Interpolation Systems to SMT. In order to extend the applicability of our results from SAT- to SMT-based verification, we investigate interpolation properties and interpolant strength in the context of satisfiability modulo theories; we present the *theory labeled interpolation systems*, an extension of the labeled interpolation systems of [DKPW10] to SMT that generalizes the interpolation frameworks of [YM05, CGS10], and examine how the constraints for some of the interpolation properties change in presence of a first order theory.

The results of our work are discussed in §3.4 and §4.4.

1.4 Resolution Proof Manipulation

In many verification environments it is not enough to find a positive or negative answer to the satisfiability problem, on the contrary it is important to obtain a certificate either as a satisfying model or as a proof of unsatisfiability that shows the nonexistence of such a model.

Resolution proofs of unsatisfiability, in particular, find application in very diverse contexts: proof-carrying code [Nec97], industrial product configuration or declarative modeling [SKK03, SSJ+03], theorem proving [Amj08, WA09, FMM+06], interpolation-based model checking [McM03, McM04b, McM04a, HJMM04]. It is thus essential to develop techniques that allow to handle proofs efficiently, so as to positively impact on the performance of the frameworks that rely on them. To this end, we focus on two main aspects: proof compression, and proof rewriting to enable the computation of interpolants.

1.4.1 Proof Compression

The size of the proofs typically affects the efficiency of the methods in which they are used. It is known that the size of resolution proofs can grow exponentially with respect to the input formula; even when they have in practice a manageable size, it might be crucial for efficiency to compress them as much as possible. Several compression technique can be found in the literature, ranging from memoization of common subproofs to partial regularization [Amj07, Amj08, Sin07, Cot10, BIFH+08, DKPW10, FMP11]; however, since the problem of finding a minimum proof is NP-hard, it is still an open challenge to design heuristics capable of obtaining good reduction in concrete situations.

Research Contribution: Proof Compression Framework. We give our contribution to the area of proof compression by developing an approach based on local transformation rules. It is a purely post-processing technique, with the benefit of being independent from the way the resolution proof is produced. It consists in identifying a particular kind of syntactical redundancies and in applying the rules in order to remove them, thus reducing the size of the proofs. The algorithm has the advantage of being easily customizable, since it is parametric in the way rules are locally applied and in the amount of transformation to be performed. The compression framework has been implemented in the tools OpenSMT and PeRIPLO (see below); experimental results on SAT and SMT benchmarks show that the framework is effective in reducing proofs, especially in combination with existing compression strategies.

The results of our work have been published in [RBS10, RBST] and are discussed in §5.3.

1.4.2 Enabling Interpolation in SMT

Whenever an unsatisfiable conjunction $A \wedge B$ is defined on a more expressive language than propositional logic, for a number of first order theories a standard approach is to resort to an SMT solver to compute interpolants. In this case the resolution proof of unsatisfiability contains original information from $A \wedge B$ plus theory lemmata, produced by the theory solver of the SMT solver, representing tautologies in the theory. Given this proof, an interpolant can be extracted using the methods described in [YM05, CGS10] (as well as our own generalization, presented in §3.4), provided that the theory solver is able to compute an interpolant from a conjunction of constraints in the theory.

The frameworks of [YM05, CGS10] have a significant limitation, since they need to assume the absence of mixed predicates (predicates defined over symbols local to A and B) in the proof; there is in fact a number of state-of-the-art techniques extensively used in SMT solvers (e.g., theory combination) that might require the addition of mixed predicates to the input formula before or during solving time [Ack54, BNOT06, BBC+05b, dMR02].

Research Contribution: Proof Manipulation to Enable Interpolation in SMT. The solution we provide is to rewrite the proof of unsatisfiability in such a way that mixed predicates are "absorbed" within existing theory lemmata. The benefit is the possibility of applying off-the-shelf techniques for solving $A \wedge B$, and, consequently, after the proof is transformed, the methods of [YM05, CGS10] for computing interpolants: in this way, interpolation becomes flexible and modular, being

decoupled into (i) generation of partial interpolants for theory lemmata by means of ad-hoc techniques and (ii) computation of the overall interpolant by means of a standard algorithm for propositional resolution proofs (e.g., [Pud97]). The approach is demonstrated to be theoretically sound and the experiments carried out within the OpenSMT framework show that it is also practically efficient.

The results of our work have been published in [BRST10, RBST] and are discussed in §5.4.

Research Contribution: Tools for Proof Manipulation And Interpolation. In the last years, the Formal Verification Group at USI has been actively developing the open-source tool OpenSMT [BPST10]; it is an SMT solver which supports reasoning in several theories and, given an input formula, is able to return either a model or a proof of unsatisfiability. As a first tool contribution we have realized within OpenSMT a framework for proof compression and interpolation, which implements the labeled interpolation systems of [DKPW10] for propositional logic and a partial extension to satisfiability modulo theories.

Then, while investigating the impact of size and strength of interpolants in concrete applications, we have recognized the need for a new tool tailored to SAT-based verification, to be employed by the model checkers FunFrog and eVolCheck, developed by our group. This has led to the creation of PeRIPLO [Rol], Proof tRansformer and Interpolator for Propositional LOgic, built on the state-of-the-art SAT solver MiniSAT 2.2.0 [ES04]. PeRIPLO contains a more efficient implementation of the labeled interpolation systems and of the compression techniques of OpenSMT, as well as new algorithms for proof compression and manipulation, both from the literature and novel. PeRIPLO is also able to produce individual interpolants and collections of interpolants, with reference to various interpolation properties and in accordance with the constraints imposed by the properties on the labeled interpolation systems [RSS12, GRS13].

The tool and its features are illustrated in [GRS13, RBST, RAF+13] and discussed in §3.6.1 and in Chapter 5.

1.5 Thesis Outline

The thesis is structured in the following manner.

Chapter 2 talks about hybrid propositional satisfiability; in §2.2 we introduce a complete DPLL-stochastic solver, based on the cross-entropy method.

Chapter 3 is dedicated to the generation of Craig interpolants and their use in model checking. In §3.2 we raise the problem of interpolants quality, and charac-

terize interpolants in terms of structure and logical strength. In §3.4 we introduce the theory labeled interpolation systems, an extension of the propositional labeled interpolation systems to first order theories that generalizes standard approaches to interpolation in SMT. In §3.5 we present a parametric interpolation framework for arbitrary theories and inference systems, that computes interpolants of different structure and strength, with or without quantifiers, from the same proof. In §3.6 we provide experimental evidence of the impact of structure and strength in bounded model checking applications that make use of interpolants as function summaries. We also describe a new tool, PeRIPLO, that offers routines for interpolation and proof manipulation, with applications to SAT-based model checking.

Chapter 4 discusses the need in model checking for collections of interdependent interpolants, that satisfy particular properties; we uniformly present the most common among these properties, analyzing the relationships among them and outlining a hierarchy. In §4.3 we address the properties in the context of labeled interpolation systems, proving sufficient and necessary conditions for families of LISs and single LISs to enjoy each property; in §4.4 we extend the investigation of interpolation properties to the theory labeled interpolation systems.

Chapter 5 is dedicated to the topic of resolution proofs manipulation. In §5.3 we describe a post-processing proof compression framework based on local transformation rules, that reduces proofs by removing redundancies of different kinds. In §5.4 we present a proof transformation technique to allow interpolation in satisfiability modulo theories, by eliminating mixed predicates from proofs.

Chapter 2

Hybrid Propositional Satisfiability

Symbolic model checking ultimately relies on efficient reasoning engines to determine the satisfiability of formulae defined on a decidable fragment of first order logic. As part of our contribution to the field of formal verification, we addressed the development of new algorithms for the propositional satisfiability problem (or SAT), one of the central decision problems of informatics. Most of the successful approaches to SAT belong to either of two main classes: deterministic solvers which implement the Davis-Putnam-Loveland-Logemann (DPLL) algorithm, and probabilistic techniques based on stochastic local search (SLS). The two classes demonstrate better performance on different kinds of problems, and can be considered in this sense complementary.

There have been several attempts in the last years to integrate DPLL and SLS in an effective manner to obtain a competitive hybrid solver. Such attempts have often given good results; still, open problems remain in terms of choosing the particular deterministic and stochastic algorithms and in combining them successfully, exploiting the strong points of both.

In this chapter we present a new complete DPLL-stochastic algorithm. We introduce an extension of the propositional satisfiability setting to a multi-valued framework, where a probability space is induced over the set of all possible truth assignments. For a given formula, a cross-entropy based algorithm (which we implemented in a tool named CROiSSANT) is used to find a satisfying assignment by applying an iterative procedure that optimizes an objective function correlated with the likelihood of satisfaction.

Our hybrid approach employs cross-entropy as a preprocessing step to SAT solving. First CROiSSANT is run to identify the areas of the search space that are more likely to contain a satisfying assignment; this information is then given to a DPLL SAT solver to suggest variables assignments in the search. Experimental results on

different sets of benchmarks show that our approach represents a valid foundation for further research on cross-entropy based SAT solvers.

The chapter is structured as follows. §2.1 and §2.1.1 provide the background, introducing the satisfiability problem, as well as DPLL and SLS in detail. §2.2 presents our contribution in terms of a new hybrid approach to SAT: the extended satisfiability framework, as well as the CROiSSANT algorithm, are illustrated in §2.2.2, while experimental results are described in §2.2.3.

2.1 Satisfiability and SAT Solving

The propositional satisfiability problem (or SAT) is one of the central decision problems of informatics, the first to be shown NP-complete [Coo71]. Assuming a set of propositional variables $V = \{p, q, r, u, v, \ldots\}$, input of the problem is a propositional formula F built over V and the set of standard connectives $\{\neg, \vee, \wedge, \ldots\}$. A *truth assignment* is a function $\sigma : V \rightarrow \{\top, \bot\}$ that assigns a value *true* ($\top$) or *false* ($\bot$) to the variables in V; if σ covers all variables in V, then it is a *total* assignment, otherwise it is *partial*. σ is a *satisfying* assignment (or a *model*) if F evaluates to $\top$ under F; if F evaluates to $\bot$, then it is a *falsifying* assignment. If, for a given F, a satisfying assignment exists, then F is said *satisfiable* otherwise F is *unsatisfiable*. SAT can thus be stated as:"Given F, does a satisfying assignment exist?".

State-of-the-art procedures to check the satisfiability of a given F rely on an encoding called *conjunctive normal form (CNF)*. F is represented as a conjunction of *clauses* $C_1 \wedge \cdots \wedge C_m$, each being a disjunction of literals $l_1 \vee \ldots \vee l_{|C|}$. A *literal* l is a propositional variable, either with positive (v) or negative ($\neg v$) *polarity* (we will use the notations $\neg v$ and $\overline{v}$ equivalently). With respect to satisfiability, the restriction to CNF does not involve any loss: in fact, any F can be translated into an equisatisfiable F' in CNF in linear time, at the cost of introducing auxiliary propositional variables corresponding to its subformulae [Tse68]. In terms of CNF, a formula is then satisfiable if a truth assignment exists that satisfies every clause, whereas a clause is satisfied if at least one of its literals evaluates to $\top$. We use the symbol $\bot$ also to denote an *empty clause*, that is a clause with no literals, which is unsatisfiable. We assume clauses do not contain both a literal and its negation. A *unit clause* is a clause having exactly one literal.

2.1.1 DPLL and SLS

Several algorithms have been developed in the satisfiability framework, both *complete* and *incomplete*; complete techniques, given an input formula, either prove that

it is unsatisfiable or return a satisfying assignment; incomplete techniques are not guaranteed to always prove satisfiability or unsatisfiability. We refer to the algorithms and to the tools which implement them as *SAT solvers*.

Most of the successful algorithms belong to two main classes: (i) the ones (complete) which implement the *Davis-Putnam-Loveland-Logemann (DPLL)* procedure [DP60, DLL62], where the tools systematically traverse and backtrack a binary tree whose nodes represent partial assignments, and (ii) the others (incomplete) based on *stochastic local search (SLS)*, where solvers guess a full assignment and, if it is not a satisfying one, use different heuristics to reassign values to variables [MS99, HS05]. Various heuristics are employed by the stochastic methods to escape local minima and to ensure that the previously tried combinations of values are not assigned in the subsequent tries.

SAT solvers have undergone a constant improvement in the last decade, and have become established tools across a variety of fields such as software and hardware verification, planning, and scheduling [BCC+99, BCCZ99, BLM01, RN10, GSMT98]. The existence of international competitions [SATb] between solvers has contributed to the development of sophisticated algorithms and implementations [GKSS08, BHMW09]; among the DPLL SAT solvers we recall MiniSAT [ES04], Lingeling [Bie13], Glucose [AS09], while known SLS solvers are GSAT [SLM92] and WalkSAT [SKC94].

Based on the annual competitions, the DPLL-based tools demonstrate better performance when applied to real-life problems as they take advantage of, and learn from the structure of the solved instances [MMZ+01, ES04]. On the other hand, stochastic approaches tend to exhibit better performance on instances characterized by a degree of randomness, especially in conjunction with other techniques [MZ06]; the development of a hybrid solver, which would encompass the benefits of both approaches, is thus a goal of theoretical interest and practical usefulness.

In the following we discuss the main features of stochastic local search (in its WalkSAT implementation) and of conflict-driven clause-learning DPLL, as well as the relationship between DPLL and the resolution inference system, after which we move on to introducing our cross-entropy based approach.

2.1.1.1 DPLL-Based Solving

Algorithm 1 illustrates a typical *conflict-driven clause-learning (CDCL)* DPLL algorithm [GKSS08, MSS96, JS97]. The Davis-Putnam-Logemann-Loveland algorithm, in its CDCL extension, is an iterative decision procedure for satisfiability that combines search, deduction and learning techniques to help prune the space of possible assignments. DPLL starts by choosing a variable (*decision variable*) which

has not been given a truth value yet, and assigns to it either true or false (line 3). It then looks for clauses where all the literals besides one have been assigned to false; since each clause must be satisfied, the remaining literal must be assigned to true. This phase is known as *unit propagation* (line 5). At this point, there are three possibilities. All clauses are satisfied (lines 10-12), in which case a satisfying assignment is returned together with the answer SAT. At least one clause (*conflict clause*) is falsified (lines 6-9): the algorithm performs *conflict analysis* (line 7) and determines a consistent state to which to backtrack (*backtrack level*), while learning new information in the form of a *learnt clause*; if the level is zero (the level where no decisions have been taken yet), it means that the formula is unsatisfiable, and the answer UNSAT is returned. If no further propagations can be done, and neither satisfiability or unsatisfiability of the formula have been determined, a new decision variable is chosen.

```
   Input: CNF formula
   Output: SAT and a satisfying assignment or UNSAT
 1 begin
 2     while true do
 3         ChooseDecisionVariable()
 4         while true do
 5             DoUnitPropagation()
 6             if conflict found then
 7                 btlevel ← AnalyzeConflict()
 8                 if btlevel = 0 then return UNSAT
 9                 Backtrack(btlevel)
10             else if all clauses satisfied then
11                 output satisfying assignment
12                 return SAT
13             else break
14         end
15     end
16 end
```

Algorithm 1: CDCL DPLL solver.

An important aspect of the DPLL algorithm is the *decision strategy* adopted, i.e. the criterion used to choose a decision variable and to assign a polarity to it, which can have a major impact on the SAT solver performance. Several heuristics can be found in the literature, among which MOMS (maximum occurrence in clauses of

minimum size) BOHM, DLIS (dynamic largest individual sum), VSIDS (variable state independent decaying sum) [JW90, BS96, MSS96, MMZ+01, MS99]; in particular VSIDS, which favours variables often involved in deriving conflicts, is at the base of MiniSAT.

The essential feature of the CDCL approach is the *clause learning* process. Each time a sequence of decisions and deductions leads to a conflict, a new clause is learned as a summary of the conflict; this information is used to prune the search space and to drive the search process. Clause learning is usually combined with *conflict-driven backtracking* [SS77], which realizes a more efficient solution than chronological backtracking based on information provided by the learnt clause.

2.1.1.2 DPLL and the Resolution System

The DPLL algorithm has a close relationship with the *resolution system*, an inference system based on a single rule, called *resolution rule*:

$$\frac{p \vee D \qquad \overline{p} \vee E}{D \vee E}\, p$$

Clauses $p \vee D$ and $\overline{p} \vee E$ are the *antecedents* (D and E being clauses themselves), $D \vee E$ is the *resolvent* and p is the *pivot* variable.

The system is sound and complete; a proof of a clause C from a set of clauses (corresponding to the clauses in a CNF formula F) can be represented as a tree or a directed acyclic graph, where (i) each node is a clause, (ii) a leaf is a clause of F (iii) an inner node is the resolvent of a resolution step and its two parents are the antecedents in the step, and (iv) the only root is C. In particular, if F is unsatisfiable, a certificate of unsatisfiability is given by a proof of $\bot$ from F; this is called *proof of unsatisfiability* or *refutation*. An example of refutation is shown in Figure 2.1.

$$\frac{\dfrac{\overline{r} \qquad r \vee p}{p}\, r \qquad \dfrac{q \qquad \overline{p} \vee \overline{q}}{\overline{p}}\, q}{\bot}\, p$$

Figure 2.1. An example of resolution refutation.

SAT solvers can be instrumented to generate resolution refutations, by logging the inference steps performed during conflict analysis [ZM03b]. Each conflict yields a *proof chain*, a proof of the newly learnt clause from the conflict clause and of (some of) the clauses involved in the sequence of decisions and propagations that led to the conflict [BKS04]. The whole proof of $\bot$ from the clauses of the input formula can be built by joining the proof chains together.

2.1.1.3 SLS-Based Solving

Algorithm 2 contains the main features of common stochastic local search based approaches, taking inspiration from the well-known WalkSAT [SKC95, GKSS08]. It describes an incomplete method, unable to prove unsatisfiability or to guarantee the discovery of a satisfiable assignment; as such, it is run with a limit on the number of search attempts (*tries limit*). If a satisfying assignment is not found within the limit, then the answer FAIL is returned.

```
   Input: CNF formula F, number of search attempts tries limit, number of flips
          per attempt flips limit, noise parameter n ∈ [0, 1]
   Output: SAT and a satisfiable assignment or FAIL
 1 begin
 2     for i ← 1 to tries limit do
 3         σ ← random assignment to F variables
 4         for j ← 1 to flips limit do
 5             if σ satisfies all clauses then
 6                 output satisfying assignment
 7                 return SAT
 8             C ← random falsified clause
 9             if exists v′ s.t. bc(v′)=0 then
10                 v ← v′
11             else
12                 with probability n :        v ← random v′ in C
13                 with probability 1 − n :  v ← v′ s.t. bc(v′) is the smallest in C
14         end
15     end
16     return FAIL
17 end
```

Algorithm 2: WalkSAT-style solver.

Each attempt begins with assigning random values to the variables of the input formula F (line 3). This assignment is then locally adjusted, by modifying (*flipping*) the value of individual variables until either all clauses are satisfied (lines 5-7), or a limit on a number of flips (*flips limit*) is reached. To perform a flip, first a clause C is randomly chosen among the falsified clauses (line 8); then, a variable is picked from C and its value is set from true to false or vice versa according to the following criteria. If C contains a variable v' s.t. flipping v' does not make any currently satisfied clause falsified (i.e., its *break count* is equal to zero, line 9), then v' is cho-

sen. Otherwise, depending on the *noise* parameter n, the algorithm probabilistically performs either a *greedy walk* step (line 13), finding a v' that minimizes the number of clauses falsified by the flip, or a *random walk* step (line 12), picking a random v' in C.

The presence of noise is helpful in addressing the problem of *local minima*. The greedy walk takes the local optimal choice each time, but, as many other greedy strategies, is not guaranteed to find a global optimum, which in this context represents a satisfying assignment (assuming at least one exists); a purely greedy approach can in fact converge to local minima, assignments from which no move can decrease the amount of falsified clauses. To escape from such situations, some kind of noise is introduced in the search, for example by making a random choice with a certain probability.

2.2 A Cross-Entropy Based Approach to Satisfiability

In the last years, considerable research has been dedicated to combining DPLL- and SLS-based algorithms in an effective manner, trying to exploit the benefits of both worlds to obtain a competitive complete hybrid solver [SKM97]. We present here a novel approach to satisfiability built on a model-based stochastic technique known as *cross-entropy method* and on its adaptation to combinatorial optimization.

The cross-entropy method is a generic approach to rare event simulation and to combinatorial optimization [RK04]. It derives its name from the cross-entropy or Kullback-Leibler divergence, which is a fundamental concept of modern information theory [KL51], and consists in an iterative approach based on minimizing the cross-entropy between probability distributions. The cross-entropy method was motivated by an adaptive algorithm for estimating probabilities of rare events in complex stochastic networks [Rub97]; then, it was realized that a simple modification allows the use of this method also for solving hard optimization problems, in which there is a performance function associated with the inputs. The cross-entropy method in optimization problems is used to find a set of inputs on which the performance function reaches its global maximum (or minimum), where the input space is assumed to be too large to allow exhaustive exploration. The method is based on an iterative sampling of the input space according to a given probability distribution over this space. Each iteration consists of the following phases:

1. Generate a set of random samples according to a specified mechanism.

2. Update the parameters of the random mechanism based on the data to produce "better" samples in the next iteration, where "better" depends on the chosen

performance function.

The initial probability distribution is assumed to be a part of the input. Samples are evaluated according to the performance function. The procedure terminates when a sample with the maximal (or the minimal) value of the performance function is generated, or when an extremum is detected.

We present an algorithm based on the cross-entropy method and on an extension of the propositional satisfiability problem to a multi-valued setting. Given a propositional formula in CNF over a set V of propositional variables, we turn the range $\{0, 1\}$ into a discrete set D such that every element of D is between 0 and 1. Intuitively, these values provide a measure of the "level of truth" of a variable. A *probability space* is derived from all possible assignments to V over D; a probability distribution P assigns a probability to each variable and value. We define the semantics of the connectives over D and examine some performance functions $\mathbb{S}$ that correlate with the evaluation of φ, in such a way that, if φ is satisfiable, then a global minimum of $\mathbb{S}$ is reached under a satisfying assignment. Starting with a uniform distribution, we execute an algorithm based on the cross-entropy method with SAT-specific optimizations to minimize the value of the chosen performance function.

We devise a new hybrid DPLL-stochastic technique by letting the cross-entropy algorithm perform an initial fast exploration of the probability space. Assuming a satisfying assignment has not been found, from the values of the best-performing sample (following the intuition about "level of truth") a partial assignment is derived, which in turn is used by the SAT solver to decide the polarities to be given to variables during the search.

We implemented our variant of the cross-entropy method in a tool named *CROiSSANT* and tested it in combination with state-of-the art SAT solvers on different sets of benchmarks of academic and industrial origin; as the experimental results show, the approach shows effectiveness in driving the search at solving time.

2.2.1 Cross-Entropy for Optimization

The cross-entropy method was originally developed in order to efficiently estimate probabilities of rare events and was later shown to be effective in solving combinatorial optimization problems. In this section, we present a brief overview of the cross-entropy method for combinatorial optimization; more details can be found in [RK04].

In this setting, we are given a very large probability space $\mathscr{X}$, a probability distribution f defined on $\mathscr{X}$, and a function $\mathbb{S}$ from this space to $\mathbb{R}^+$, and we are

searching for an element X of $\mathscr{X}$ on which $\mathbb{S}$ reaches its global maximum. The function $\mathbb{S}$ is called the *performance function*, and intuitively, it measures "fitness" of the elements of $\mathscr{X}$. The space is assumed to be very large and infeasible to search exhaustively, and the global maximum is rare, so that the probability of finding it by random sampling according to the given probability distribution f is very low. In addition, in many problems we do not know the value of the global maximum in advance. The problem would be solved if we had another distribution g that increases the probability of finding the global maximum. The ideal distribution here would be $g_{\mathbb{S}}$, which gives probability 1 to inputs on which $\mathbb{S}$ reaches its global maximum and 0 to all other inputs. The cross-entropy method attempts to iteratively approximate $g_{\mathbb{S}}$ by changing the probability distribution on $\mathscr{X}$ so that the distance between the current distribution and the ideal distribution decreases in each iteration. The notion of distance used in this approximation is the Kullback-Leibler divergence or distance (also called cross-entropy). The Kullback-Leibler divergence between g and h is defined as:

$$\mathscr{D}(g,h) = E_g \ln \frac{g(X)}{h(X)} = \int g(x) \ln g(x) dx - \int g(x) \ln h(x) dx$$

Note that this is not a distance in the formal sense, since in general it is not symmetric. Since $g_{\mathbb{S}}$ is unknown, the approximation is done iteratively from the input distribution $f = f_0$, where in iteration t we draw a random sample according to the current distribution f_t and compute the (approximate) cross-entropy between the f_t and $g_{\mathbb{S}}$ based on this sample. The probability distribution f_{t+1} for iteration $t+1$ is computed from f_t and the sample in iteration t so that $\mathscr{D}(g_{\mathbb{S}}, f_{t+1}) < \mathscr{D}(g_{\mathbb{S}}, f_t)$. The exact update formula depends on the formulation of the problem, but, roughly speaking, the probability distribution is modified based on the best quantile of the current sample, so that the elements close to the best quantile get a higher probability in the next iteration.

The process can be halted once the global maximum is reached, if it is known in advance. When this is not the case, a common criterion, which we adopt, is to stop when the best quantile does not significantly change for a number of subsequent iterations. The convergence of our cross-entropy algorithm is discussed in §2.2.2.2.

2.2.2 The CROiSSANT Approach

In this section, we present an extension of the propositional model to the multi-valued one, together with the semantics of the propositional connectives; we also describe a probability space induced over the multi-valued space and discuss the choice of an appropriate performance function (§2.2.2.1). We define the variant of

the cross-entropy setting adopted and illustrate the core algorithm of CROiSSANT in detail (§2.2.2.2, §2.2.2.3). Then, we set out in §2.2.2.4 the mechanics of the interaction between CROiSSANT and a DPLL SAT solver and the intuitions behind that. Finally, we discuss in §2.2.2.5 the relationships between the parameters of the CROiSSANT algorithm and their tuning in our experiments.

2.2.2.1 The Multi-Valued Model

Intuitively, in order to guarantee convergence to a satisfying assignment of a given CNF formula φ, one needs a probability space and a performance function that allow an "evolution" of the assignments towards the satisfying assignment. In particular, this means that the performance function reaches its global maximum on a satisfying assignment, and that the value of the performance function increases when the sample is getting closer to a satisfying assignment. Moreover, the sample space should be dense enough to allow gradual changes of the probability distribution. In the standard propositional logic every variable can have only two values, and thus its value cannot change gradually. Our approach introduces a multi-valued framework, inspired by *fuzzy logic* [Zad88] and its use of *t-norms*.

Assuming a domain $D \subseteq [0,1]$, an assignment function $e : V \to D$ extends a standard propositional assignment by associating each variable of V with a value in D. The function e^* evaluates an arbitrary propositional formula φ to a number in $[0,1]$, given an assignment e of values to its variables:

- $e^*(v) = d$ if $e(v) = d$
- $e^*(\neg\psi) = 1 - e^*(\psi)$.
- $e^*(\psi \vee \eta) = e^*(\psi) \times e^*(\eta)$.
- $e^*(\psi \wedge \eta) = \max(e^*(\psi), e^*(\eta))$.

In our framework, 0 stands for *true* and 1 stands for *false*, contrary to the traditional way of defining their meaning – this allows a more efficient computation of the value of propositional formulae.

Remark. While the above definition of e^* does not correspond to standard algebraic operations (in particular, there is no distributivity), it is easy to see that e^* applied to $\wedge$ is a t-norm, that is, generalizes conjunction, and that e^* applied to $\vee$ is a t-conorm, that is, generalizes disjunction. In particular, when $D = \{0,1\}$, e^* matches the standard definitions of $\wedge$ and $\vee$ in De Morgan algebra. Indeed, we have for $d \in D$:

- $e^*(0 \vee d) = 0$, that is, *true* $\vee\, d =$ *true*;
- $e^*(1 \vee d) = d$, that is, *false* $\vee\, d = d$;
- $e^*(1 \wedge d) = 1$, that is, *false* $\wedge\, d =$ *false*;
- $e^*(0 \wedge d) = d$, that is, *true* $\wedge\, d = d$.

The domain D could in theory be the interval $[0, 1]$. Practically, working in a continuous environment is unfeasible from a computational point of view for a combinatorial problem like satisfiability; thus we adopt a framework that *approximates* the continuous domain by a discrete one, allowing a more efficient computation of the performance function and a simpler mechanism to update probabilities in the cross-entropy algorithm. For $K \in \mathbb{N}, K \geq 2$, the range D_K of all variables in V is the set of values $\{0, 1/(K-1), 2/(K-1), \ldots, 1\}$. A larger K provides finer granularity and thus a denser space, while a smaller K leads to more efficient computations.

In what follows, we assume an input formula φ in CNF over a set of variables V, with $|V| = n$. A probability space is defined over the set E of all possible complete assignments e over V, where $e(v) \in D_K$ for each $v \in V$. A probability distribution $P : E \to [0, 1]$ gives to each assignment its probability in the following way. Probabilities for individual variables v and values d are directly specified as $P(v = d)$. In the cross-entropy framework, variables are treated independently; thus, the probability of an assignment $e(v_1) = d_1, \ldots, e(v_n) = d_n$ is computed as $P((v_1 = d_1), \ldots, (v_n = d_n)) = \prod_i P(v_i = d_i)$. Due to this independence condition, the probability distribution is completely represented by means of a matrix $\mathbf{p}$ of size $n \times K$, whose generic element $\mathbf{p}_{i,j}$ is exactly $P(v_i = d_j)$.

A performance function $\mathbb{S}_\varphi : E \to [0, 1]$ assigns to each $e \in E$ a value that denotes the "truth level" of φ, that is how close φ is to being satisfied under the assignment e. Dealing with formulae in CNF, the most straightforward choice for $\mathbb{S}$ is the max function based on e^* as defined above: the value of a clause c is $e^*(c)$, the product of its literals values, and the value of a formula is given by the maximum value among its clauses. This way, a single clause having value 1 (falsified) results in the whole formula having value 1, and assignments that satisfy all but one clause (a common situation in structured SAT instances) are very far from the global minimum. However, based on our experiments, the max function appears to not be smooth enough for our purposes, and it does not allow for a sufficiently accurate discrimination among assignments: indeed, all assignments which falsify at least one clause have the same value 1 and are thus indistinguishable. A better choice is the function $\mathbb{S}$ which we define as follows.

Definition 2.2.1 (Performance Function $\mathbb{S}$)**.** Given a formula in CNF $\varphi = C_1 \wedge \cdots \wedge C_m$ and an assignment e, the performance function $\mathbb{S}$ is defined for a clause C_i as $\mathbb{S}_{C_i}(e) = \prod_{l \in C_i} e(l)$, and for φ as $\mathbb{S}_\varphi(e) = (\sum_{i=1}^{m} \mathbb{S}_{C_i}(e)/m)$.

Note that if φ is satisfiable, then the global minimum of $\mathbb{S}$ is 0 and it is reached on a satisfying assignment to φ. In other words, we replace the max function by the arithmetic average of the clauses values; as we demonstrate in §2.2.3, this definition allows us to achieve satisfactory convergence towards a global minimum in a reasonable time for satisfiable instances.

2.2.2.2 The Parameter Setting

A generic cross-entropy algorithm for optimization involves two main steps:

- Generate a number R of independent and identically distributed samples in the search space according to a specified probability distribution $\mathbf{p}_t$.
- Update the parameters of $\mathbf{p}_t$, based on a certain amount R_b of best performing samples (the *elite samples set*), by means of cross-entropy minimization.

The cross-entropy method creates a sequence of *levels* $\gamma_0, \ldots$ and *parameter matrices* $\mathbf{p}_0, \ldots$ such that the first sequence converges to an optimal performance, while the second one converges to the corresponding optimal parameter matrix. Usually a smoothed updating rule is used, in which the parameter matrix is taken as a linear combination (with *smoothing parameter* α) of the previous matrix and the new one obtained from the elite samples set.

Several variants of the cross-entropy method ([RK04, Mar05, KTB11, CJK07]) are described in the literature: with or without smoothing, with or without memory (the elite samples set is entirely computed from the current iteration rather than accounting for the previous generations), with fixed or adaptive α, R and R_b. We adopt a memoryless approach, since it is more efficient in this framework. For the same reason, we keep both R and R_b fixed during the algorithm execution. As commonly done for combinatorial optimization problems, we choose a smoothed updating procedure to avoid the degeneration of the sequence of parameter matrices to a $\{0, 1\}$-matrix.

Different proofs can be found certifying the theoretical convergence of the algorithms both in the continuous and in the combinatorial setting; given the approach chosen and dealing with a discrete problem, we mainly refer to [KTB11] and [CJK07]. In particular, [CJK07] compares the effects of adopting a constant or adaptive smoothing parameter: an appropriately decreasing sequence of α is *guaranteed to reach an optimal solution*; using a constant but sufficiently small α, the

algorithm converges to an optimal solution with probability arbitrarily close to 1. The price is in terms of speed of convergence; this is why, following the considerations in the literature [RK04] and our own experience, we use a constant smoothing parameter, even though it might lead to non global optima.

2.2.2.3 The CROiSSANT Algorithm

Algorithm 3 illustrates our application of the cross-entropy method to the satisfiability problem. The algorithm takes as input a CNF formula φ, a coarseness K of the discretization D_K, a number of samples to be generated in each iteration R, the performance function $\mathbb{S}$ introduced in §2.2.2.1, a size of the elite samples set R_b, a smoothing parameter α, a time limit.

```
Input: CNF formula, discretization D_K, number of samples per iteration R,
       number of elite samples R_b, performance function S, smoothing
       parameter α, time limit
begin
    repeat
        Initialize p_0
        t ← 1
        repeat
            Generate R assignments e_1, ..., e_R according to p_{t-1}
            Evaluate S(e_i), sort from smallest to largest S_(1) ≤ ... ≤ S_(R)
            Extract the elite samples set e_(R-R_b+1), ..., e_(R)
            Set γ_t ← S_(R-R_b+1)
            Calculate q_t from the elite set using (*)
            Update p_t ← α q_t + (1 - α) p_{t-1}
            t ← t + 1
        until a minimum is detected or the time limit is reached
    until the time limit is reached
    Return information (see §2.2.2.4)
end
```

Algorithm 3: The CROiSSANT algorithm.

The algorithm follows these steps. It starts by initializing the probability matrix $\mathbf{p}_0$ to the uniform distribution over the variables V and the values D_K, moving then to the main loop. In each iteration t, a number R of i.i.d. sample assignments from D_K^n is drawn according to the probability matrix $\mathbf{p}_{t-1}$ of the previous iteration. The performance function $\mathbb{S}$ (which gives a measure of the "truth level" of the formula

φ) is evaluated on the samples, and the results are sorted in ascending order; the R_b best samples are taken to form the new elite set and the new level γ_t (corresponding to the performance of the worst sample in the elite set) is determined. From this set, for each variable v and value d a probability $\mathbf{q}_{t,v,d}$ is derived according to the following equation:

$$\mathbf{q}_{t,v,d} = \frac{\sum_{z=1}^{R} I_{\{\mathbb{S}(e_z)\geq\gamma_t\}} I_{\{e_z(v)=d\}}}{\sum_{z=1}^{R} I_{\{\mathbb{S}(e_z)\geq\gamma_t\}}} \qquad (*)$$

where the function I is an *indicator function* (or a *characteristic function*): for a propositional expression b the value of $I_{\{b\}}$ is 1 if b is true, 0 otherwise. In other words, the probability $\mathbf{q}_{t,v,d}$ is the proportion of samples in the elite set for which v has value d (recall definitions from §2.2.2.1). Finally, the probability matrix $\mathbf{p}_{t-1}$ is updated by "shifting" it (based on the smoothing parameter α) towards the matrix $\mathbf{q}_t$ generated from the elite set, thus making it biased towards the current most successful samples.

We use a simple criterion in order to detect whether a minimum has been found: the sequence of γ_t steadily decreases (while the probability matrix converges to an optimal one) although it exhibits oscillations due to the stochastic nature of the algorithm. To account for this phenomenon, we keep track of the current lowest value among the γ_t: if after a certain number of iterations such value has not improved, we determine that a minimum has been reached.

Since, as discussed, the algorithm is not guaranteed to converge to a global minimum, the whole process is repeated for a series of runs until the time limit is met.

In the end, the algorithm returns some information, for example the overall best sample obtained throughout the various runs (see §2.2.2.4).

2.2.2.4 Interaction with SAT Solving

As a first investigation of a new hybrid framework that exploits the features of DPLL and stochastic optimization, we have employed our cross-entropy algorithm as a preprocessing step before SAT solving. CROiSSANT performs a preliminary exploration of the search space, storing the best assignment found; this is used to generate a partial propositional assignment which is given as input to the DPLL solver. Then, at solving time, when branching on a variable v, the search is driven by choosing its polarity according to that partial propositional assignment (of course if the variable is affected by the assignment).

The reason behind this approach lies in the relationship between the granularity of the search space and the way the propositional assignments are produced. In

§2.2.2.1 we introduced the multi-valued model and discussed the extension of the propositional domain to an arbitrary discrete domain D_K with K values between 0 and 1. Intuitively, the association of a value with a variable measures how that variable is close to being either true or false: 2 values can distinguish between true and false, 3 values allow "unknown" variables, and so on. An initial exploration of the search space by CROiSSANT can provide the SAT solver with useful information on the likelihood of variables to be either true or false. By analyzing the best sample found, we extract the subset of variables which have value 0 or 1 and use these values to guide the decision phase at solving time.

2.2.2.5 Tuning of the CROiSSANT Algorithm

The input of the algorithm consists of a CNF formula φ, a coarseness K of the discretization D_K, a number R of samples to generate in each iteration, a performance function $\mathbb{S}$, a size R_b of the elite samples set, a smoothing parameter α and a time limit.

Since our goal is to use cross-entropy as a fast preprocessing technique, we give CROiSSANT a short timeout compared to the total time dedicated to preprocessing plus solving. The reasons for the choice of a specific $\mathbb{S}$ have been presented in §2.2.2.1; in order to set a framework for experimentation we performed some preliminary benchmarking, analyzing the relationships among R, R_b, α, with respect to the timeout and to the size of the search space, dependent on φ and D_K.

Larger values of R allow for deeper exploration at each iteration; on the other hand, the generation of more samples comes at the price of a smaller amount of iterations that can be carried out within a given time limit. For our experimentation we chose a fixed R, but we plan to investigate how to make the parameter adaptive to the complexity of the instance at hand. A similar tradeoff exists for the size of the elite set: values of R_b close to R increase the preciseness of the estimates (almost all samples are considered to update the distribution), while smaller values allow for faster convergence; we use a fixed R_b of 10% of R.

The adoption of a constant α entails an additional tradeoff between accuracy and speed of convergence: a smaller α increases the probability of finding an optimal solution, but at the expense of a greater amount of iterations.

Another parameter was added, to account for the possibility of running into non global minima, as discussed in §2.2.2.1. Stopping condition for the inner loop of the algorithm is the detection of a minimum, in terms of a lack of improvement of the current lowest value in the sequence of γ_t for a certain number of iterations; this amount, to which we will refer as *"no improvement threshold"* (NIT), is also given as input. Notice that a good choice for the threshold depends both on R and

R_b (more exploration increases the chances of improvement), and on the size and the characteristics of the search space.

With regard to the coarseness of the discretization, a greater K entails a more dense space, with advantages in terms of gradualness of convergence of the cross-entropy technique (§2.2.2.2) and accuracy of the information supplied to the SAT solver (§2.2.2.4); nevertheless, a larger space might make convergence slower and contain more non global minima. We tested this tradeoff using different values of K.

An interesting question that arose, while developing our approach, was how likely it would be to come across the same minimum in different runs, and, in that case, how to avoid the issue. We experimented with manipulating the probability matrix in order to direct the search away from the minima already found. Unfortunately, one of the major points of the cross-entropy algorithm, that is being able to treat variables individually, turned out to be a hindrance to our idea: in fact, by modifying the probabilities of the variables, we would end up reducing the probability of generating not only the minima themselves, but also many other unrelated sample assignments. In practice, we achieved the best performance by simply restarting from the uniform distribution: for a given timeout, taking as input formulae of sufficiently high complexity (so as to have a sufficiently large search space), we noticed that the algorithm tended to visit different areas of the space in each run, leading to different minima.

2.2.3 Experimental Results

We implemented our variant of the cross-entropy based optimization algorithm in a tool named CROiSSANT written in C++. In this section we present the results of executing CROiSSANT as a preprocessor on a collection of benchmarks with different settings of tunable parameters, following the considerations of §2.2.2.5.

Since our approach employs the cross-entropy method with the aim of identifying the areas of the search space that are more likely to contain a satisfying assignment, we considered it natural to focus on *satisfiable* benchmarks. We collected satisfiable instances from the SAT Competition 2011 and the SAT Race 2010 [SATb] (which contain benchmarks of both academic and industrial origin) and compared the running time of the state-of-the-art CDCL SAT solver MiniSAT 2.2.0 alone with the running time of MiniSAT after preprocessing by CROiSSANT. Given the stochastic nature of our technique, we took into account not only the default deterministic strategy of MiniSAT (to always assign negative polarity), but also the randomized strategy that assigns polarity positive or negative at each branching step

with equal probability[1].

Table 2.1. MiniSAT on SAT Competition 2011 and SAT Race 2010 benchmarks.

		(1) MiniSAT only		(2) MiniSAT random pol.	
	#Bench	#Solved	Time(s)	#Solved	Time(s)
SAT Comp.2011	305	140.0	2126.56	145.7	2074.63
application	47	41.0	638.70	37.7	778.08
crafted	58	45.0	971.98	41.7	1226.05
random	200	54.0	2811.05	66.3	2625.40
SAT Race 2010	22	20.0	627.17	19.3	756.42
cryptography	11	11.0	556.39	10.3	704.38
hardware-verif.	1	1.0	106.45	1.0	12.71
mixed	7	5.0	1070.14	5.0	1144.97
software-verif.	3	3.0	26.72	3.0	288.56

		(3) 2 mins, $K = 2$		(4) 2 mins, $K = 3$		(5) 2 mins, $K = 10$	
	#Bench	#Solved	Time(s)	#Solved	Time(s)	#Solved	Time(s)
SAT Comp.2011	305	143.7	2017.71	139.7	2039.56	143.3	2026.79
application	47	42.0	619.41	39.7	654.00	40.0	687.45
crafted	58	47.7	798.19	48.3	794.50	50.7	769.28
random	200	54.0	2699.97	51.7	2726.23	52.7	2725.45
SAT Race 2010	22	19.3	736.80	20.3	624.69	19.3	668.95
cryptography	11	10.3	791.53	10.7	623.13	10.0	687.31
hardware-verif.	1	1.0	107.62	1.0	172.85	1.0	40.50
mixed	7	5.0	1035.19	5.7	942.05	5.3	1000.07
software-verif.	3	3.0	49.57	3.0	40.50	3.0	38.52

Experimental Data. A set of 305 + 22 satisfiable benchmarks was initially extracted by running MiniSAT on the 900 new instances of SAT Competition 2011 and on the 100 instances of SAT Race 2010, setting a time limit of 3h. The benchmarks were run with a total (preprocessing plus solving) time limit of 1h on machines equipped with a Quad-Core AMD Opteron(tm) processor 2344 HE 1000 MHz and 3Gb of RAM memory.

Table 2.1 presents the results of executing CROiSSANT as a preprocessor to MiniSAT in different configurations compared to the results of MiniSAT alone; we

[1]The full experimental data is available at `http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

show the average values obtained over three runs using different seeds. The first and the second configurations in the table consist in running just MiniSAT respectively with default and random polarities. The other configurations include preprocessing using 50 as number of samples per iteration R, 0.1 as smoothing factor α and 50 as no improvement threshold NIT, whereas preprocessing time and coarseness of the discretization K (as number of values) are shown; the best overall (partial) assignment, not taking into account variables with values different from 0 and 1, was given to MiniSAT to suggest variable polarities during branching. In the *#Solved* column, the numbers represent the amount of instances for which MiniSAT could find a satisfiable assignment. The *Time(s)* column shows the average time in seconds spent (after the preprocessing phase) by MiniSAT on the different families of instances, both solved and non solved; preprocessing time is not included, apart from the few cases when CROiSSANT was able to find a satisfiable assignment itself, and clearly MiniSAT was not run.

Analysis. The experiments give some interesting results. Table 2.1 shows that in the first configuration (2 minutes preprocessing, $K = 2$) the use of CROiSSANT indeed improves MiniSAT performance on all the three classes of the SAT Competition instances. CRoiSSANT is also noticeably superior (both in number of solved benchmarks and average running times) on the crafted class: this mainly contains problems of combinatorial origin, such as graph isomorphism, Green Tao and Van der Waerden numbers, automata synchronization. Instances of this kind tend to show regularities in the solution space: global and non global minima might be close, so that the information provided by CROiSSANT in terms of the assignment with the best performance (that is, closest to a satisfying assignment) would be particularly useful. The adoption of a random in opposition to a non-random approach in assigning polarities has a major effect on the results over structured and random instances; in particular, the random strategy performs better than all the other techniques on random problems, but systematically worse on structured problems.

2.2.4 Related Work

It is natural to view satisfiability as an optimization problem on the space of complete variables assignments, with the performance function evaluating, for example, the number of satisfied clauses. As such, there is considerable ongoing research on applying optimization techniques to satisfiability. A number of powerful stochastic solvers is based on various extensions of the random walk model, see for example GSAT [SLM92], WalkSAT [SKC94], HSAT [GW93], SDF [SS01], Novelty and R-Novelty [MSK97], Novelty+ and R-Novelty+ [Hoo99], UnitWalk [HK05].

In contrast to these techniques, our approach is to iteratively adjust the probability distribution, according to which assignments are generated. Also, while nearly all known stochastic approaches for propositional satisfiability use a binary range for the variables, we use smoother ranges, which allows a more gradual improvement. Surprisingly there is no prior work on the application of the cross-entropy method to satisfiability, to the best of our knowledge, apart from a brief mention in [RK04]. At the same time, the cross-entropy method is used in many other areas, including buffer allocation [AKRR05], neural computation [Dub02], DNA sequence alignment [KK02], scheduling [Mar02], graph problems [Rub02], and, more recently, testing [CFGN07] and replay [CFGN09] of concurrent programs. It was shown to be very effective in discovering or approximating solutions to very hard problems for which it is possible to define a "good" (smooth and gradually improving) performance function.

In the last years there has been extensive research on combining SLS and DPLL. Three main research branches have been followed: using DPLL as an aid for SLS or vice versa SLS as a helper for DPLL, either before or at solving time [Cra93, MSG98, FF04, JO02, FR04, HLDV06]; letting the two techniques interact in a synergistic way [FH07, LMS08, BHG09, ALMS09a, ALMS09b].

In [MSG98] it is suggested, for an unsatisfiable formula, to use a stochastic solver to try to identify its minimal unsatisfiable cores, and then to use a complete solver on the smaller resulting subformula. Moreover, the scores delivered by the stochastic solver could be used to guide the branching strategy of the complete solver.

Using a stochastic solver to determine polarities for branching variables is probably even more natural, however to our knowledge it has not been explicitly mentioned in the literature. There is also a strong connection between setting the initial branching polarities and the progress saving heuristic, suggested in [PD07] and used in all state-of-the-art CDCL solvers.

2.2.5 Summary and Future Developments

We presented a novel approach to the satisfiability problem based on the cross-entropy method. Our framework extends the binary propositional model to a multi-valued setting, where variables can be assigned values in a discrete set; we induce a probability space over the enlarged search space and define a performance function that correlates with the likelihood of the input formula to be satisfiable. We implemented our variant of the cross-entropy algorithm in a tool named CROiSSANT, written in C++. We employ CROiSSANT as a preprocessor to a DPLL-based solver; CROiSSANT is executed with a time limit and the best assignment found is ex-

ploited to suggest variables polarities at solving time. We conducted experiments on different sets of benchmarks, using CROiSSANT in combination with the state-of-the-art solver MiniSAT; the results show an improvement both in running times and number of solved instances, providing evidence that our framework is a sound basis for further research on cross-entropy based SAT solvers.

Future work can move in a number of directions. We tested one among several ways of interaction between CROiSSANT and a SAT solver. Other approaches, still having cross-entropy minimization as a preprocessing step, might include: employing a partial propositional assignment also taking into account how many and which variables are involved; deriving from the best sample a vector of probabilities based on which to assign polarities; storing and using the probability matrix from which the best sample was generated; exploiting structural dependencies among clauses.

More complex approaches could be based on an integrated framework, where a cross-entropy algorithm and a SAT solver are run alternately while exchanging information; learning techniques could be tested based on resolution, in line with [FR04], where new clauses are added to reshape the space to remove current local minima.

Cross-entropy can be combined with other stochastic optimization methods, such as simulated annealing, swarm and evolutionary algorithms. A promising direction is also to enhance CROiSSANT with local techniques, for example stochastic local search: intuitively, a cross-entropy-based algorithm can perform an initial wide-scale narrowing of the search space, while local search will be applied in a second phase for finding a global minimum in a much smaller area. It would be also useful to characterize the classes of instances (as well as the structure of the corresponding search spaces) and the areas, such as combinatorics, where the cross-entropy method is most effective.

Chapter 3

Craig Interpolation in Model Checking

Craig's interpolation theorem is a classical result in first order logic, which has found wide and successful application as an overapproximation technique in symbolic model checking.

Interpolation is defined with respect to an unsatisfiable conjunction $A \wedge B$, and interpolants are typically generated from a proof of unsatisfiability of the conjunction. Interpolants are not unique, since various algorithms exist to generate different interpolants from the same proof, and, in turn, proofs can be transformed by means of manipulation techniques, consequently allowing additional interpolants. An important goal in model checking – and the one addressed in this research – is thus to assess which interpolants have the highest "quality" and which features characterize good interpolants in the individual verification frameworks. In this chapter we address interpolation from both a theoretical and a practical point of view, focusing on syntactic and semantic features and on their impact on verification.

§3.3 formally defines Craig interpolation and introduces the relevant notation; it discusses the state-of-the art of interpolants generation in first order theories, focusing on proof-based interpolation techniques. In particular, §3.3.3 illustrates the *local proofs framework* [KV09, HKV12], able to compute interpolants from a class of proofs in arbitrary first order theories and inference systems; §3.3.4 addresses the *labeled interpolation systems (LISs)* [DKPW10, Wei12], that perform interpolation in the propositional resolution and hyper-resolution inference systems; §3.3.5 describes *interpolation in satisfiability modulo theories*, as realized by [YM05, CGS10]. We reformulate the aforementioned frameworks, providing a uniform notation that makes it easier to compare them with our contribution in the following sections.

The LISs are an appropriate choice for investigating interpolant strength in model checking, since they consist of a parametric algorithm that allows to systematically generate interpolants of different strength from the same proof; however, they are

limited to propositional logic. We address this limitation in §3.4 by extending LISs to the context of SMT, at the same time subsuming [YM05, CGS10] as special cases; we also provide a concrete application of our extension to a first order theory.

The approach of [KV09, HKV12] is appreciated for its generality, since it applies to any theory and inference system in first order logic. In §3.5 we show how the methods of [DKPW10, Wei12, KV09, HKV12] belong to a same class of interpolation algorithms, based on recursive procedures that generate interpolants from refutations. We analyze this type of algorithms and develop a new parametric interpolation framework for arbitrary first order theories and inference systems; it is able to derive interpolants of different structure and strength, with or without quantifiers, from the same proof. Moreover, we demonstrate that [DKPW10, Wei12, KV09, HKV12] can be considered instantiations of our framework.

After providing the theoretical infrastructure, we move on to experimental evaluation. In §3.6 we address the problem of generating effective interpolants in the context of software SAT-based bounded model checking, and examine the impact of size and logical strength on verification. We take into account two bounded model checking applications: verification of a given source code incrementally with respect to various properties, and verification of software upgrades with respect to a fixed set of properties. Both applications use interpolation for generating function summaries. We provide evidence that size and logical strength of interpolants significantly affect the performance of verification, and that these characteristics depend on the role interpolants play in the verification process.

3.1 Interpolation in Symbolic Model Checking

Craig's interpolation theorem [Cra57a, Cra57b] is a classical result in first order logic, and has had a strong impact on the formal verification community since the work carried out in [McM03, McM04b, McM04a], where it was shown that interpolation can be applied as an effective overapproximation technique. Interpolation has stimulated a considerable amount of research in the area; in the following we introduce some of the most successful applications to model checking, which are relevant to the topics discussed in this thesis.

A *Craig interpolant* I is formally defined with respect to an unsatisfiable conjunction of formulae $A \wedge B$. I is a formula that is implied by A (i.e., $A \models I$), is unsatisfiable in conjunction with B (i.e., $I \wedge B \models \bot$) and is defined on the common symbols of A and B. In other words an interpolant is an overapproximation of A that is still unsatisfiable with B.

Interpolants have been exploited in conjunction with bounded model checking

in [McM03] to realize unbounded symbolic model checking; [McM03] presents an algorithm that iteratively increases the bound and at the same time tries to overapproximate the set of reachable states. Also, the method is guaranteed to terminate, in the worst case when the bound reaches the diameter of the system.

In the context of predicate abstraction, interpolation can be used to approximate the transition relation of a system, yielding a complete technique (assuming an adequate set of predicates) which does not require exact image computation [JM05].

Craig interpolation is employed in [HJMM04] for counterexample-guided abstraction refinement: new predicates are synthesized from a spurious abstract behavior, which is then removed by means of a local refinement of the abstraction.

Synthesis is also the goal of [JM06], which presents an approach to guide the derivation of predicates from interpolants, by restricting the language of the interpolants. The procedure is guaranteed to terminate if the property is provable.

In [McM06], an application of interpolation is proposed within the lazy abstraction framework of [HJMS02]. Following the abstraction-refinement paradigm, an abstract model is built and iteratively refined; refinement is carried out locally, based on interpolants generated from the refutation of program behaviors.

[VG09] presents an extension of bounded model checking, which combines it with the use of interpolants as done in [McM06] and [JM05].

Interpolants are integrated with the theory of nested words in [HHP10], developing a software model checking approach for recursive programs.

A recent application of interpolation to verification involves the use of interpolants to abstract of the behavior of function calls (the approach is known as *function summarization*); the goal of summarization is to store and reuse information about already analyzed portions of a program, to make subsequent verification checks more efficient. Function summarization is combined with bounded model checking in [SFS11] and in [SFS12b] to perform incremental verification of software programs. Function summarization is also employed in the Whale framework of [AGC12] to achieve inter-procedural verification; interpolants play a twofold role, since they can represent either overapproximations of set of reachable states, or summaries of function bodies.

3.1.1 Interpolation, BMC and IC3

In the context of symbolic model checking (see §1.1) a system is usually represented as a pair of formulae (S, T), where S denotes the set of initial states and the transition relation T describes the system behaviour. A property to be verified is encoded as a formula P, so that the system is safe if the error states where $\neg P$ holds are not reachable from S.

Verifying that the system satisfies P reduces to prove that P is an *inductive invariant* property:

$$S \models P \qquad\qquad P \wedge T \models P' \tag{3.1}$$

If (i) the initial states satisfy P and, (ii) assuming P holds, it also holds after applying the transition relation, then P holds in all reachable states. When the inductiveness of P cannot be directly proved, it might be possible to show that another formula $\hat{P}$, stronger than P ($\hat{P} \models P$), is an inductive invariant, from which P would follow as a consequence; the approach of [McM03], which combines interpolation and bounded model checking (BMC), is based on iteratively building such a $\hat{P}$, as illustrated below.

```
Input: Transition system (S, T), property P
Output: SAFE or UNSAFE and a counterexample
Data: k: bound, R(i): overapproximation of states at distance at most i from
      S, I(i): interpolant
begin
    if S ∧ ¬P is satisfiable then return UNSAFE, counterexample
    k ← 1, i ← 0
    R(i) ← S
    while true do
        A ← R(i) ∧ T^0
        B ← ⋀_{j=1}^{k-1} T^j ∧ ⋁_{l=0}^{k} ¬P^l
        if A ∧ B is satisfiable then
            if R(i) = S then return UNSAFE, counterexample
            else
                k ← k + 1, i ← 0
                R(i) ← S
        else
            I(i) ← Itp(A, B)
            if I(i) ⊨ R(i) then return SAFE
            else
                R(i + 1) ← R(i) ∨ I(i)
                i ← i + 1
    end
end
```

Algorithm 4: Interpolation and BMC.

BMC performs verification w.r.t. a fixed bound k; P is considered within distance k from S, by checking the satisfiability of a propositional formula of the kind $S \wedge T^0 \wedge \cdots \wedge T^{k-1} \wedge (\neg P^0 \vee \cdots \vee \neg P^k)$. [McM03] uses interpolation to effectively realize unbounded model checking, as shown in Algorithm 4. Standard BMC examines states at distance at most k from the initial states S, while increasing k; the approach of [McM03] instead computes overapproximations of the states reachable from S, step by step, and checks whether from these overapproximations the property P can be violated within distance k, while trying to keep k fixed.

The algorithm first makes sure P is not violated by S (line 2), otherwise it halts and immediately returns a counterexample. After that, the bound k is set to 1, and the initial overapproximation $R(0)$ to S. In the main loop, for a bound k and a current overapproximation $R(i)$ of the states at distance at most i from S, the algorithm checks if P is violated by the states reachable from $R(i)$ in at most k steps.

If the formula $R(i) \wedge \bigwedge_{j=0}^{k-1} T^j \wedge \bigvee_{l=0}^{k} \neg P^l$ is unsatisfiable (lines 13-18), then the formula is partitioned into $A \wedge B$ and an interpolant $I(i)$ is computed, which represents an approximation of the image of $R(i)$ (i.e., of the states reachable from $R(i)$ in one step). Next, a fixpoint check is carried out: if $I(i) \models R(i)$, it means that all states have been covered, and the system is safe; otherwise, $R(i+1)$ is set to $R(i) \vee I(i)$ and the procedure continues.

Suppose instead that the formula is satisfiable (lines 8-12): the error might be real or spurious, caused by an insufficient value of k. In the former case, the system is unsafe (line 9); in the latter, k is increased to allow finer overapproximations, and the algorithm restarts from S.

When Algorithm 4 returns SAFE, the current $R(i)$ corresponds to an inductive invariant $\hat{P}$ stronger than P: on one side, $S \models R(i)$, moreover $R(i) \wedge T \models I'(i)$ and $I(i) \models R(i)$ imply $R(i) \wedge T \models R'(i)$; on the other side, the fact that at each iteration $0 \leq h \leq i$, $R(h) \wedge \bigwedge_{j=0}^{k-1} T^j \models \bigwedge_{l=0}^{k} P^l$, together with $R(i)$ being an inductive invariant, yield $R(i) \models P$.

A different approach to strengthening is adopted by the *IC3* (also called *property directed reachability*) verification framework, introduced in [Bra11] and further discussed in [SB11, Bra12], which in the last years has established itself as a noteworthy alternative to interpolation-based model checking as performed in [McM03]. The algorithms of [McM03] and [Bra11] have some aspects in common: they both compute a sequence $R(0), \ldots, R(h), \ldots$ of overapproximations of the states within distance $1, \ldots, h, \ldots$ from S, until an inductive invariant that implies P is found; also, for any h, $R(h) \models R(h+1)$ and $R(h) \wedge T \models R'(h+1)$.

The main feature that distinguishes [Bra11] from [McM03] is the absence of unrolling: reasoning is in fact localized to one application of the transition relation (e.g. $R(h) \wedge T \rightarrow P'$), which helps reducing the solving effort. Another important

difference is in the way the two methods behave when an error is detected: whenever [McM03] generates an $R(h)$ from which a state violating P can be reached, it discards the whole sequence of overapproximations and restarts from S, after increasing the bound k; on the contrary, [Bra11] exploits a counterexample to the validity of $R(h) \wedge T \rightarrow P'$ in order to learn new information which allows to refine the sequence itself. IC3 has undergone developments in several directions [CG12, HB12, BR13, HBS13], and interesting results are expected from its integration with interpolation-based techniques.

3.2 Interpolants Quality

Interpolation is by now well-established in the area of model checking and several algorithms are available to construct interpolants for an unsatisfiable conjunction of formulae. While interpolation-based techniques crucially depend on to which extent "good" interpolants can be generated, there is no general criterion for defining the notion of "good" with respect to a particular verification framework. Finding a good interpolant is a hard problem, whose solution justifies spurious behaviors or helps proving correctness of system properties. Given a system to be verified, a natural question to ask is therefore "what makes some interpolants better than others?", or "what are the essential properties for which a good interpolant can be derived"?

In this thesis we address the problem and characterize quality by means of two features which are intuitively relevant to model checking, and for which preliminary evidence has been provided in the literature: *logical strength* and *syntactic structure*.

Strength. A formula F_1 is said to be *stronger* than F_2 if $F_1 \models F_2$ (resp. F_2 is *weaker* than F_1). Since interpolants are inherently overapproximations, a stronger or weaker interpolant is expected to drive the verification process in terms of a finer or coarser approximation.

The works of [DKPW10, JM05] remark that interpolants of different strength can be beneficial in different verification frameworks. On one side, [JM05] emphasizes the usefulness of logically strong interpolants when approximating a transition relation by means of interpolation and bounded model checking, with direct application to predicate abstraction. On the other side, [DKPW10] provides examples where weak interpolants lead to a faster convergence of model checking. [DKPW10] and [Wei12] in particular offer an adequate framework to conduct an investigation of interpolant strength: they present in fact the *labeled interpolation systems*, a parametric algorithm which allows to systematically generate interpolants of different strength from the same proof.

Structure. Besides semantic features like logical strength, syntactic features are also relevant to verification.

A first one is the presence of *quantifiers*: most of the approaches presented in §3.1 rely on quantifier-free interpolants, to guarantee that the satisfiability checks involving interpolants are complete. Whenever quantifier-free interpolants cannot be generated, it might help to minimize the amount of quantifiers, as discussed in [HKV12].

[AM13] also remarks the importance of structure, by showing the effectiveness of interpolants built as conjunctions of linear inequalities over program properties, for programs implementing linear arithmetic operations.

Size (measured by the number of propositional connectives) is another reasonable candidate, since generating, storing and using smaller and less redundant formulae naturally require fewer computational resources. Supporting evidence is given in this sense by [CLV13]: the authors show that compact interpolants are helpful in the context of hardware unbounded model checking, realized by means of the method of [McM03]. The usefulness of small interpolants is also intuitively clear for the function summarization based approaches considered in [AGC12, SFS12b, SFS11], where interpolants correspond to summaries that are used multiple times in subsequent verification attempts.

3.3 Generation of Interpolants

In this section we introduce Craig interpolation and discuss the generation of interpolants. We begin by providing notation and some basic definitions, then move to illustrate a number of state-of-the-art approaches to interpolation in first order theories, focusing on algorithms that generate interpolants from proofs of unsatisfiability.

3.3.1 Craig Interpolation

Interpolation is considered in the context of standard first order logic. We assume countable sets of individual variables (x, y, z), function (f, g) and predicate (P, Q) symbols. A function symbol of 0-arity is called a constant (a, b, c), while a predicate symbol of 0-arity corresponds to a propositional variable (o, p, q, r, including for convenience the logical constants $\top, \bot$). A term is a constant or is built from function symbols and individual variables ($f(c, g(x))$); an atom is a propositional variable or is built from predicate symbols and terms ($P(x, f(c))$). A formula is built from atoms, propositional connectives, and quantifiers; in the rest of the chapter we

will denote formulae by $C, D, E, F, \ldots$. A formula is called *closed*, or a *sentence*, if it has no free individual variables; terms and formulae are *ground* if they have no occurrences of individual variables.

Given a formula F whose free variables are $x_1, \ldots, x_m$, $\forall F$ is the *universal closure* of F, that is $\forall x_1 \ldots \forall x_m\, F$; similarly, $\exists F$ is the *existential closure* of F, that is $\exists x_1 \ldots \exists x_m\, F$.

We call a *signature* Σ a set of function and predicate symbols. A *first order theory* consists of a signature together with a set of sentences (*theory axioms*) which fix the semantics of the symbols in the signature; in the following we consider theories with the equality predicate $=$. Theories which are relevant to formal verification include equality with uninterpreted functions, linear arithmetic, bit-vectors, lists and other data structures (more details are given in §3.3.5). For example, the theory of equality and uninterpreted functions has as signature all the function symbols and the equality predicate $=$; the axioms that define the semantics of equality are reflexivity, symmetry, transitivity and congruence.

When focusing on some particular formula F, we use Σ_F to denote the set of function and predicate symbols occurring in F, and $\mathscr{L}_F$ (the *language* of F), to denote the set of all formulae built on Σ_F. We equivalently represent the negation of a formula F as $\overline{F}$ or $\neg F$.

Inference Systems and Proofs. An *inference system* is a set of inference rules; an *inference rule* is an $n+1$-ary relation on formulae ($n \geq 0$), usually written as $\frac{C_1 \quad \cdots \quad C_n}{C}$, where $C_1, \ldots, C_n$ are the *premises* and C the *conclusion*. An *axiom* C is the conclusion of an inference with 0 premises; it will be represented without the bar line as C. An example of inference system, introduced in §2.1.1.2, is the resolution system, which has the resolution rule as unique inference rule.

A formula F is *provable* ($\vdash F$) in an inference system if there exists a proof Π of F: Π is a finite tree obtained by applications of the inference rules, such that the root is F and the leaves are axioms; an inner node C with parents $C_1, \ldots, C_n$ represents the conclusion C of an inference with premises $C_1, \ldots, C_n$. F is provable *from assumptions* $F_1, \ldots, F_n$ ($F_1 \wedge \cdots \wedge F_n \vdash F$) if there exists a proof where every leaf is an axiom or one the formulae $F_1, \ldots, F_n$. A *refutation* is a proof of $\bot$. We refer to a *subproof* as subtree of a proof, containing *subleaves* and a *subroot*.

A formula F is a *logical consequence* of $F_1, \ldots, F_n$ ($F_1 \wedge \cdots \wedge F_n \models F$) if every model of $F_1, \ldots, F_n$ is also a model of F; in particular, F is *valid* ($\models F$) if every interpretation is a model. When dealing with a first order theory $\mathscr{T}$, we will use $\models_{\mathscr{T}}$ to denote logical consequence w.r.t. the models of the axioms of $\mathscr{T}$. In some cases it will be convenient to use $F_1 \Rightarrow F_2$ in place of $F_1 \models F_2$, and $F_1 \Leftrightarrow F_2$ in place of

$F_1 \models F_2$ and $F_2 \models F_1$.

An inference system is *sound* if, for any $F, F_1, \ldots, F_n$, $F_1 \wedge \cdots \wedge F_n \vdash F$ entails $F_1 \wedge \cdots \wedge F_n \models F$.

Craig Interpolation. Craig's interpolation theorem was first presented in [Cra57a, Cra57b]:

Theorem 3.3.1 (Craig's Interpolation Theorem)**.** *Let A and C be first order formulae such that $A \rightarrow C$ is valid. Then, there exists an "intermediate" formula I s.t.:*

$$A \models I \qquad\qquad I \models C \qquad\qquad \mathscr{L}_I \subseteq \mathscr{L}_A \cap \mathscr{L}_C$$

I is an interpolant *for $A \rightarrow C$.*

A slightly different definition of interpolant, which we adopt, has been later introduced by [McM03] in the context of model checking:

Definition 3.3.1 (Craig Interpolant)**.** Let A and B be first order formulae such that $A \wedge B$ is unsatisfiable. An *interpolant* for $A \wedge B$ is a formula I s.t.:

$$A \models I \qquad\qquad I \wedge B \models \bot \qquad\qquad \mathscr{L}_I \subseteq \mathscr{L}_A \cap \mathscr{L}_B$$

We will alternatively use the notation $A \models I$ and $B \models \overline{I}$. With respect to the original formulation, I is an interpolant for $A \rightarrow \neg B$; I is implied by A, unsatisfiable with B and defined on the common symbols of A and B. Craig interpolation can also be defined in presence of a first order theory $\mathscr{T}$, so that inconsistency and implication must hold modulo the axioms of the theory:

Definition 3.3.2 (Craig Interpolant Modulo Theory)**.** Let A and B be first order formulae such that $A \wedge B$ is unsatisfiable in $\mathscr{T}$. A *theory interpolant* for $A \wedge B$ is a formula I s.t.:

$$A \models_{\mathscr{T}} I \qquad\qquad I \wedge B \models_{\mathscr{T}} \bot \qquad\qquad \mathscr{L}_I \subseteq \mathscr{L}_A \cap \mathscr{L}_B$$

[YM05] relaxes the constraint on $\mathscr{L}_I$ in presence of a theory $\mathscr{T}$, allowing the *interpreted* symbols of $\mathscr{T}$ (that is, the symbols whose semantics is specified by the axioms of $\mathscr{T}$) to appear in the interpolants, even if they do not appear either in A or in B; an interpolant I must then satisfy:

$$\mathscr{L}_I \subseteq (\mathscr{L}_A \cap \mathscr{L}_B) \cup \mathscr{L}_{\mathscr{T}}$$

where $\mathscr{L}_{\mathscr{T}}$ is the language of the axioms of $\mathscr{T}$. The choice is justified by showing that there are theories (e.g. the theory of lists) where the existence of an interpolant for any unsatisfiable $A \wedge B$ is not guaranteed if interpreted symbols are not always allowed in $\mathscr{L}_I$.

Symbols and Formulae. Symbols in the context of interpolation are distinguished depending on where they appear in a given $A \wedge B$.

We define $\Sigma_{AB} = \Sigma_A \cap \Sigma_B$ as the set of symbols occurring both in A and B and take $\mathscr{L}_{AB} = \mathscr{L}_A \cap \mathscr{L}_B$. The signature symbols of Σ_{AB} are called *AB-common*. Signature symbols occurring only in $\Sigma_A \setminus \Sigma_{AB}$ are *A-local*, and symbols occurring only in $\Sigma_B \setminus \Sigma_{AB}$ are *B-local*. A symbol that is not common is called local.

Terms, atoms, formulae are called *clean* if they contain only common symbols (clean formulae are thus in $\mathscr{L}_{AB}$), otherwise they are *dirty*: *A-dirty* if they contain only A-local and common symbols, but at least one A-local symbol; *B-dirty* if they only contain B-local and common symbols, but at least one B-local symbol. Terms, atoms and formulae, which are clean, A-dirty, or B-dirty, are also called *AB-pure*; if they are dirty but they contain at least one A-local and one B-local symbol, they are called *AB-mixed*: for example, if c is an A-local symbol and d is B-local, then the atom $c = d$ is AB-mixed.

When dealing with formulae in the form of clauses, we will make use of a *restriction operator* $|_\Sigma$, for a certain signature Σ. The application of $|_\Sigma$ to a clause C yields a clause $C|_\Sigma$, corresponding to the disjunction of the literals of C that are in Σ. We set $C|_A = C|_{\Sigma_A \setminus \Sigma_{AB}}$ and say that $C|_A$ is restricted to the A-local symbols of C. Similarly, we write $C|_B = C|_{\Sigma_B \setminus \Sigma_{AB}}$ and $C|_{AB} = C|_{\Sigma_{AB}}$, where $C|_B$ and $C|_{AB}$ are restricted to the B-local and AB-common symbols of C, respectively. Hence, a clause C can be written as $C = C|_B \vee C|_A \vee C|_{AB}$.

3.3.2 Interpolation Systems

Several techniques might exist to obtain interpolants from an unsatisfiable conjunction $A \wedge B$, depending on the language on which the two formulae are defined; we refer in general to these techniques as *interpolation systems*.

Definition 3.3.3 (Interpolation System)**.** An *interpolation system* Itp_S is a function that, given an unsatisfiable $A \wedge B$, returns a Craig interpolant for $A \wedge B$.

Various interpolation systems are known in the literature. The work of [RSS07] allows to compute interpolants using linear programming procedures for constraint solving; [SNA12] adopts a machine learning perspective, where interpolants are viewed as classifiers and can be generated by means of standard classification techniques; [CIM12] proposes a technique that incrementally generates a interpolant in disjunctive normal form in propositional logic; even verification methods like IC3 [Bra11] can be viewed as computing interpolants.

3.3.2.1 Proof-Based Interpolation Systems

A class of interpolation systems, which is particularly interesting for its applications to model checking, is characterized by the extraction of interpolants for $A \wedge B$ from a proof of unsatisfiability of $A \wedge B$. This approach grants two important benefits: the generation can be achieved in linear time with respect to the proof size and interpolants themselves only contain information relevant to determine the unsatisfiability of $A \wedge B$.

Proof-based systems recursively generate an interpolant for an unsatisfiable $A \wedge B$ from a refutation of $A \wedge B$. These procedures initially compute *partial interpolants* for (some of) the leaves in the refutation of $A \wedge B$, then, following the refutation structure, generate partial interpolants for (some of) the inner nodes, relying on the partial interpolants computed for the inference premises. The partial interpolant of $\perp$ corresponds to the overall interpolant for $A \wedge B$.

The precise method for obtaining an interpolant depends on the first order theory in which A and B are defined. Most of the proof-based interpolation systems produce a *quantifier-free* formula whenever A and B are themselves quantifier-free. The absence of quantifiers is an important requirement to make sure that satisfiability checks involving interpolants are complete and efficient. Note that Craig's theorem does not guarantee the existence of quantifier-free interpolants for every theory; for example the theory of array does not admit quantifier-free interpolation [KMZ06].

[McM03] introduces an interpolation algorithm for propositional logic, and [McM04a] extends it to generate interpolants in the combined theory of uninterpreted functions and linear arithmetic. [McM04a] builds over the previous work by [Pud97], which shows how to compute interpolants from resolution refutations in propositional logic and for systems of inequalities in linear arithmetic. The frameworks of [Kra97, Hua95] independently present the same system as [Pud97] for propositional logic. More recently, [DKPW10] has introduced a method that generalizes [Pud97, McM03], by analyzing the logical strength of interpolants. The work of [DKPW10] has been extended by [Wei12] to interpolation in the propositional hyper-resolution system. [McM03, Pud97, Kra97, Hua95, DKPW10, Wei12] are discussed in §3.3.4.

Promising results have been achieved also in the context of SMT, where several ad-hoc interpolation systems have been proposed for conjunctions of atoms in individual theories and combinations of them. We recall the work of [CGS08], which addresses linear arithmetic, difference logic and combinations of convex theories; another technique for theory combination is presented in [GKT09]. [CGS09] discusses UTVPI, [LT08] covers linear arithmetic, [FGG+09] deals with equality and uninterpreted functions. Presburger arithmetic (in its original formulation) instead

does not admit quantifier-free interpolants; however the addition of stride predicates to the language makes the theory quantifier-free interpolating [Pug91, BKRW11, JCG08]. [KLR10] provides an efficient alternative to [BKRW11], but at the cost of preventing the use of standard SMT solving techniques; an extension of the signature of linear integer arithmetic allows [GLS12] to improve on [KLR10] and [BKRW11]. Interpolation for the theory of bit-vectors is discussed in [Gri11], which builds on the works of [KW07] and [KW09]. Interpolation is also possible for several other data structures, as shown in [KMZ06]. [CGS10] covers linear arithmetic, difference logic, UTVPI, and proposes an algorithm to deal with combination of theories in presence of the delayed theory combination framework of [BCF+09, BBC+05b]; combination of theories is also addressed in [YM05], which presents a technique based on the Nelson-Oppen framework [NO79] that computes interpolants for the combination by employing interpolation procedures for the individual theories.

[YM05] and [CGS10] are of particular interest for SMT since they also show how to compute interpolants for formulae with an arbitrary boolean structure by combining an interpolation system for conjunctions of atoms in a theory with an interpolation system for propositional logic; in particular [YM05] builds on [Pud97], while [CGS10] on [McM04a]. This approach has the advantage of being modular and flexible: it reduces in fact the extension of interpolant generation to an additional theory to providing a procedure that is able to generate theory interpolants for conjunctions of ground literals in that theory. More details are given in §3.3.5.

Compared to the interpolation systems mentioned above, the methods described in [KV09] and [HKV12] give a more general algorithm that can be used with arbitrary first order calculi and inference systems. This algorithm is, however, restricted to proofs with a particular structure, called local proofs [KV09] or split proofs [JM06]. [KV09, HKV12] are discussed in §3.3.3.

3.3.2.2 Interpolation and Quantifier Elimination

A first order theory $\mathscr{T}$ is said to admit *quantifier elimination (QE)* if there exists an algorithm able to transform a quantified formula F, defined on $\mathscr{T}$, to a logically equivalent F' without quantifiers. Quantifier elimination procedures, applied for the purpose of exact image computation in model checking [BCM+92, McM92, WBCG00, ABE00], have been gradually replaced by interpolation as a more efficient alternative, since its introduction by [McM03] in the area of verification.

Although computationally more expensive, the generation of quantifier-free interpolants could in principle be achieved by means of QE, rather than by means of proof-based systems as described in §3.3.2.1, if the formulae under account are propositional or defined on a theory that admits QE. Whenever an unsatisfiable

$A \wedge B$ contains individual (resp. propositional) variables as the only local symbols, the strongest and the weakest interpolant can in fact be expressed by the formulae $\exists x_{x \in \Sigma_A \setminus \Sigma_{AB}} A$ (resp. $\exists p_{p \in \Sigma_A \setminus \Sigma_{AB}} A$) and $\forall x_{x \in \Sigma_B \setminus \Sigma_{AB}} \neg B$ (resp. $\forall p_{p \in \Sigma_B \setminus \Sigma_{AB}} \neg B$), that is by quantifying away the local variables themselves (see, for example, [KLR10] and [EKS08]); it is possible at this point to resort to a QE algorithm in order to obtain a quantifier-free interpolant.

Quantified Propositional Logic. A simple procedure, that applies to formulae with arbitrary propositional structure, relies on the *expansions* $\exists p\, F \Longleftrightarrow F[\top/p] \vee F[\bot/p]$ and $\forall p\, F \Longleftrightarrow F[\top/p] \wedge F[\bot/p]$, where $F[\top/p]$ denotes the formula obtained by replacing the occurrences of p in F by $\top$; in case of a formula with n quantifiers, they are eliminated iteratively from the innermost to the outermost. Data structures such as BDDs (see §1.1), introduced for representation and manipulation of propositional functions, can be used to perform this kind of computation, although their application suffers from heavy memory requirements.

A different approach is available when F is in CNF. A universal quantifier $\forall p\, F$ can be eliminated by simply removing the occurrences of p from F. In case of an existential quantifier $\exists p\, F$, the resolution rule can be exploited in order to get rid of p: all pairs of clauses where p has opposite polarity are resolved on p, the resolvents are added to F, and finally all the clauses containing p are discarded.

Other more sophisticated algorithms realize QE by eliminating quantified variables iteratively in a certain order. [WBCG00] discusses QE as a means to perform image computation in the context of a symbolic model checking algorithm based on an extension of BDDs; similarly, [ABE00] combines SAT solving and the use of ad-hoc data structures in order to perform reachability analysis. The main limitation of such iterative techniques is that their performance critically depends on the order in which variables are eliminated, possibily leading to formulae that grow exponentially in the number of the quantified variables.

Another possibility is to rely on SAT solving for an enumeration and collection of the satisfying assignments, incrementally adding blocking constraints that exclude the solutions previously found; building on these assignments, a quantifier-free formula equivalent to the original one can be obtained. [McM02] shows how the standard CDCL approach can be extended to generate, from a universally quantified formula, an equivalent formula in CNF from which the universal quantifiers are easily removed. [GGA04] adopts a circuit-based approach in order to cover a larger part of the solution space compared to a simple enumeration of assignments, thus reducing the overall amount of required enumeration steps; the same goal is aimed at by [JS05], which proposes an optimization based on lifting propositional

variables, as a means to generate minimal satisfying assignments. [BKK11] improves over [McM02] with an algorithm that produces a quantifier-free formula in CNF, by joining enumeration of assignments and computation of shortest implicants (i.e., minimum satisfying assignments). [GM12] introduces a compositional technique for the elimination of existential quantifiers from formulae in CNF. Based on the resolution system, it adds resolvents to the original formula until the quantified variables become redundant, so that the clauses containing them can be discarded leading to an equivalent quantifier-free formula.

First Order Theories. Most algorithms that perform QE in the context of first order theories have been developed for arithmetic. Fourier-Motzkin elimination [KS08] addresses systems of inequalities in LRA, and is extended by the Omega Test [Pug91] to linear integer arithmetic; both these techniques are applicable to conjunctions of theory literals, so that formulae with aribitrary structure must first be appropriately transformed. A different kind of methods relies on replacing existential quantification $\exists x\, F$, which corresponds to an infinite disjunction, with a finite disjunction $\bigvee_i F[x_i/x]$, where the x_i depend on the free variables of F: we recall Ferrante and Rackoff's [FR75] and Loos and Weispfenning's algorithms for LRA [LW93], and Cooper's algorithm for LIA [Coo72]. Moving from linear to non-linear arithmetic, a well-known approach to the theory of real closed fields builds on cylindrical algebraic decomposition [Col75], while QE on algebrically closed fields can be realized by means of Gröbner bases [CLO07]. [Nip08] presents QE procedures for dense linear orders, LRA and LIA, while [LNO06], [Mon10] and [Bjø10] adopt an SMT-oriented approach to perform QE for LRA: [LNO06] uses enumeration of assignments by an SMT solver to perform predicate abstraction and incremental refinement of approximations, [Mon10] combines "lazy" enumeration with QE over conjunctions of literals, [Bjø10] proposes QE procedures for LRA and LIA as theory solvers.

QE over bit-vectors can be reduced to QE over the constituent propositional variables, as well as to QE over LIA [KS08]. [JC11] directly targets bit-vector arithmetic as modular arithmetic on integers, and presents a QE procedure for linear modular equalities and disequalities, later extended to also deal with inequalities [JC13].

3.3.3 Interpolation in Arbitrary First Order Theories

[KV09] introduces an interpolation system to generate interpolants from a class of proofs, called *local proofs*, characterized by syntactic requirements at the level of individual inferences; for a given local proof Π, an interpolant can be obtained as

a propositional combination of some of the formulae in Π. The algorithm has the advantage of being independent from the particular theory under account, allowing to deal with arbitrary first order theories and inference systems, although it is not applicable to arbitrary proofs. A local proof does not necessarily exist for any formula provable from a set of axioms and assumptions. However, when the presence of non-local inferences only depends on uninterpreted constants, as it is usually the case in bounded model checking, non-local proofs can be turned into local ones, by existentially quantifying the constants [HKV12]. This kind of transformation comes thus at the price of allowing quantifiers in the interpolants. [KV09] proves instead that an extension of the quantifier-free superposition calculus with quantifier-free linear rational arithmetic guarantees local proofs.

[KV09, HKV12] address interpolation in presence of a first order theory $\mathscr{T}$ and a sound inference system. Fixed A and B, an AB-proof of a formula F is so defined:

Definition 3.3.4 (AB-Proof)**.** An *AB-proof* is a proof such that:

(AB1) For every leaf C: $A \models_{\mathscr{T}} \forall C$ and $C \in \mathscr{L}_A$ or $B \models_{\mathscr{T}} \forall C$ and $C \in \mathscr{L}_B$

(AB2) For every inference $\dfrac{C_1 \quad \cdots \quad C_n}{C}$: $\forall C_1, \ldots, \forall C_n \models_{\mathscr{T}} \forall C$.

An AB-proof of $\bot$ is called an *AB-refutation*.

The authors consider, within AB-proofs, the class of *local refutations*, where every inference is a *local inference*:

Definition 3.3.5 (Local Inference)**.** An inference $\dfrac{C_1 \quad \cdots \quad C_n}{C}$ is called *local* if it satisfies the following conditions:

(L1) Either $\{C_1, \ldots, C_n, C\} \subseteq \mathscr{L}_A$ or $\{C_1, \ldots, C_n, C\} \subseteq \mathscr{L}_B$

(L2) If $C_1, \ldots, C_n$ are clean, then C is clean as well.

Condition (L1) says that no inference contains both A-local and B-local symbols, (L2) that inferences do not introduce superfluous symbols.

The interpolation system generates an interpolant for $A \wedge B$ as a *propositional combination of clean formulae*, conclusions of inferences called *symbol eliminating*:

Definition 3.3.6 (Symbol Elimination)**.** An inference $\dfrac{C_1 \quad \cdots \quad C_n}{C}$ is called *symbol eliminating* if:

(J1) C is a clean leaf (i.e. $n = 0$), or

(J2) C is a clean conclusion and, for some i, C_i is dirty.

The idea is that the conclusion C eliminates dirty symbols present in the premises; if C is a leaf, either $\frac{A}{C}$ or $\frac{B}{C}$, so C is considered as eliminating a dirty symbol in A or B.

The framework of [KV09, HKV12] identifies in a proof Π an *A-subproof* Π' (resp. *B-subproof*) as follows:

- The last inference of Π' satisfies (J2) and, for some i, C_i is A-dirty (resp. B-dirty)
- Π' is a maximal subproof s.t. every formula belongs to $\mathscr{L}_A$ (resp. $\mathscr{L}_B$)

Inferences where all formulae are clean can equivalently belong to an A-subproof or to a B-subproof; A-dirty formulae must belong to A-subproofs, while B-dirty to B-subproofs.

As a consequence of this characterization, the proof Π can be partitioned, starting from the root up to the leaves, into an alternation of A- and B- subproofs: in fact, given an A- (resp. B-)subproof, its leaves are either leaves of Π or conclusions of symbol eliminating inferences, representing the roots of B- (resp. A-)subproofs.

The interpolation system recursively constructs an interpolant I for a formula $A \wedge B$ from a local refutation of $A \wedge B$, according to the partitioning. I is a propositional combination of partial interpolants, each associated with a clean leaf or with the clean root of a maximal subproof, as illustrated in Table 3.1.

Table 3.1. Local Proofs Interpolation System.

Leaf:	$C\,[I]$
	$I = \begin{cases} C & \text{if } A \models_{\mathscr{T}} C \\ \overline{C} & \text{if } B \models_{\mathscr{T}} C \end{cases}$
Inner node:	$\dfrac{C_1\,[I_{C_1}] \quad \cdots \quad C_m\,[I_{C_m}] \quad D}{C\,[I]}$
	$I = \begin{cases} \bigwedge(C_i \vee I_{C_i}) \wedge \bigvee \overline{C_i} & \text{for an } A\text{-subproof} \\ \bigwedge(C_i \vee I_{C_i}) & \text{for a } B\text{-subproof} \end{cases}$

In the table, $C[I]$ denotes that the formula C has a partial interpolant I. The partial interpolant of a clean leaf C is the leaf itself, positive if $A \models_{\mathscr{T}} C$, negated if $B \models_{\mathscr{T}} C$. The partial interpolant of a clean inference conclusion C, root of a maximal subproof, is either $\bigwedge(C_i \vee I_{C_i}) \wedge \bigvee \overline{C_i}$, in case of an A-subproof, $\bigwedge(C_i \vee I_{C_i})$, in

case of a B-subproof. Note that the partial interpolant I for an inference conclusion is computed from the interpolants associated with the clean premises $C_1, \ldots, C_m$, whereas the dirty premises D are not taken into account.

3.3.4 Interpolation in SAT

Among the most commonly used interpolation systems for propositional logic are the system independently developed by Pudlák [Pud97], Huang [Hua95] and Krajíček [Kra97], and the one by McMillan [McM03], all of them generalized by the labeled interpolation systems (LISs) of [DKPW10]; the LISs additionally allow to systematically generate interpolants of different strength from the same proof, making them a suitable instrument in our research on logical strength.

These methods generate interpolants from refutations obtained in the *resolution system* (introduced in §2.1.1.2), a sound and complete inference system for propositional logic based on the *resolution rule*:

$$\frac{C^+ \vee p \qquad C^- \vee \overline{p}}{C^+ \vee C^-}$$

C^+, C^- are clauses, $C^+ \vee p$ and $C^- \vee \overline{p}$ are the *antecedents*, $C^+ \vee C^-$ the *resolvent* and p is the *pivot* of the resolution step. For brevity, to denote that a propositional variable p is A-local, B-local or AB-common we will write $p \in A, p \in B, p \in AB$.

3.3.4.1 McMillan and Pudlák's Interpolation Systems

Tables 3.2 and 3.3 respectively illustrate the interpolation systems of McMillan [McM03] and Pudlák [Pud97, Hua95, Kra97]; Table 3.4 shows a system "dual" to that of McMillan, first discussed in [DKPW10]. We refer to them as $Itp_M, Itp_P, Itp_{M'}$. In the tables, $C|_{AB}$ denotes the restriction of clause C to AB-common variables; $C \in A$ (resp. $C \in B$) means that C is one of the clauses of the CNF formula A (resp. B). By $C[I]$ we represent that clause C has a partial interpolant I. I^+, I^- and I are the partial interpolants respectively associated with the two antecedents and the resolvent of a resolution step.

3.3.4.2 The Labeled Interpolation Systems

[DKPW10] generalizes the abovementioned interpolation systems by introducing the notion of *labeled interpolation system* (LIS), focusing on the concept of *logical strength*. If F_1 and F_2 are two formulae, and $F_1 \models F_2$, then we say that F_1 is *stronger* than F_2 (F_2 is *weaker* than F_1).

Table 3.2. McMillan's interpolation system Itp_M.

Leaf:	$C\,[I]$
$I = \begin{cases} C\|_{AB} & \text{if } C \in A \\ \top & \text{if } C \in B \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \\ I^+ \wedge I^- & \text{if } p \in B \text{ or } p \in AB \end{cases}$	

Table 3.3. Pudlák's interpolation system Itp_P.

Leaf:	$C\,[I]$
$I = \begin{cases} \bot & \text{if } C \in A \\ \top & \text{if } C \in B \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \\ I^+ \wedge I^- & \text{if } p \in B \\ (I^+ \vee p) \wedge (I^- \vee \overline{p}) & \text{if } p \in AB \end{cases}$	

Table 3.4. McMillan′'s interpolation system $Itp_{M'}$.

Leaf:	$C\,[I]$
$I = \begin{cases} \bot & \text{if } C \in A \\ \neg C\|_{AB} & \text{if } C \in B \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \text{ or } p \in AB \\ I^+ \wedge I^- & \text{if } p \in B \end{cases}$	

The system of [DKPW10] relies on a so-called *labeling* L. Given a refutation of $A \wedge B$, L assigns a *label* $L(p, C)$ among $\{\bot, a, b, ab\}$ to each variable p in each clause C in the refutation; we assume that no clause is tautological (i.e., it has both a literal and its negation), so assigning a label to variables or literals is equivalent. The set of possible labelings is restricted by ensuring that A-local variables have label a and B-local variables label b; freedom is left for AB-common variables to be labeled

either a, b or ab. A label $\perp$ means that p does not appear in a clause. Labels are independently set for all variables occurrences in the leaves of the refutation, and recursively computed for the inner nodes. The label of a variable p in the resolvent of a resolution step is computed from the labels of p in the antecedents as $L(p, C^+ \vee C^-) = L(p, C^+ \vee p) \sqcup L(p, C^- \vee \overline{p})$. If a total order $b \preceq ab \preceq a \preceq \perp$ is set on the labels, then $\sqcup$ represents the join operator of the lattice determined by the total order, shown in Figure 3.1. For example, $a \sqcup b = ab$ and $b \sqcup \perp = b$. The labels of pivots are also computed in this way.

$$\begin{tikzcd}[column sep=small] & ab \arrow[dl, dash] \arrow[dr, dash] & \\ a \arrow[dr, dash] & & b \arrow[dl, dash] \\ & \perp & \end{tikzcd}$$

Figure 3.1. The Hasse Diagram of $\sqcup$.

A LIS is defined as a procedure Itp_L (shown in Table 3.5) that, given $A \wedge B$, a refutation Π and a labeling L, outputs a partial interpolant $I_L(C)$ for any clause C in Π; this depends on the clause being in A or B (if leaf) and on the label of the pivot associated with the resolution step (if inner node). $I_L = I_L(\perp)$ represents the interpolant for $A \wedge B$. In Table 3.5, $p : \alpha$ indicates that variable p has label α.

Table 3.5. Labeled interpolation system Itp_L.

Leaf:	$C\,[I]$
	$I = \begin{cases} C\|_b & \text{if } C \in A \\ \neg C\|_a & \text{if } C \in B \end{cases}$
Inner node:	$\dfrac{C^+ \vee p : \alpha\,[I^+] \qquad C^- \vee \overline{p} : \beta\,[I^-]}{C^+ \vee C^-\,[I]}$
	$I = \begin{cases} I^+ \vee I^- & \text{if } \alpha \sqcup \beta = a \\ I^+ \wedge I^- & \text{if } \alpha \sqcup \beta = b \\ (I^+ \vee p) \wedge (I^- \vee \overline{p}) & \text{if } \alpha \sqcup \beta = ab \end{cases}$

Itp_L generalizes Itp_M, Itp_P and $Itp_{M'}$; it can be seen, by comparing Table 3.5 with Tables 3.2, 3.3, 3.4, that the three systems are obtained as special cases by labeling all the occurrences of AB-common variables with b, ab and a, respectively. Consider for example Itp_M. Given a leaf $C \in A$, C can only contain A-local and AB-common variables; the first ones must be labeled a, while we choose to label the second ones b. Similarly happens if $C \in B$. Thus, $C|_b$ corresponds to $C|_{AB}$

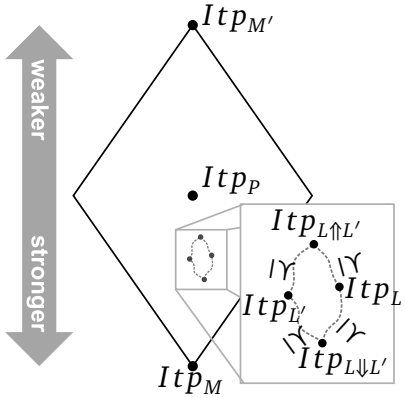


Figure 3.2. Lattice of labeled interpolation systems.

when $C \in A$, and $\neg C|_a$ to $\top$ when $C \in B$. Given a resolution step with pivot p in the refutation, the only possible cases are $\alpha \sqcup \beta = a$ and $\alpha \sqcup \beta = b$; the first case corresponds to $p \in A$, the second to $p \in B$ or $p \in AB$.

The total order over labels can be extended to a partial order over labelings: $L \preceq L'$ if, for every clause C and variable p in C, $L(p,C) \preceq L'(p,C)$. This allows to directly compare the logical strength of the interpolants produced by two systems. In fact, for any refutation Π of a formula $A \wedge B$ and labelings L, L' such that $L \preceq L'$, we have $I_L \models I_{L'}$ and we say that the system Itp_L is *stronger* than $Itp_{L'}$.

Two interpolation systems Itp_L and $Itp_{L'}$ can generate new systems $Itp_{L\Uparrow L'}$ and $Itp_{L\Downarrow L'}$ by combining the labelings L and L' in accordance with the relation $\preceq$: $(L \Uparrow L')(p,C) = \max_{\preceq}\{L(p,C), L'(p,C)\}$ and, vice versa, $(L \Downarrow L')(p,C) = \min_{\preceq}\{L(p,C), L'(p,C)\}$. The collection of LISs over a refutation, together with the order $\preceq$ and the operators $\Uparrow, \Downarrow$, represent a *complete lattice*, where Itp_M is the greatest element and $Itp_{M'}$ is the least, with Itp_P being in between (see Figure 3.2).

Figure 3.3 shows the computation of interpolants by means of Itp_M and $Itp_{M'}$ on an unsatisfiable formula $A \wedge B$, where $A = (p \vee \overline{q}) \wedge r$ and $B = (\overline{p} \vee \overline{r}) \wedge q$. The three propositional variables p, q, r are AB-common: M assigns label b to all of them, while M' assigns a. Since $M \preceq M'$, we have $I_M \models I_{M'}$, where $I_M = (p \vee \overline{q}) \wedge r$ and $I_{M'} = (p \wedge r) \vee \overline{q}$.

$$\dfrac{\dfrac{\dfrac{p\overline{q}\,[p \vee \overline{q}] \qquad \overline{pr}\,[\top]}{\overline{qr}\,[p \vee \overline{q}]} \qquad r\,[r]}{\overline{q}\,[(p \vee \overline{q}) \wedge r]} \qquad q\,[\top]}{\bot\,[(p \vee \overline{q}) \wedge r]} \qquad \dfrac{\dfrac{\dfrac{p\overline{q}\,[\bot] \qquad \overline{pr}\,[p \wedge r]}{\overline{qr}\,[p \wedge r]} \qquad r\,[\bot]}{\overline{q}\,[p \wedge r]} \qquad q\,[\overline{q}]}{\bot\,[(p \wedge r) \vee \overline{q}]}$$

Figure 3.3. Computation of interpolants with $Itp_M, Itp_{M'}$.

The Labeled Interpolation Systems for Hyper-Resolution. The framework of LISs is generalized by [Wei12] from the resolution system to the hyper-resolution system. The *hyper-resolution (HR) system* is an inference system that uses a single inference rule, called the *hyper-resolution rule*:

$$\frac{\overline{p_1} \vee \cdots \vee \overline{p_{n-1}} \vee E \qquad D_1 \vee p_1 \qquad \ldots \qquad D_{n-1} \vee p_{n-1}}{\bigvee D_i \vee E}$$

where $p_1, \ldots, p_n$ are the *pivots*, $D_1, \ldots, D_{n-1}, E$ are clauses and $n \geq 2$. In the following we will refer for brevity to the resolvent as C and to the n antecedents as $C_0, \ldots, C_{n-1}$, setting $C_0 = \overline{p_1} \vee \cdots \vee \overline{p_{n-1}} \vee E$ and $C_i = D_i \vee p_i$.

The labeling mechanism is naturally extended to deal with the presence of more than two antecedents: given an HR step, the label $L(p, C)$ of a variable in the resolvent is computed from the labels in the antecedents as $L(p, C) = L(p, C_0) \sqcup \cdots \sqcup L(p, C_{n-1})$, according to the diagram of Figure 3.1. Labels for the pivots are computed in a similar way.

The notion of LIS is generalized to that of *HR-LIS*, and the procedure to compute interpolants is illustrated in Table 3.6:

Table 3.6. HR labeled interpolation system $HR\text{-}Itp_L$.

Leaf:	$C\,[I]$
	$I = \begin{cases} C\|_b & \text{if } C \in A \\ \neg C\|_a & \text{if } C \in B \end{cases}$
Inner node:	$\dfrac{\overline{p_1} \vee \cdots \vee \overline{p_{n-1}} \vee E\,[I_{C_0}] \qquad D_1 \vee p_1\,[I_{C_1}] \qquad \ldots \qquad D_{n-1} \vee p_{n-1}\,[I_{C_{n-1}}]}{\bigvee D_i \vee E\,[I]}$
	$I = \begin{cases} I_{C_0} \vee \bigvee_{i=1}^{n-1} I_{C_i} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = a \\ I_{C_0} \wedge \bigwedge_{i=1}^{n-1} I_{C_i} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = b \\ \begin{cases} (I_{C_0} \vee \bigvee \overline{p_i}) \wedge \bigwedge_{i=1}^{n-1} (I_{C_i} \vee p_i) \\ (I_{C_0} \wedge \bigwedge p_i) \vee \bigvee_{i=1}^{n-1} (I_{C_i} \wedge \overline{p_i}) \end{cases} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = ab \end{cases}$

A necessary condition for the generation of interpolants by means of the HR-LISs is that in each HR step of a refutation all the pivots must be labeled uniformly, that is $\forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = \alpha$, for a certain label α. However, this is not a heavy restriction, since, as shown in [Wei12], it is always possible to split an HR step into two or more steps, until a uniform labeling is achieved; in the worst case, an HR step can be split into a sequence of resolution steps.

Note from Table 3.6 that, whenever $n = 2$, $HR\text{-}Itp_L$ reduces to Itp_L; if the pivot label is ab, the additional interpolant $(I^- \wedge p) \vee (I^+ \wedge \overline{p})$ can be generated, which is logically equivalent to $(I^- \vee \overline{p}) \wedge (I^+ \vee p)$.

3.3.5 Interpolation in SMT

After covering in §3.3.4 the topic of interpolation in propositional logic (for the resolution and hyper-resolution systems), in this section we address interpolation in satisfiability modulo theories.

We introduce in §3.3.5.1 background notions on SMT, and discuss interpolation in the context of lazy SMT solving in §3.3.5.2. We focus on the approaches of [YM05] and [CGS10], which extend the propositional systems of [Pud97] and [McM03] to first order theories, by integrating them with procedures that compute interpolants for conjunctions of atoms in a theory; we reformulate the interpolation systems of [YM05, CGS10] to provide a uniform notation with §3.3.5.1, §3.3.3, and to ease the comparison with a more general method which we will present in §3.4.

3.3.5.1 Lazy SMT Solving

SMT extends the expressiveness of purely propositional logic by allowing reasoning in *first order theories*, where the semantics of some function and predicate symbols is fixed a priori; *SMT($\mathcal{T}$)* denotes the problem of deciding the satisfiability, w.r.t. an underlying theory $\mathcal{T}$, of formulae containing propositional variables and atomic expressions in $\mathcal{T}$.

One of the most successful approaches to SMT($\mathcal{T}$) relies on efficiently combining engines that respectively deal with the propositional and the theory-specific aspects of reasoning. *Lazy SMT solving* [Seb07] consists of integrating a CDCL SAT solver with one or more *theory solvers*, algorithms that decide whether a conjunction of literals is satisfiable in a theory $\mathcal{T}$: the SAT solver treats theory atoms as if they were propositional variables, and enumerates the truth assignments satisfying the propositional abstraction of the input formula, while the theory solver checks the consistency, within the theory, of the sets of atoms corresponding to the assignments. If a conjunction of literals is unsatisfiable in $\mathcal{T}$, then its negation is valid and is called a *$\mathcal{T}$-lemma*: intuitively, $\mathcal{T}$-lemmata are formulae that encode facts valid in the theory $\mathcal{T}$.

Theories of interest in formal verification include equality and uninterpreted functions (EUF), dealing with atoms of the kind $f(d) = g(b,c)$; linear arithmetic over the rationals (LRA) or the integers (LIA), with atoms like $5c + 7d - 3e = 2$; difference logic logic (DL) and unit-two-variable-per-inequality ($UTVPI$), subtheories of linear arithmetic where atoms are of the kind $c - d \leq 5$; bit-vectors (BV, $c^{[32]} +_{32} d^{[32]}$); arrays (AX, $read(a,i) = read(write(b,j,e),i)$); lists ($LI$, $car(d) = car(cdr(e))$). Noteworthy SMT solvers include Z3 [dMB08], CVC4 [CVC] MathSAT [BBC+05a], OpenSMT [BPST10].

$$\dfrac{\dfrac{\dfrac{c \neq d \vee d \neq e \vee c = e \qquad c = d}{d \neq e \vee c = e} \qquad d = e}{c = e} \qquad c \neq e}{\bot}$$

Figure 3.4. An example of resolution refutation in SMT.

An SMT solver can be instrumented to generate, as a SAT solver, a *resolution refutation* of an unsatisfiable formula. Refutations in SMT are different from their propositional counterparts in that they contain both propositional variables and theory atoms, and the leaves are original clauses as well as $\mathcal{T}$-lemmata made of original predicates, generated by the prover during the solving process. Figure 3.4 shows a small refutation in the theory of EUF; note the presence of the lemma $c \neq d \vee d \neq e \vee c = e$, which is an instance of the transitivity axiom for equality.

3.3.5.2 Interpolation in Lazy SMT Solving

A natural approach to interpolation in lazy SMT is the one proposed in [YM05]. The authors present a generalization of Pudlák's method that can compute a *theory interpolant* (as for Definition 3.3.2) for a formula defined on a theory $\mathcal{T}$. Their recursive algorithm is still based on the notion of partial interpolant, with the difference that, in SMT refutations, partial interpolants are associated both with the original clauses of the problem and with the theory lemmata. In this context, we refer to partial interpolants also as *theory partial interpolants*.

The algorithm makes use of an external procedure that computes theory partial interpolants for the leaves in the refutation which correspond to theory lemmata; after that, Pudlák's method can be run as if the refutation was purely propositional, yielding the global theory interpolant for the input formula $A \wedge B$.

The approach of [YM05] provides a method to generate interpolants for a class of first order theories and combinations of them, by using as black boxes interpolation procedures for the component theories.

The interpolation system $\mathcal{T}$-Itp_P is illustrated in Table 3.7, where $TPI_P(C)$ denotes the theory partial interpolant for the theory lemma C, as returned by a dedicated procedure TPI_P. Note that in an SMT refutation a pivot p can be a propositional variable or stand for a theory atom; in analogy with LISs notation, $p \in A, p \in B, p \in AB$ respectively mean p is A-dirty, p is B-dirty, p is clean.

An approach similar to that of [YM05] is proposed in [CGS10], with the difference that [CGS10] builds on the interpolation system developed by McMillan [McM03] rather than that of Pudlák. The $\mathcal{T}$-Itp_M interpolation system is shown in Table 3.8.

Table 3.7. Pudlák's interpolation system $\mathscr{T}$-Itp_P.

Leaf:	$C\,[I]$
$I = \begin{cases} \bot & \text{if } C \in A \\ \top & \text{if } C \in B \\ TPI_P(C) & \text{if } C \text{ is a } \mathscr{T}\text{-lemma} \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \\ I^+ \wedge I^- & \text{if } p \in B \\ (I^+ \vee p) \wedge (I^- \vee \overline{p}) & \text{if } p \in AB \end{cases}$	

Table 3.8. McMillan's interpolation system $\mathscr{T}$-Itp_M.

Leaf:	$C\,[I]$
$I = \begin{cases} C\vert_{AB} & \text{if } C \in A \\ \top & \text{if } C \in B \\ TPI_M(C) & \text{if } C \text{ is a } \mathscr{T}\text{-lemma} \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \\ I^+ \wedge I^- & \text{if } p \in B \text{ or } p \in AB \end{cases}$	

Both [YM05] and [CGS10] assume that refutations do not contain *AB-mixed predicates*, characterized by the presence of both A-local and B-local symbols. The problem of AB-mixed predicates, as well as a proof manipulation approach to remove such predicates from refutations, will be discussed in §5.4.

3.4 Theory Labeled Interpolation Systems

The labeled interpolation systems, introduced in §3.3.4.2, are an effective instrument for the investigation of logical strength in model checking, since they allow to systematically generate interpolants of different strength from the same resolution refutation; however, their applicability is restricted to the ambit of propositional logic.

In this section we address such limitation by presenting in §3.4.1 a new frame-

work, the *theory labeled interpolation systems ($\mathcal{T}$-LISs)*, that extends LISs to satisfiability modulo theories, and show in §3.4.2 its instantiation in the theory of difference logic.

Soundness of SMT Interpolation. In §3.3.5.2 we discussed two interpolation systems that generate theory interpolants in the context of lazy SMT, [YM05] and [CGS10]; they respectively build upon Pudlák's [Pud97] and McMillan's [McM03] propositional systems by combining them with dedicated procedures to compute theory partial interpolants for theory lemmata.

An important feature, common to the two approaches, is that the soundness of the interpolation systems is proved by inductive reasoning: it is shown in fact that the partial interpolant $I(C)$ of each clause C in a refutation satisfies an invariant property, and that this property, at the level of the refutation root, reduces to the conditions that define Craig interpolants. Assuming an unsatisfiable $A \wedge B$ and a reference theory $\mathcal{T}$, in [YM05] a partial interpolant must satisfy:

$$\begin{aligned} &A \wedge \neg(C|_A \vee C|_{AB}) \models_{\mathcal{T}} I(C) \\ &B \wedge \neg(C|_B \vee C|_{AB}) \wedge I(C) \models_{\mathcal{T}} \bot \\ &\mathcal{L}_{I(C)} \subseteq \mathcal{L}_{AB} \cup \mathcal{L}_{\mathcal{T}} \end{aligned} \tag{3.2}$$

while in [CGS10]:

$$\begin{aligned} &A \wedge \neg(C|_A) \models_{\mathcal{T}} I(C) \\ &B \wedge \neg(C|_B \vee C|_{AB}) \wedge I(C) \models_{\mathcal{T}} \bot \\ &\mathcal{L}_{I(C)} \subseteq \mathcal{L}_{AB} \cup \mathcal{L}_{\mathcal{T}} \end{aligned} \tag{3.3}$$

It is easy to see that both sets of requirements reduce to the definition of interpolant for $C = \bot$.

These invariants also influence how theory partial interpolants for theory lemmata are computed by means of an ad-hoc procedure TPI. Consider a theory lemma C: C is a valid formula in $\mathcal{T}$, while $\neg C$ is unsatisfiable. Moreover, $\neg C$ can be represented as $\neg(C|_A \vee C|_{AB} \vee C|_B)$, depending on where the symbols of A, B appear in the literals of C. Note that a theory atom in C does not necessarily appear in A or in B, since it might have been introduced e.g. at solving time (see §5.4); still, it contains symbols originally present in A, B.

TPI receives as input the unsatisfiable conjunction of literals $\neg C$, splits it into an A part D_1 (including $\neg C|_A$) and a B part D_2 (including $\neg C|_B$), assigning the literals in $\neg C|_{AB}$ to either part, and computes a formula $I(C)$ for $D_1 \wedge D_2$ that satisfies:

$$\begin{aligned}
&D_1 \models_{\mathcal{T}} I(C) \\
&D_2 \wedge I(C) \models_{\mathcal{T}} \bot \\
&\mathcal{L}_{I(C)} \subseteq (\mathcal{L}_{D_1} \cap \mathcal{L}_{D_2}) \cup \mathcal{L}_{\mathcal{T}}
\end{aligned} \tag{3.4}$$

For an appropriate assignment of the literals in $\neg C|_{AB}$ to D_1 and D_2, (3.4) corresponds to:

$$\begin{aligned}
&\neg C|_A \wedge \neg C|_{AB} \models_{\mathcal{T}} I(C) \\
&\neg C|_B \wedge \neg C|_{AB} \wedge I(C) \models_{\mathcal{T}} \bot \\
&\mathcal{L}_{I(C)} \subseteq \mathcal{L}_{AB} \cup \mathcal{L}_{\mathcal{T}}
\end{aligned}$$

and to:

$$\begin{aligned}
&\neg(C|_A) \models_{\mathcal{T}} I(C) \\
&\neg C|_B \wedge \neg C|_{AB} \wedge I(C) \models_{\mathcal{T}} \bot \\
&\mathcal{L}_{I(C)} \subseteq \mathcal{L}_{AB} \cup \mathcal{L}_{\mathcal{T}}
\end{aligned}$$

$I(C)$ thus satisfies the requirements of partial interpolant defined in (3.2) and (3.3).

Soundness of LISs. Consider now the labeled interpolation systems. In [DKPW10] the soundness of a system Itp_L is proved by showing that, for each clause C and corresponding partial interpolant $I(C)$ in the refutation of an unsatisfiable conjunction $A \wedge B$, the following invariant holds:

$$\begin{aligned}
&A \wedge \neg(C|_a \vee C|_{ab}) \models I_L(C) \\
&B \wedge \neg(C|_b \vee C|_{ab}) \wedge I_L(C) \models \bot \\
&\mathcal{L}_{I_L(C)} \subseteq \mathcal{L}_{AB}
\end{aligned} \tag{3.5}$$

We discussed how Pudlák's and McMillan's interpolation systems Itp_P, Itp_M can be regarded as instances of the LISs, obtained by labeling all AB-common propositional variables respectively as ab and as b. We will apply the same kind of reasoning to prove that (3.5), when generalized to the level of first order theories, subsumes both (3.2) and (3.3).

An invariant is also the basis on which the relationship between labelings and interpolant strength is established in [DKPW10]; given two labelings such that $L \preceq L'$, the interpolants I_L and $I_{L'}$, generated for a each clause C in a refutation, satisfy:

$$I_L(C) \models I_{L'}(C) \vee C|_{AB} \tag{3.6}$$

3.4.1 Interpolant Strength in SMT

Our goal is to make use of (3.5) and (3.6) to develop an extension of LISs to SMT. To this end, we first need to lift the notion of labeling to that of $\mathcal{T}$*-labeling*:

Definition 3.4.1 ($\mathcal{T}$-Labeling)**.** A $\mathcal{T}$*-labeling* is a function L that assigns a *label* $L(p, C)$ among $\{\bot, a, b, ab\}$ to each *atom* p in each clause C in a refutation of $A \wedge B$. A-dirty atoms have label a, B-dirty atoms have label b; clean atoms in the clauses of $A \wedge B$ can have label a, b, ab, clean atoms in theory lemmata can have label a, b.

Note that the above definition does not correspond to labeling atoms appearing only in A as a, atoms appearing only in B as b and common atoms as b, a, ab, as in the propositional case. In fact, A-dirty atoms must appear in A or in theory lemmata, B-dirty in B or in theory lemmata, clean atoms can belong to A, to B, to both or to theory lemmata. Remember that refutations are assumed not to contain AB-mixed predicates.

As in the LISs, labels are independently set for all atoms occurrences in the leaves of the refutation, and recursively computed for the inner nodes, according to the join operator $\sqcup$ introduced in §3.3.4.2.

We can now proceed to the generalization of a LIS to a theory LIS:

Definition 3.4.2 (Theory Labeled Interpolation System ($\mathcal{T}$-LIS))**.** A *theory labeled interpolation system ($\mathcal{T}$-LIS)* is a procedure $\mathcal{T}$-Itp_L that, given A, B defined on a theory $\mathcal{T}$, a refutation, a labeling L, and a procedure TPI_L, outputs a *theory partial interpolant* $I_L(C)$ for each clause C in the refutation. If C is an original clause, $I_L(C)$ satisfies the following constraints:

$$\begin{aligned} &A \wedge \neg(C|_a \vee C|_{ab}) \models_{\mathcal{T}} I_L(C) \\ &B \wedge \neg(C|_b \vee C|_{ab}) \wedge I_L(C) \models_{\mathcal{T}} \bot \\ &\mathcal{L}_{I_L(C)} \subseteq \mathcal{L}_{AB} \cup \mathcal{L}_{\mathcal{T}} \end{aligned} \tag{3.7}$$

If C is a theory lemma, we require:

$$\begin{aligned} &\overline{C|_a} \models_{\mathscr{T}} I_L(C) \\ &\overline{C|_b} \wedge I_L(C) \models_{\mathscr{T}} \bot \\ &\mathscr{L}_{I_L(C)} \subseteq \mathscr{L}_{AB} \cup \mathscr{L}_{\mathscr{T}} \end{aligned} \tag{3.8}$$

Note that the conditions of (3.8) imply those of (3.7).

Due to the given definition, the invariant that in LISs guarantees the relationship between labelings and interpolant strength can be naturally extended to $\mathscr{T}$-LISs as:

$$I_L(C) \models_{\mathscr{T}} I_{L'}(C) \vee C|_{AB} \tag{3.9}$$

Table 3.9. Theory Labeled Interpolation System $\mathscr{T}$-Itp_L.

Leaf:	$C\,[I]$
$I = \begin{cases} C\|_b & \text{if } C \in A \\ \neg C\|_a & \text{if } C \in B \\ TPI_L(C) & \text{if } C \text{ is a } \mathscr{T}\text{-lemma} \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p : \alpha\,[I^+] \qquad C^- \vee \overline{p} : \beta\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } \alpha \sqcup \beta = a \\ I^+ \wedge I^- & \text{if } \alpha \sqcup \beta = b \\ (I^+ \vee p) \wedge (I^- \vee \overline{p}) & \text{if } \alpha \sqcup \beta = ab \end{cases}$	

Based on [CGS10, YM05], a $\mathscr{T}$-LIS (shown in Table. 3.9) generalizes a LIS by explicitly considering the presence of theory lemmata as leaves. It is parametric in a procedure TPI_L able to compute a theory partial interpolant for any $\mathscr{T}$-lemma C; TPI_L must be compliant with (3.7), (3.8), (3.9).

The approaches of [YM05] and [CGS10], respectively characterized by the invariants of (3.2) and (3.3), are special cases of (3.7). Suppose in fact that all clean atoms are labeled ab; then $C|_a = C|_A, C|_b = C|_B, C|_{ab} = C|_{AB}$, so that $C|_a \vee C|_{ab} = C|_A \vee C|_{AB}$ and $C|_b \vee C|_{ab} = C|_B \vee C|_{AB}$. Suppose now that clean atoms are given label b; then $C|_a = C|_A, C|_b = C|_B \vee C|_{AB}, C|_{ab} = \bot$, so that $C|_a \vee C|_{ab} = C_A$ and $C|_b \vee C|_{ab} = C|_B \vee C|_{AB}$.

The definition we give here has the advantage of explicitly relying on the invariant of (3.5) extended to a theory $\mathscr{T}$; the consequence is that, to show that a $\mathscr{T}$-LIS $\mathscr{T}$-Itp_L is sound for a particular TPI_L, it is necessary and sufficient to prove that TPI_L indeed generates formulae satisfying (3.8) and (3.9).

We put our approach to the test by applying it to interpolation in the theory of *difference logic (DL)*.

3.4.2 Interpolation in Difference Logic

In DL theory atoms are of the form $d - e \leq c$ where d, e are uninterpreted constants and c is a numeric constant, integer (IDL) or rational (RDL). Checking the consistency of a conjunction of atoms in DL relies on building a directed graph, where nodes are uninterpreted constants and there is an edge $d \xrightarrow{c} e$ for each atom $d \leq e + c$. The conjunction of atoms is unsatisfiable if and only if the graph has a cycle of negative weight [Seb07] (as in Figure 3.5); moreover, a theory interpolant can be computed based on the structure of the cycle, as shown in [CGS10].

Consider a theory lemma C in a refutation of $A \wedge B$; $\overline{C}$ is unsatisfiable and a cycle can be built as a witness of unsatisfiability. The edges in the cycle are partitioned into A-edges, B-edges and AB-edges, depending on whether the corresponding atoms are A-dirty, B-dirty or clean; note that the methods of [Seb07] do not introduce AB-mixed atoms in a refutation. The AB-edges are assigned, according to some criterion, to the A-edges and the B-edges.

We can compute a *summary* of a set of A-edges if some consecutive edges, associated with DL atoms, are replaced by new edges associated with the sums of these atoms, so that in the end all atoms are *clean* (i.e., all A-local variables have been removed). A *maximum* and a *minimum* summary are summaries of maximum and minimum size; they respectively correspond to summing atoms exactly as needed to remove A-local symbols, and to summing atoms as much as possible. Consider in Figure 3.5 the sequence of edges $d_1 \xrightarrow{0} d_2 \cdots d_5$: a maximum summary is $d_1 \xrightarrow{-1} d_3 \xrightarrow{0} d_5$ (associated with $d_1 \leq d_3 - 1 \wedge d_3 \leq d_5$), while a minimum summary is $d_1 \xrightarrow{-1} d_5$ (associated with $d_1 \leq d_5 - 1$).

A summary of all the A-edges in the cycle yields a theory interpolant; the resulting formula is in fact clean, implied by the A-edges and unsatisfiable in conjunction with the B-edges. Note that a sum of atoms is implied by the conjunction of the atoms; for this reason the maximum summary of the A-edges is a theory interpolant *stronger* than the minimum summary.

Different interpolants can be obtained from the cycle of Figure 3.5, depending on (i) the summary of A-edges and (ii) the assignment of the AB-edges to A-edges or

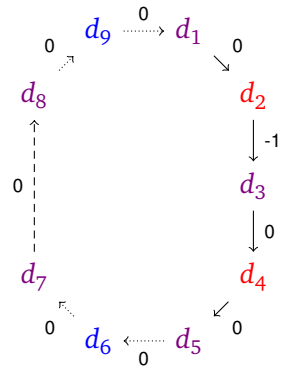


Figure 3.5. Example of a cycle of negative weight. Arrows denote A-edges, dotted arrows B-edges, dashed arrows AB-edges. Red symbols are A-local, blue B-local, purple AB-common.

to B-edges. The four interpolants are:

(1) $d_1 \leq d_5 - 1$: minimum summary, $d_7 \xrightarrow{0} d_8$ assigned to B-edges

(2) $d_1 \leq d_5 - 1 \wedge d_7 \leq d_8$: minimum summary, $d_7 \xrightarrow{0} d_8$ assigned to A-edges

(3) $d_1 \leq d_3 - 1 \wedge d_3 \leq d_5$: maximum summary, $d_7 \xrightarrow{0} d_8$ assigned to B-edges

(4) $d_1 \leq d_3 - 1 \wedge d_3 \leq d_5 \wedge d_7 \leq d_8$: maximum summary, $d_7 \xrightarrow{0} d_8$ assigned to A-edges

The interpolants produced by means of the syntactic manipulations of (i) and (ii) are of different strength, and related by logical implication in the following manner: (2)⇒(1), (4)⇒(3), (4)⇒(2), (3)⇒(1) in DL, (3) and (2) are incomparable.

(i) and (ii) are the basis on which to define a DL-LIS. A DL-labeling gives label a to a clean atom if it is assigned to the A-edges, b if it is assigned to the B-edges. Any theory interpolant I obtained for a conjunction of DL atoms by computing a summary of the edges with label a could be in principle used as a theory partial interpolant in the DL-LISs, as proved by the following result:

Theorem 3.4.1. *Let TPI_L be a procedure that computes a theory interpolant $I_L(C)$ for a DL-lemma C, as a summary of the atoms labeled a in the cycle of negative weight built over $\overline{C}$. Then $I_L(C)$ satisfies* (3.8).

Proof. A DL-labeling assigns label a or b to all atoms in C, so $C = C|_a \vee C|_b$. Since C is a DL-lemma, then $\overline{C|_a} \wedge \overline{C|_b} \models_{DL} \bot$. Let $I_L(C)$ be any clean summary of

$\overline{C|_a}$; by the above considerations, $\overline{C|_a} \models_{DL} I_L(C)$ and $I_L(C) \wedge \overline{C|_b} \models_{DL} \bot$, moreover $\mathscr{L}_{I_L(C)} \subseteq \mathscr{L}_{AB} \cup \mathscr{L}_{DL}$. So $I_L(C)$ is a theory partial interpolant according to (3.8). □

However, the use of arbitrary summaries does not guarantee that the theory partial interpolants obtained with different labelings can be related in terms of logical strength; for example, the interpolants (2) and (3) are not comparable. Thus, in order to satisfy (3.9), we restrict ourselves to the *strongest* theory interpolants, i.e. maximum summaries:

Theorem 3.4.2. *Let TPI_L be a procedure that computes a theory interpolant $I_L(C)$ for a DL-lemma C, as a maximum summary of the atoms labeled a in the cycle of negative weight built over $\overline{C}$. Then, given two labeling functions L, L' s.t. $L \preceq L'$, $I_L(C) \models_{DL} I_{L'}(C) \vee C|_{AB}$.*

Proof. $L \preceq L'$ implies that some of the AB-edges labeled by b in L are instead labeled by a in L'; this in turn yields that $I_{L'}(C)$ corresponds to $I_L(C)$ plus some additional conjuncts $\overline{D}$ which are a subset of the conjuncts of $\overline{C|_{AB}}$, that is $I_{L'}(C) = I_L(C) \wedge \overline{D}$. So we have:

$$\begin{aligned} I_L(C) &\Rightarrow \\ (I_L(C) \wedge \overline{D}) \vee D &\Rightarrow \\ (I_L(C) \wedge \overline{D}) \vee C|_{AB} &= \\ I_{L'}(C) \vee C|_{AB} & \end{aligned}$$

and (3.9) is satisfied, since $F_1 \models F_2$ implies $F_1 \models_{\mathscr{T}} F_2$ for any $F_1, F_2, \mathscr{T}$. □

3.4.3 Summary and Future Developments

We introduced the new framework of $\mathscr{T}$-LISs that extends the labeled interpolation systems from propositional logic to first order theories. The $\mathscr{T}$-LISs generalize the approaches to SMT interpolation of [YM05] and [CGS10], allowing to obtain interpolants of different logical strength from the same proof, provided that a procedure TPI is available to generate theory partial interpolants according to (3.8) and (3.9). We then discussed interpolation in difference logic and showed an instantiation of the $\mathscr{T}$-LISs to this theory.

Future work can address the development of procedures TPI tailored to other theories, where methods exist to compute theory interpolants for a given conjunction of atoms: these include linear rational arithmetic, unit-two-variables-per-inequality, equality and uninterpreted functions, lists. Besides individual theories, combinations of them can also be taken into account, in the Nelson-Oppen setting or in the delayed theory combination framework, as discussed in [YM05, CGS10].

3.5 A Parametric Interpolation Framework for First Order Theories

We discussed in §3.2 the challenge of determining what features of interpolants are most relevant to verification, and justified the adoption of logical strength and structure as parameters with respect to which the quality of interpolants can be evaluated.

In this section we provide the theoretical formalization of a *parametric interpolation framework* for arbitrary theories and inference systems, which supports the generation of multiple interpolants of different strength and structure from the same proof (§3.5.1). It can generate quantifier-free interpolants on examples where current methods are only able to compute quantified interpolants and provides flexibility in adjusting logical expressiveness, yielding interpolants that are stronger/weaker than the ones generated by current methods.

The development of the new framework is based on the characterization of a class of interpolation algorithms, relying on recursive procedures that generate interpolants from refutations; we show how the local proofs framework of [KV09, HKV12] (§3.5.2) and the LISs of [DKPW10, Wei12] for the resolution and hyper-resolution systems (§3.5.3) belong to this class, and how they can be considered instantiations of our method.

Interpolation Example. We start with an example showing the interpolant generation capabilities of the new framework.

Example 3.5.1. Let us take $\forall z(z = c) \wedge a = c \wedge g(b) = g(h)$ as A, and $f(a) \neq f(h) \wedge h = b$ as B. Then, c, g are A-local symbols, a, b, h are AB-common, and f is B-local. Clearly, $A \wedge B$ is unsatisfiable. A refutation Π of $A \wedge B$ is given in Figure 3.6. A possible interpolant of A and B is the quantified formula $\forall z(z = a)$, which would be obtained, for example, by the interpolation system of [KV09].

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall z(z = c) \qquad a = c}{\forall z(z = a)}}{a = b}}{f(a) = f(b)} \qquad \dfrac{f(a) \neq f(h) \qquad \dfrac{h = b}{f(h) = f(b)}}{f(a) \neq f(b)}}{\bot}$$

Figure 3.6. Local refutation Π of $A \wedge B$.

However, our method is able to compute $a = b$ and $h \neq b \vee (a = b \wedge h = b)$ as interpolants of $A \wedge B$ from the refutation of Figure 3.6, besides $\forall z(z = a)$. Note

that these two additional formulae are quantifier-free, and of different strength: our method thus allows to compute quantifier-free interpolants for problems on which [KV09] could only derive quantified interpolants. When addressing quantifier-free inference systems, for example the propositional hyper-resolution system, our approach also generates a range of quantifier-free interpolants, including those coming from [Wei12]. The main advantage provided by our framework comes with the flexibility of choosing between multiple interpolants and generating interpolants of different structure and strength, with or without quantifiers, from the same proof.

3.5.1 A Parametric Interpolation Framework

In this section we present a new interpolation framework that describes a class of recursive interpolation systems computing *partial interpolants* from refutations. These procedures start by deriving partial interpolants for the leaves; then, they derive partial interpolants for (some of) the inner nodes, by relying on the previously computed partial interpolants.

We begin by defining the notion of partial interpolant, then we illustrate a *parametric interpolation system* in Algorithm 5, and discuss the soundness of our approach. The parametric system will be later instantiated into two specific interpolation algorithms, in §3.5.2 and §3.5.3.

In the following we assume an underlying first order theory $\mathscr{T}$ on which formulae are defined; for brevity $\models$ can stand for $\models_{\mathscr{T}}$ and $\mathscr{L}_{AB}$ for $\mathscr{L}_{AB} \cup \mathscr{L}_{\mathscr{T}}$. Let Π be an AB-refutation of $A \wedge B$. Similarly to [KV09], we generate an interpolant I of A and B such that I is a propositional combination of formulae of Π. Our framework is parametric in a chosen *partition* of Π, i.e. a set of proofs $\mathscr{P} = \{\Pi'_i\}$ such that (i) each Π'_i is a subproof of Π, (ii) a leaf of a subproof Π'_i represents the root of another subproof Π'_j or a leaf of Π, (iii) each inference of Π belongs to some $\Pi'_i \in \mathscr{P}$. We call the leaves of a subproof $\Pi'_i \in \mathscr{P}$ *subleaves* of Π'_i; note that a subleaf might also be a leaf of Π. Similarly, the root of a subproof Π'_i is called a *subroot* of Π'_i. The aim of our algorithm is to build an interpolant from Π, by using the partition $\mathscr{P}$ of Π. To this end, we first define the notion of a *partial interpolant* of a formula C. We are then interested in computing the partial interpolants of the subroots C of the subproofs in $\mathscr{P}$.

Definition 3.5.1. [Partial Interpolant] Let C be a formula, and let f and g denote functions over formulae such that $f(\bot) = g(\bot) = \bot$. A formula I_C is called a *partial interpolant of* C with respect to A and B if it satisfies:

$$A \models I_C \vee f(C) \qquad B \models \overline{I_C} \vee g(C) \qquad I_C \in \mathscr{L}_{AB} \tag{3.10}$$

Note that when C is $\bot$, a partial interpolant I_C is an interpolant of A and B, since we have $A \models I_C$ and $B \models \overline{I_C}$. We also note that Definition 3.5.1 generalizes the notion of C-interpolants from [KV09]. Namely, by taking $f(C) = C$ and $g(C) = C$ a partial interpolant I_C is just a C-interpolant in the sense of [KV09], when C is clean.

Let us emphasize that in Definition 3.5.1 we are not restricted to a particular choice of f and g, which can be arbitrary functions over formulae. For example, the value of $f(C)$ and $g(C)$ might not even depend on C, or f and g can be defined using $\mathscr{P}$; the only restriction we impose is that (3.10) holds. Such a generality allows to build various (partial) interpolants, as presented later in §3.5.2 and §3.5.3.

Given a partition $\mathscr{P}$ of Π, we first compute partial interpolants of the leaves of Π. Next, for each subproof $\Pi'_i \in \mathscr{P}$ with root C and leaves $C_1, \ldots, C_n$, we build a partial interpolant I_C of C, proceeding inductively. We use the subleaves $C_1, \ldots, C_n$, and respectively compute their partial interpolants $I_{C_1}, \ldots, I_{C_n}$. I_C is then obtained as a propositional combination of (some of) C, $C_1, \ldots, C_n$, and $I_{C_1}, \ldots, I_{C_n}$ As a consequence, a partial interpolant of the root $\bot$ of Π is an interpolant I of A and B.

When computing partial interpolants of a formula C, we make a case distinction whether C is a leaf (base case) or a subroot of Π (induction step). We now address each case separately and formulate requirements over a formula to be a partial interpolant of C (see (3.11) and (3.14)).

Partial Interpolants of Leaves. Let C be a leaf of Π. Then, by the property (AB1) of AB-proofs (see §3.3.3), we need to distinguish between $A \models C$ and $B \models C$. The following conditions over a partial interpolant I_C of C are therefore imposed in order to satisfy (3.10):

$$\begin{array}{llll} A \models C \wedge \overline{f(C)} \rightarrow I_C & B \models I_C \rightarrow g(C) & I_C \in \mathscr{L}_{AB} & \text{if } A \models C \\ A \models \overline{f(C)} \rightarrow I_C & B \models I_C \rightarrow \overline{C} \vee g(C) & I_C \in \mathscr{L}_{AB} & \text{if } B \models C \end{array} \quad (3.11)$$

Partial Interpolants of Subroots. Let C be the root of a subproof Π' of Π. We assume that Π' consists of more than one formula (otherwise, we are in the base case) and that the leaves of Π' are $C_1, \ldots, C_n$. By the property (AB2), we conclude $\bigwedge C_i \models C$. By the induction hypothesis over $C_1, \ldots, C_n$, we assume that the partial interpolants $I_{C_1}, \ldots, I_{C_n}$ of the subleaves C_i are already computed. Using (3.10), we have:

$$A \models I_{C_i} \vee f(C_i) \qquad B \models \overline{I_{C_i}} \vee g(C_i) \qquad I_{C_i} \in \mathscr{L}_{AB} \quad (3.12)$$

From a simple combination of $\bigwedge C_i \models C$ and (3.12), we have:

$$A \models \bigwedge(I_{C_i} \vee f(C_i)) \wedge (\bigvee \overline{C_i} \vee C) \quad B \models \bigwedge(\overline{I_{C_i}} \vee g(C_i)) \wedge (\bigvee \overline{C_i} \vee C) \tag{3.13}$$

Using (3.10) in conjunction with (3.13), we derive the following constraints over a partial interpolant I_C of C:

$$\begin{aligned} &A \models \bigwedge(I_{C_i} \vee f(C_i)) \wedge (\bigvee \overline{C_i} \vee C) \wedge \overline{f(C)} \rightarrow I_C && I_C \in \mathscr{L}_{AB} \\ &B \models I_C \rightarrow \bigvee(I_{C_i} \wedge \overline{g(C_i)}) \vee (\bigwedge C_i \wedge \overline{C}) \vee g(C) && (3.14) \end{aligned}$$

Parametric Interpolation Framework. Our interpolation system is given in Algorithm 5. It takes as input an AB-proof Π, a partition $\mathscr{P}$ of Π, and the functions f and g. In addition, Algorithm 5 depends on a *construct* function which builds partial interpolants of leaves and subroots of Π, by using f and g. That is, for a formula C, *construct* returns a *set* Φ *of partial interpolants* I_C by making a case distinction whether C is a leaf or a subroot of Π. Hence, setting $f_C = f(C), g_C = g(C), f_i = f(C_i), g_i = g(C_i), I_i = I(C_i)$, *construct* is defined as:

$$construct(C, C_i, I_i, f_C, g_C, f_i, g_i) = \begin{cases} \Phi_1 & \text{if } C \text{ is a leaf} \\ \Phi_2 & \text{if } C \text{ is a subroot} \end{cases} \tag{3.15}$$

where each $I_C \in \Phi_1$ satisfies (3.11) and each $I_C \in \Phi_2$ satisfies (3.14). Note that the arguments $C_i, I_{C_i}, f(C_i), g(C_i)$ of *construct* become trivially empty whenever C is a leaf. For simplicity of notation, we therefore write $construct(C, f(C), g(C))$ whenever C is a leaf. The behavior of *construct*, in particular the choice of Φ_1 and Φ_2, is specific to the inference system in which Π was produced. We will address choices of Φ_1 and Φ_2 in §3.5.2 and §3.5.3.

Assuming *construct*, f, g are fixed, Algorithm 5 returns an interpolant I of A and B as follows. First, the leaves of Π are identified (line 2); for each leaf C of Π, a set Φ_1 of partial interpolants satisfying (3.11) is constructed, then, the partial interpolant of C is selected from Φ_1 (line 5). Next, partial interpolants of the subroots C of Π are recursively computed (lines 11-19); to this end, each subproof $\Pi' \in \mathscr{P}$ with root C and leaves $C_1, \ldots, C_n$ is analyzed. A set Φ_2 of partial interpolants of C is built by using the partial interpolants of $C_1, \ldots, C_n$ (line 14), and the partial interpolant of C is selected from Φ_2 (line 15). Finally, the partial interpolant of $\perp$ is returned as the interpolant of A and B (line 21).

Input: A, B s.t. $A \wedge B \models \bot$, AB-refutation Π of $A \wedge B$, partition $\mathscr{P}$ of Π, functions f, g, *construct*; f and g satisfy (3.10), *construct* produces clean formulae
Output: Interpolant I for $A \wedge B$

```
// Compute partial interpolants of leaves
L ←leaves(Π)
for each formula C ∈ L do
    Φ1 ←construct(C, f(C), g(C))
    I_C ←select(Φ1)
end
// Compute partial interpolants of subroots
𝒥 ← ⋃_{C∈L} I_C,  where  𝒥[C] ← I_C
𝒫* = {}
repeat
    for each Π′ in 𝒫 such that leaves(Π′) ⊆ L do
        C ←root(Π′)
        for each C_i ∈ leaves(Π′) do  I_{C_i} ← 𝒥[C_i]
        Φ2 ←construct(C, C_i, I_{C_i}, f(C), g(C), f(C_i), g(C_i))
        I_C ←select(Φ2)
        𝒥 ← 𝒥 ∪ {I_C}
        L ← L ∪ {C}
    end
    𝒫* ← 𝒫* ∪ {Π′}
until 𝒫* = 𝒫
return 𝒥[⊥]
```

Algorithm 5: Parametric Interpolation System.

Algorithm 5 depends on the choice of f, g, and *construct*, as well as of the partition $\mathscr{P}$; $select$ denotes a function that picks and returns a formula from a set of formulae. A *parametric interpolation framework* is thus implicitly defined by f, g, *construct*, and $\mathscr{P}$, yielding different interpolation algorithms based on Algorithm 5.

In the sequel we present two concrete choices of f, g and *construct*, together with $\mathscr{P}$. The first one, illustrated in §3.5.2, gives rise to an interpolation procedure for arbitrary first order inference systems. The second one, discussed in §3.5.3, addresses the propositional hyper-resolution system.

3.5.2 Interpolation in First Order Systems

In this section we present an interpolation procedure for arbitrary first order inference systems, by fixing the definition of f, g, *construct* and $\mathscr{P}$ in Algorithm 5.

Definition of f and g. We take f and g such that $f(C) = g(C) = C$, for every formula C. Clearly, the condition $f(\bot) = g(\bot) = \bot$ from Definition 3.5.1 is satisfied.

Definition of $\mathscr{P}$. We are interested in a special kind of partition, which we call *AB-partition* and define below.

Definition 3.5.2. [*AB*-partition] Let Π be an *AB*-proof and consider a partition $\mathscr{P} = \{\Pi'_j\}$ of Π into a set of subproofs Π'_j. $\mathscr{P}$ is called an *AB-partition* if the following conditions hold:

- the subroot C of each Π'_j is clean;
- the subleaves C_i of each Π'_j satisfy one of the following conditions: (a) every C_i is clean, or (b) if some of the C_i are dirty, then the dirty subleaves C_j are also leaves of Π and C_j are either all *A*-dirty or all *B*-dirty. Then, a dirty subleaf C_j cannot contain both *A*-local and *B*-local symbols.

In this section we fix $\mathscr{P}$ to be an *AB*-partition. We are now left with defining the input function *construct* of Algorithm 5. We make a case distinction on the subroots of the proof, and define the sets Φ_1 and Φ_2 of partial interpolants.

Definition of *construct* for Partial Interpolants of Leaves. Let C be a leaf of Π. Since $f(C) = g(C) = C$, (3.11) yields the following constraints over I_C:

$$\begin{array}{llll} A \models \bot \to I_C & B \models I_C \to C & I_C \in \mathscr{L}_{AB} & \text{if } A \models C \\ A \models \overline{C} \to I_C & B \models I_C \to \top & I_C \in \mathscr{L}_{AB} & \text{if } B \models C \end{array}$$

In principle, any formula $I_C \in \mathscr{L}_{AB}$ such that $\models \overline{C} \to \overline{I_C}$ if $A \models C$, and $\models \overline{C} \to I_C$ if $B \models C$ can be chosen as partial interpolant. Depending on whether C is clean or not, we define the set Φ_1 of partial interpolants as follows:

- If C is clean, we take: $\Phi_1 = \begin{cases} \{C, \bot\} & \text{if } A \models C \\ \{\overline{C}, \top\} & \text{if } B \models C \end{cases}$
- If C is dirty, we take: $\Phi_1 = \begin{cases} \{\bot\} & \text{if } A \models C \\ \{\top\} & \text{if } B \models C \end{cases}$

Definition of *construct* for Partial Interpolants of Subroots. Let C be the root of a subproof $\Pi' \in \mathscr{P}$, and let $C_1, \ldots, C_n$ denote the subleaves of Π'. As $f(C) = g(C) = C$ and $f(C_i) = g(C_i) = C_i$, (3.14) yields the following constraints over $I_C \in \mathscr{L}_{AB}$:

$$\begin{aligned} A &\models \bigwedge(I_{C_i} \vee C_i) \wedge (\bigvee \overline{C_i} \vee C) \wedge \overline{C} \rightarrow I_C \\ B &\models I_C \rightarrow \bigvee(I_{C_i} \wedge \overline{C_i}) \vee (\bigwedge C_i \wedge \overline{C}) \vee C \end{aligned} \tag{3.16}$$

Any formula $I_C \in \Phi_2$ needs to satisfy (3.16). A potential set Φ_2 of partial interpolants consists of the following ten formulae (annotated from (a) to (j)):

$$\begin{array}{ll} \text{(a) } \bigwedge(I_{C_i} \vee C_i) \wedge (\bigvee \overline{C_i} \vee C) \wedge \overline{C} & \text{(f) } \bigvee(I_{C_i} \wedge \overline{C_i}) \\ \text{(b) } \bigwedge(I_{C_i} \vee C_i) \wedge (\bigvee \overline{C_i}) & \text{(g) } \bigvee(I_{C_i} \wedge \overline{C_i}) \vee C \\ \text{(c) } \bigwedge(I_{C_i} \vee C_i) \wedge (\bigvee \overline{C_i} \vee C) & \text{(h) } \bigvee(I_{C_i} \wedge \overline{C_i}) \vee (\bigwedge C_i \wedge \overline{C}) \\ \text{(d) } \bigwedge(I_{C_i} \vee C_i) \wedge \overline{C} & \text{(i) } \bigvee(I_{C_i} \wedge \overline{C_i}) \vee (\bigwedge C_i) \\ \text{(e) } \bigwedge(I_{C_i} \vee C_i) & \text{(j) } \bigvee(I_{C_i} \wedge \overline{C_i}) \vee (\bigwedge C_i \wedge \overline{C}) \vee C \end{array} \tag{3.17}$$

It is not hard to argue that every formula from (3.17) satisfies (3.16). However, not any formula from (3.17) could be used as a partial interpolant I_C, as partial interpolants need to be clean. Note however that $\mathscr{P}$ is an AB-partition; this means that the root of Π' is clean, yielding that $f(C) = g(C) = C$ are clean formulae. Hence, whether a formula from (3.17) is clean depends only on whether the leaves of Π' are also clean. To define the set Φ_2, we therefore exploit the definition of AB-partitions and adjust (3.17) to the following three cases. In the sequel we refer by (a), ..., (j) to the formulae denoted by (a), ..., (j) in (3.17).

Case (i). All leaves C_i of Π' are clean. Any formula from (3.17) is a partial interpolant and:

$$\Phi_2 = \{(\text{a}), (\text{b}), (\text{c}), (\text{d}), (\text{e}), (\text{f}), (\text{g}), (\text{h}), (\text{i}), (\text{j})\}$$

Case (ii). Some leaves of Π' are A-dirty. Let us write $\{C_i\} = \{D_k\} \cup \{C_j\}$, where C_j are the clean leaves and D_k denote the A-dirty leaves of Π'. By the definition of AB-partitions, D_k are also leaves of Π. From property (AB1), we conclude $A \models \bigwedge D_k$ and take $I_{D_k} = \bot$ as the partial interpolants of D_k. From (AB2), we have $\models \bigvee \overline{C_i} \vee C$. Then from $A \models \bigwedge D_k$ and $\models \bigvee \overline{C_i} \vee C$, we derive $A \models \bigvee \overline{C_j} \vee C$. Thus, restricting ourselves to the clean leaves C_j, the constraints (3.14) become:

$$A \models \bigwedge(I_{C_j} \vee C_j) \wedge (\bigvee \overline{C_j} \vee C) \wedge \overline{C} \rightarrow I_C \qquad B \models I_C \rightarrow \bigvee(I_{C_j} \wedge \overline{C_j}) \vee C$$

Let (a′),(b′),(c′),(f′),(g′) denote the formulae obtained from (a),(b),(c),(f),(g), by replacing C_i with C_j; then any formula (a′),(b′),(c′),(f′),(g′) can be taken as a partial interpolant I_C of C. Hence:

$$\Phi_2 = \{(\mathrm{a}'), (\mathrm{b}'), (\mathrm{c}'), (\mathrm{f}'), (\mathrm{g}')\}$$

Case (iii). Some leaves of Π' *are B-dirty.* Using the notation of Case (ii), (3.14) imposes the following constraints over I_C:

$$A \models \bigwedge (I_{C_j} \vee C_j) \wedge \overline{C} \rightarrow I_C, \qquad B \models I_{C_j} \rightarrow \bigvee (I_{C_j} \wedge \overline{C_j}) \vee (\bigwedge C_j \wedge \overline{C}) \vee C.$$

Let (d′),(e′),(h′),(i′),(j′) denote the formulae obtained from (d),(e),(h),(i),(j), by replacing C_i with C_j. Then, (d′),(e′),(h′),(i′),(j′) are partial interpolants I_C of C. Hence:

$$\Phi_2 = \{(\mathrm{d}'),(\mathrm{e}'),(\mathrm{h}'),(\mathrm{i}'),(\mathrm{j}')\}$$

Interpolation Algorithm for First Order Inference Systems. Algorithm 5 yields a new interpolation procedure for arbitrary first order inference systems. It takes as input an *AB*-refutation Π and an *AB*-partition $\mathscr{P}$ of Π; the functions f, g satisfy the condition $f(C) = g(C) = C$, for every C, whereas *construct* is defined by using the above given sets Φ_1 and Φ_2 in (3.15). With these considerations on its inputs, Algorithm 5 returns an interpolant I of A and B by recursively computing the partial interpolants of leaves and subroots of Π. The (partial) interpolants derived by Algorithm 5 are of different strength and structure and are generated from the same proof.

Logical Relations among Partial Interpolants. Figure 3.7 shows the relationship among the formulae from (3.17) in terms of logical strength. An arrow is drawn between two formulae denoted by (x) and (y) if (x)⊨(y). All implications are valid, which can be proved by simply applying resolution on (x)∧$\overline{(\mathrm{y})}$. The logical relations of Figure 3.7 correspond to Case (i) above; the relations corresponding to Cases (ii) and (iii) are special instances of Figure 3.7.

The partial interpolants cover a range of degrees of logical strength, and the choice of each partial interpolant directly affects the strength of the overall interpolant returned by Algorithm 5. We can in fact state:

Theorem 3.5.1. *Given an AB-refutation* Π *and an AB-partition* $\mathscr{P}$*, assume that partial interpolants have been chosen from the sets* Φ_1 *and* Φ_2 *for each leaf and root*

in the subproofs $\Pi' \in \mathscr{P}$, so that Algorithm 5 yields an interpolant I. Then, if any partial interpolant I_C is replaced by a stronger partial interpolant I'_C, Algorithm 5 generates an interpolant I' s.t. $I' \models I$.

Proof. The result follows from the monotonicity of the propositional connectives $\vee, \wedge$. Partial interpolants are recursively built by combining previously computed partial interpolants, taken with *positive sign*, and other formulae by means of $\vee, \wedge$. Replacing in I every occurrence of a certain I_C by means of an I'_C s.t. $I'_C \models I_C$ thus yields a new I' s.t. $I' \models I$. □

While previous methods, e.g. [JM05, DKPW10, Wei12], also address logical strength, they are restricted to quantifier-free interpolants. Our approach instead characterizes interpolants strength in full first order logic, and can be employed in arbitrary first order theories.

The quality of the generated interpolants indeed depends on the verification techniques in which they are used, so that weaker or stronger interpolants might be beneficial in different verification environments: this subject will be addressed in §3.6.

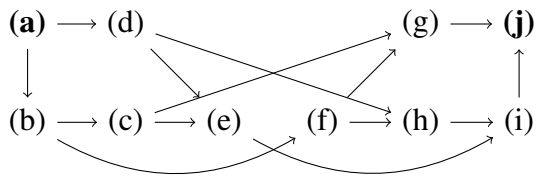


Figure 3.7. Implication graph of partial interpolants in first order inference systems.

The Local Proofs Framework. The interpolation algorithm of [KV09] extracts interpolants from *local proofs* or *split proofs*. An inference in a local proof cannot use both A-local and B-local symbols; inferences of local proofs are called local inferences. It is easy to see that local proofs are special cases of AB-proofs.

Given a local AB-proof Π, by making use of our notation, the algorithm of [KV09] can be summarized as follows. A partition $\mathscr{P}$ of Π is first created such that each subproof Π' of Π is an A-subproof or a B-subproof. Next, partial interpolants are constructed as:

- If C is a clean subleaf of Π, then: $\Phi_1 = \begin{cases} \{C\} & \text{if } A \models C \\ \{\overline{C}\} & \text{if } B \models C \end{cases}$

- If C is a clean subroot of a subproof Π' with leaves $C_1, \ldots, C_n$, then $C_1, \ldots, C_n \models C$. Let $\{C_j\}$ denote the set of clean leaves of Π'. Hence, $\{C_j\} \subseteq \{C_1, \ldots, C_n\}$ and:

$$\Phi_2 = \begin{cases} \{\bigwedge_j (C_j \vee I_{C_j}) \wedge \bigvee_j \overline{C_j}\} & \text{if } \Pi' \text{ is an } A\text{-dirty subproof} \\ \{\bigwedge_j (C_j \vee I_{C_j})\} & \text{if } \Pi' \text{ is a } B\text{-dirty subproof} \end{cases}$$

It is therefore not hard to prove that the algorithm of [KV09] is a special case of Algorithm 5. The partial interpolants generated by [KV09] are in fact a subset of the partial interpolants we compute; in particular, if Π' is an A-dirty (respectively, B-dirty) subproof, then the partial interpolant of the subroot C of Π' in [KV09] corresponds to our formula (b′) (respectively, (e′)) defined before.

Note that the sets Φ_1 and Φ_2 computed by [KV09] contain exactly one formula, giving thus exactly one interpolant, while the cardinality of Φ_1 and Φ_2 in our method can be greater than one (lines 4 and 14 of Algorithm 5). Moreover, some of our interpolants cannot be obtained by other methods, as shown in Example 3.5.1.

Example 3.5.2. We illustrate our first order interpolation system by using the formulae A and B of Example 3.5.1. Consider the AB-refutation Π given in Figure 3.6 and take the AB-partition $\mathscr{P} = \{\Pi', \Pi''\}$, where Π' and Π'' are respectively given in Figure 3.8 and Figure 3.9.
By applying Algorithm 5, we first visit the subproof Π'' and compute $I_{a=b}$. Since Π'' has A-dirty leaves, the set of partial interpolants corresponding to the root $a = b$ of Π'' is: {(a′), (b′), (c′), (f′), (g′)}. Since all the subleaves of Π'' are dirty leaves, (a′),(b′),(c′),(f′),(g′) respectively reduce to $a = b \wedge a \neq b$, $\bot$, $a = b$, $\bot$, $a = b$. The set of partial interpolants $I_{a=b}$ is thus given by: $\{a = b, \bot\}$.

$$\dfrac{\dfrac{a = b}{f(a) = f(b)} \qquad \dfrac{f(a) \neq f(h) \qquad \dfrac{h = b}{f(h) = f(b)}}{f(a) \neq f(b)}}{\bot}$$

Figure 3.8. Subproof Π'.

$$\dfrac{\dfrac{\forall z(z = c) \qquad a = c}{\forall z(z = a)}}{a = b}$$

Figure 3.9. Subproof Π''.

Next, we visit the subproof Π'. As Π' has B-dirty leaves, the set of partial interpolants corresponding to the root $\bot$ of Π' is {(d′), (e′), (h′), (i′), (j′)}. Since Π' has two clean subleaves, namely $a = b$ and $h = b$, the formulae (d′),(e′),(h′),(i′),(j′) are simplified, yielding the following set of partial interpolants $I_\bot$: $\{(I_{a=b} \vee a = b) \wedge (I_{h=b} \vee h = b), (I_{a=b} \wedge a \neq b) \vee (I_{h=b} \wedge h \neq b) \vee (a = b \wedge h = b)\}$. To derive $I_{h=b}$, note that $h = b$ is the only clean leaf of B. Therefore, the set of partial

interpolants $I_{h=b}$ is given by $\{\top, h \neq b\}$. Based on these results, the set of (partial) interpolants $I_\perp$ is finally given by $\{a = b, h \neq b \vee (a = b \wedge h = b)\}$.

The AB-partition we used here is different from the one used in [KV09]. The flexibility in choosing AB-partitions in Algorithm 5 allows us to derive quantifier-free interpolants from Figure 3.6.

3.5.3 Interpolation in the Hyper-Resolution System

In this section, we instantiate the parametric interpolation framework in the propositional hyper-resolution system; we turn Algorithm 5 into an interpolation procedure for the propositional hyper-resolution system, by fixing the choices of f, g, *construct*, and $\mathscr{P}$. We also prove that our approach generalizes the work of [Wei12].

We begin by introducing some notation specific to the HR system. For each clause C and inference in Π, we define two arbitrary subsets $\Delta_A^C, \Delta_B^C \subseteq \Sigma_{AB}$ of AB-common symbols, where $\Delta_A^C \cup \Delta_B^C = \Sigma_{AB}$. We then write $C|_{A\Delta_A^C} = C|_A \vee C|_{\Delta_A^C}$, $C|_{B\Delta_B^C} = C|_B \vee C|_{\Delta_B^C}$. It is important to remark that Δ_A^C, Δ_B^C need not to be the same for the inferences where C is involved: for example, an AB-common symbol of C can be treated as A-local in the inference where C is the conclusion, and as B-local in an inference where C is a premise. With these notations at hand, we now define the input parameters f, g, *construct* and $\mathscr{P}$ of Algorithm 5 in the HR system.

Definition of f and g. We take f and g such that, for every C:

$$f(C) = C|_{A\Delta_A^C} \qquad g(C) = C|_{B\Delta_B^C} \tag{3.18}$$

Note that $f(C) \vee g(C) = C|_A \vee C|_B \vee C|_{AB} = C$ for any C. Similarly to [DKPW10], $f(C)$ and $g(C)$ separate the symbols of C into sets of A-local and B-local symbols, where the AB-common symbols in Σ_{AB} can be treated either as A-local, B-local, or AB-common.

Definition of $\mathscr{P}$. We fix the partition of an AB-proof to a so-called *HR-partition*, as defined below.

Definition 3.5.3. [*HR*-partition] Let Π be an AB-proof and consider a partition $\mathscr{P} = \{\Pi'_j\}$ of Π into a set of subproofs Π'_j. The partition $\mathscr{P}$ of Π is called an *HR-partition* if the following condition holds:

- for each subproof Π'_j with root C and leaves $C_0, \ldots, C_{n-1}$, the inference $\frac{C_0 \ \ldots \ C_{n-1}}{C}$ is an application of the hyper-resolution rule. That is, for some clauses $E, D_1, \ldots, D_{n-1}$ and literals $p_1, \ldots, p_{n-1}$, C_0 can be written as $\overline{p_1} \vee \cdots \vee \overline{p_{n-1}} \vee E$, $C_1, \ldots, C_{n-1}$ denote respectively $D_1 \vee p_1, \ldots, D_{n-1} \vee p_{n-1}$, and C is $\bigvee D_i \vee E$.

In this section we fix $\mathscr{P}$ *to be an HR-partition*, and proceed to the definition of the *construct* function for partial interpolants.

Definition of *construct* for Partial Interpolants of Leaves. Let C be a leaf of Π. Note that, if $A \models C$, then we have $C \in \mathscr{L}_A$. Similarly, if $B \models C$ then $C \in \mathscr{L}_B$ holds. Therefore, by using the definition of f and g from (3.18), the constraints of (3.11) over the partial interpolants $I_C \in \mathscr{L}_{AB}$ reduce to:

$$A \models C \wedge \overline{C|_{A\Delta_A^C}} \rightarrow I_C \qquad B \models I_C \rightarrow C|_{\Delta_B^C} \qquad \text{if } A \models C$$
$$A \models \overline{C|_{\Delta_A^C}} \rightarrow I_C \qquad B \models I_B \rightarrow \overline{C} \vee C|_{B\Delta_B^C} \qquad \text{if } B \models C$$

A set Φ_1 of partial interpolants is defined as:

$$\Phi_1 = \begin{cases} \{C_{\Delta_B^C}\} & \text{if } A \models C \\ \{\overline{C|_{\Delta_A^C}}\} & \text{if } B \models C \end{cases}$$

Definition of *construct* for Partial Interpolants of Subroots. Let C be the root of a subproof $\Pi' \in \mathscr{P}$, and let $C_0, \ldots, C_{n-1}$ denote the leaves of Π'. Further, set $\Delta_A^i = \Delta_A^{C_i}$ and $\Delta_B^i = \Delta_B^{C_i}$. Considering the definitions of f and g from (3.18), the constraints of (3.14) over the partial interpolants $I_C \in \mathscr{L}_{AB}$ are simplified to:

$$\begin{aligned} A &\models \bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i}) \wedge (\bigvee \overline{C_i} \vee C) \wedge \overline{C|_{A\Delta_A^C}} \rightarrow I_C \\ B &\models I_C \rightarrow \bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}}) \vee (\bigwedge C_i \wedge \overline{C}) \vee C|_{B\Delta_B^C} \end{aligned} \tag{3.19}$$

Any formula $I_C \in \Phi_2$ thus satisfies (3.19). A potential set Φ_2 of partial interpolants therefore consists of the following ten formulae (similarly to §3.5.2, annotated from (a) to (j)):

(a) $\bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i}) \wedge (\bigvee \overline{C_i} \vee C) \wedge \overline{C|_{A\Delta_A^C}}$
(b) $\bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i}) \wedge (\bigvee \overline{C_i})$
(c) $\bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i}) \wedge (\bigvee \overline{C_i} \vee C)$
(d) $\bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i}) \wedge \overline{C|_{A\Delta_A^C}}$
(e) $\bigwedge(I_{C_i} \vee C_i|_{A\Delta_A^i})$
(f) $\bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}})$
(g) $\bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}}) \vee C|_{B\Delta_B^C}$
(h) $\bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}}) \vee (\bigwedge C_i \wedge \overline{C})$
(i) $\bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}}) \vee (\bigwedge C_i)$
(j) $\bigvee(I_{C_i} \wedge \overline{C_i|_{B\Delta_B^i}}) \vee (\bigwedge C_i \wedge \overline{C}) \vee C|_{B\Delta_B^C}$

(3.20)

To use the formulae from (3.20) as partial interpolants we need to ensure that they are clean. Similarly to §3.5.2, the definition of Φ_2 comes by considering the following three cases.

Case (i). The root C and all leaves C_i of Π' are clean. As C is clean, we have $C|_{A\Delta_A^C} = C|_{\Delta_A^C}$ and $C|_{B\Delta_B^C} = C|_{\Delta_B^C}$. A similar result for C_i is also derived. The formulae (a),(d),(g),(j) of (3.20) are therefore partial interpolants in I_C. Moreover, if $C = C|_{\Delta_A^C}$, (b),(f),(h) are also partial interpolants. On the other hand, if $C = C|_{\Delta_B^C}$, (c),(e),(i) also yield partial interpolants. We thus have:

$$\Phi_2 = \begin{cases} \{(a),(b),(d),(f),(g),(h),(j)\} & \text{if } C = C|_{\Delta_A^C} \\ \{(a),(c),(d),(e),(g),(i),(j)\} & \text{if } C = C|_{\Delta_B^C} \\ \{(a),(d),(g),(j)\} & \text{otherwise} \end{cases}$$

Case (ii). Some leaves of Π' are A-dirty. We write $\{C_i\} = \{D_k\} \vee \{C_j\}$, where C_j are the clean leaves and D_k represent the A-dirty leaves of Π'. Let (a′),(b′),(c′),(f′),(g′) denote the formulae obtained from (a),(b),(c),(f),(g), by replacing C_i with C_j. We then have:

$$\Phi_2 = \begin{cases} \{(a'),(b'),(f'),(g')\} & \text{if } C = C|_{\Delta_A^C} \\ \{(a'),(c'),(g')\} & \text{if } C = C|_{\Delta_B^C} \\ \{(a'),(g')\} & \text{otherwise} \end{cases}$$

Case (iii). Some leaves of Π' are B-dirty. Using the notation of Case (ii), let (d′),(e′),(h′),(i′),(j′) denote the formulae obtained from (d),(e),(h),(i),(j), by replacing C_i with C_j. Then:

$$\Phi_2 = \begin{cases} \{(d'),(h'),(j')\} & \text{if } C = C|_{\Delta_A^C} \\ \{(d'),(e'),(i'),(j')\} & \text{if } C = C|_{\Delta_B^C} \\ \{(d'),(j')\} & \text{otherwise} \end{cases}$$

We will show later how the set Φ_2 can be extended by exploiting the peculiarities of the HR system; in particular, we will decribe how to obtain additional clean formulae by removing local literals thanks to the HR rule.

Interpolation Algorithm for the HR System. Algorithm 5 yields a new interpolation system for the HR system. It takes as input an AB-refutation Π and an HR-partition $\mathscr{P}$ of Π; the functions f, g satisfy (3.18), whereas *construct* is defined by using the above specified Φ_1 and Φ_2 in (3.15). With such specification, Algorithm 5 computes an interpolant I of A and B in the HR system; our method benefits from computing partial interpolants of different strength and structure, from the same proof.

Logical Relations among Partial Interpolants. The logical relations among the formulae of (3.20) are given in the implication graph of Figure 3.10. Similarly to Figure 3.7, an arrow in Figure 3.10 is drawn between two formulae denoted by (x) and (y) if (x)$\models$(y). Furthermore, an arrow annotated by $\vartriangle$ (resp. $\triangledown$) in Figure 3.10

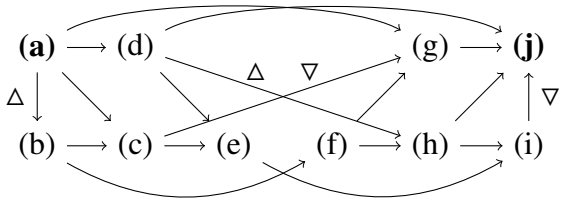


Figure 3.10. Implication graph of partial interpolants in the HR system.

is drawn between two formulae (x) and (y) if (x)$\models$(y) under the assumption that $C = C|_{\Delta_A^C}$ (resp. $C = C|_{\Delta_B^C}$). Figure 3.10 shows that some of the implications do not always hold: (a)$\models$(b) and (d)$\models$(h) only if $C = C|_{\Delta_A^C}$, while (c)$\models$(g) and (i)$\models$(j) only if $C = C|_{\Delta_B^C}$.

The Labeled Hyper-Resolution Framework. We now relate Algorithm 5 to the approach of [Wei12] in the HR system. Given an AB-refutation Π, using our notation, the algorithm of [Wei12] can be summarized as follows:

- If C is a leaf of Π, then:

$$\Phi_1 = \begin{cases} C|_b & \text{if } A \models C \\ \overline{C|_a} & \text{if } B \models C \end{cases}$$

- If C is the conclusion of the HR rule with premises $C_0, \ldots, C_{n-1}$, then, for some literals $p_1, \ldots, p_{n-1}$ and clauses $D_1, \ldots, D_{n-1}, E$, we have $C_0 = \overline{p_1} \vee \cdots \vee \overline{p_{n-1}} \vee E$, $C_1 = D_1 \vee p_1, \ldots, C_{n-1} = D_{n-1} \vee p_{n-1}$, and $C = \bigvee D_i \vee E$. The pivots p_i are assumed to be uniformly labeled in [Wei12]. Then:

$$\Phi_2 = \begin{cases} I_{C_0} \vee \bigvee_{i=1}^{n-1} I_{C_i} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = a \\ I_{C_0} \wedge \bigwedge_{i=1}^{n-1} I_{C_i} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = b \\ \begin{cases} (I_{C_0} \vee \bigvee \overline{p_i}) \wedge \bigwedge_{i=1}^{n-1} (p_i \vee I_{C_i}) \\ (I_{C_0} \wedge \bigwedge p_i) \vee \bigvee_{i=1}^{n-1} (\overline{p_i} \wedge I_{C_i}) \end{cases} & \text{if } \forall i\, L(p_i, C_0) \sqcup L(p_i, C_i) = ab \end{cases}$$

We argue that Algorithm 5 in the HR system generalizes the method of [Wei12]. To this end, we will show how the behavior of the labeling function on the AB-common literals can be simulated in our framework, by assigning appropriate sets Δ_R^C, Δ_B^C to every clause C in every inference.

Consider a leaf C of the AB-refutation Π, such that $C \in \mathscr{L}_A$. Using [Wei12], the A-local literals of C are labeled with a, and the AB-common literals with one of the labels a, b, ab. The partial interpolant $C|_b$ is thus a subclause of $C|_{AB}$. We then fix Δ_B^C such that $C|_b = C|_{\Delta_B^C}$, and hence our partial interpolant is also a partial interpolant of [Wei12]. A similar argument holds when $C \in \mathscr{L}_B$.

Consider an arbitrary HR inference in Π, with root C and leaves C_i. The goal is to exploit the HR rule to remove local literals in order to make dirty formulae clean. This allows to obtain an additional set of partial interpolants for subproofs containing dirty subleaves/subroots. Let $\Pi' \in \mathscr{P}$ be a subproof with root C and leaves $C_0, \ldots, C_{n-1}$. As $\mathscr{P}$ is an HR-partition, formulae C and C_i are as given in Definition 3.5.3. Consider now the formulae (d) and (g) from (3.20), and replace C and C_i with the clauses from Definition 3.5.3. We thus obtain the new formulae (d) and (g):

$$\text{(d)} \quad (I_{C_0} \vee (E \vee \bigvee \overline{p_i})|_{A\Delta_A^0}) \wedge \bigwedge (I_{C_i} \vee (D_i \vee p_i)|_{A\Delta_A^i}) \wedge \overline{(\bigvee D_i \vee E)}|_{A\Delta_A^C}$$

$$\text{(g)} \quad (I_{C_0} \wedge \overline{(E \vee \bigvee \overline{p_i})}|_{B\Delta_B^0}) \vee \bigvee (I_{C_i} \wedge \overline{(D_i \vee p_i)}|_{B\Delta_B^i}) \vee (\bigvee D_i \vee E)|_{B\Delta_B^C}$$

From Figure 3.10, we have (a)$\models$(d) and (g)$\models$(j). It can be further derived that (d)$\models$(g), since (d) $\wedge$ $\overline{\text{(g)}}$ is unsatisfiable as shown below:

$$\begin{aligned}
&(I_{C_0} \vee (E \vee \bigvee \overline{p_i})|_{A\Delta_A^0}) \wedge \bigwedge (I_{C_i} \vee (D_i \vee p_i)|_{A\Delta_A^i}) \wedge \overline{(\bigvee D_i \vee E)}|_{A\Delta_A^C} \wedge \\
&(\overline{I_{C_0}} \vee (E \vee \bigvee \overline{p_i})|_{B\Delta_B^0}) \wedge \bigwedge (\overline{I_{C_i}} \vee (D_i \vee p_i)|_{B\Delta_B^i}) \wedge \overline{(\bigvee D_i \vee E)}|_{B\Delta_B^C} \quad \Rightarrow \\
&((E \vee \bigvee \overline{p_i})|_{A\Delta_A^0} \vee (E \vee \bigvee \overline{p_i})|_{B\Delta_B^0}) \wedge \bigwedge ((D_i \vee p_i)|_{A\Delta_A^i} \vee (D_i \vee p_i)|_{B\Delta_B^i}) \wedge \\
&\overline{(\bigvee D_i \vee E)}|_{A\Delta_A^C} \wedge \overline{(\bigvee D_i \vee E)}|_{B\Delta_B^C} \quad \Rightarrow \\
&(E \vee \bigvee \overline{p_i}) \wedge \bigwedge (D_i \vee p_i) \wedge \bigwedge \overline{D_i} \wedge \overline{E} \quad \Rightarrow \quad \bot
\end{aligned}$$

The relation (d)$\models$(g) can be exploited to obtain intermediate pairs of formulae (x),(y), such that (d)$\models$(x), (x)$\models$(y) and $\overline{\text{(g)}} \models \overline{\text{(y)}}$. We now apply the HR rule to obtain new partial interpolants by removing the local literals of D_i and E in (d) and $\overline{\text{(g)}}$; this gives us the following formulae:

$$\text{(m)} \quad (I_{C_0} \vee E|_{\Delta_A^0} \vee \bigvee \overline{p_i}|_{A\Delta_A^0}) \wedge \bigwedge (I_{C_i} \vee D_i|_{\Delta_A^i} \vee p_i|_{A\Delta_A^i}) \wedge \bigwedge \overline{D_i}|_{\Delta_A^C} \wedge \overline{E}|_{\Delta_A^C}$$

$$\text{(n)} \quad (I_{C_0} \wedge \overline{E}|_{\Delta_B^0} \wedge \bigwedge p_i|_{B\Delta_B^0}) \vee \bigvee (I_{C_i} \wedge \overline{D_i}|_{\Delta_B^i} \wedge \overline{p_i}|_{B\Delta_B^i}) \vee \bigvee D_i|_{\Delta_B^C} \vee E|_{\Delta_B^C}$$

It is always possible to split a HR inference into a sequence of HR inferences so that a uniform labeling of the pivots is achieved (see [Wei12] and §3.3.4). In

turn, the presence of a uniform labeling allows to further simplify (m) and (n), thus deriving the partial interpolants of [Wei12] as special cases of our formulae.

We show how the choice of sets Δ_A, Δ_B is affected in our setup, by analyzing the three possible cases:

(A) if the label is a, then the Δ_A, Δ_B sets are chosen so that, if p_i is AB-common, then $p_i \in \Delta_A^i \setminus \Delta_B^i$ and $\overline{p_i} \in \Delta_A^0 \setminus \Delta_B^0$;

(B) if the label is b then, if p_i is AB-common, then $p_i \in \Delta_B^i \setminus \Delta_A^i$ and $\overline{p_i} \in \Delta_B^0 \setminus \Delta_A^0$;

(C) if the label is ab then $p_i \in \Delta_A^i \cap \Delta_B^i$ and $\overline{p_i} \in \Delta_A^0 \cap \Delta_B^0$.

In case (A), (n) reduces to:

$$\text{(o)} \qquad (I_{C_0} \wedge \overline{E}|_{\Delta_B^0}) \vee \bigvee (I_{C_i} \wedge \overline{D_i}|_{\Delta_B^i}) \vee \bigvee D_i|_{\Delta_B^C} \vee E|_{\Delta_B^C}$$

Formula (o) can be thus also used as an additional partial interpolant in Φ_2. A special case of (o) is the partial interpolant $\bigvee_{i=0}^{n-1} I_{C_i}$.

In case (B), (m) reduces to:

$$\text{(p)} \qquad (I_{C_0} \vee E|_{\Delta_A^0}) \wedge \bigwedge (I_{C_i} \vee D_i|_{\Delta_A^i}) \wedge \bigwedge \overline{D_i}|_{\Delta_A^C} \wedge E|_{\Delta_A^C}$$

Formula (p) can then be also used as a partial interpolant in Φ_2. A special case of (p) is $\bigwedge_{i=0}^{n-1} I_{C_i}$.

In case (C), (m) and (n) can also be used as partial interpolants in Φ_2. Special cases of (m) and (n) are the two partial interpolants of [Wei12].

The relations between Figure 3.10 and the formulae (m), (n), (p), and (o) are shown in Figure 3.11. Similarly to Figure 3.7, an arrow in Figure 3.11 is drawn between two formulae denoted by (x) and (y) if (x)$\models$(y). In addition, an arrow annotated by $\star$ (respectively, by $*$ and $\bullet$) is drawn between (x) and (y) if (x)$\models$(y) under the assumption that $p_i \in (\Delta_A^i \cap \Delta_B^i)$ (respectively, $p_i \in (\Sigma_{B \setminus AB} \cup (\Delta_B^i \setminus \Delta_A^i))$ and $p_i \in (\Sigma_{A \setminus AB} \cup (\Delta_A^i \setminus \Delta_B^i))$).

Example 3.5.3. Consider a proof Π with an HR inference, as in Figure 3.12. Assume that the following partial interpolants are given: $I_0 = I_{\overline{p_1}\overline{p_2}q_1}$, $I_1 = I_{p_1q_2}$, $I_2 = I_{p_2q_3}$. We assume that all literals belong to $\Sigma|_{AB}$, and the pivots p_1 and p_2 are both labeled as ab.
The algorithm of [Wei12] yields the partial interpolant $I_{q_1q_2q_3} = (I_0 \vee \overline{p_1} \vee \overline{p_2}) \wedge (I_1 \vee p_1) \wedge (I_2 \vee p_2)$. W.l.o.g., we assume that the sets Δ_A, Δ_B have been chosen such that,

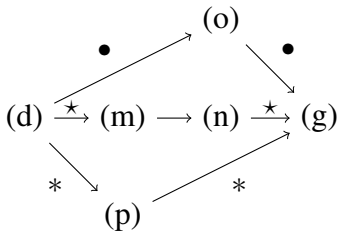


Figure 3.11. Additional implication graph of partial interpolants in the HR system.

$$\frac{\begin{array}{c}\vdots\\ \overline{p_1}\,\overline{p_2}q_1\end{array}\qquad\begin{array}{c}\vdots\\ p_1q_2\end{array}\qquad\begin{array}{c}\vdots\\ p_2q_3\end{array}}{\begin{array}{c}q_1q_2q_3\\ \vdots\end{array}}$$

Figure 3.12. AB-proof Π with HR inferences.

for every clause C, all common variables of C are in $\Delta_A^C \cap \Delta_B^C$[1]. Then, our method generates the partial interpolant $I_{q_1q_2q_3}$ as the formula (m), simplified below:

$$I = (I_0 \vee q_1 \vee \overline{p_1} \vee \overline{p_2}) \wedge (I_1 \vee q_2 \vee p_1) \wedge (I_2 \vee q_3 \vee p_2) \wedge \overline{q_1} \wedge \overline{q_2} \wedge \overline{q_3}$$

Our interpolant I, which is stronger than the interpolant I' of [Wei12], cannot be generated in [Wei12]. Moreover, our method can also generate the interpolant I' of [Wei12], by applying the HR rule with pivots q_i.

We thus conclude that Algorithm 5 generalizes and extends the method of [Wei12] in a number of ways. While in [Wei12] the label $L(p, C)$ of a variable p in the conclusion C of an HR inference is derived in a unique manner from the labels of the inference premises, in our framework the sets Δ_A^C and Δ_B^C can be chosen independently for every clause C and every inference.

An important aspect of the labeled systems is that they allow to systematically compare the strength of the interpolants resulting from different labelings. In particular, in [DKPW10] a total order $\preceq$ is defined over the labels $\{a, b, ab, \bot\}$ as $b \preceq ab \preceq a \preceq \bot$. Then, $\preceq$ is extended to a partial order over labeling functions L, L', as follows: $L \preceq L'$ is defined if, for every clause C and variable p in C, $L(p, C) \preceq L'(p, C)$. If $L \preceq L'$, then the interpolant given by L is stronger than the one given by L'. Our framework also benefits from such a comparison, since in any *AB*-refutation we are able to simulate the labeling function L by an appropriate

[1]This is case (C). The other choices of Δ_R, Δ_B yield other generalizations of [Wei12].

choice of Δ_A^C, Δ_B^C for every clause in the refutation. Therefore, for any AB-refutation and labelings L, L' such that $L \preceq L'$, the interpolant obtained with L is stronger than the one obtained with L'.

3.5.4 Related Work

Our approach offers a theoretical characterization of a class of interpolation systems which have in common specific structural properties, generalizing existing interpolation systems for first order theories and for propositional logic; in particular, we show how the systems of [KV09, HKV12] and of [DKPW10, Wei12], respectively illustrated in §3.3.3 and in §3.3.4, can be regarded as special cases of our method in the context of arbitrary first order and hyper-resolution inference systems.

When compared to [KV09], the differences and benefits of our approach can be summarized as follows. We derive an algorithm for arbitrary first order theories and inference systems, which extracts interpolants as propositional combinations of formulae from a refutation. Our algorithm can be applied to a class of proofs strictly larger than the class of local proofs in [KV09]; it can also produce a family of interpolants which contains the interpolants of [KV09]. Within this family, we relate and compare the interpolants by their logical strength. The results of [KV09] about the existence of local proofs in the superposition calculus and turning non-local proofs into local ones in the style of [HKV12] can be naturally extended to our framework. Remarkably, our method allows to compute quantifier-free interpolants for problems on which [KV09] can only derive quantified interpolants.

Referring to [Wei12, DKPW10], our approach is different in the following aspects. We integrate the hyper-resolution system into our first order interpolation algorithm, and discuss the applicability of the family of interpolants proposed there. We then extend the class of proofs from first order theories to arbitrary hyper-resolution refutations, and show how the structure of the formulae and inference rules allows to obtain additional interpolants, containing those generated by [Wei12]. Finally, we also compare the produced interpolants by their logical strength.

3.5.5 Summary and Future Developments

In this section we contributed to a theoretical formalization of a generic interpolation approach, based on the adoption of structure and strength as features that affect the quality of interpolants, as well as on the characterization of a class of recursive interpolation systems.

We developed a new parametric interpolation framework for arbitrary first order theories and inference systems, which is able to compute interpolants of different

structure and strength, with or without quantifiers, from the same proof. We described the framework in relation with well-known interpolation algorithms, that respectively address local proofs in first order logic and the propositional hyper-resolution system, and showed that they can be regarded as instantiations of our method.

Future work can consist in adjusting the parametric interpolation framework to cover other existing interpolation systems, as well as to develop new efficient algorithms specialized to various inference systems and theories; a first line of research could concern the generalization of our results about the HR labeled system to the ambit of satisfiability modulo theories, addressing the theory labeled interpolation systems presented in §3.4.

The approaches we focus on in this thesis obtain interpolants from refutations, and the quality of the interpolants is naturally dependent on the quality of the proofs from which they are generated. A second promising direction of study can thus involve a characterization of proofs and of the features that make them "good"; for example, an analysis could be performed of proofs containing only ground formulae or with restricted quantification.

We introduced in §3.5.2 the notion of AB-partition, as a technical requirement that enables interpolation in arbitrary inference systems. A natural question that arises is whether a given refutation admits an AB-partition; even more interesting is to understand which inference systems are able to produce, for any unsatisfiable $A \wedge B$, an AB-refutation that allows at least an AB-partition. To some extent, the works of [KV09, HKV12] answer these questions by considering local proofs, which always admit an AB-partitioning, as seen in §3.5.2. In [HKV12], it is shown that non-local proofs in some cases can be translated into local ones, by existentially quantifying away dirty uninterpreted constants; [KV09] proves instead that an extension of the quantifier-free superposition calculus with quantifier-free linear rational arithmetic always guarantees local proofs. Deriving sufficient and necessary conditions over AB-partitions of AB-refutations is an interesting task to be further investigated.

Relevant structural and semantic aspects of formulae, besides quantification, include size, intended as number of connectives, and content in terms of predicates. On one hand, storing and employing small formulae can make verification more efficient; on the other hand, works as [AM13] show that the presence of particular kinds of predicates improves the likelihood of convergence of certain verification techniques. On the practical side, the impact our work could be assessed on problems related for example to bounded model checking or predicate synthesis, similarly to §3.6, where we provide an experimental evaluation of the effect of interpolants strength and size in SAT-based BMC.

3.6 Impact of Interpolant Strength and Structure

In this section we put into practice the theoretical framework developed in the first part of the chapter and analyze the concrete impact of interpolants features in verification. We address the problem of generating effective interpolants in the context of SAT-based software bounded model checking (BMC) [BCC+03], and study the impact of size and strength. Specifically, we present the PeRIPLO [Rol] framework and discuss its ability to drive interpolation by providing routines that act on complementary levels: (i) manipulation (including compression) of the resolution refutations generated by a SAT solver, from which interpolants are computed, and (ii) systematic variation of the strength of the interpolants, as allowed by the labeled interpolation systems.

As case studies we consider two applications of BMC: verification of a C program incrementally with respect to a number of different properties (as in the FunFrog tool [SFS11]), and incremental verification of different versions of a C program with respect to a fixed set of properties (as in the eVolCheck tool [SFS12b]). Both applications rely on interpolation to generate abstractions of the behavior of function calls (*function summaries*); the goal of summarization is to store and reuse information about already analyzed portions of a program, to make subsequent verification checks more efficient. If summaries (i.e. interpolants) are fit, a remarkable performance improvement is usually achieved; if spurious errors have been introduced due to overapproximation, (some of) the summaries need to be refined, which might be resource-consuming. The challenge we address is to use PeRIPLO to drive the generation of interpolants so as to obtain effective summaries.

In this section we contribute to the state-of-the-art in interpolation-based model checking in the following ways. We present an interpolation framework, PeRIPLO, able to generate individual interpolants and collections of interpolants satisfying particular properties. PeRIPLO offers a set of tunable techniques to manipulate refutations and to obtain interpolants of different strength from them; it can be integrated in any SAT based verification framework which makes use of interpolants.

We provide solid experimental evidence that compact interpolants improve performance in the context of software BMC. To the best of our knowledge, the only previous work to concretely assess the impact of the size of interpolants is [CLV13], which, as discussed in §3.2, addresses the use of interpolants in hardware unbounded model checking.

We present a first systematic evaluation of the impact of interpolant strength in a specific verification domain. We target function summarization in software BMC and show that interpolants of different strength are beneficial to different applications; in particular, stronger and weaker interpolants are respectively suitable for the

FunFrog and eVolCheck approaches. These results match the intuition behind the use of interpolants as function summaries.

3.6.1 PeRIPLO

PeRIPLO (Proof tRansformer and Interpolator for Propositional LOgic) is an open-source SAT solver, built on MiniSAT 2.2.0 [ES04], that provides proof logging, proof manipulation routines and propositional interpolation. It can be used as a standalone tool or as a library; its routines are accessible via configuration file or API. Figure 3.13 illustrates the tool architecture.

PeRIPLO receives as input a propositional formula F from the *verification environment*, and passes it to the *SAT solver*, that checks satisfiability while performing proof logging. If the formula is unsatisfiable, a resolution refutation Π is built in form of a directed acyclic graph.

Π can be further processed by the *proof transformer*, for example it can be compressed or manipulated as a preliminary step to interpolation.

Once Π is available, the environment can ask the *interpolator* for the generation of an individual or a collection of interpolants $\{I_i\}$ by means of an interpolation system Itp, providing a subdivision of F into $A \wedge B$; if the collection is related to some interpolation property P, then an additional checking phase can be enabled to ensure that P is satisfied.

Interpolant Strength. PeRIPLO realizes the labeled interpolation systems of [DKPW10] and allows to systematically vary the strength of the interpolants. It is able to produce both individual interpolants and collections of interpolants, w.r.t. various interpolation properties (e.g., tree interpolation, see §3.6.3) and in accordance with the constraints imposed by the properties on the LISs [RSS12, GRS13]. Chapter 4 will provide a detailed description of such properties and of their relationship with the LISs.

Proof Compression. PeRIPLO allows to compress refutations by means of the following techniques, which target different kinds of redundancies in proofs: (i) the *RecyclePivotsWithIntersection* (RPI) algorithm of [FMP11, BIFH$^+$08], (ii) the *PushdownUnits* (PU) algorithm based on [FMP11], (iii) a *structural hashing* based approach (SH) similar to that of [Cot10], (iv) the *Local Transformation Framework* of [RBS10, RBST, BRST10]. Some manipulation routines are available depending on the LIS chosen: for example, in case of McMillan's system Itp_M it is possible to perform a fast transformation of the refutation to achieve a partial CNFization of the

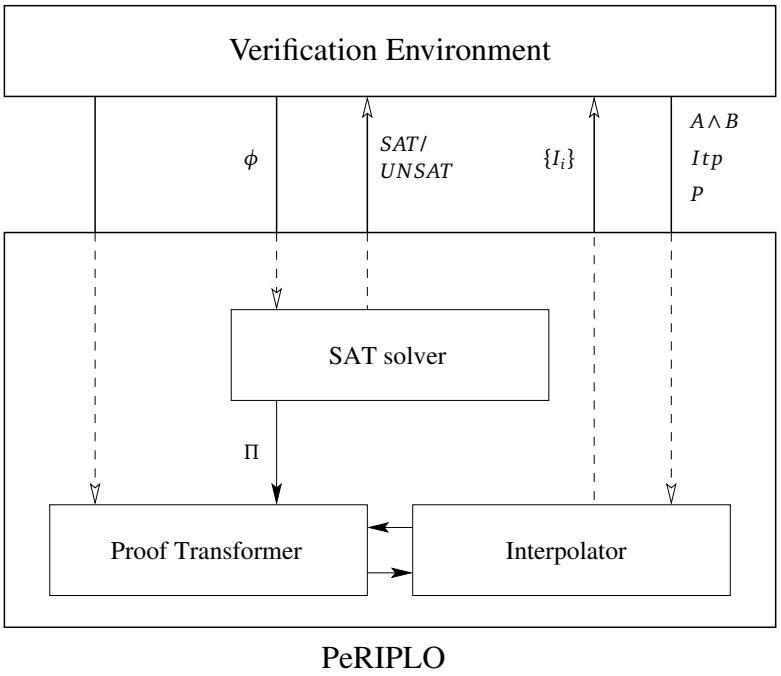


Figure 3.13. PeRIPLO architecture.

interpolant [JM05, RBST]. The local rewriting rules can also be applied to further strengthen or weaken the interpolant with respect to a given LIS [JM05]. PeRIPLO does not implement techniques to directly minimize the interpolants after their generation; nevertheless, structural hashing is performed while building logic formulae, for a more efficient representation in memory. Chapter 5, which is dedicated to the topic of proof manipulation, will discuss the abovementioned approaches to compression in detail.

3.6.2 Function Summaries in Bounded Model Checking

SAT-based BMC is one of the most successful approaches to software verification. It checks a program w.r.t. a property by (i) unwinding loops and recursive function calls up to a given bound, (ii) encoding program and negated property into a propositional formula (*BMC formula*), and (iii) using a SAT solver to check the BMC formula. If the formula is unsatisfiable, the program is safe w.r.t. the bound; otherwise, a satisfying assignment identifies a behavior that violates the property.

We describe in the following two BMC applications which employ interpolation-based *function summaries* as overapproximations of function calls. These applications, respectively implemented in the FunFrog [SFS12a] and eVolCheck [FSS13]

tools, prove suitable to assess, by means of PeRIPLO, the impact size and strength of interpolants can have on verification.

FunFrog. [SFS11] presents a framework to perform incremental verification of a set of properties. Summaries are used to store information about the already analyzed portions of the program, which helps to check subsequent properties more efficiently.

A summary I_f for a function f is an interpolant constructed from an unsatisfiable BMC formula $A_f \wedge B_\pi$, where A_f encodes f and its nested calls, B_π the rest of the program and the negated property π (which holds for the program). While checking the program w.r.t. another property π', the BMC formula changes to $A_f \wedge B_{\pi'}$; I_f is used in place of f: if $I_f \wedge B_{\pi'}$ turns out to be unsatisfiable, then the summary is still valid and π' is proved to hold in the program. If instead $I_f \wedge B_{\pi'}$ is satisfiable, satisfiability could be caused by the overapproximation due to I_f: I_f is replaced by the precise encoding of f and the check is repeated. If $A_f \wedge B_{\pi'}$ is satisfiable, the error is real; if $A_f \wedge B_{\pi'}$ is unsatisfiable, then the error is spurious and I_f is *refined* to a new I'_f.

The ability to reuse summaries depends on their quality. According to our intuition, *accurate summaries* (i.e. *strong interpolants*) are effective in FunFrog: a summary in fact overapproximates the behavior of a function call w.r.t. an assertion; the more precise the summary is, the more closely it reflects the behavior of the corresponding function and the more likely it is to be employed in the verification of subsequent assertions.

eVolCheck. The upgrade checking algorithm of [SFS12b] uses function summarization for BMC in a different way. Verification is done simultaneously w.r.t. a fixed set of properties, but for a program that undergoes modifications. Summaries $\{I_i\}$ are computed for the function calls $\{A_{f_i}\}$ of the original version of the program, and applied to perform local incremental checks of the new version. If the old summaries are general enough to overapproximate the new behavior of the modified functions $\{A_{f'_j}\}$ (i.e. $A_{f'_j} \models I_j$) then the new version is safe. Otherwise, the summaries of the caller functions of the $\{A_{f'_j}\}$ are checked in the same way. If the check succeeds, new summaries $\{I'_j\}$ are generated that *refine* the old $\{I_j\}$. This process continues up to the root of the call tree. If in the end the summary of the main function is proven invalid, then the new version is buggy.

In contrast with FunFrog, *coarse summaries* (i.e. *weak interpolants*) are more suitable for eVolCheck; the underlying intuition is that weaker interpolants represent abstractions which are more "tolerant" and are more likely to remain valid when the

functions are updated.

Compact summaries are expected to yield a more efficient verification both in the FunFrog and eVolCheck frameworks: on one hand, storing and reusing smaller formulae is less expensive, on the other hand, summary reduction via proof compression allows to remove redundancies while keeping the relevant information; in the FunFrog approach, additionally, new summaries are built when possible from refutations involving previously computed summaries.

3.6.3 Experimental Evaluation

We evaluated FunFrog and eVolCheck on a collection of 50 crafted C benchmarks characterized by a non trivial call tree structure reflecting the structure of real C programs used in previous experimentation [SFS12a, FSS13]. The benchmarks contain assertions distributed on different levels of the tree, which makes them particularly suitable for summary-based verification. FunFrog and eVolCheck employ PeRIPLO for symbolic reasoning and interpolation; they provide as input BMC formulae and receive as output interpolants, specifying a LIS depending on the desired interpolant strength. Proof compression techniques can also be applied in order to produce smaller summaries. The experiments were carried out on a 64-bit Ubuntu server featuring a Quad-Core 4GHz Xeon CPU, with a memory threshold of 13GB[2].

FunFrog. In a first phase, FunFrog was run to check the assertions of each benchmark incrementally w.r.t. the call tree, with the goal of maximizing the reuse of summaries.

Consider a program with the following chain of nested calls:

$$main()\{f()\{g()\{h()\{\}Assert_g\}Assert_f\}Assert_{main}\}$$

where $Assert_x$ denotes an assertion in the body of a function x. In a successful scenario, (i) $Assert_g$ is checked and a summary I_h for h is created; (ii) $Assert_f$ is efficiently verified by exploiting I_h (I_g is then built over I_h) and (iii) so is $Assert_{main}$ by means of I_g. Each benchmark was tested in different configurations: with/without performing proof compression before interpolation and choosing one among Itp_M, Itp_P, $Itp_{M'}$ to compute all the interpolants. Compression consisted of a sequential run of LU,SH,RPI (see §3.6.1); this particular combination is effective in reducing proofs, as shown in [RBST].

[2]The full experimental data is available at `http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

eVolCheck. In a second phase, new versions of the benchmarks were created, modifying syntax/semantics of the original programs. First eVolCheck was run to check all assertions at once, yielding a collection of function summaries; then the new program versions were verified w.r.t. the same assertions by using the summaries. As discussed in [SFS12b], while performing upgrade checking the interpolants need to satisfy a property known as *tree interpolation*. In §4.3.2 we will show that tree interpolation is satisfied by Itp_M, Itp_P but not by $Itp_{M'}$; for this reason we only took into account Itp_M and Itp_P for experimentation. Compression was performed as in FunFrog.

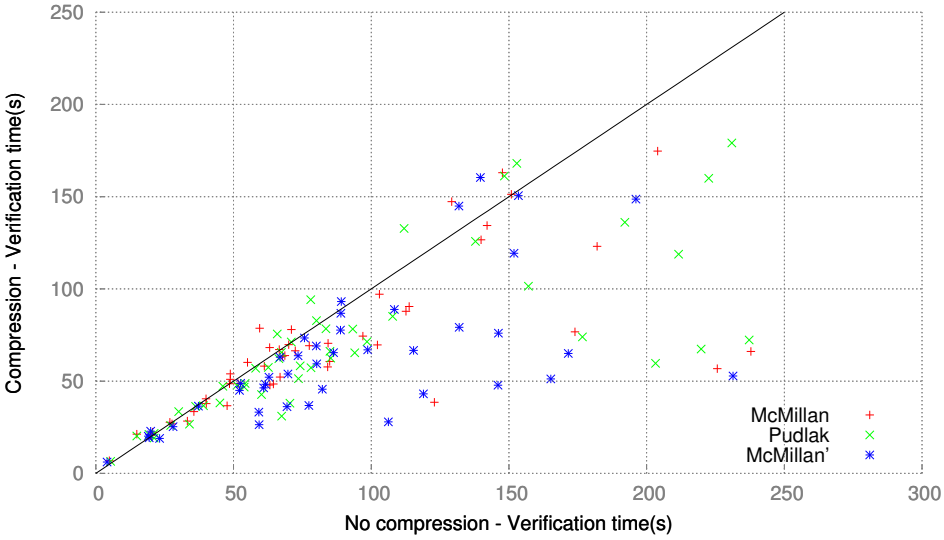
Experimental Results. *Small interpolants* indeed have a strong impact on the performance in both frameworks. Figure 3.14 compares the verification times for the benchmarks in FunFrog (a) and eVolCheck (b), with and without performing proof compression before interpolation. Table 3.10 provides additional statistics for the individual interpolation systems: *#Refinements* denotes the total amount of summary refinements in FunFrog, while *#Invalid summaries* the total number of summaries that in eVolCheck were made invalid because of program updates; $Avg|I|$ and *Time(s)* indicate the average size of interpolants and the average verification time over all the benchmarks; $Time_C/Time_V$ *ratio* is the ratio between the time spent for proof compression and the verification time.

Figure 3.14 and Table 3.10 show the remarkable performance improvement achieved by exploiting proof compression; FunFrog, e.g., obtains a reduction in the average interpolants size Avg$|I|$ up to 95% and a speedup up to 54%. Note also in Figure 3.14 that the effect of compression increases with the complexity of the benchmark; the overhead due to applying compression techniques becomes in fact less and less significant as the benchmark verification time grows.

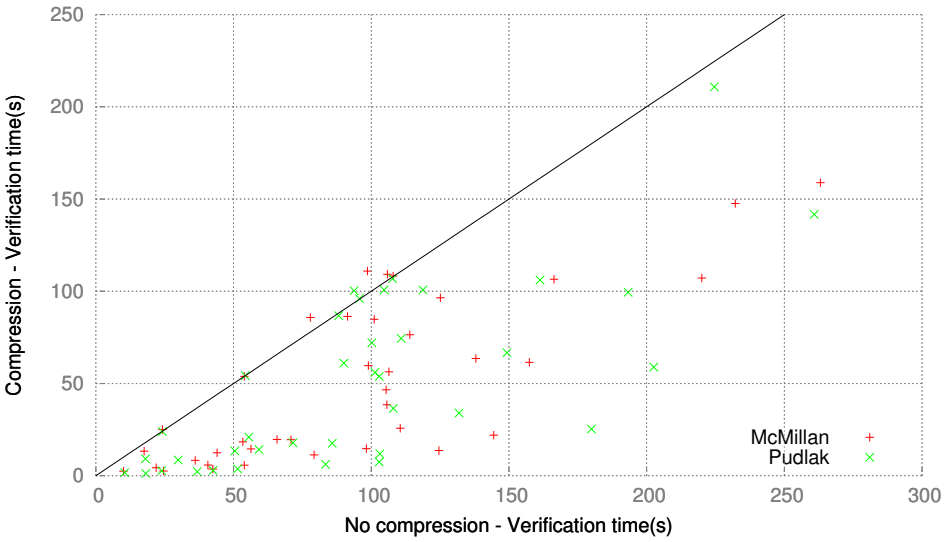
According to the intuitions discussed in §3.6.2, *strong interpolants* prove beneficial in FunFrog, while *weak interpolants* are more suitable for eVolCheck; this is represented in Table 3.10 by a smaller amount of summary refinements in FunFrog and of invalidated summaries in eVolCheck.

The results show that the size of interpolants seems to have definitely an overall greater impact than interpolant strength. Verification time, in fact, is principally determined by the size of the summaries, so that, even in presence of a larger amount of refinements or invalidated summaries, smaller summaries tend to lead to a better performance.

It is important to remark that both size and strength are dependent on the features of the refutations from which the interpolants are produced, as well as on the specific interpolation algorithms, and that these aspects cannot be considered separately. For



(a) FunFrog



(b) eVolCheck

Figure 3.14. Proof compression effect on verification time.

example, in our experimentation we found considerable differences in the size of the interpolants generated by the three LISs, and in the effect of proof compression: interpolants generated with $Itp_{M'}$ in FunFrog were on average twice as big as those generated with Itp_M, but they benefited the most from compression.

Moreover, among all existing refutations for a certain unsatisfiable formula (in-

Table 3.10. Verification statistics for FunFrog and eVolCheck.

(a) FunFrog

No Compression	Itp_M	Itp_P	$Itp_{M'}$
#Refinements	290	298	308
Avg $\|I\|$	38886.62	39372.07	72994.08
Time(s)	4568.08	4929.93	6805.81
Compression	Itp_M	Itp_P	$Itp_{M'}$
#Refinements	293	293	294
Avg $\|I\|$	4336.21	3402.58	3255.69
Time(s)	3327.56	3450.17	3201.72
$\text{Time}_C/\text{Time}_V$ ratio	0.32	0.33	0.32

(b) eVolCheck

No Compression	Itp_M	Itp_P
#Invalid summaries	65	63
Avg $\|I\|$	334554.64	377903.11
Time(s)	4322.57	4402.00
Compression	Itp_M	Itp_P
#Invalid summaries	63	62
Avg $\|I\|$	12579.89	12929.82
Time(s)	2073.79	2057.34
$\text{Time}_C/\text{Time}_V$ ratio	0.19	0.19

cluding those obtained via compression), there might be some which are of better "quality" w.r.t. interpolation by means of LISs. A good refutation could be characterized by a large logical "distance" between the interpolant I yielded by Itp_M and I' yielded by $Itp_{M'}$, where the distance between I and I' – remember that $I \models I'$ – is defined as the number of models of I' that are not models of I. A large distance in this sense would allow for a higher degree of variation in the coarseness of summaries, with direct impact on verification.

3.6.4 Summary and Future Developments

In this section we addressed the key problem of generating effective interpolants in verification by evaluating the impact of size and logical strength in the context of software SAT-based BMC.

To this end, we introduced PeRIPLO, a novel framework that drives interpolation by providing routines for manipulation of the resolution refutations from which the interpolants are computed and for systematic variation of the interpolants strength. As case studies we considered two BMC applications which use interpolation to generate function summaries: (i) verification of a C program incrementally with respect to a number of different properties, and (ii) incremental verification of different versions of a C program with respect to a fixed set of properties. We provided solid experimental evidence that compact interpolants improve the verification performance in the two applications. We also carried out a first systematic evaluation of the impact of strength in a specific verification domain, showing that different applications benefit from interpolants of different strength: specifically, stronger and weaker interpolants are respectively desirable in (i) and (ii).

Future work can move both in a theoretical and in an experimental directions. On the experimental side, new collections of benchmarks can be extracted from pieces of software encoding problems of industrial and academic origin. It would be an interesting task to subject the benchmarks to verification by means of both SAT-based techniques – as we did, employing the FunFrog and eVolCheck frameworks developed by our group – and SMT-based ones, comparing the effect of strength and size reduction by means of proof compression in the two cases.

On the theoretical side, it could be useful to identify families of problems based on their call tree structure, on the position of the assertions within the call tree, and on the content of the assertions themselves.

For example, in application (i), an interpolant is computed with respect to an assertion; the interpolant represents an overapproximation of the behavior of a function, and it is certainly dependent on how much information the assertion (assuming it holds) gives on the behavior of the function. Intuitively, the less tight the assertion is w.r.t. the behavior, the more room there is to tune the strength of interpolants.

It might be even possible to formally characterize problems where, for a certain interpolation-based technique, a stronger or weaker interpolant can be mathematically demonstrated to make convergence faster; such a research, however, should be paired with an experimental investigation to guarantee that similar problems correspond to real software programs.

Chapter 4

Interpolation Properties in Model Checking

In many verification tasks, a single interpolant, i.e., a single subdivision of constraints into two groups A and B, is not sufficient.

Properties such as path interpolation, simultaneous abstraction, interpolation sequence, symmetric interpolation, tree interpolation are used in existing tools like IMPACT [McM06], Whale [AGC12], FunFrog [SFS12a] and eVolCheck [FSS13], which implement instances of predicate abstraction [JM06], lazy abstraction with interpolants (LAWI) [McM06], interpolation-based function summarization [SFS12b, SFS11]. These properties, to which we refer as *collectives* since they concern collections of interpolants, are not satisfied by arbitrary Craig interpolants and must be established for each interpolation algorithm and verification framework, in order to make verification possible.

In this chapter we perform a systematic study of collectives in verification and identify the particular constraints they impose on interpolation systems for propositional logic and first order theories respectively used in SAT- and SMT-based model checking.

A first limitation in the state-of-the-art is that there is no framework which correlates the existing interpolation systems and compares the various collectives; we address the problem and, for the first time, we gather, identify, and uniformly present the most common collectives imposed on interpolation by existing verification approaches (§4.1).

In addition to the issues related to a diversity of interpolation properties, it is desirable to have flexibility in choosing different algorithms for computing different interpolants in a collection, rather than using a single interpolation system. To guarantee such a flexibility, we present a framework which generalizes the tradi-

tional setting consisting of a single interpolation system to allow for collections, or *families*, of interpolation systems.

Families find practical applicability in several contexts. One example is LAWI-style verification, where it is desirable to obtain a collection of path interpolants $\{I_i\}$ with weak interpolants at the beginning (i.e., $I_1, I_2, \ldots$) and strong interpolants at the end (i.e., $\ldots, I_{n-1}, I_n$). This would increase the likelihood of convergence and the performance of the model checking algorithm, and can be achieved by using a family of systems of different strength. Another example is the computation of function summaries by means of interpolation. We discussed in §3.6 how different applications of summarization could require different levels of abstraction by means of interpolation. A system that generates stronger interpolants can yield a tighter abstraction, more closely reflecting the behavior of the corresponding function. On the other hand, a system that generates weaker interpolants would give an abstraction which is more "tolerant" and is more likely to remain valid when the function is updated.

In §4.2 we systematically examine the collectives and the relationships among them; in particular, we show that for families of interpolation systems the collectives form a hierarchy, whereas for a single system all but two (i.e., path interpolation and simultaneous abstraction) are equivalent.

Another issue which this chapter deals with is the fact that there exist different approaches for generating interpolants, as presented in §3.3. The variety of interpolation algorithms makes it difficult to reason about their properties in a systematic manner. At a low level of representation, the challenge is determined by the complexity of individual algorithms and by the diversity among them, which makes it hard to study them uniformly. On the other hand, at a high level, where the details are hidden, not many interesting results can be obtained. For this reason, we adopt a twofold approach, working both at a high and at a low level of representation: at the high level, we give a global view of the entire collection of properties and of their relationships and hierarchy; at the low level, we obtain additional stronger results for concrete interpolation systems. In particular, we first investigate the properties of interpolation systems treating them as black boxes, and then focus on the propositional LISs. In this chapter, the results of §4.2 apply to arbitrary interpolation algorithms, while those of §4.3 apply to LISs.

For the first time, both sufficient and necessary conditions are given for a family of LISs and for a single LIS to enjoy each of the collectives. In particular, we show that in case of a single system path interpolation is common to all LISs, while simultaneous abstraction is as strong as all other properties. Our results have also concrete applications, as discussed in §4.3.2.

Finally, in §4.4, we move to first order theories, and extend some of the results

obtained for LISs to the theory labeled interpolation systems in the context of SMT solving, building on the framework we outlined in §3.4.

4.1 Interpolation Systems

In this section we adapt the notation of §3.3 to deal with collections of interpolants, and then proceed to discuss the collectives, highlighting their use in the context of model checking. We employ the standard convention of identifying conjunctions of formulae with sets of formulae and concatenation with conjunction, whenever convenient. For example, we interchangeably use $\{\phi_1, \ldots, \phi_n\}$ and $\phi_1 \cdots \phi_n$ for $\phi_1 \wedge \cdots \wedge \phi_n$.

Definition 4.1.1 (Interpolation System)**.** An *interpolation system* Itp_S is a function that, given an inconsistent $\Phi = \{\phi_1, \phi_2\}$, returns a Craig interpolant, a formula $I_{\phi_1,S} = Itp_S(\phi_1 \mid \phi_2)$ such that:

$$\phi_1 \models I_{\phi_1,S} \qquad I_{\phi_1,S} \wedge \phi_2 \models \bot \qquad \mathscr{L}_{I_{\phi_1,S}} \subseteq \mathscr{L}_{\phi_1} \cap \mathscr{L}_{\phi_2}$$

The new notation is appropriate to address the generation of multiple interpolants from the same collection of formulae $\Phi = \{\phi_1, \ldots, \phi_n\}$, by splitting the formulae in Φ into two groups A and B in different manners; we write $I_{\phi_1 \cdots \phi_i,S}$ to denote $Itp_S(\phi_1 \cdots \phi_i \mid \phi_{i+1} \cdots \phi_n)$, where $A = \phi_1 \cdots \phi_i$ and $B = \phi_{i+1} \cdots \phi_n$. W.l.o.g., we assume that, for any Itp_S and any formula ϕ, $Itp_S(\top \mid \phi) = \top$ and $Itp_S(\phi \mid \top) = \bot$, where we equate the constant true $\top$ with the empty formula. We omit S whenever clear from the context.

An interpolation system Itp is called *symmetric* if for any inconsistent $\Phi = \{\phi_1, \phi_2\}$:

$$Itp(\phi_1 \mid \phi_2) \Leftrightarrow \overline{Itp(\phi_2 \mid \phi_1)}$$

Definition 4.1.2 (Family of Interpolation Systems)**.** A collection $\mathscr{F} = \{Itp_{S_1}, \ldots, Itp_{S_n}\}$ of interpolation systems is called a *family*.

As in §3.5, we implicitly assume an underlying first order theory when dealing with generic interpolation systems.

4.1.1 Collectives

In the following, we formulate the properties of interpolation systems that are required by existing verification algorithms. Furthermore, we generalize the collectives by presenting them over families of interpolation systems (i.e., we allow the use different systems to generate different interpolants in a collection). Later, we restrict the properties to the more traditional setting of the singleton families.

n-Path Interpolation (PI) has been first defined in [JM06], where it is employed in the refinement phase of CEGAR-based predicate abstraction. It has also appears in [VG09] under the name *interpolation-sequence*, where it is used for a specialized interpolation-based hardware verification algorithm.

Formally, a family of $n+1$ interpolation systems $\{Itp_{S_0}, \ldots, Itp_{S_n}\}$ has the *n-path interpolation* property (n-PI) iff for any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$ and for $0 \leq i \leq n-1$ (recall that $I_\top = \top$ and $I_\Phi = \bot$):

$$I_{\phi_1 \ldots \phi_i, S_i} \wedge \phi_{i+1} \models I_{\phi_1 \ldots \phi_{i+1}, S_{i+1}}$$

n-Generalized Simultaneous Abstraction (GSA) is the generalization of *simultaneous abstraction*, a property that has been introduced, under the name *symmetric interpolation*, in [JM05], where it is used for approximation of a transition relation for predicate abstraction. We changed the name to avoid confusion with the notion of *symmetric interpolation system* (see above). The reason for generalizing the property will be apparent later.

Formally, a family of $n+1$ interpolation systems $\{Itp_{S_1}, \ldots, Itp_{S_{n+1}}\}$ has the *n-generalized simultaneous abstraction* property (n-GSA) iff for any inconsistent $\Phi = \{\phi_1, \ldots, \phi_{n+1}\}$:

$$\bigwedge_{i=1}^{n} I_{\phi_i, S_i} \models I_{\phi_1 \ldots \phi_n, S_{n+1}}$$

The case $n = 2$ is called *Binary GSA (BGSA)*: $I_{\phi_1, S_1} \wedge I_{\phi_2, S_2} \models I_{\phi_1 \phi_2, S_3}$.
If $\phi_{n+1} = \top$, the property is called *n-simultaneous abstraction* (n-SA):

$$\bigwedge_{i=1}^{n} I_{\phi_i, S_i} \models \bot (= I_{\phi_1 \ldots \phi_n, S_{n+1}})$$

and, if $n = 2$, *binary SA (BSA)*. In n-SA $Itp_{S_{n+1}}$ is irrelevant and is often omitted.

n-State-Transition Interpolation (STI) is defined as a combination of PI and SA in a single family of systems. It has been introduced in [AGC12] as part of the inter-procedural verification algorithm Whale. Intuitively, the "state" interpolants overapproximate the set of reachable states, and the "transition" interpolants summarize the transition relations (or function bodies). The STI requirement ensures that state overapproximation is "compatible" with the summarization.

Formally, a family of interpolation systems $\{Itp_{S_0}, \ldots, Itp_{S_n}, Itp_{T_1}, \ldots, Itp_{T_n}\}$ has the *n-state-transition interpolation* property (n-STI) iff for any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$ and for $0 \leq i \leq n-1$:

$$I_{\phi_1 \ldots \phi_i, S_i} \wedge I_{\phi_{i+1}, T_{i+1}} \models I_{\phi_1 \ldots \phi_{i+1}, S_{i+1}}$$

T-Tree Interpolation (TI) is a generalization of classical interpolation used in model checking applications, in which partitions of an unsatisfiable formula naturally correspond to a tree structure such as call tree or program unwinding. The collective has been first defined in [MR13] for analysis of recursive programs, and is equivalent to the nested interpolants of [HHP10].

Formally, let $T = (V, E)$ be a tree with n nodes $V = [1, \ldots, n]$. A family of n interpolation systems $\{Itp_{S_1}, \ldots, Itp_{S_n}\}$ has the *T-tree interpolation property* (T-TI) iff for any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$:

$$\bigwedge_{(i,j)\in E} I_{F_j,S_j} \wedge \phi_i \models I_{F_i,S_i}$$

where $F_i = \{\phi_j \mid i \sqsubseteq j\}$, and $i \sqsubseteq j$ iff node j is a descendant of node i in T. Note that for the root i of T, $F_i = \Phi$ and $I_{F_i,S_i} = \bot$.

An interpolation system Itp_S is said to *have a property P* (or, simply, to have P), where P is one of the properties defined above, if every family induced by Itp_S has P. For example, Itp_S has GSA iff for every k the family $\{Itp_{S_1}, \ldots, Itp_{S_k}\}$, where $Itp_{S_i} = Itp_S$ for all i, has k-GSA.

4.2 Collectives of Interpolation Systems

In this section, we study collectives of general interpolation systems, that is, we treat interpolation systems as black boxes. In section §4.3 we will extend the study to the implementation-level details of LISs.

4.2.1 Collectives of Single Systems

We begin by studying the relationships among the various collectives of single interpolation systems.

Theorem 4.2.1. *Let Itp_S be an interpolation system. The following are equivalent: Itp_S has BGSA (1), Itp_S has GSA (2), Itp_S has TI (3), Itp_S has STI (4).*

Proof. We show that $1 \Rightarrow 2$, $2 \Rightarrow 3$, $3 \Rightarrow 4$, $4 \Rightarrow 1$.

($1 \Rightarrow 2$) Assume Itp_S has BGSA. Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_{n+1}\}$. Then, for $2 \leq i \leq n$:

$$I_{\phi_1 \cdots \phi_{i-1}} \wedge I_{\phi_i} \models I_{\phi_1 \cdots \phi_i}$$

which together yield:

$$\bigwedge_{i=1}^{n} I_{\phi_i} \models I_{\phi_1 \ldots \phi_n}$$

Hence, Itp_S has GSA.

$(2 \Rightarrow 3)$ Let $T = ([1, \ldots, n], E)$, take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. Since Itp_S has GSA:

$$\bigwedge_{(i,j) \in E} I_{F_j} \wedge I_{\phi_i} \models I_{F_i}$$

and, from the definition of Craig interpolation, $\phi_i \models I_{\phi_i}$. Hence, Itp_S has T-TI.

$(3 \Rightarrow 4)$ Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$ and extend it to a Φ' by adding n copies of $\top$ at the end. Define a tree $T_{STI} = ([1, \ldots, 2n], E)$ s.t.:

$$E = \{(n+i, i) \mid 1 \leq i \leq n\} \cup \{(n+i, n+i-1) \mid 1 \leq i \leq n\}$$

Then, for $1 \leq i \leq n$, $F_i = \{\phi_i\}$ and $F_{n+i} = \{\phi_1, \ldots, \phi_i\}$, where F_i is as in the definition of T-TI. By the T-TI property:

$$I_{F_{n+i}} \wedge I_{F_{i+1}} \wedge \top \models I_{F_{n+i+1}}$$

which is equivalent to STI.

$(4 \Rightarrow 1)$ Follows from STI being syntactically equivalent to BGSA for $i = 1$. □

Theorem 4.2.1 has a few simple extensions. First, GSA implies SA directly from the definitions. Similarly, since $\phi \models I_\phi$, STI implies PI. Finally, we conjecture that both SA and PI are strictly weaker than the rest. In §4.3.2 (Theorem 4.3.11), we prove that, for LISs, PI is strictly weaker than SA. As for SA, Proposition 4.2.1 shows that it is equivalent to BGSA in symmetric interpolation systems; but, in the general case, the conjecture remains open.

Proposition 4.2.1. *SA implies BGSA in symmetric interpolation systems.*

Proof. Take any inconsistent $\Phi = \{\phi_1, \phi_2, \phi_3\}$. If an interpolation system has SA, then:

$$I_{\phi_1} \wedge I_{\phi_2} \wedge I_{\phi_3} \models \bot$$

Equivalently,

$$I_{\phi_1} \wedge I_{\phi_2} \models \overline{I_{\phi_3}}$$

For a symmetric system, $\overline{I_{\phi_3}} = I_{\phi_1 \phi_2}$. □

These results define a hierarchy of collectives which is summarized in Figure 4.1, where the edges indicate implications among the collectives. Note that *SA* $\Rightarrow$ *GSA* holds only for symmetric systems.

In summary, the main contribution in the setting of a single system is the proof that almost all collectives are equivalent and the hierarchy of the collectives collapses. From a practical perspective, this means that McMillan's interpolation system Itp_M (implemented by most interpolating SMT solvers) has all of the collective properties, including the recently introduced tree interpolation.

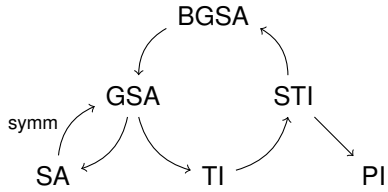


Figure 4.1. Collectives of single systems.

4.2.2 Collectives of Families of Systems

Here, we move from single systems to studying collectives of families of interpolation systems. We first show that the collectives introduced in §4.1 directly extend from families to subfamilies. Second, we examine the hierarchy of the relationships among the properties. Finally, we conclude by discussing the practical implications of these results.

Collectives of Subfamilies. If a family of interpolation systems $\mathscr{F}$ has a property P, then subfamilies of $\mathscr{F}$ have P as well. We prove this statement for the various collectives individually, remarking that k-STI has an application later in the proof of Theorem 4.3.3.

Theorem 4.2.2. *A family $\{Itp_{S_0},\ldots,Itp_{S_n},Itp_{T_1},\ldots,Itp_{T_n}\}$ has n-STI iff for all $k \leq n$ the subfamily $\{Itp_{S_0},\ldots,Itp_{S_k}\} \cup \{Itp_{T_1},\ldots,Itp_{T_k}\}$ has k-STI.*

Proof. ($\Rightarrow$) Assume an inconsistent $\Phi = \{\phi_1,\ldots,\phi_k\}$. We can extend it to a $\Phi' = \{\phi'_1,\ldots,\phi'_n\}$ such that $\phi'_i = \phi_i$, by adding $n-k$ empty formulae $\top$. If $\mathscr{F}$ has the n-STI property, for $0 \leq j \leq k-1$:

$$I_{\phi_1\ldots\phi_j,S_j} \wedge I_{\phi_{j+1},T_{j+1}} \models I_{\phi_1\ldots\phi_{j+1},S_{j+1}}$$

($\Leftarrow$) Follows from $k = n$. □

A simple modification of the previous proof yields:

Theorem 4.2.3. *A family $\{Itp_{S_0},\ldots,Itp_{S_n}\}$ has n-PI iff for all $k \leq n$ the subfamily $\{Itp_{S_0},\ldots,Itp_{S_k}\}$ has k-PI.*

Theorem 4.2.4. *A family $\mathscr{F} = \{Itp_{S_1},\ldots,Itp_{S_{n+1}}\}$ has n-GSA iff for all $k \leq n$ all the subfamilies $\{Itp_{S_{i_1}},\ldots,Itp_{S_{i_{k+1}}}\}$ have k-GSA.*

Proof. ($\Rightarrow$) Let n be a natural number. Take any inconsistent $\Phi = \{\phi_1,\ldots,\phi_{k+1}\}$ such that $k \leq n$. Let $\{i_1,\ldots,i_{k+1}\}$ be a subset of $\{1,\ldots,n+1\}$. Extend Φ to a

$\Phi' = \{\phi'_1, \ldots, \phi'_{n+1}\}$ by adding $(n-k)$ copies of $\top$, so that $\phi'_{i_1} = \phi_1, \ldots, \phi'_{i_k} = \phi_k$, $\phi'_{i_{k+1}} = \phi_{n+1}$. Since $\mathscr{F}$ has n-GSA:

$$\bigwedge_{j=1}^{n} I_{\phi'_j, S_j} \models I_{\phi'_1 \ldots \phi'_n, S_{n+1}}$$

and, since $\phi'_j = \top$ for $j \notin \{i_1, \ldots, i_k\}$:

$$\bigwedge_{j \in \{i_1 \ldots i_k\}} I_{\phi_j, S_j} \models I_{\phi_{i_1 \ldots i_k}, S_{i_{k+1}}}$$

($\Leftarrow$) Follows from $k = n$. □

With the same reasoning we obtain:

Theorem 4.2.5. *A family $\{Itp_{S_1}, \ldots, Itp_{S_n}\}$ has n-SA iff for all $k \leq n$ all the subfamilies $\{Itp_{S_{i_1}}, \ldots, Itp_{S_{i_k}}\}$ have k-SA.*

Theorem 4.2.6. *For a given tree $T = (V, E)$, a family $\{Itp_{S_i}\}_{i \in V}$ has T-TI iff for every subtree $T' = (V', E')$ of T, the family $\{Itp_{S_j}\}_{j \in V'}$ has T'-TI.*

Proof. ($\Rightarrow$) Assume an inconsistent $\Phi = \{\phi_{i_1}, \ldots, \phi_{i_k}\}$ decorating T'. We can extend Φ with $|V'| - |V|$ empty formulae $\top$ to $\Phi' = \{\phi'_1, \ldots, \phi'_n\}$ decorating T. If $\{Itp_{S_i}\}_{v_i \in V}$ has the T-TI property, for all v'_i in V and in particular for all v_i in V'‘:

$$\bigwedge_{(v_i, v_j) \in E'} I_{F_j, S_j} \wedge \phi_i \models I_{F_i, S_i}$$

($\Leftarrow$). Follows from $T' = T$. □

Relationships Among Collectives. We now show the relationships among collectives. First, we note that n-SA and BGSA are equivalent for symmetric interpolation systems. Whenever a family $\mathscr{F} = \{Itp_{S_1}, \ldots, Itp_{S_{n+1}}\}$ has $(n+1)$-SA and $Itp_{S_{n+1}}$ is symmetric, then $\mathscr{F}$ has n-GSA, as proved by Proposition 4.2.2, which is the analogue of Proposition 4.2.1 for single systems.

Proposition 4.2.2. *If a family $\mathscr{F} = \{Itp_{S_1}, \ldots, Itp_{S_{n+1}}\}$ has $(n+1)$-SA and $Itp_{S_{n+1}}$ is symmetric, then $\mathscr{F}$ has n-GSA.*

Proof. Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. Since $\mathscr{F}$ has $(n+1)$-SA, then $I_{\phi_1, S_1} \wedge \cdots \wedge I_{\phi_{n+1}, S_{n+1}} \models \bot$. Assuming $Itp_{S_{n+1}}$ is symmetric, $\overline{I_{\phi_{n+1}, S_{n+1}}} = I_{\phi_1, \ldots, \phi_n, S_{n+1}}$ and the thesis is proved. □

In the rest of the section, we delineate the hierarchy of collectives. In particular, we show that T-TI is the most general collective, immediately followed by n-GSA, which is followed by BGSA and n-STI, which are equivalent, and at last by n-SA and n-PI. The first result is that the n-STI property implies both the n-PI and n-SA properties separately:

Theorem 4.2.7. *If a family $\mathscr{F} = \{Itp_{S_0}, \ldots, Itp_{S_n}, Itp_{T_1}, \ldots, Itp_{T_n}\}$ has n-STI then (1) $\{Itp_{S_0}, \ldots, Itp_{S_n}\}$ has n-PI and (2) $\{Itp_{T_1}, \ldots, Itp_{T_n}\}$ has n-SA.*

Proof. (1) It follows from $\phi_i \models I_{\phi_i, S_i}$ for every i.

(2) Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. If $\mathscr{F}$ has n-STI, then, for $0 \leq i \leq n-1$:

$$I_{\phi_1 \cdots \phi_i, S_i} \wedge I_{\phi_{i+1}, T_{i+1}} \models I_{\phi_1 \cdots \phi_{i+1}, S_{i+1}}$$

Since $I_{\phi_1 \cdots \phi_n} = \bot$, we get $I_{\phi_1, T_1} \wedge \cdots \wedge I_{\phi_n, T_n} \models \bot$. □

A natural question to ask is whether the converse of Theorem 4.2.7 is true. That is, whether the family $\mathscr{F}_1 \cup \mathscr{F}_2$ that combines two arbitrary families $\mathscr{F}_1$ and $\mathscr{F}_2$ that independently enjoy n-PI and n-SA, respectively, has n-STI. We show in §4.3, Theorem 4.3.3, that this is not the case.

As for BGSA, the n-STI property is closely related to it: deciding whether a family $\mathscr{F}$ has n-STI is in fact reducible to deciding whether a collection of subfamilies of $\mathscr{F}$ has BGSA.

Theorem 4.2.8. *A family $\mathscr{F} = \{Itp_{S_0}, \ldots, Itp_{S_n}, Itp_{T_1}, \ldots, Itp_{T_n}\}$ has n-STI iff $\{Itp_{S_i}, Itp_{T_{i+1}}, Itp_{S_{i+1}}\}$ has BGSA for all $0 \leq i \leq n-1$.*

Proof. ($\Rightarrow$) Take any inconsistent $\Phi = \{\phi_1, \phi_2, \phi_3\}$. For $0 \leq i \leq n-1$, extend Φ to a $\Phi' = \{\phi'_1, \ldots, \phi'_n\}$ by adding $(n-3)$ copies of $\top$, so that $\phi'_i = \phi_1$, $\phi'_{i+1} = \phi_2$, $\phi'_{i+2} = \phi_3$. Since $\mathscr{F}$ has n-STI:

$$I_{\phi'_1 \cdots \phi'_i, S_i} \wedge I_{\phi'_{i+1}, T_{i+1}} \models I_{\phi'_1 \cdots \phi'_{i+1}, S_{i+1}}$$

Hence, by construction:

$$I_{\phi_1, S_i} \wedge I_{\phi_2, T_{i+1}} \models I_{\phi_1 \phi_2, S_{i+1}}$$

($\Leftarrow$) Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. Since $\{Itp_{S_i}, Itp_{T_{i+1}}, Itp_{S_{i+1}}\}$ has BGSA, it follows that for $\{\phi'_1, \phi'_2, \phi'_3\}$, where $\phi'_1 = \phi_1 \wedge \cdots \wedge \phi_i$, $\phi'_2 = \phi_{i+1}$, $\phi'_3 = \phi_{i+2} \wedge \cdots \wedge \phi_n$:

$$I_{\phi'_1, S_i} \wedge I_{\phi'_2, T_{i+1}} \models I_{\phi'_1 \phi'_2, S_{i+1}}$$

Hence, by construction:

$$I_{\phi_1 \ldots \phi_i, S_i} \wedge I_{\phi_{i+1}, T_{i+1}} \models I_{\phi_1 \ldots \phi_{i+1}, S_{i+1}}$$

□

From Theorem 4.2.8 and Theorem 4.2.7 we derive:

Corollary 4.2.3. *If there exists a family* $\{Itp_{S_0}, \ldots, Itp_{S_n}\} \cup \{Itp_{T_1}, \ldots, Itp_{T_n}\}$ *s.t.* $\{Itp_{S_i}, Itp_{T_{i+1}}, Itp_{S_{i+1}}\}$ *has BGSA for all* $0 \leq i \leq n-1$*, then* $\{Itp_{T_1}, \ldots, Itp_{T_n}\}$ *has* n*-SA.*

We now relate T-TI and n-GSA. Note that the need for two theorems with different statements arises from the asymmetry between the two properties: all ϕ_i are abstracted by interpolation in n-GSA, whereas in T-TI a formula is not abstracted, when considering the correspondent parent together with its children.

Theorem 4.2.9. *Given a tree* $T = (V, E)$ *if a family* $\mathscr{F} = \{Itp_{S_i}\}_{i \in V}$ *has* T*-TI, then, for every parent* i_{k+1} *and its children* $i_1, \ldots, i_k$*:*

1. *If* i_{k+1} *is the root,* $\{Itp_{S_{i_1}}, \ldots, Itp_{S_{i_k}}\}$ *has* k*-SA.*
2. *Otherwise,* $\{Itp_{S_{i_1}}, \ldots, Itp_{S_{i_k}}, Itp_{S_{i_{k+1}}}\}$ *has* k*-GSA.*

Proof. Take any inconsistent $\Phi = \{\phi_{i_1}, \ldots, \phi_{i_{k+1}}\}$. Consider a parent i_{k+1} and its children $i_1, \ldots, i_k$. If i_{k+1} is not the root, extend Φ to a Φ' in such a way that: the children are decorated with $\phi_{i_1}, \ldots, \phi_{i_k}$, all their descendants and i_{k+1} with $\top$, all the nodes external to the subtree rooted in i_{k+1} with ϕ_{n+1}. Since $\mathscr{F}$ has T-TI, then at node i_{k+1}:

$$\bigwedge_{(i_{k+1}, j) \in E} I_{F_j, S_j} \wedge \phi_{i_{k+1}} \models I_{F_{i_{k+1}}, S_{i_{k+1}}}$$

that is:

$$\bigwedge_{i \in \{i_1 \ldots i_k\}} I_{\phi_i, S_i} \wedge \top \models I_{\phi_{i_1} \cdots \phi_{i_k}, S_{k+1}}$$

If i_{k+1} is the root, the proof simply ignores the presence of $\phi_{i_{k+1}}$ and $S_{i_{k+1}}$. □

Theorem 4.2.10. *Given a tree* $T = (V, E)$*, a family* $\mathscr{F} = \{Itp_{S_i}\}_{i \in V}$ *has* T*-TI if, for every node* i_{k+1} *and its children* $i_1, \ldots, i_k$*, there exists* $T_{i_{k+1}}$ *such that:*

1. *If* i_{k+1} *is the root,* $\{Itp_{S_{i_1}}, \ldots, Itp_{S_{i_k}}, Itp_{T_{i_{k+1}}}\}$ *has* $(k+1)$*-SA.*
2. *Otherwise,* $\{Itp_{S_{i_1}}, \ldots, Itp_{T_{i_{k+1}}}, Itp_{S_{i_{k+1}}}\}$ *has* $(k+1)$*-GSA.*

Proof. Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. Consider a parent i_{k+1} different from the root and its children $i_1, \ldots, i_k$.
If $\{Itp_{S_{i_1}}, \ldots, Itp_{T_{i_{k+1}}}, Itp_{S_{i_{k+1}}}\}$ has k-GSA, for $\{F_{i_1}, \ldots, F_{i_k}, \phi_{i_{k+1}}, \Phi \backslash (\bigcup F_{i_j} \cup \{\phi_{i_{k+1}}\})\}$:

$$\bigwedge_{i \in \{i_1 \ldots i_k\}} I_{F_i, S_i} \wedge I_{\phi_{i_{k+1}}, T_{i_{k+1}}} \models I_{F_{i_{k+1}}, S_{i_{k+1}}}$$

The thesis follows since $\phi_{i_{k+1}} \models I_{\phi_{i_{k+1}},T_{i_{k+1}}}$. If i_{k+1} is the root, $I_{F_{i_{k+1}},S_{i_{k+1}}} = \bot$ and $S_{i_{k+1}}$ is superfluous. □

An important observation is that the T-TI property is the most general, in the sense that it realizes any of the other properties, given an appropriate choice of the tree T. We prove that n-GSA and n-STI can be implemented by T-TI for some T^n_{GSA} and T^n_{STI}; the remaining cases can be derived in a similar manner. Note that the converse implications are not necessarily true in general, since the tree interpolation requirement is stronger.

Theorem 4.2.11. *If a family $\mathscr{F} = \{Itp_{S_{n+1}}, Itp_{S_1}, \ldots, Itp_{S_{n+1}}\}$ has T^n_{GSA}-TI, then $\{Itp_{S_1}, \ldots, Itp_{S_{n+1}}\}$ has n-GSA.*

Proof. Let $T^n_{GSA} = (V, E)$ be the tree shown in Figure 4.2, where $V = \{0, \ldots, n+1\}$ and $E = \{(0, i) \mid 1 \leq i \leq n\} \cup \{(n+1, 0)\}$.

Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_{n+1}\}$. We decorate node 0 with $\top$, all other nodes i with ϕ_i, for $1 \leq i \leq n+1$. Since $\mathscr{F}$ has T-TI, then at node 0:

$$\bigwedge_{(0,j)\in E} I_{F_j,S_j} \wedge \top \models I_{F_0,S_{n+1}}$$

Hence, by construction:

$$\bigwedge_{i=1}^{n} I_{\phi_i,S_i} \models I_{\phi_1 \ldots \phi_n,S_{n+1}}$$

□

Theorem 4.2.12. *If a family $\mathscr{F} = \{Itp_{S_0}, \ldots, Itp_{S_n}\} \cup \{Itp_{T_1}, \ldots, Itp_{T_n}\}$ has T^n_{STI}-TI, then it has n-STI.*

Proof. Let $T^n_{STI} = (V, E)$ be the tree shown in Figure 4.3, where $V = \{1, \ldots, 2n\}$ and $E = \{(n+i, i) \mid 1 \leq i \leq n\} \cup \{(n+i, n+i-1) \mid 1 \leq i \leq n\}$.

Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. For $1 \leq i \leq n$, we decorate i with ϕ_i, $n+i$ with $\top$; similarly we associate i with Itp_{T_i} and $n+i$ with Itp_{S_i}. Since $\mathscr{F}$ has T-TI, then at every node $n+i+1$, for $0 \leq i \leq n-1$:

$$(I_{F_{n+i},S_i} \wedge I_{F_{i+1},T_{i+1}}) \wedge \top \models I_{F_{n+i+1},S_{i+1}}$$

Hence, by construction,

$$I_{\phi_1 \ldots \phi_i,S_i} \wedge I_{\phi_{i+1},T_{i+1}} \models I_{\phi_1 \ldots \phi_{i+1},S_{i+1}}$$

□

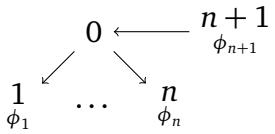


Figure 4.2. T^n_{GSA}.

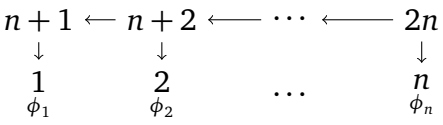


Figure 4.3. T^n_{STI}.

The results of so far (including Theorem 4.3.3 of §4.3) define a hierarchy of collectives which is summarized in Figure 4.4. The solid edges indicate direct implication between properties; $SA \Rightarrow GSA$ requires symmetry, while $GSA \Rightarrow TI$ requires the existence of an additional set of interpolation systems. The dashed edges represent the ability of TI to realize all the other properties for an appropriate tree; only the edges to STI and GSA are shown, the other ones are implicit. The dash-dotted edges represent the subfamily properties.

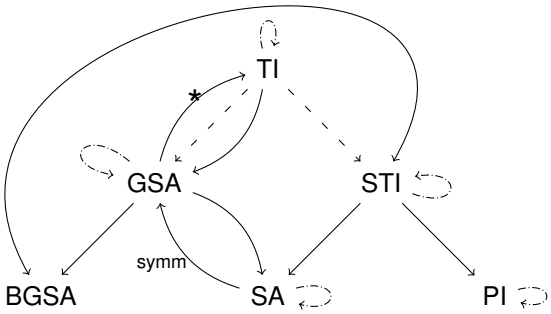


Figure 4.4. Collectives of families of systems.

An immediate application of our results is that they show how to overcome limitations of existing implementations. For example, they enable the trivial construction of tree interpolants in the MathSAT SMT solver [BCF+08] (currently only available in iZ3) – thus enabling its usability for upgrade checking [SFS12b] – by reusing existing BGSA-interpolation implementation of MathSAT. Similarly, our results enable construction of BGSA and GSA interpolants in the iZ3 SMT solver [McM11] (currently only available in MathSAT) – thus enabling the use of iZ3 in Whale [AGC12].

4.3 Collectives of Labeled Interpolation Systems

In this section, we move from the abstract level of general interpolation systems to the implementation level of the propositional labeled interpolation systems; we study collectives of families, then summarize the results for single LISs, also answering the questions left open in §4.2.

In the following, we will make use of the notation introduced in §3.3.4 for the resolution system and the labeled interpolation systems, adapting it to collections of interpolants and systems. Since a labeled system Itp_L is uniquely determined by the labeling L, when discussing a family of LISs $\{Itp_{L_1}, \ldots, Itp_{L_n}\}$ we will refer to the correspondent *family of labelings* as $\{L_1, \ldots, L_n\}$.

Labeling Notation. In the previous sections, we saw how the various collectives involve the generation of multiple interpolants from a single inconsistent formula $\Phi = \{\phi_1, \ldots, \phi_n\}$ for different subdivisions of Φ into an A and a B parts; we refer to these ways of splitting Φ as *configurations*. Remember that a labeling L has freedom in assigning labels only to occurrences of AB-common propositional variables; each configuration identifies these variables. We will say that a variable p has *class* A, B, AB to indicate that p is A-local, B-local, AB-common.

Since we deal with several configurations at a time, it is useful to separate the variables into *partitions* of Φ depending on whether the variables are local to a ϕ_i or shared, taking into account all possible combinations. For example, Table 4.1 is the *labeling table* that characterizes 3-SA.

Table 4.1. 3-SA.

p in ?	Variable $class, label$		
	$\phi_1 \mid \phi_2\phi_3$	$\phi_2 \mid \phi_1\phi_3$	$\phi_3 \mid \phi_1\phi_2$
ϕ_1	A, a	B, b	B, b
ϕ_2	B, b	A, a	B, b
ϕ_3	B, b	B, b	A, a
$\phi_1\phi_2$	AB, α_1	AB, α_2	B, b
$\phi_2\phi_3$	B, b	AB, β_2	AB, β_3
$\phi_1\phi_3$	AB, γ_1	B, b	AB, γ_3
$\phi_1\phi_2\phi_3$	AB, δ_1	AB, δ_2	AB, δ_3

Recall that in 3-SA we are given an inconsistent $\Phi = \{\phi_1, \phi_2, \phi_3\}$ and a family of labelings $\{L_1, L_2, L_3\}$ and generate three interpolants I_{ϕ_1,L_1}, I_{ϕ_2,L_2}, I_{ϕ_3,L_3}. The labeling L_i is associated with the ith configuration. For example, the table shows

that L_1 can independently assign a label from $\{a, b, ab\}$ to each occurrence of each variable shared between ϕ_1 and ϕ_2, ϕ_1 and ϕ_3 or ϕ_1, ϕ_2 and ϕ_3 (as indicated by the presence of $\alpha_1, \gamma_1, \delta_1$).

When talking about an occurrence of a variable p in a certain partition $\phi_{i_1} \cdots \phi_{i_k}$, it is convenient to associate to p and the partition a *labeling vector* $(\eta_{i_1}, \ldots, \eta_{i_k})$, representing the labels assigned to p by $L_{i_1}, \ldots, L_{i_k}$ in configuration $i_1, \ldots, i_k$ (all other labels are fixed). Strength of labeling vectors is compared pointwise, extending the linear order $b \preceq ab \preceq a \preceq \bot$ as described in §3.3.4.2.

We reduce the problem of deciding whether a family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_n}\}$ has an interpolation property P to showing that all labeling vectors of $\{L_1, \ldots, L_n\}$ satisfy a certain set of *labeling constraints*. For simplicity of presentation, in the rest of the chapter we assume that all occurrences of a variable are labeled uniformly. The extension to differently labeled occurrences is straightforward.

4.3.1 Collectives of Families of LISs

We derive in the following both *necessary* and *sufficient* conditions for the collectives to hold in the context of LISs families. The practical significance of our results is to identify which LISs satisfy which collectives. In particular, for the first time, we show that not all labeled interpolation systems satisfy all collectives. The results of our research provide an essential guide for using interpolant strength results when collectives are required (such as in upgrade checking).

We proceed as follows. First, we identify necessary and sufficient labeling constraints to characterize BGSA. Second, we extend them to n-GSA and to n-SA. Third, we exploit the connections between BGSA and n-GSA on one side, and n-STI and T-TI on the other (Theorem 4.2.8, Lemma 4.2.9, Lemma 4.2.10) to derive the labeling constraints both for n-STI and T-TI, thus completing the picture.

4.3.1.1 BGSA

Let $\Phi = \{\phi_1, \phi_2, \phi_3\}$ be an unsatisfiable formula in CNF, $\mathscr{F} = \{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ a family of LISs. We want to identify the restrictions on the labeling vectors of $\{L_1, L_2, L_3\}$ for which $\mathscr{F}$ has BGSA, i.e., $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \models I_{\phi_1\phi_2,L_3}$. We define a set of *BGSA constraints* CC_{BGSA} on labelings as follows:

Definition 4.3.1 (BGSA Constraints CC_{BGSA})**.** A family of labelings $\{L_1, L_2, L_3\}$

satisfies CC_{BGSA} iff:

$$(\alpha_1, \alpha_2), (\delta_1, \delta_2) \preceq \{(ab, ab), (b, a), (a, b)\}$$
$$\beta_2 \preceq \beta_3 \qquad \gamma_1 \preceq \gamma_3$$
$$\delta_1 \preceq \delta_3 \qquad \delta_2 \preceq \delta_3$$

hold for all variables, where α_i, β_i, γ_i and δ_i are as shown in Table 4.2, the labeling table for $BGSA$. $* \preceq \{*_1, *_2\}$ denotes that $* \preceq *_1$ or $* \preceq *_2$ (both can be true).

Table 4.2. BGSA.

p in ?	Variable $class, label$		
	$\phi_1 \mid \phi_2\phi_3$	$\phi_2 \mid \phi_1\phi_3$	$\phi_1\phi_2 \mid \phi_3$
ϕ_1	A, a	B, b	A, a
ϕ_2	B, b	A, a	A, a
ϕ_3	B, b	B, b	B, b
$\phi_1\phi_2$	AB, α_1	AB, α_2	A, a
$\phi_2\phi_3$	B, b	AB, β_2	AB, β_3
$\phi_1\phi_3$	AB, γ_1	B, b	AB, γ_3
$\phi_1\phi_2\phi_3$	AB, δ_1	AB, δ_2	AB, δ_3

We aim to prove that CC_{BGSA} is necessary and sufficient for a family of LISs to have BGSA. On one hand, we claim that, if $\{L_1, L_2, L_3\}$ satisfies CC_{BGSA}, then $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA. It is sufficient to prove the thesis for a set of *restricted BGSA constraints* CC^*_{BGSA}, defined as follows:

Definition 4.3.2 (Restricted BGSA Constraints CC^*_{BGSA})**.** A family of labelings $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA} iff:

$$(\alpha_1, \alpha_2), (\delta_1, \delta_2) \in \{(ab, ab), (b, a), (a, b)\}$$
$$\beta_2 = \beta_3 \qquad \gamma_1 = \gamma_3$$
$$\delta_3 = \max\{\delta_1, \delta_2\}$$

Note that the conditions on the δ_i are equivalent to $(\delta_1, \delta_2, \delta_3) \in \{(ab, ab, ab), (b, a, a), (a, b, a)\}$, due to the order $b \preceq ab \preceq a \preceq \bot$.

Lemma 4.3.1. *If $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA}, then $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA.*

Proof by structural induction. We show that, given a refutation of Φ, for any clause C in the refutation the partial interpolants satisfy:

$$I_{\phi_1, L_1}(C) \wedge I_{\phi_2, L_2}(C) \models I_{\phi_1\phi_2, L_3}(C)$$

that is

$$I_{\phi_1,L_1}(C) \wedge I_{\phi_2,L_2}(C) \wedge \overline{I_{\phi_1\phi_2,L_3}(C)} \models \bot$$

For simplicity, we write I_1, I_2, I_3 to refer to the three partial interpolants for C and, if C has antecedents, we denote their partial interpolants with I_1^+, I_2^+, I_3^+ and I_1^-, I_2^-, I_3^-.

Base case (leaf). Case splitting on C (refer to Table 4.2):

$$C \in \phi_1 \ : \quad I_1 = C|_{1,b} \quad I_2 = \overline{C|_{2,a}} \quad \overline{I_3} = \overline{C|_{3,b}}$$

$$C \in \phi_2 \ : \quad I_1 = \overline{C|_{1,a}} \quad I_2 = C|_{2,b} \quad \overline{I_3} = \overline{C|_{3,b}}$$

$$C \in \phi_3 \ : \quad I_1 = \overline{C|_{1,a}} \quad I_2 = \overline{C|_{2,a}} \quad \overline{I_3} = C|_{3,a}$$

The goal is to show that in each case $I_1 \wedge I_2 \wedge \overline{I_3} \models \bot$. Representing C by grouping variables into the different partitions, with overbraces to show the label assigned to each variable, we have:

$C \in \phi_1$:

$$C|_{1,b} = \overbrace{C_{\phi_1}|_b}^{a} \vee \overbrace{C_{\phi_1\phi_2}|_b}^{\alpha_1} \vee \overbrace{C_{\phi_1\phi_3}|_b}^{\gamma_1} \vee \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_1}$$

$$\overline{C|_{2,a}} = \overbrace{C_{\phi_1}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_2}|_a}^{\alpha_2} \wedge \overbrace{C_{\phi_1\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_2}$$

$$\overline{C|_{3,b}} = \overbrace{C_{\phi_1}|_b}^{a} \wedge \overbrace{C_{\phi_1\phi_2}|_b}^{a} \wedge \overbrace{C_{\phi_1\phi_3}|_b}^{\gamma_3} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_3}$$

$C \in \phi_2$:

$$\overline{C|_{1,a}} = \overbrace{C_{\phi_2}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_2}|_a}^{\alpha_1} \wedge \overbrace{C_{\phi_2\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_1}$$

$$C|_{2,b} = \overbrace{C_{\phi_2}|_b}^{a} \vee \overbrace{C_{\phi_1\phi_2}|_b}^{\alpha_2} \vee \overbrace{C_{\phi_2\phi_3}|_b}^{\beta_2} \vee \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_2}$$

$$\overline{C|_{3,b}} = \overbrace{C_{\phi_2}|_b}^{a} \wedge \overbrace{C_{\phi_1\phi_2}|_b}^{a} \wedge \overbrace{C_{\phi_2\phi_3}|_b}^{\beta_3} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_3}$$

$C \in \phi_3$:

$$\overline{C|_{1,a}} = \overbrace{C_{\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_2\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_3}|_a}^{\gamma_1} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_1}$$

$$\overline{C|_{2,a}} = \overbrace{C_{\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_2\phi_3}|_a}^{\beta_2} \wedge \overbrace{C_{\phi_1\phi_3}|_a}^{b} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_2}$$

$$C|_{3,a} = \overbrace{C_{\phi_3}|_a}^{b} \vee \overbrace{C_{\phi_2\phi_3}|_a}^{\beta_3} \vee \overbrace{C_{\phi_1\phi_3}|_a}^{\gamma_3} \vee \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_3}$$

We can carry out some simplifications, due to the equality constraints in CC^*_{BGSA} and the fact that variables with label a restricted w.r.t. b (and vice versa) are removed, leading (with the help of the resolution rule) to the constraints:

$$(\overbrace{C_{\phi_1\phi_2}|_b}^{\alpha_1} \vee \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_1}) \wedge \overbrace{C_{\phi_1\phi_2}|_a}^{\alpha_2} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_2} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_3} \models \bot$$

$$\overbrace{C_{\phi_1\phi_2}|_a}^{\alpha_1} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_1} \wedge (\overbrace{C_{\phi_1\phi_2}|_b}^{\alpha_2} \vee \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_2}) \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_b}^{\delta_3} \models \bot$$

$$\overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_1} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_2} \wedge \overbrace{C_{\phi_1\phi_2\phi_3}|_a}^{\delta_3} \models \bot$$

Finally, the constraints on (α_1, α_2) and $(\delta_1, \delta_2, \delta_3)$ guarantee that the remaining variables are simplified away, proving the base case.

Inductive step (inner node). The inductive hypothesis (i.h.) consists of $I_1^+ \wedge I_2^+ \wedge I_3^+ \models \bot$, $I_1^- \wedge I_2^- \wedge \overline{I_3^-} \models \bot$. We do a case splitting on the pivot p:

Case 1 (p in ϕ_1).

$$\begin{aligned} I_1 \wedge I_2 \wedge \overline{I_3} &\Leftrightarrow \\ (I_1^+ \vee I_1^-) \wedge (I_2^+ \wedge I_2^-) \wedge \overline{(I_3^+ \vee I_3^-)} &\Leftrightarrow \\ (I_1^+ \vee I_1^-) \wedge I_2^+ \wedge I_2^- \wedge \overline{I_3^+} \wedge \overline{I_3^-} &\Rightarrow \\ (I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee (I_1^- \wedge I_2^- \wedge \overline{I_3^-}) &\Rightarrow^{\text{i.h.}} \bot \end{aligned}$$

Case 2 (p in ϕ_2).

$$\begin{aligned} I_1 \wedge I_2 \wedge \overline{I_3} &\Leftrightarrow \\ (I_1^+ \wedge I_1^-) \wedge (I_2^+ \vee I_2^-) \wedge \overline{(I_3^+ \vee I_3^-)} &\Leftrightarrow \\ I_1^+ \wedge I_1^- \wedge (I_2^+ \vee I_2^-) \wedge \overline{I_3^+} \wedge \overline{I_3^-} &\Rightarrow \\ (I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee (I_1^- \wedge I_2^- \wedge \overline{I_3^-}) &\Rightarrow^{\text{i.h.}} \bot \end{aligned}$$

Case 3 (p in ϕ_3).

$$
\begin{aligned}
I_1 \wedge I_2 \wedge \overline{I_3} &\Leftrightarrow \\
(I_1^+ \wedge I_1^-) \wedge (I_2^+ \wedge I_2^-) \wedge \overline{(I_3^+ \wedge I_3^-)} &\Leftrightarrow \\
I_1^+ \wedge I_1^- \wedge I_2^+ \wedge I_2^- \wedge (\overline{I_3^+} \vee \overline{I_3^-}) &\Rightarrow \\
(I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee (I_1^- \wedge I_2^- \wedge \overline{I_3^-}) &\Rightarrow^{\text{i.h.}} \bot
\end{aligned}
$$

Case 4 (p in $\phi_1\phi_2$). If $(\alpha_1, \alpha_2) = (ab, ab)$:

$$
\begin{aligned}
I_1 \wedge I_2 \wedge \overline{I_3} &\Leftrightarrow \\
(I_1^+ \vee p) \wedge (I_1^- \vee \overline{p}) \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \overline{p}) \wedge \overline{(I_3^+ \vee I_3^-)} &\Rightarrow \\
(I_1^+ \vee p) \wedge (I_1^- \vee \overline{p}) \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \overline{p}) \wedge (\overline{I_3^+} \vee p) \wedge (\overline{I_3^-} \vee \overline{p}) &\Rightarrow \\
((I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee p) \wedge ((I_1^- \wedge I_2^- \wedge \overline{I_3^-}) \vee \overline{p}) &\Rightarrow^{\text{resol}} \\
(I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee (I_1^- \wedge I_2^- \wedge \overline{I_3^-}) &\Rightarrow^{\text{i.h.}} \bot
\end{aligned}
$$

Case 5 (p in $\phi_1\phi_2\phi_3$). If $(\delta_1, \delta_2, \delta_3) = (ab, ab, ab)$:

$$
\begin{aligned}
I_1 \wedge I_2 \wedge \overline{I_3} &\Leftrightarrow \\
(I_1^+ \vee p) \wedge (I_1^- \vee \overline{p}) \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \overline{p}) \wedge \overline{((I_3^+ \vee p) \wedge (I_3^- \vee \overline{p}))} &\Leftrightarrow \\
(I_1^+ \vee p) \wedge (I_1^- \vee \overline{p}) \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \overline{p}) \wedge ((\overline{I_3^+} \wedge \overline{p}) \vee (\overline{I_3^-} \wedge p)) &\Rightarrow \\
((I_1^+ \vee p) \wedge (I_2^+ \vee p) \wedge \overline{I_3^+} \wedge \overline{p}) \vee ((I_1^- \vee \overline{p}) \wedge (I_2^- \vee \overline{p}) \wedge \overline{I_3^-} \wedge p) &\Rightarrow^{\text{resol}} \\
(I_1^+ \wedge I_2^+ \wedge \overline{I_3^+}) \vee (I_1^- \wedge I_2^- \wedge \overline{I_3^-}) &\Rightarrow^{\text{i.h.}} \bot
\end{aligned}
$$

All the remaining cases are treated in a similar manner, to reach a point (possibly after a resolution step if some of the labels are ab) where the inductive hypothesis can be applied. □

Recall that, given two labelings L, L' s.t. $L \preceq L'$, the interpolant I_L obtained from a refutation with L is stronger than the interpolant $I_{L'}$ obtained with L', that is $I_L \models I_{L'}$. The connection between partial order on labelings and LISs and strength of the generated interpolants allows to relax CC^*_{BGSA} to CC_{BGSA}.

Lemma 4.3.2. *The CC^*_{BGSA} constraints:*

$$
\begin{aligned}
&(\alpha_1, \alpha_2), (\delta_1, \delta_2) \in \{(ab, ab), (b, a), (a, b)\} \\
&\beta_2 = \beta_3 \qquad \gamma_1 = \gamma_3 \\
&\delta_3 = \max\{\delta_1, \delta_2\}
\end{aligned}
$$

can be relaxed to CC_{BGSA}:

$$(\alpha_1, \alpha_2), (\delta_1, \delta_2) \preceq \{(ab, ab), (b, a), (a, b)\}$$
$$\beta_2 \preceq \beta_3 \qquad \gamma_1 \preceq \gamma_3$$
$$\delta_1 \preceq \delta_3 \qquad \delta_2 \preceq \delta_3$$

Proof. Let $\Phi = \{\phi_1, \phi_2, \phi_3\}$ be an unsatisfiable formula in CNF. Consider an arbitrary family of LISs $\mathscr{F} = \{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ s.t. $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA}; then, by Lemma 4.3.1, $\mathscr{F}$ has BGSA, i.e.:

$$I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \wedge \overline{I_{\phi_1\phi_2,L_3}} \models \bot$$

Note from Table 4.2, the labeling table for $BGSA$, that $\alpha_1, \gamma_1, \delta_1$ refer to L_1, $\alpha_2, \beta_2, \delta_2$ refer to L_2, and $\beta_3, \gamma_3, \delta_3$ refer to L_3. Consider now an arbitrary $\{L'_1, L'_2, L'_3\}$ that satisfies CC_{BGSA}; due to the definitions of CC_{BGSA} and CC^*_{BGSA}, it derives that $L'_1 \preceq L_1$, $L'_2 \preceq L_2$ and $L'_3 \succeq L_3$. This in turn implies $I_{\phi_1,L'_1} \models I_{\phi_1,L_1}$, $I_{\phi_2,L'_2} \models I_{\phi_2,L_2}$, and $I_{\phi_1\phi_2,L_3} \models I_{\phi_1\phi_2,L'_3}$, yielding:

$$I_{\phi_1,L'_1} \wedge I_{\phi_2,L'_2} \wedge \overline{I_{\phi_1\phi_2,L'_3}} \Rightarrow I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \wedge \overline{I_{\phi_1\phi_2,L_3}} \Rightarrow \bot$$

Thus, the family $\mathscr{F}' = \{Itp_{L'_1}, Itp_{L'_2}, Itp_{L'_3}\}$ has BGSA. □

The above considerations lead to the following result:

Corollary 4.3.3. *If $\{L_1, L_2, L_3\}$ satisfies CC_{BGSA}, then $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA.*

On the other hand, it holds that the satisfaction of the CC_{BGSA} constraints is necessary for BGSA:

Lemma 4.3.4. *If $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA, then $\{L_1, L_2, L_3\}$ satisfies CC_{BGSA}.*

Proof by contradiction. We show that, if any of the CC_{BGSA} constraints is violated, there exist an unsatisfiable formula $\Phi = \{\phi_1, \phi_2, \phi_3\}$ and a refutation such that $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$. The possible violations for the CC_{BGSA} constraints consist of:

1. $(\alpha_1, \alpha_2), (\delta_1, \delta_2) \in \{(a, a), (ab, a), (a, ab)\}$
2. $(\beta_2, \beta_3), (\gamma_1, \gamma_3), (\delta_1, \delta_3), (\delta_2, \delta_3) \in \{(a, ab), (a, b), (ab, b)\}$

It is sufficient to take into account $(\alpha_1, \alpha_2) \in \{(a, a), (a, ab)\}$ and $(\beta_2, \beta_3) \in \{(a, ab), (a, b), (ab, b)\}$. The remaining cases follow by symmetry.

(1) $(\alpha_1, \alpha_2) = (a, a) : \phi_1 = (p \vee \overline{q}) \wedge r,\ \phi_2 = (\overline{p} \vee \overline{r}) \wedge q,\ \phi_3 = s$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vee \overline{q}\,[\bot] \qquad \overline{p} \vee \overline{r}\,[p \wedge r]}{\overline{q} \vee \overline{r}\,[p \wedge r]} \qquad r\,[\bot]}{\overline{q}\,[p \wedge r]} \qquad q\,[\overline{q}]}{\bot\,[(p \wedge r) \vee \overline{q}]}$$

$A = \phi_1 \quad B = \phi_2, \phi_3$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vee \overline{q}\,[\overline{p} \wedge q] \qquad \overline{p} \vee \overline{r}\,[\bot]}{\overline{q} \vee \overline{r}\,[\overline{p} \wedge q]} \qquad r\,[\overline{r}]}{\overline{q}\,[(\overline{p} \wedge q) \vee \overline{r}]} \qquad q\,[\bot]}{\bot\,[(\overline{p} \wedge q) \vee \overline{r}]}$$

$A = \phi_2 \quad B = \phi_1, \phi_3$

We have $I_{\phi_1,L_1} = (p \wedge r) \vee \overline{q}$, $I_{\phi_2,L_2} = (\overline{p} \wedge q) \vee \overline{r}$, $I_{\phi_1\phi_2,L_3} = \bot$ since s is absent from the proof. Then, $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$: a counterexample is $\overline{q}, \overline{r}$.

(2) $(\alpha_1, \alpha_2) = (a, ab) : \phi_1 = (p \vee \overline{q}) \wedge r,\ \phi_2 = (\overline{p} \vee \overline{r}) \wedge q,\ \phi_3 = s$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vee \overline{q}\,[\top] \qquad \overline{p} \vee \overline{r}\,[\bot]}{\overline{q} \vee \overline{r}\,[\overline{p}]} \qquad q\,[\bot]}{\overline{r}\,[(\overline{p} \vee \overline{q}) \wedge q]} \qquad r\,[\top]}{\bot\,[((\overline{p} \vee \overline{q}) \wedge q) \vee \overline{r}]}$$

$A = \phi_2 \quad B = \phi_1, \phi_3$

We have $I_{\phi_1,L_1} = (p \wedge r) \vee \overline{q}$ and $I_{\phi_1\phi_2,L_3} = \bot$ as in (1), while $I_{\phi_2,L_2} = ((\overline{p} \vee \overline{q}) \wedge q) \vee \overline{r}$. Then, $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$: a counterexample is $\overline{q}, \overline{r}$.

(3) $(\beta_2, \beta_3) = (a, b) : \phi_1 = s,\ \phi_2 = (\overline{p} \vee \overline{r}) \wedge q,\ \phi_3 = (p \vee \overline{q}) \wedge r$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vee \overline{q}\,[\top] \qquad \overline{p} \vee \overline{r}\,[\overline{p} \vee \overline{r}]}{\overline{q} \vee \overline{r}\,[\overline{p} \vee \overline{r}]} \qquad r\,[\top]}{\overline{q}\,[\overline{p} \vee \overline{r}]} \qquad q\,[q]}{\bot\,[(\overline{p} \vee \overline{r}) \wedge q]}$$

$A = \phi_1, \phi_2 \quad B = \phi_3$

We have $I_{\phi_1,L_1} = \top$, since s is absent from the proof, while $I_{\phi_2,L_2} = (\overline{p} \wedge q) \vee \overline{r}$ as in (1); $I_{\phi_1\phi_2,L_3} = (\overline{p} \vee \overline{r}) \wedge q$. Then, $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$: a counterexample is $\overline{q}, \overline{r}$.

(4) $(\beta_2, \beta_3) = (a, ab) : \phi_1 = s,\ \phi_2 = (\overline{p} \vee \overline{r}) \wedge q,\ \phi_3 = (p \vee \overline{q}) \wedge r$

$$
\begin{array}{c}
A = \phi_1, \phi_2 \quad B = \phi_3 \\
\dfrac{p \vee \overline{q}\,[\top] \qquad \overline{p} \vee \overline{r}\,[\bot]}{\overline{q} \vee \overline{r}\,[\overline{p}]} \qquad r\,[\top] \\
\dfrac{\overline{q}\,[\overline{p} \vee \overline{r}] \qquad q\,[\bot]}{\bot\,[(\overline{p} \vee \overline{r} \vee \overline{q}) \wedge q]}
\end{array}
$$

$I_{\phi_1,L_1} = \top$ as in (3), $I_{\phi_2,L_2} = (\overline{p} \wedge q) \vee \overline{r}$ as in (1), $I_{\phi_1\phi_2,L_3} = (\overline{p} \vee \overline{r} \vee \overline{q}) \wedge q$. Then, $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$: a counterexample is $\overline{q}, \overline{r}$.

(5) $(\beta_2, \beta_3) = (ab, b) : \phi_1 = s,\ \phi_2 = (\overline{p} \vee \overline{r}) \wedge q,\ \phi_3 = (p \vee \overline{q}) \wedge r$

$I_{\phi_1,L_1} = \top$ as in (3), $I_{\phi_2,L_2} = ((\overline{p} \vee \overline{q}) \wedge q) \vee \overline{r}$ as in (2), $I_{\phi_1\phi_2,L_3} = (\overline{p} \vee \overline{r}) \wedge q$ as in (3). Then, $I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models I_{\phi_1\phi_2,L_3}$: a counterexample is $\overline{q}, \overline{r}$.

□

Having proved that CC_{BGSA} is both sufficient and necessary, we conclude:

Theorem 4.3.1. *A family* $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ *has BGSA if and only if* $\{L_1, L_2, L_3\}$ *satisfies* CC_{BGSA}.

4.3.1.2 n-GSA

After addressing the binary case, we move to defining necessary and sufficient conditions for n-GSA. A family of LISs $\{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA if, for any inconsistent $\Phi = \{\phi_1, \ldots, \phi_{n+1}\}$:

$$I_{\phi_1,L_1} \wedge \cdots \wedge I_{\phi_n,L_n} \models I_{\phi_1 \ldots \phi_n, L_{n+1}}$$

As we defined a set of labeling constraints for BGSA, we now introduce the *n-GSA constraints* (CC_{nGSA}) on a family of labelings.

Definition 4.3.3 (n-GSA Constraints CC_{nGSA})**.** A family of labelings $\{L_1, \ldots, L_{n+1}\}$ satisfies CC_{nGSA} if for every variable with labeling vector $(\alpha_{i_1}, \ldots, \alpha_{i_{k+1}})$, $1 \leq k \leq n$, letting $m = i_{k+1}$ if $i_{k+1} \neq n+1$, $m = i_k$ otherwise:

(1) $(\exists j \in \{i_1, \ldots, i_m\}\ \alpha_j = a) \rightarrow (\forall h \in \{i_1, \ldots, i_m\}\ h \neq j \rightarrow \alpha_h = b)$

Also, if $i_{k+1} = n+1$:

(2) $\forall j \in \{i_1, \ldots, i_k\}, \alpha_j \preceq \alpha_{i_{k+1}}$

That is, if a variable is not shared with ϕ_{n+1}, then, if one of the labels is a, all the others must be b; if the variable is shared with ϕ_{n+1}, condition (1) still holds for $(\alpha_{i_1}, \ldots, \alpha_{i_{k-1}})$, and all these labels must be stronger or equal than $\alpha_{i_{k+1}} = \alpha_{n+1}$. We can prove that these constraints are necessary and sufficient for a family of LISs to have n-GSA; in order to do so, we make use of a set of restricted constraints:

Definition 4.3.4 (Restricted n-GSA Constraints CC^*_{nGSA})**.** A family of labelings $\{L_1, \ldots, L_{n+1}\}$ satisfies CC^*_{nGSA} if for every variable with labeling vector $(\alpha_{i_1}, \ldots, \alpha_{i_{k+1}})$, $1 \leq k \leq n$:

(1) If $i_{k+1} \neq n+1$, then either $(\alpha_{i_1}, \ldots, \alpha_{i_{k+1}}) = (ab, \ldots, ab)$ or

$$\exists! j \in \{i_1, \ldots, i_k\}\ \alpha_j = a \wedge (\forall h \in \{i_1, \ldots, i_m\}\ h \neq j \rightarrow \alpha_h = b)$$

(2) If $i_{k+1} = n+1$, then $\forall j \in \{i_1, \ldots, i_k\}, \alpha_j = \alpha_{i_{k+1}}$

The restriction mechanism is the generalization of the one shown in Definition 4.3.2.

Lemma 4.3.5. *If $\{L_1, \ldots, L_{n+1}\}$ satisfies CC^*_{nGSA}, then the family $\{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA.*

Proof by structural induction. We prove that, given a refutation of Φ, for any clause C in the refutation the partial interpolants satisfy:

$$I_{\phi_1, L_1}(C) \wedge \cdots \wedge I_{\phi_n, L_n}(C) \models I_{\phi_1 \ldots \phi_n, L_{n+1}}(C)$$

that is

$$I_{\phi_1, L_1}(C) \wedge \cdots \wedge I_{\phi_n, L_n}(C) \wedge \overline{I_{\phi_1 \ldots \phi_n, L_{n+1}}(C)} \models \bot$$

Base case (leaf). Remember that, if $C \in \phi_i, i \neq n+1$, $C \in A$ in configuration i (hence the partial interpolant is $C|_{i,b}$) and in configuration $n+1$ ($\overline{C|_{n+1,b}}$) and $C \in B$ in all the other configurations $j \neq i, n+1$ ($\overline{C|_{j,a}}$). If $C \in \phi_{n+1}$, $C \in B$ in all

configurations ($C|_{n+1,a}$ in configuration $n+1$, $\overline{C|_{i,a}}$ everywhere else). So we need to prove:

$$\overline{C|_{1,a}} \wedge \cdots \wedge \overline{C|_{i-1,a}} \wedge C|_{i,b} \wedge \overline{C|_{i+1,a}} \wedge \cdots \wedge \overline{C|_{n,a}} \wedge \overline{C|_{n+1,b}} \models \bot$$

$$\overline{C|_{1,a}} \wedge \cdots \wedge \overline{C|_{i-1,a}} \wedge \overline{C|_{i,a}} \wedge \overline{C|_{i+1,a}} \wedge \cdots \wedge \overline{C|_{n,a}} \wedge C|_{n+1,a} \models \bot$$

respectively for $i \neq n+1$ and $i = n+1$.

We can divide the variables of $C \in \phi_i$ into partitions, obtaining $C = C_{\phi_i} \vee C_{\phi_i\phi_2} \vee \cdots \vee C_{\phi_1 \ldots \phi_n}$, leading to a system of constraints as shown for BGSA; the conjunction of:

$$\overline{(C_{\phi_i} \vee C_{\phi_i\phi_2} \vee \cdots \vee C_{\phi_1 \ldots \phi_n})|_{1,a}}$$

$$\vdots$$

$$(C_{\phi_i} \vee C_{\phi_i\phi_2} \vee \cdots \vee C_{\phi_1 \ldots \phi_n})|_{i,b}$$

$$\vdots$$

$$\overline{(C_{\phi_i} \vee C_{\phi_i\phi_2} \vee \cdots \vee C_{\phi_1 \ldots \phi_n})|_{n,a}}$$

must imply $\bot$ for every ϕ_i, $i \neq n+1$ (similarly for ϕ_{n+1}). All the simplifications are carried out in line with the proof of Lemma 4.3.1.

Inductive step (inner node). The proof is a again a direct generalization of the proof of Lemma 4.3.1.

Performing a case splitting on the pivot and on its labeling vector, the starting point is a conjunction of the partial interpolants $I_1 \wedge \cdots \wedge I_n \wedge \overline{I_{n+1}}$ of C, which is then expressed in terms of the partial interpolants for the antecedents. The goal is to reach a formula $\psi = (I_1^+ \wedge \cdots \wedge I_n^+ \wedge \overline{I_{n+1}^+}) \vee (I_1^- \wedge \cdots \wedge I_n^- \wedge \overline{I_{n+1}^-})$ where the inductive hypothesis can be applied.

The key observation is that the restricted CC_{nGSA} constraints give rise to a combination of boolean operators (after the dualization of the ones in $\overline{I_{n+1}}$ due to the negation) which makes it always possible to obtain the desired ψ, possibly with the help of the resolution rule. □

Similarly to what done for BGSA in Lemma 4.3.2, the CC^*_{nGSA} constraints can be relaxed to CC_{nGSA}, yielding the following result:

Lemma 4.3.6. *If $\{L_1, \ldots, L_{n+1}\}$ satisfies CC_{nGSA}, then the family $\{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA.*

We move now from the sufficient to the necessary conditions:

Lemma 4.3.7. *If a family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA, then $\{L_1, \ldots, L_{n+1}\}$ satisfies CC_{nGSA}.*

Proof by induction and contradiction. We prove the theorem by strong induction on $n \geq 2$.

Base Case ($n = 2$). Follows by Lemma 4.3.4.

Inductive Step. Assume the thesis holds for all $k \leq n-1$, we prove it for $k = n$. By Lemma 4.2.4, if a family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA, then any subfamily of size $k+1 \leq n$ has k-GSA. Combined with the inductive hypothesis, this implies that it is sufficient to establish the theorem for every variable p and labeling vectors $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ and $\vec{\beta} = (\beta_1, \ldots, \beta_{n+1})$ corresponding to partitions $\phi_1 \cdots \phi_n$ and $\phi_1 \cdots \phi_{n+1}$, respectively.

We only show the case of $\vec{\alpha}$. The proof for $\vec{\beta}$ is analogous. W.l.o.g., assume that there is a p such that $\vec{\alpha}$ violates CC_{nGSA} for $\alpha_1 = \alpha_2 = a$ (other cases are symmetric). Construct a family of labelings $\{L'_1, L'_2, L'_{n+1}\}$ from $\{L_1, \ldots, L_{n+1}\}$ by (1) taking all labelings of partitions involving only subsets of ϕ_1, ϕ_2 and ϕ_{n+1}. For example, vectors (η_3, η_4) and $(\eta_1, \eta_2, \eta_3, \eta_{n+1})$ would be discarded, while (η_1, η_2) and $(\eta_1, \eta_2, \eta_{n+1})$ would be kept; and (2) for p, set the labeling vector of partition $\phi_1\phi_2$ to $(\alpha_1, \alpha_2) = (a, a)$. By Lemma 4.3.4, $\{L'_1, L'_2, L'_{n+1}\}$ does not have BGSA. Let $\Phi' = \{\phi_1, \phi_2, \phi_{n+1}\}$ be such that $I_{\phi_1, L'_1} \wedge I_{\phi_2, L'_2} \not\models I_{\phi_1\phi_2, L'_{n+1}}$, and let Π be the corresponding resolution refutation.

Construct $\Phi = \{\phi_1, \phi_2, p, \ldots, p, \phi_{n+1}\}$ by adding $(n-2)$ copies of p to Φ'. Φ is unsatisfiable, and Π is also a valid refutation for Φ. From this point, we assume that all interpolants are generated from Π.

Assume, by contradiction, that $\mathscr{F}$ has n-GSA. Then,

$$I_{\phi_1, L_1} \wedge \cdots \wedge I_{\phi_n, L_n} \models I_{\phi_1 \cdots \phi_n, L_{n+1}}$$

But, because $\phi_3, \ldots, \phi_n$ do not contribute any clauses to Π, $I_{\phi_i, L_i} = \top$ for $3 \leq i \leq n$. Hence,

$$I_{\phi_1, L_1} \wedge I_{\phi_2, L_2} \models I_{\phi_1\phi_2, L_{n+1}}$$

However, by construction:

$$I_{\phi_1, L_1} = I_{\phi_1, L'_1} \qquad I_{\phi_2, L_2} = I_{\phi_2, L'_2} \qquad I_{\phi_1\phi_2, L_{n+1}} = I_{\phi_1\phi_2, L'_{n+1}}$$

which leads to a contradiction. Hence $\vec{\alpha}$ must satisfy CC_{nGSA}.

□

We can now state the main theorem:

Theorem 4.3.2. *A family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_{n+1}}\}$ has n-GSA if and only if $\{L_1, \ldots, L_{n+1}\}$ satisfies CC_{nGSA}.*

4.3.1.3 n-SA

The constraints for n-SA follow as a special case of CC_{nGSA}:

Definition 4.3.5 (n-SA Constraints CC_{nSA})**.** A family of labelings $\{L_1, \ldots, L_n\}$ satisfies CC_{nSA} if for every variable with labeling vector $(\alpha_{i_1}, \ldots, \alpha_{i_k})$, $2 \leq k \leq n$, it holds that:

$$(\exists j \in \{i_1, \ldots, i_k\}\, \alpha_j = a) \rightarrow (\forall h \in \{i_1, \ldots, i_k\}\, h \neq j \rightarrow \alpha_h = b)$$

Corollary 4.3.8. *A family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_n}\}$ has n-SA if and only if $\{L_1, \ldots, L_n\}$ satisfies CC_{nSA}.*

In particular, for $n = 2$, we have:

Corollary 4.3.9. *A family $\{Itp_{L_1}, Itp_{L_2}\}$ has 2-SA if and only if $\{L_1, L_2\}$ satisfies $(\alpha_1, \alpha_2) \preceq \{(ab, ab), (a, b), (b, a)\}$.*

Recall from §3.3.4 that the LIS corresponding to Pudlák's system Itp_P is obtained by labeling all *AB*-common propositional variables with ab. The set of all families of labelings $\{L'_1, \ldots, L'_n\}$ s.t. every $Itp_{L'_i}$ is stronger or equivalent to Itp_P is thus a subset of the set of all families of labelings $\{L_1, \ldots, L_n\}$ satisfying CC_{nSA}; the inclusion is strict since CC_{nSA} also allows for tuples of labels (e.g., $(\alpha_1, \alpha_2) = (a, b)$ or $(\delta_1, \delta_3, \delta_2) = (a, b, b)$) not comparable with Pudlák's labeling.

The result of Corollary 4.3.8 allows then to formally relate Itp_P and n-SA:

Corollary 4.3.10. *Suppose that, in a family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_n}\}$, each Itp_{L_i} is stronger or equivalent to Itp_P. Then $\mathscr{F}$ has n-SA.*

Moreover, a family that has $(n+1)$-SA also has n-GSA if the last member of the family is Pudlák's system. In fact, from Proposition 4.2.2 and Pudlák's system being symmetric (as shown in [Hua95]), it follows that *if a family $\{Itp_{L_1}, \ldots, Itp_{L_n}, Itp_P\}$ has $(n+1)$-SA, then it has n-GSA*; this in turn yields:

Corollary 4.3.11. *Suppose that, in a family $\mathscr{F} = \{Itp_{L_1}, \ldots, Itp_{L_n}\}$, each Itp_{L_i} is stronger or equivalent to Itp_P. Then $\mathscr{F}' = \{Itp_{L_1}, \ldots, Itp_{L_n}, Itp_P\}$ has n-GSA.*

Besides directly proving the necessary and sufficient labeling constraints for n-GSA, and deriving those for n-SA as a corollary, we can show an alternative way to build a family $\{Itp_{L_1}, \ldots, Itp_{L_n}\}$ which enjoys n-SA by applying the following Lemma 4.3.12.

Lemma 4.3.12. *If a family $\{Itp_{L_0}, Itp_{L_1}, Itp_{L_3}, Itp_{L_4}\}$ has 3-PI , then an L_2 can always be found s.t. $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA.*

Proof. Follows as a corollary of Theorem 4.3.5 and from the fact that, in 3-PI, the only relevant condition is:

$$I_{\phi_1,L_1} \wedge \phi_2 \models I_{\phi_1\phi_2,L_2}$$

since, for any L_0, L_1, L_2, L_3, $\top = I_{\top,L_0} \wedge \phi_1 \models I_{\phi_1,L_1}$ and $I_{\phi_1\phi_2,L_2} \wedge \phi_3 \models I_{\phi_1\phi_2\phi_3,L_3} = \bot$. □

We can combine Lemma 4.3.12 and Theorem 4.3.8 of §4.3.2 concerning PI for a single LIS in order to build a family $\{Itp_{L_1}, \ldots, Itp_{L_n}\}$ that has n-SA.

Theorem 4.3.8 states that, given any L, Itp_L has PI, in particular for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$:

$$I_{\phi_1,L} \wedge \phi_2 \models I_{\phi_1\phi_2,L}$$

Lemma 4.3.12 shows that $\{L, L\}$ can be extended to an $\{L, L', L\}$ such that $\{Itp_L, Itp_{L'}, Itp_L\}$ has BGSA, that is, for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$:

$$I_{\phi_1,L} \wedge I_{\phi_2,L'} \models I_{\phi_1\phi_2,L}$$

We iteratively generate a collection of families $\{L, L'_i, L\}$ that have BGSA, i.e.:

$$I_{\phi_1,L} \wedge I_{\phi_2,L'_i} \models I_{\phi_1\phi_2,L}$$

for $0 \leq i \leq n$; following Theorem 4.2.8 relating n-STI and BGSA, we then obtain a family $\mathscr{F}$ that has n-STI. Finally, from $\mathscr{F}$ we extract, as in Theorem 4.2.7, a subfamily $\{Itp_{L'_1}, \ldots, Itp_{L'_n}\}$ that has n-SA.

After investigating n-GSA and n-SA, we address two questions which were left open in §4.2.2: do n-SA and n-PI imply n-STI? Is the requirement of additional interpolation systems necessary to obtain T-TI from n-GSA? We show here that n-SA and n-PI do not necessarily imply n-STI, and that, for LISs, n-GSA and T-TI are equivalent.

4.3.1.4 n-STI

Theorem 4.2.7 shows that if a family has n-STI, then it has both n-SA and n-PI. We prove that the converse is not necessarily true. First, it is not difficult to prove that any family $\{Itp_{L_0}, Itp_{L_1}, Itp_{L_2}\}$ has 2-PI:

Proposition 4.3.13. *Any family* $\{Itp_{L_0}, Itp_{L_1}, Itp_{L_2}\}$ *has* 2*-PI.*

Proof. Recall that $I_{\top,L_0} = \top$ and $I_{\phi_1\phi_2,L_2} = \bot$ for any L_0, L_2. Hence, 2-PI reduces to the following two conditions: $\phi_1 \models I_{\phi_1,L_1}$, $I_{\phi_1,L_1} \wedge \phi_2 \models \bot$, which are true of any Craig interpolant. □

A second result is that:

Lemma 4.3.14. *There exists a family* $\{Itp_{L_0}, Itp_{L_1}, Itp_{L_2}\}$ *that has* 2*-PI and a family* $\{Itp_{L'_1}, Itp_{L'_2}\}$ *that has* 2*-SA, but the family* $\{Itp_{L_0}, Itp_{L_1}, Itp_{L_2}, Itp_{L'_1}, Itp_{L'_2}\}$ *does not have* 2*-STI.*

Proof. By Theorem 4.2.8, a necessary condition for 2-STI is that $\{Itp_{L_1}, Itp_{L'_2}, Itp_{L_2}\}$ has BGSA. By Proposition 4.3.13, $\{L_0, L_1, L_2\}$ can be arbitrary. By Theorem 4.3.1 and Corollary 4.3.9, there exists $\{L'_1, L'_2\}$ such that $\{Itp_{L'_1}, Itp_{L'_2}\}$ has 2-SA, but $\{Itp_{L_1}, Itp_{L'_2}, Itp_{L_2}\}$ does not have BGSA. □

We obtain the main result applying the STI subfamily property (Theorem 4.2.2):

Theorem 4.3.3. *There exists a family* $\{Itp_{S_0}, \ldots, Itp_{S_n}\}$ *that has* n*-PI, and a family* $\{Itp_{T_1}, \ldots, Itp_{T_n}\}$ *that has* n*-SA, but the family* $\{Itp_{S_0}, \ldots, Itp_{S_n}\} \cup \{Itp_{T_1}, \ldots, Itp_{T_n}\}$ *does not have* n*-STI.*

4.3.1.5 T-TI

In this section we discuss tree interpolation. Theorem 4.2.10 shows how T-TI can be obtained by multiple applications of GSA at the level of each parent and its children, provided that we can find an appropriate labeling to generate an interpolant for the parent. We prove here that, in the case of LISs, this requirement is not needed, and derive explicit constraints on labelings for T-TI.

Let us define *n-GSA strengthening* any property derived from n-GSA by not abstracting any of the subformulae ϕ_i, for example:

$$I_{\phi_1,L_1} \wedge \cdots \wedge I_{\phi_{n-1},L_{n-1}} \wedge \phi_n \models I_{\phi_1 \ldots \phi_n, L_{n+1}}$$

It can be proved that:

Lemma 4.3.15. *The set of labeling constraints of any* n*-GSA strengthening is a subset of constraints of* n*-GSA.*

Proof. Assume w.l.o.g we strengthen the first subformula ϕ_1. Then any variable in any partition which does not involve ϕ_1 has the same labeling vector and its n-GSA labeling constraints are also the same. Instead, variables in any partition $\phi_1\phi_{i_2}\ldots\phi_{i_k}$ have now a labeling vector $(\alpha_{i_2}, \ldots, \alpha_{i_k})$, where the first component α_1 is missing. Referring to the definition of CC_{nGSA}, it is easy to verify that the set of the constraints for the strengthening are a subset of the constraints for n-GSA. □

From Theorem 4.2.10 and Lemma 4.3.15, it follows that:

Lemma 4.3.16. *Given a tree $T = (V, E)$ a family $\{Itp_{S_i}\}_{i \in V}$ has T-TI if, for every parent i_{k+1} and its children $i_1, \ldots, i_k$, the family of labelings of the $(k+1)$-GSA strengthening obtained by non abstracting the parent satisfies the correspondent subset of $(k+1)$-GSA constraints.*

Note that, in contrast to Theorem 4.2.10, in the case of LISs we do not need to ensure the existence of an additional set of interpolation systems to abstract the parents. The symmetry between the necessary and sufficient conditions given by Theorem 4.2.10 and Theorem 4.2.9 is restored, and we establish:

Theorem 4.3.4. *Given a tree $T = (V, E)$ a family $\{Itp_{S_i}\}_{i \in V}$ has T-TI if and only if for every parent i_{k+1} and its children $i_1, \ldots, i_k$, the family of labelings of the $(k+1)$-GSA strengthening obtained by non abstracting the parent satisfies the correspondent subset of $(k+1)$-GSA constraints.*

Alternatively, in the case of LISs, the additional interpolation systems can be constructed explicitly:

Theorem 4.3.5. *Any $\mathscr{F} = \{Itp_{L_{i_1}}, \ldots, Itp_{L_{i_k}}, Itp_{L_{n+1}}\}$ s.t. $k < n$ that has an n-GSA strengthening property can be extended to a family that has n-GSA.*

Proof. Refer to the definition of CC_{nGSA} and to Lemma 4.3.15. We can complete $\mathscr{F}$ for example by introducing $n - k$ instances of McMillan's system Itp_M. Both constraints (1) and (2) for n-GSA are satisfied, since Itp_M always assigns label b (recall the order $b \preceq ab \preceq a$). Note that Itp_M is not necessarily the only possible choice. □

4.3.1.6 n-PI

Theorem 4.2.7 derives n-SA from n-STI, which in turn is reduced to BGSA by Theorem 4.2.8: a family $\mathscr{F} = \{Itp_{S_0}, \ldots, Itp_{S_n}, Itp_{T_1}, \ldots, Itp_{T_n}\}$ has n-STI iff $\{Itp_{S_i}, Itp_{T_{i+1}}, Itp_{S_{i+1}}\}$ has BGSA for all $0 \leq i \leq n-1$.

We now prove a variant of Theorem 4.2.8 for n-PI, by exploiting the notion of n-GSA strengthening introduced while discussing tree interpolation.

Theorem 4.3.6. *A family $\mathscr{F} = \{Itp_{L_0}, \ldots, Itp_{L_n}\}$ has n-PI iff, for all $0 \leq i \leq n-1$, $\{Itp_{L_i}, Itp_{L_{i+1}}\}$ has the BGSA strengthening* BST *obtained from BGSA by non abstracting the second subformula.*

Proof. ($\Rightarrow$). Take any inconsistent $\Phi = \{\phi_1, \phi_2, \phi_3\}$. For $0 \leq i \leq n-1$, extend Φ to a $\Phi' = \{\phi'_1, \ldots, \phi'_n\}$ by adding $(n-3)$ copies of $\top$, so that $\phi'_i = \phi_1$, $\phi'_{i+1} = \phi_2$, $\phi'_{i+2} = \phi_3$. Since $\mathscr{F}$ has n-PI:

$$I_{\phi'_1 \cdots \phi'_i, S_i} \wedge \phi'_{i+1} \models I_{\phi'_1 \cdots \phi'_{i+1}, L_{i+1}}$$

Hence, by construction, BST holds:

$$I_{\phi_1, L_i} \wedge \phi_2 \models I_{\phi_1 \phi_2, L_{i+1}}$$

($\Leftarrow$) Take any inconsistent $\Phi = \{\phi_1, \ldots, \phi_n\}$. Since $\{Itp_{L_i}, Itp_{L_{i+1}}\}$ has BST, it follows that for $\{\phi'_1, \phi'_2, \phi'_3\}$, where $\phi'_1 = \phi_1 \wedge \cdots \wedge \phi_i$, $\phi'_2 = \phi_{i+1}$, $\phi'_3 = \phi_{i+2} \wedge \cdots \wedge \phi_n$:

$$I_{\phi'_1, L_i} \wedge \phi'_2 \models I_{\phi'_1 \phi'_2, L_{i+1}}$$

Hence, by construction:

$$I_{\phi_1 \ldots \phi_i, L_i} \wedge \phi_{i+1} \models I_{\phi_1 \ldots \phi_{i+1}, L_{i+1}}$$

□

Table 4.3. BST.

p in ?	Variable $class, label$	
	$\phi_1 \mid \phi_2\phi_3$	$\phi_1\phi_2 \mid \phi_3$
ϕ_1	A, a	A, a
ϕ_2	B, b	A, a
ϕ_3	B, b	B, b
$\phi_1\phi_2$	AB, α_1	A, a
$\phi_2\phi_3$	B, b	AB, β_3
$\phi_1\phi_3$	AB, γ_1	AB, γ_3
$\phi_1\phi_2\phi_3$	AB, δ_1	AB, δ_3

Thanks to Lemma 4.3.15, we derive the labeling constraints of BST, and as a consequence the constraints of n-PI, as a subset of the BGSA constraints:

Definition 4.3.6 (BST Constraints CC_{BST})**.**

$$\gamma_1 \preceq \gamma_3 \qquad \delta_1 \preceq \delta_3$$

As BST is obtained from BGSA by non abstracting the second formula, Table 4.3, the labeling table for BST, follows from Table 4.2 by omitting the second column.

We conclude stating the main result:

Theorem 4.3.7. *A family $\mathscr{F} = \{Itp_{L_0}, \ldots, Itp_{L_n}\}$ has n-PI iff, for all $0 \leq i \leq n-1$, $\{L_i, L_{i+1}\}$ satisfies CC_{BST}.*

4.3.2 Collectives of Single LISs

In the following, we highlight the fundamental results in the context of single LISs, which represent the most common application of the framework of [DKPW10] to SAT-based model checking.

A first result, important for its practical applications, is that any LIS satisfies PI:

Theorem 4.3.8. *PI holds for all single LISs.*

Proof. In the previous section we addressed n-PI for a family of LISs $\{Itp_{L_0}, \ldots, Itp_{L_n}\}$. We identified a set of constraints for each $\{L_i, L_{i+1}\}$ as:

$$\gamma_1 \preceq \gamma_3 \qquad \delta_1 \preceq \delta_3$$

In case of a single LIS, $\gamma_1 = \gamma_3$ and $\delta_1 = \delta_3$, so all constraints are trivially satisfied for all $0 \leq i \leq n-1$. □

Second, recall that in §4.2 we proved that BGSA, STI, TI, GSA are equivalent for single interpolation systems, and that SA ⇒ BGSA for symmetric ones. We now show that for a single LIS, SA is equivalent to BGSA and that PI is not.

Theorem 4.3.9. *If a LIS has SA, then it has BGSA.*

Proof. We show that, for any L, the labeling constraints of SA imply those of BGSA. Refer to Table 4.2, Table 4.1, Theorem 4.3.2 and Corollary 4.3.8. In case of a family $\{L_1, L_2, L_3\}$, the constraints for 3-SA are:

$$(\alpha_1, \alpha_2), (\beta_2, \beta_3), (\gamma_1, \gamma_3) \preceq \{(ab, ab), (b, a), (a, b)\}$$
$$(\delta_1, \delta_2, \delta_3) \preceq \{(ab, ab, ab), (a, b, b), (b, a, b), (b, b, a)\}$$

When $L_1 = L_2 = L_3$, they simplify to $\alpha, \beta, \gamma, \delta \in \{ab, b\}$; this means that, in case of a single LIS, only Pudlák's or stronger systems are allowed. For a family $\{L_1, L_2, L_3\}$, the BGSA constraints are:

$$(\alpha_1, \alpha_2), (\delta_1, \delta_2) \preceq \{(ab, ab), (b, a), (a, b)\}$$
$$\beta_2 \preceq \beta_3 \qquad \gamma_1 \preceq \gamma_3$$
$$\delta_1 \preceq \delta_3 \qquad \delta_2 \preceq \delta_3$$

When $L_1 = L_2 = L_3$, they simplify to $\alpha, \delta \in \{ab, b\}$, so that the constraints for 3-SA imply those for BGSA. □

From the proof of Theorem 4.3.9, we obtain:

Theorem 4.3.10. *A LIS Itp_L has the BGSA property iff it is stronger or equivalent to Pudlák's system Itp_P. In particular, McMillan's system Itp_M has BGSA.*

Finally, Theorem 4.3.8, Theorem 4.3.10, and the fact that $Itp_{M'}$ is strictly weaker than Itp_P yield:

Theorem 4.3.11. *The system $Itp_{M'}$ has PI but does not have BGSA.*

Note that the necessary and sufficient conditions for LISs to support each of the collectives simplify implementing procedures with a given property, or, more importantly from a practical perspective, determine which implementation supports which property.

We summarize the results about labeled interpolation systems in Figure 4.5 and in Figure 4.6; in comparison with the hierarchy of general interpolation systems outlined in Figure 4.1 and Figure 4.4, TI can be reduced to GSA in case of LISs families without additional assumptions, while GSA and SA are equivalent for single LISs.

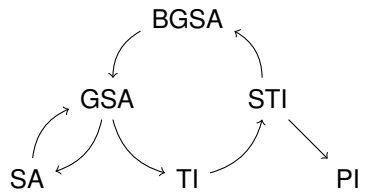


Figure 4.5. Collectives of single LISs.

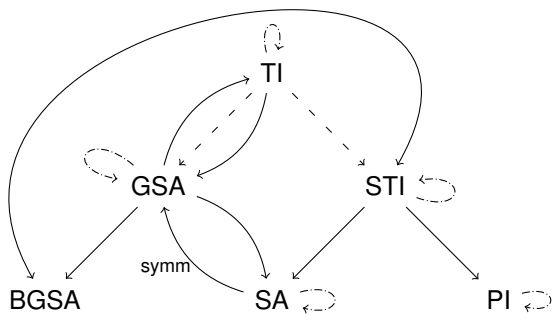


Figure 4.6. Collectives of families of LISs.

4.4 Collectives of Theory Labeled Interpolation Systems

After investigating propositional LISs, we move on to first order theories. The results of §4.2.1 and §4.2.2 about generic interpolation systems are directly applicable to the theory labeled interpolation systems; the goal of this section is to analyze the concrete constraints on $\mathscr{T}$-labelings that allow collectives to be satisfied by $\mathscr{T}$-LISs.

In §3.4 we discussed the two invariant properties that characterize LISs; the soundness of a LIS Itp_L is proved by showing that in a refutation of $A \wedge B$, for each clause C and partial interpolant $I(C)$:

$$\begin{aligned}
&A \wedge \neg(C|_a \vee C|_{ab}) \models I_L(C)\\
&B \wedge \neg(C|_b \vee C|_{ab}) \wedge I_L(C) \models \bot\\
&\mathscr{L}_{I_L(C)} \subseteq \mathscr{L}_{AB}
\end{aligned}$$

while the relationship between labelings and interpolant strength is established by the fact that, given two labelings L, L' s.t. $L \preceq L'$, for each clause C:

$$I_L(C) \models I_{L'}(C) \vee C|_{AB}$$

The above conditions are at the base of the results about collectives and LISs presented so far.

In §3.4.1 we introduced the notions of $\mathscr{T}$-labeling and $\mathscr{T}$-LIS, for a first order theory $\mathscr{T}$, and generalized the conditions to:

$$\begin{aligned}
&A \wedge \neg(C|_a \vee C|_{ab}) \models_{\mathscr{T}} I_L(C)\\
&B \wedge \neg(C|_b \vee C|_{ab}) \wedge I_L(C) \models_{\mathscr{T}} \bot\\
&\mathscr{L}_{I_L(C)} \subseteq \mathscr{L}_{AB} \cup \mathscr{L}_{\mathscr{T}}
\end{aligned}$$

and:

$$I_L(C) \models_{\mathscr{T}} I_{L'}(C) \vee C|_{AB}$$

Clearly, the invariant properties of a $\mathscr{T}$-LIS correspond to those of a LIS, modulo the underlying $\mathscr{T}$. The main difference we need to consider is the presence of theory lemmata. In fact, if a proof does not contain theory lemmata, then a $\mathscr{T}$-labeling is equivalent to a labeling; moreover, a $\mathscr{T}$-LIS behaves as a LIS in generating interpolants (compare Table 3.9 and Table 3.5), and consequently all the proofs related to LISs can be extended in a straightforward manner to $\mathscr{T}$-LISs.

4.4.1 Collectives of Families and of Single $\mathscr{T}$-LISs

In the following, we systematically address some of the main results of §4.3.1 and §4.3.2, adapting them to take into account the presence of theory lemmata: in particular, on one side we focus on BGSA and n-GSA, for their importance in the hierarchy of collectives; on the other side, we examine n-PI and instantiate it in the context of difference logic, further developing the theme of §3.4.2.

4.4.1.1 GSA

Let us recall here the CC_{BGSA} constraints from §4.3.1.1:

$$\begin{array}{ll} (\alpha_1,\alpha_2),(\delta_1,\delta_2) \preceq \{(ab,ab),(b,a),(a,b)\} & \\ \beta_2 \preceq \beta_3 & \gamma_1 \preceq \gamma_3 \\ \delta_1 \preceq \delta_3 & \delta_2 \preceq \delta_3 \end{array}$$

and the correspondent restricted CC^*_{BGSA}:

$$\begin{array}{ll} (\alpha_1,\alpha_2),(\delta_1,\delta_2) \in \{(ab,ab),(b,a),(a,b)\} & \\ \beta_2 = \beta_3 & \gamma_1 = \gamma_3 \\ \delta_3 = \max\{\delta_1,\delta_2\} & \end{array}$$

Lemma 4.3.1 states that if a family of labelings $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA}, then the family of LISs $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ has BGSA. We extend the lemma to $\mathscr{T}$-LISs:

Lemma 4.4.1. *Assume a family of $\mathscr{T}$-labelings $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA}. Assume also that there exists a procedure TPI_L s.t. for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$, $\{L_1, L_2, L_3\}$ satisfying CC^*_{BGSA}, refutation Π of Φ and theory lemma C in Π:*

$$TPI_{\phi_1,L_1}(C) \wedge TPI_{\phi_2,L_2}(C) \models_{\mathscr{T}} TPI_{\phi_1\phi_2,L_3}(C)$$

Then, $\{\mathscr{T}\text{-}Itp_{L_1}, \mathscr{T}\text{-}Itp_{L_2}, \mathscr{T}\text{-}Itp_{L_3}\}$ has BGSA.

Proof by structural induction. The proof relies on showing that, for any clause C in any refutation of any Φ, the partial interpolants satisfy:

$$I_{\phi_1,L_1}(C) \wedge I_{\phi_2,L_2}(C) \models_{\mathscr{T}} I_{\phi_1\phi_2,L_3}(C)$$

.

Base case (leaf). The cases $C \in \phi_1, C \in \phi_2, C \in \phi_3$ are dealt exactly as in Lemma 4.3.1; the remaining case, i.e. C is a theory lemma, follows from the hypothesis on the existence of TPI_L.

Inductive step (inner node). There are no theory lemmata among the inner nodes, so the inductive step is as in Lemma 4.3.1. □

The relaxation of CC^*_{BGSA} to CC_{BGSA} derives from the relationship between partial order on $\mathcal{T}$-labelings and interpolant strength:

Theorem 4.4.1. *Assume a family of $\mathcal{T}$-labelings $\{L_1, L_2, L_3\}$ satisfies CC^*_{BGSA}. Assume also that there exists a procedure TPI_L s.t. for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$, $\{L_1, L_2, L_3\}$ satisfying CC^*_{BGSA}, refutation Π of Φ and theory lemma C in Π:*

$$TPI_{\phi_1,L_1}(C) \wedge TPI_{\phi_2,L_2}(C) \models_{\mathcal{T}} TPI_{\phi_1\phi_2,L_3}(C)$$

Then, $\{\mathcal{T}\text{-}Itp_{L_1}, \mathcal{T}\text{-}Itp_{L_2}, \mathcal{T}\text{-}Itp_{L_3}\}$ has BGSA.

We stress that the assumptions on TPI_L are important since the procedure must be able to compute a theory partial interpolant for any theory lemma C in $\mathcal{T}$ and for any way of labeling the clean atoms of C as a or b, independently from the refutation where the lemma appears.

These assumptions are sufficient but not necessary. Consider in fact the contrapositive of the converse of Theorem 4.4.1: if (i) $\{L_1, L_2, L_3\}$ does not satisfy CC_{BGSA} or (ii) such a TPI_L does not exist, then (iii) $\{Itp_{L_1}, Itp_{L_2}, Itp_{L_3}\}$ does not have BGSA. Now, on one hand, (i)⇒(iii) follows from Lemma 4.3.4; on the other hand, the existence of some theory lemma C in some refutation Π with:

$$TPI_{\phi_1,L_1}(C) \wedge TPI_{\phi_2,L_2}(C) \not\models_{\mathcal{T}} TPI_{\phi_1\phi_2,L_3}(C)$$

does not necessarily entail that, at the level of the root of Π:

$$I_{\phi_1,L_1} \wedge I_{\phi_2,L_2} \not\models_{\mathcal{T}} I_{\phi_1\phi_2,L_3}$$

In the same manner as Lemma 4.3.1 for BGSA is generalized to Lemma 4.3.5 for n-GSA, we derive:

Theorem 4.4.2. *Assume a family of $\mathcal{T}$-labelings $\{L_1, \ldots, L_{n+1}\}$ satisfies CC_{nGSA}. Assume also that there exists a procedure TPI_L s.t. for any $\Phi = \{\phi_1, \ldots, \phi_{n+1}\}$, $\{L_1, \ldots, L_{n+1}\}$ satisfying CC_{nGSA}, refutation Π of Φ and theory lemma C in Π:*

$$TPI_{\phi_1,L_1}(C) \wedge \cdots \wedge TPI_{\phi_n,L_n}(C) \models_{\mathcal{T}} TPI_{\phi_1\ldots\phi_n,L_{n+1}}(C)$$

Then, $\{\mathcal{T}\text{-}Itp_{L_1}, \ldots, \mathcal{T}\text{-}Itp_{L_{n+1}}\}$ has n-GSA.

4.4.1.2 PI

In §4.3.1.6 we deduced the CC_{BST} constraints:

$$\gamma_1 \preceq \gamma_3 \qquad \delta_1 \preceq \delta_3$$

as a subset of the BGSA constraints and proved that a family of interpolation systems $\{Itp_{L_0}, \ldots, Itp_{L_n}\}$ has n-PI iff, for all $0 \leq i \leq n-1$, $\{L_i, L_{i+1}\}$ satisfies CC_{BST}.

We employ Theorem 4.4.1 to derive:

Theorem 4.4.3. *Assume a family of $\mathcal{T}$-labelings $\{L_0, \ldots, L_n\}$ s.t., for all $0 \leq i \leq n-1$, $\{L_i, L_{i+1}\}$ satisfies CC_{BST}. Assume also there exists a procedure TPI_L s.t. for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$, $\{L_1, L_3\}$ satisfying CC_{BST}, refutation Π of Φ and theory lemma C in Π:*

$$TPI_{\phi_1, L_1}(C) \wedge \phi_2 \models_{\mathcal{T}} TPI_{\phi_1\phi_2, L_3}(C)$$

Then, $\{\mathcal{T}\text{-}Itp_{L_0}, \ldots, \mathcal{T}\text{-}Itp_{L_n}\}$ has n-PI.

Theorem 4.3.8 from §4.3.2 shows that, in case of any single LIS Itp_L, the CC_{BST} constraints are trivially satisfied since $\gamma_1 = \gamma_3$ and $\delta_1 = \delta_3$. When dealing with $\mathcal{T}$-LISs, we thus need only to ensure the existence of an appropriate procedure TPI_L:

Theorem 4.4.4. *Assume there exists a procedure TPI_L s.t. for any $\Phi = \{\phi_1, \phi_2, \phi_3\}$, $\mathcal{T}$-labeling L, refutation Π of Φ and theory lemma C in Π:*

$$TPI_{\phi_1, L}(C) \wedge \phi_2 \models_{\mathcal{T}} TPI_{\phi_1\phi_2, L}(C)$$

Then, any $\mathcal{T}\text{-}Itp_L$ has n-PI.

In the case of difference logic, a suitable TPI_L indeed exists, and is the one introduced in Theorem 3.4.2, based on computing a theory interpolant $TPI_L(C)$ for a DL lemma C as a maximum summary of the atoms labeled a in the cycle of negative weight built over $\overline{C}$.

Consider in fact the labeling table of BST, where A changes from ϕ_1 to $\phi_1 \wedge \phi_2$ when moving from the first to the second configuration. As a result, edges corresponding to original atoms appearing in ϕ_2 might pass from B or AB to A; similarly, new theory atoms containing symbols local to ϕ_2 might change from B-dirty to A-dirty. Moreover, $\gamma_1 = \gamma_3$, $\delta_1 = \delta_3$, and $\alpha_1, \beta_3 \in \{a, b\}$ because we are labeling atoms in a theory lemma. This implies that some of the edges of the cycle, that are labeled b in the first configuration, are instead labeled a in the second one.

Following the same reasoning as in Theorem 3.4.2, the computation of maximum summaries yields $TPI_{\phi_1\phi_2, L}(C) = TPI_{\phi_1, L}(C) \wedge \phi_2'$, where ϕ_2' is a subset of

Table . BST.

p in ?	Atom $class, label$	
	$\phi_1 \vert \phi_2 \phi_3$	$\phi_1 \phi_2 \vert \phi_3$
ϕ_1	A, a	A, a
ϕ_2	B, b	A, a
ϕ_3	B, b	B, b
$\phi_1 \phi_2$	AB, α_1	A, a
$\phi_2 \phi_3$	B, b	AB, β_3
$\phi_1 \phi_3$	AB, γ_1	AB, γ_3
$\phi_1 \phi_2 \phi_3$	AB, δ_1	AB, δ_3

the conjuncts of ϕ_2. Now, $\phi_2 \models \phi_2'$, which in turn allows to derive:

$$TPI_{\phi_1,L}(C) \wedge \phi_2 \Rightarrow$$
$$TPI_{\phi_1,L}(C) \wedge \phi_2' =$$
$$TPI_{\phi_1\phi_2,L}(C)$$

We can finally state:

Theorem 4.4.5. *PI holds for all single DL-LISs, if theory interpolants of DL-lemmata are computed by means of maximum summarization of the atoms labeled a.*

4.5 Summary and Future Developments

In this chapter we carried out a systematic investigation of the most common interpolation properties exploited in verification, focusing on the constraints they impose on interpolation systems used in SAT and in SMT-based model checking.

We systematized and unified the collectives and proved that, for families of interpolation systems, the properties form a hierarchy, whereas for a single system all properties except path interpolation and simultaneous abstraction are in fact equivalent.

In the context of propositional logic, we defined and proved both sufficient and necessary conditions for families of labeled interpolation systems. In particular, we demonstrated that in case of a single system path interpolation is common to all LISs, while simultaneous abstraction is as strong as all other more complex properties. Moreover, every LIS stronger or equivalent to Pudlák's Itp_P (including thus Itp_M) enjoys all properties, while only path interpolation is satisfied by $Itp_{M'}$; this fact has been used in §3.6.3 with reference to tree interpolation.

We extended some of these results to first order theories in the framework of theory labeled interpolation systems, where ad-hoc procedures are needed to compute partial interpolants for theory lemmata, and discussed the case of difference logic.

Future work can concern, on one hand, a theoretical analysis of the requirements these procedures must meet in order for $\mathcal{T}$-LISs to satisfy the various collectives, and of whether and how the necessary and sufficient conditions identified for LISs are affected. On the other hand, such analysis can be complemented by the development of concrete procedures tailored both for individual and combination of theories, as delineated in §3.4.3.

On a practical side, experimentation can be performed with families of systems, exploiting the flexibility given by computing different interpolants in a collection by means of different systems. A first step could be to extend the SAT-based function summarization framework of §3.6.3, allowing to tune the coarseness degree of individual summaries by varying the strength of interpolants, both in the verification of the same program incrementally with respect to various properties, and of various versions of a program with respect to the same properties.

After that, our results could be applied to SMT-based model checking, for example to LAWI-based verification algorithms relying on path interpolation. An interesting aspect to investigate is that LAWI benefits from having weaker interpolants at the beginning of a path and stronger at the end; our work shows instead that PI is not guaranteed to hold if, in the sequence of LISs generating the interpolants, a weaker LIS precedes a stronger one. A possible solution would be to start with the weakest $\mathcal{T}$-$Itp_{M'}$, strengthening step by step as long as PI is satisfied until reaching a certain $\mathcal{T}$-LIS $\mathcal{T}$-Itp_L, and then to keep $\mathcal{T}$-Itp_L until the end of the path.

Chapter 5

Proof Manipulation

Resolution proofs find application in many verification techniques: first and foremost, all model checking approaches presented in §3.1 rely on interpolants, that can be generated from refutations by means of the interpolation systems illustrated in §3.3.2.

Proofs can also be used as justifications of specifications of inconsistency in various industrial applications (e.g., product configuration or declarative modeling [SKK03, SSJ+03]). In the context of proof-carrying code [Nec97] a system can verify a property about an application exploiting a proof provided with the application executable code. SAT solvers and SMT solvers can be integrated into interactive theorem provers as automated engines to produce proofs, that can be later replayed and verified within the provers [Amj08, WA09, FMM+06]. An unsatisfiable core, that is an inconsistent subset of clauses, can be extracted from a proof, to be exploited for example during the refinement phase in model checking [AM03, GLST05].

It is essential to develop techniques that allow to handle proofs efficiently, so as to positively impact on the performance of the frameworks that rely on them. In this chapter we focus on two important aspects: compression of resolution proofs, and rewriting to facilitate the computation of interpolants. These approaches, which are motivated and discussed in detail in §5.3 and §5.4, are both realized by means of a set of local rewriting rules that enable restructuring and compression.

The chapter begins in §5.1 by recalling the notion of resolution proof, and describing the ways in which proofs are concretely represented. In §5.2 we introduce a new proof transformation framework consisting of a set of local rewriting rules and we prove its soundness.

In §5.3 we address the problem of compression, and present a collection of algorithms based on the transformation framework. We compare them against existing

compression techniques and provide experimental results that show their effectiveness over SMT and SAT benchmarks.

In §5.4 we discuss the inability of state-of-the-art interpolation systems for first order theories to deal with AB-mixed predicates, addressing a limitation already mentioned in §3.3.5 and §3.4.1. We then illustrate an application of our transformation framework aimed at reordering resolution refutations, in such a way that interpolation is made possible. The approach is demonstrated to be theoretically sound and experiments are provided to show that it is also practically efficient.

As a theoretical investigation, we also examine the interaction between LISs and proof manipulation by discussing algorithms that guarantee the generation of interpolants in conjunctive or disjunctive normal form by reordering resolution steps in a propositional refutation.

In §5.5 we describe some of the heuristics adopted in the application of rules by the transformation framework, with reference to §5.3 and §5.4, while in §5.6 we review the existing literature on proof manipulation.

5.1 Resolution Proofs

So far it has been sufficient to consider resolution proofs as trees: leaves are original clauses or theory lemmata, while inner nodes are resolvents of resolution steps whose antecedents are the two parents in the tree.

In this chapter, which focuses on techniques to manipulate proofs, it is necessary to pay more attention to the ways proofs are actually represented; for this reason we distinguish between a *resolution proof tree* and a *resolution proof DAG*.

Definition 5.1.1 (Resolution Proof Tree)**.** A *resolution proof tree* of a clause C from a set of clauses $\mathbb{C}$ is a tree such that:

1. Each node n is labeled by a clause $C(n)$.
2. If n is a leaf, $C(n) \in \mathbb{C}$
3. The root is a node n s.t. $C(n) = C$.
4. An inner node n has pivot $piv(n)$ and exactly two parents n^+, n^- s.t. $C(n) = Res_{piv(n)}(C(n^+), C(n^-)))$, that is, $C(n)$ is the resolvent of a *resolution step* with pivot p and antecedents $C(n^+), C(n^-)$. $C(n^+)$ and $C(n^-)$ respectively contain the positive and negative occurrence of the pivot.
5. Each non-root node has exactly one child.

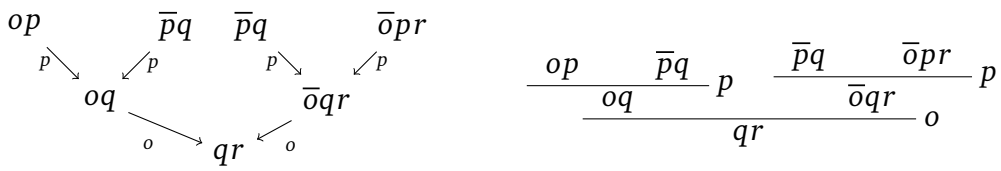


Figure 5.1. Resolution proof tree.

In the following, we equivalently use a graph-based representation (left) or an inference rule-based representation (right); for brevity, clauses can be shown as sequences of literals and subclauses omitting the "∨" symbol, as $p\overline{q}D$.

In real-world applications proofs are rarely generated or stored as trees; for instance proofs produced by CDCL solvers are represented as DAGs (directed acyclic graph). We therefore introduce the following notion of resolution proof, which is more suitable for describing the graph-based transformation algorithms illustrated in this chapter.

Definition 5.1.2 (Resolution Proof DAG)**.** A *resolution proof DAG* of a clause C from a set of clauses $\mathbb{C}$ is a directed acyclic graph such that:

1. - 4. hold as in Definition 5.1.1.

5. Each non-root node has one or more children.

Resolution proof DAGs extend the notion of resolution proof trees by allowing a node to participate as antecedent in multiple resolution steps.

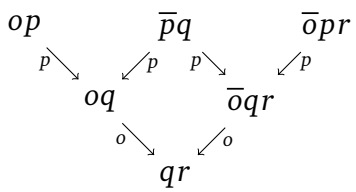


Figure 5.2. Resolution proof DAG.

We identify a node n with its clause $C(n)$ whenever convenient; in general, different nodes can be labeled by the same clause, that is $C(n_i) = C(n_j)$ for $i \neq j$. A proof P is a *refutation* if $C = \bot$. A *subproof* P', with *subroot* n, of a proof P is the subtree that derives $C(n)$ from a subset of clauses that label leaves of P; when referring to P and its root compared to P', we call P *global proof* and its root *global root*.

It is always possible to turn a resolution proof tree into a resolution proof DAG, by merging two or more nodes labeled by a same clause into a single node, which inherits the children of the merged nodes. On the other hand, a resolution proof DAG can be "unrolled" into a resolution proof tree, possibly at exponential cost: it is in fact sufficient to traverse the DAG bottom-up, duplicating nodes with multiple children so that each node is left with at most one child.

Similarly to [BIFH+08], we distinguish between a *legal* and an *illegal* proof; an illegal proof is a proof which has undergone transformations in such a way that some clauses might not be the resolvents of their antecedents anymore. In this chapter however an illegal proof represents an intermediate transformation step in an algorithm, and the proof can always be *reconstructed* into a legal one, as explained in the next sections.

In the following, we will consider refutations as obtained by means of modern CDCL SAT solvers and lazy SMT solvers, involving both propositional and theory atoms. Whenever the theory content is not relevant to the problem at hand, it is convenient to represent each theory atom with a new propositional variable called its *propositional abstraction*: for example an atom $x + y < 1$ will be represented by a certain variable q.

We use $C_1 \subseteq C_2$ to indicate that C_1 *subsumes* C_2, that is the set of literals C_1 is a subset of the set of literals C_2; $v(s)$ denotes the variable associated with a literal s.

5.2 The Local Transformation Framework

This section introduces a proof transformation framework based on local rewriting rules. We start by assuming a resolution proof tree, and then extend the discussion to resolution proof DAGs. All results related to proofs hold in particular for refutations.

The framework is built on a set of rewriting rules that transform a subproof with root C into one whose subroot C' is logically equivalent or stronger than C (that is, $C' \models C$). Each rewriting rule is defined to match a particular *context*, identified by two consecutive resolution steps (see Figure 5.3). A context involves

Figure 5.3. Rule context.

two pivots p and q and five clauses C_1, C_2, C_4, C_3, C; we call C the *context root*; the subproof rooted in C is the *context subproof*. Clearly p is contained in C_1 and C_2 (with opposite polarity), and q is contained in C_4 and C_3 (again with opposite polarity); q must be contained in $C_1 \cup C_2$.

A clause C might be the root of two different contexts, depending on whether C_1 and C_2 are taken as the antecedents of either of the two antecedents of C; in that case, to distinguish among them we talk about *left* and *right context*.

Figure 5.4 shows a set of proof transformation rules. Each rule is associated with a unique context, and, conversely, each context can be mapped to at least one rule (i.e., the set of rules is exhaustive, modulo symmetry, for every possible context). A first property that characterizes the set of rules is *locality*: only the limited information represented by a context is in fact needed to determine which rule is applicable. A second property is *strengthening*: the rules either keep the context root unchanged or turn it into a logically stronger formula.

The classification of rules into S (swapping) and R (reducing) depends on the effect of the rules on the context rooted in C: $S1$ and $S2$ swap the two resolution steps in the context without modifying C, while $R1$, $R2$, $R2'$ and $R3$ replace C with a new C' such that $C' \subseteq C$; in other words, the R rules generate subproofs with stronger roots.

The influence of the S rules does not extend beyond the context where they are applied, while that of the R rules possibly propagates down to the global root. The R rules essentially simplify the proof and their effect cannot be undone, while an application of an S rule can be reversed. In particular, the effect of rule $S2$ can be canceled out simply by means of another application of the same $S2$. $S1$ has $S1'$ as its inverse (note the direction of the arrow); $S1'$ is actually a derived rule, since it corresponds to the sequential application of $S2$ and $R2$.

The rules $R2$ and $R2'$ are associated with the same context; they respectively behave as $S2$ (with an additional simplification of the root) and $R1$. The decision whether to apply either rule depends on the overall goal of the transformation. Note that the application of rule $R2$ to a context turns it into a new context which matches rule $S1$.

5.2.1 Extension to Resolution Proof DAGs

If the proof to be transformed is a DAG rather than a tree, some constraints are necessary on the application of the rules.

Consider rules $S1, S1', S2, R2$, and suppose clause C_4 is involved in more than one resolution step, having thus at least another resolvent C_5 besides C. If C_4 is modified by a rule, it is not guaranteed that the correctness of the resolution step

$S1$: $s \notin C_3$, $t \in C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}tE \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] \\ & & C : DEF & \end{tikzcd} \Rrightarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] & & C_3 : \bar{t}F \arrow[dr, "v(t)"'] & & C_2 : \bar{s}tE \arrow[dl, "v(t)"] \\ & C_4' : sDF \arrow[drr, "v(s)"'] & & & & C_4'' : \bar{s}EF \arrow[dll, "v(s)"] & \\ & & & C : DEF & & & \end{tikzcd}$$

$S1'$: $s \notin C_3$, $t \in C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}tE \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] \\ & & C : DEF & \end{tikzcd} \Lleftarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] & & C_3 : \bar{t}F \arrow[dr, "v(t)"'] & & C_2 : \bar{s}tE \arrow[dl, "v(t)"] \\ & C_4' : sDF \arrow[drr, "v(s)"'] & & & & C_4'' : \bar{s}EF \arrow[dll, "v(s)"] & \\ & & & C : DEF & & & \end{tikzcd}$$

$S2$: $s \notin C_3$, $t \notin C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] \\ & & C : DEF & \end{tikzcd} \Rrightarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : \bar{t}F \arrow[dl, "v(t)"] & \\ & C_4' : sDF \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] \\ & & C' : DEF & \end{tikzcd}$$

$R1$: $s \in C_3$, $t \in C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}tE \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] \\ & & C : sDEF & \end{tikzcd} \Rrightarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] \\ & C' : sDF & \end{tikzcd}$$

$R2$: $s \in C_3$, $t \notin C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] \\ & & C : sDEF & \end{tikzcd} \Rrightarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] & \\ & C_4' : sDF \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] \\ & & C' : DEF & \end{tikzcd}$$

$R2'$: $s \in C_3$, $t \notin C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] \\ & & C : sDEF & \end{tikzcd} \Rrightarrow \begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(t)"'] & & C_3 : s\bar{t}F \arrow[dl, "v(t)"] \\ & C' : sDF & \end{tikzcd}$$

$R3$: $\bar{s} \in C_3$, $t \notin C_2$

$$\begin{tikzcd}[column sep=small] C_1 : stD \arrow[dr, "v(s)"'] & & C_2 : \bar{s}E \arrow[dl, "v(s)"] & \\ & C_4 : tDE \arrow[dr, "v(t)"'] & & C_3 : \bar{s}\bar{t}F \arrow[dl, "v(t)"] \\ & & C : \bar{s}DEF & \end{tikzcd} \Rrightarrow C' = C_2 : \bar{s}E$$

Figure 5.4. Local transformation rules for resolution proof trees.

having C_5 as resolvent (and in turn of the resolution steps on the path from C_5 to the global root) is preserved. This problem does not concern clauses C_1, C_2, C_3 and the subproofs rooted in them, which are not changed by any rule.

A simple solution consists in creating a copy of C_4, to which all resolvents of C_4 besides C are assigned, so that C_4 is left with exactly one resolvent; at that point any modification to C_4 will affect only the context rooted in C. Since duplications increase the size of the proof, they should be carried out sparingly (see §5.5).

A more efficient alternative exists in case of rules $R1$, $R2'$, $R3$, where C_4 is detached by the context rooted in C and loses C as resolvent, but keeps the other resolvents (if any). The effect of the transformation rules is shown in Figure 5.5: the presence of additional resolvents for C_4 is denoted by a dotted arrow.

5.2.2 Soundness of the Local Transformation Framework

In this section we first prove that the rewriting rules preserve the legality of the subproofs rooted in the contexts where the rules are applied; then we discuss how the rules affect the global proof and what steps must be taken to maintain it legal.

Effect on a Context. Based on the following observations, we claim that after a single application of a rule to a context with root C, the legal subproof rooted in C is replaced by a legal subproof rooted in $C' \subseteq C$.

Refer to Figure 5.5. No additional subproofs are introduced by the rules and no modifications are brought to the subproofs rooted in C_1, C_2, C_3, which are simply recombined or detached from the context. As for the S rules, C_4 is either replaced by the resolvent of C_1, C_3 ($S2$) or by the resolvent of the resolvents of C_1, C_3 and C_3, C_2 ($S1$, where a new clause $C_4'' = Res_{v(s)}(C_2, C_3)$ is also introduced). Note that in both cases C is not modified. The R rules instead yield a more substantial change in the form of a stronger context root $C' \subseteq C$:

- In $R1$ and $R2'$, the subproofs with root C_1 and C_3 are combined to obtain a subproof with root $sDF \subseteq sDEF$.
- $R2$ has a swap effect similar to $S2$, but replaces the root $sDEF$ with DEF, removing a single literal.
- In $R3$, the whole subproof is substituted by the subproof rooted in $C_2 = \bar{s}E$, which subsumes $C = \bar{s}DEF$.

All the above transformations involve updating the relevant clauses by means of sound applications of the resolution rule.

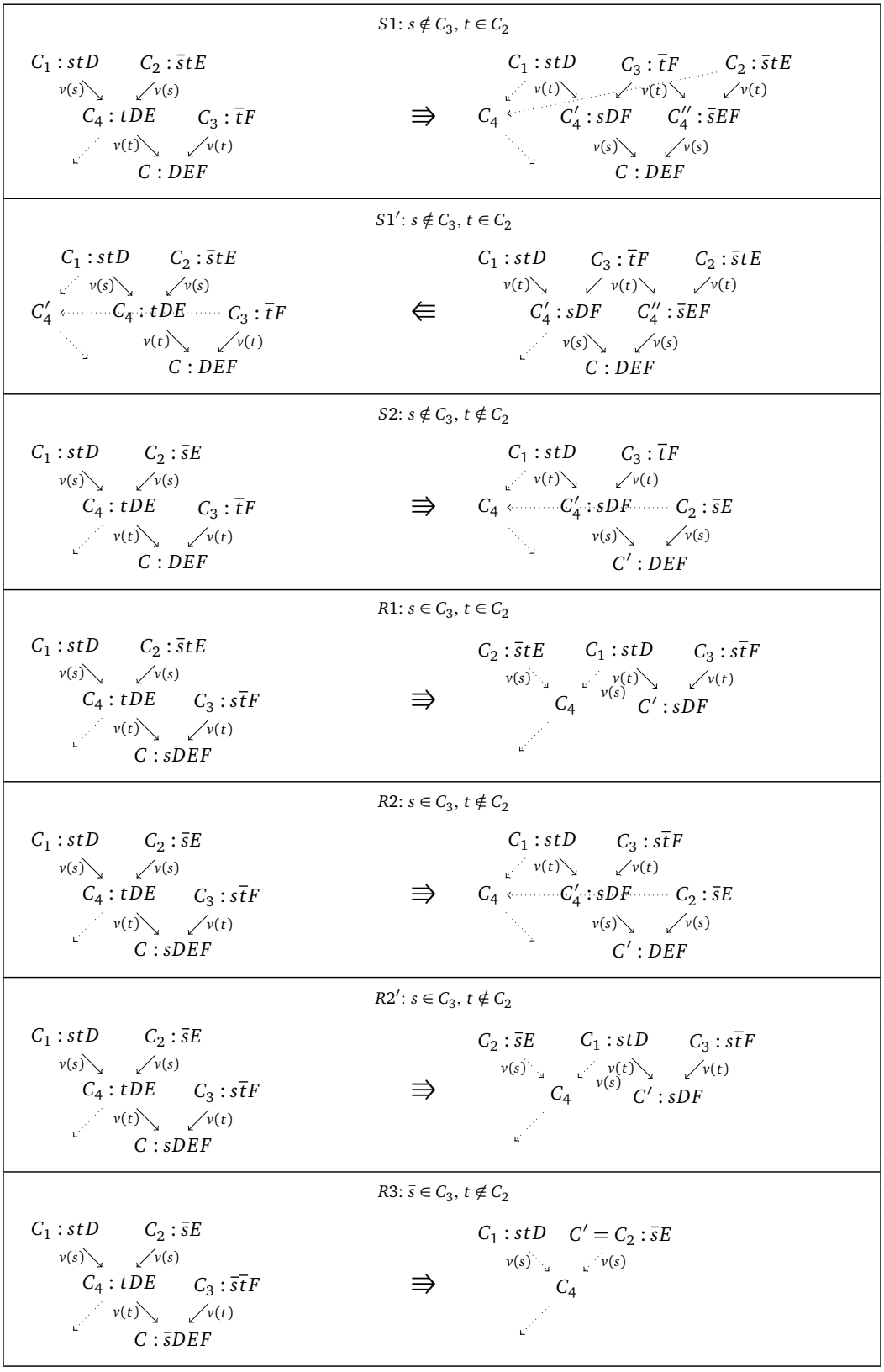


Figure 5.5. Local transformation rules for resolution proof DAGs.

Effect on the Global Proof. The application of a rule to a context yields a legal subproof rooted in a clause $C' \subseteq C$; however, the global proof could turn into an illegal one. In fact, the deletion of literals from C affects the sequence of resolution steps that extends from C to the global root: some of these steps might become superfluous, because they resolve upon a variable which was introduced by C (but does not appear in C'), and they should be appropriately removed. In the same way, the elimination of a resolution step could itself lead to the disappearance of more literal occurrences, leading to a chain reaction.

The following Algorithm 6, *SubsumptionPropagation*, has the purpose of propagating the effect of the replacement of C by $C' \subseteq C$ along the path leading from C to the global root.

The algorithm restructures the proof in a top-down manner analyzing the sequence of resolution steps to ensure their correctness while propagating the effect of the initial subsumption. We prove that, after an execution of SubsumptionPropagation following the application of an R rule to a legal proof, the result is still a legal proof.

The idea at the base of the algorithm reflects the mechanisms of the restructuring procedures first proposed in [BIFH$^+$08, DKPW10]:

1. It determines the effect range of the substitution of C by C', which corresponds to the set of nodes reachable from the node labeled by C'.

2. It analyzes, one by one, all reachable nodes; it is necessary that the antecedents of a node n have already been visited (and possibly modified), in order to guarantee a correct propagation of the modifications to n.

3. Due to the potential vanishing of literals from clauses, it might happen that in some resolution step the pivot is not present in both antecedents anymore; if that is the case, the resolution step is deleted, by replacing the resolvent with the antecedent devoid of the pivot (if the pivot is missing in both antecedents, either of them is arbitrarily chosen), otherwise, the resolution step is kept and the resolvent clause updated. At the graph level, n is substituted by n^+ or n^-, assigning the children of n (if any) to it.

Theorem 5.2.1. *Assume a legal proof P. The application of an R rule, followed by an execution of SubsumptionPropagation, yields a legal proof P', whose new global root subsumes the previous one.*

Proof by structural induction. **Base case**. Assume an R rule is applied to a context rooted in a clause C; C is replaced by $C' \subseteq C$ and the subproof rooted in C' is legal,

```
Input: A legal proof modified by an R rule
Output: A legal proof
Data: W: set of nodes reachable from C', U: set of visited nodes
begin
    U ← ∅
    Determine W, e.g. through a visit from C'
    while W \ U ≠ ∅ do
        Choose n ∈ W \ U such that:
        (n⁺ ∈ W or n⁺ ∉ W) and (n⁻ ∈ W or n⁻ ∉ W)
        U ← U ∪ {n}
        p ← piv(n)
        if p ∈ C(n⁺) and p̄ ∈ C(n⁻) then
            C(n) ← Res_p(C(n⁺), C(n⁻))
        else if p ∉ C(n⁺) and p̄ ∈ C(n⁻) then
            Substitute n with n⁺
        else if p ∈ C(n⁺) and p̄ ∉ C(n⁻) then
            Substitute n with n⁻
        else if p ∉ C(n⁺) and p̄ ∉ C(n⁻) then
            Heuristically choose a parent, replace n with it
    end
end
```

Algorithm 6: SubsumptionPropagation.

as previously shown. The subproofs rooted in the clauses of nodes not reachable from C are not affected and thus remain legal.

Inductive step. All nodes reachable from C are visited; in particular, a node n is visited after its reachable parents. By inductive hypothesis $C'(n^+) \subseteq C(n^+)$, $C'(n^-) \subseteq C(n^-)$ and the subproofs rooted in $C'(n^+)$ and $C'(n^-)$ are legal. We show that, after visiting n, $C'(n) \subseteq C(n)$ and the subproof rooted in $C'(n)$ is legal. Let $p = piv(n)$. We have three possibilities:

- Case 1: the pivot still appears both in $C'(n^+)$ and in $C'(n^-)$; $C'(n) = Res_p(C'(n^+), C'(n^-))$, thus $C'(n) \subseteq C(n)$.

- Case 2: the pivot is present only in one antecedent, let us say $C'(n^+)$; the subproof rooted in $C(n)$ is replaced by the one rooted in $C'(n^-)$ (legal by hypothesis). But $C'(n) = C'(n^-) \subseteq C(n)$ since $C'(n^-)$ does not contain the pivot.

Step 1: Application of rule $R2$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{\overline{p}r}}{\mathbf{qr}}\,\mathbf{p} \quad \mathbf{p\overline{q}}}{\mathbf{pr}}\,\mathbf{q} \quad \overline{p}u}{ru}\,p \quad \overline{r}v}{uv}\,r \quad \Rrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{p\overline{q}}}{\mathbf{p}}\,\mathbf{q} \quad \mathbf{\overline{p}r}}{\mathbf{r}}\,\mathbf{p} \quad \overline{p}u}{ru}\,p \quad \overline{r}v}{uv}\,r$$

Step 2: Elimination of an unnecessary resolution step (Case 2)

$$\dfrac{\dfrac{\dfrac{\dfrac{pq \quad p\overline{q}}{p}\,q \quad \overline{p}r}{\mathbf{r}}\,p \quad \mathbf{\overline{p}u}}{\mathbf{ru}}\,\mathbf{p} \quad \overline{r}v}{uv}\,r \quad \Rrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{pq \quad p\overline{q}}{p}\,q \quad \overline{p}r}{\mathbf{r}}\,p \quad \overline{r}v}{uv}\,r$$

Step 3: Update of a resolving clause (Case 1)

$$\dfrac{\dfrac{\dfrac{pt \quad p\overline{t}}{p}\,t \quad \overline{p}r}{\mathbf{r}}\,p \quad \mathbf{\overline{r}v}}{\mathbf{uv}}\,\mathbf{r} \quad \Rrightarrow \quad \dfrac{\dfrac{\dfrac{pt \quad p\overline{t}}{p}\,t \quad \overline{p}r}{\mathbf{r}}\,p \quad \mathbf{\overline{r}v}}{\mathbf{v}}\,\mathbf{r}$$

Figure 5.6. Example of rule application and subsumption propagation.

- Case 3: the pivot is not present in either antecedent. Same reasoning as for Case 2, but arbitrarily choosing an antecedent for the substitution.

In all three cases the subproof rooted in $C'(n)$ is legal and $C'(n) \subseteq C(n)$. □

Figure 5.6 shows the effect of $R2$ and the subsequent application of Subsumption-Propagation on a small proof.

5.2.3 A Transformation Meta-Algorithm

The Local Transformation Framework defined by our rules leaves to the user the flexibility of choosing a particular *strategy* and a *termination criterion* for their application.

Whenever a sizeable amount of rules has to be applied, rather than running SubsumptionPropagation multiple times, it is more efficient to combine the application of all rules and the propagation of the modifications into a single traversal of the proof.

Algorithm 7, *TransformAndReconstruct*, illustrates this approach. At first it performs a topological sorting of the proof (line 2), in order to ensure that each node is visited after its parents. Then it analyzes one node at a time, checking if the corresponding resolution step is still sound (line 6). If the resolution step is sound, it updates the resolvent clause, determining the node contexts (if any) and the associated rules. At most one rule is applied, and the decision is based on local heuristic considerations (line 10). If the resolution step is not sound and either antecedent does not contain the pivot (lines 11, 13, 15), then the resolution step is removed by replacing the resolvent with that antecedent (which, missing the pivot, subsumes the resolvent); at the graph level, n is substituted by n^+ or n^-.

Note that the antecedent not responsible for the substitution might have lost all its resolvents and thus does not contribute to the proof anymore; in that case it is pruned away, together with the portion of the subproof rooted in it which has become detached from the global proof.

A key point of the algorithm is the call to *ApplyRule(left context, right context)*: this method heuristically chooses at most one context (possibly none) rooted in n and applies the corresponding rule. The instantiation of ApplyRule with different procedures yields concrete algorithms suitable for particular applications, as illustrated in the next sections.

Based on the above observations and on Theorem 5.2.1, we have the following result:

Theorem 5.2.2. *TransformAndReconstruct outputs a legal proof.*

5.3 Proof Compression

Resolution proofs, as generated by modern solvers, find application in many verification techniques. In most cases, the size of the proofs affects the efficiency of the methods in which they are used. It is known that the size of a resolution proof can grow exponentially with respect to the size of the input formula: even when proofs are representable in a manageable memory space, it might be crucial for efficiency to reduce or compress them as much as possible. Several compression technique have been developed and can be found in literature, ranging from memoization of common subproofs to partial regularization [Amj07, Amj08, Sin07, Cot10, BIFH+08,

```
Input: A legal proof, an instance of ApplyRule
Output: A legal proof
Data: TS: nodes topological sorting vector
begin
    TS ← topological_sorting_top_down(proof)
    foreach n ∈ TS do
        if n is not a leaf then
            p ← piv(n)
            if p̄ ∈ C(n⁻) and p ∈ C(n⁺) then
                C(n) ← Res_p(C(n⁻), C(n⁺))
                Determine left context lc of n, if any
                Determine right context rc of n, if any
                ApplyRule(rc, lc)
            else if p̄ ∉ C(n⁻) and p ∈ C(n⁺) then
                Substitute n with n⁻
            else if p̄ ∈ C(n⁻) and p ∉ C(n⁺) then
                Substitute n with n⁺
            else if p̄ ∉ C(n⁻) and p ∉ C(n⁺) then
                Heuristically choose a parent, substitute n with it
    end
end
```

Algorithm 7: TransformAndReconstruct.

DKPW10, FMP11]; however, since the problem of finding a minimum proof is NP-hard, it is still an open challenge to design heuristics capable of obtaining good reduction in practical situations.

This section discusses algorithms aimed at compressing proofs. We identify two kinds of *redundancies* in resolution proofs and present a set of post-processing techniques aimed at removing them; the techniques are independent from the way the refutation is produced and can be applied to an arbitrary resolution proof of unsatisfiability. We also illustrate how to combine these algorithms in an effective manner, and show the results of experimenting on a collection of SAT and SMT benchmarks.

We do not address directly the problem of core minimization, that is nonetheless achieved as a side effect of proof reduction. A rich literature exists on techniques aimed at obtaining a minimum, minimal, or small unsatisfiable core, that is a subset of the initial set of clauses that is still unsatisfiable [LMS04, CGS07, ZM03a,

OMA+04, Hua05, Bru03, DHN06, GMP07, MLA+05].

5.3.1 Proof Redundancies

This chapter focuses on two particular kinds of redundancies in resolution proofs.

The first one stems from the observation that, along each path from a leaf to the root, it is unnecessary to resolve upon a certain pivot more than once. The proof can be simplified, for example by keeping (for a given variable and a path) only the resolution step closest to the root, while cutting the others away. In the literature, a proof such that each variable is used as a pivot at most once along each path from a leaf to the root is said to be *regular* [Tse68].

The second kind of redundancy is related to the content of a proof. It might be the case that there exist multiple nodes associated with equal clauses; such nodes can be merged, keeping only one pair of parents and grouping together all the children. In particular, we call a proof *compact* if $C(n_i) = C(n_j) \Rightarrow i = j$ for any i, j, that is, different nodes are labeled by different clauses.

5.3.2 Proof Regularity

In this section we discuss how to make a proof (partially) regular. We show how to employ Algorithm 7 for this purpose and present two algorithms explicitly devised for regularization, namely RecyclePivots [BIFH+08] and its refinement RecyclePivotsWithIntersection [FMP11]. We illustrate them individually and explain how they can be combined to obtain more powerful algorithms.

5.3.2.1 Regularization in the Local Transformation Framework

The R rules are, as a matter of fact, a means to perform a "local" regularization; they are applied to contexts where a resolution step on a pivot $v(s)$ is immediately followed by a reintroduction of the pivot with positive ($R1, R2, R'2$) or negative ($R3$) polarity (see Figure 5.5).

Resolving on $v(s)$ is redundant, since the newly introduced occurrence of the pivot will be later resolved upon along the path to the global root; the R rules have the effect of simplifying the context, possibly pruning subproofs which do not contribute to the global proof anymore. Moreover, the rules replace the root of a context with a stronger one, which allows to achieve further compression as shown below.

Consider, for example, the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{\overline{p}o}}{\mathbf{qo}}\,p \quad \mathbf{p\overline{q}}}{\mathbf{po}}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{or}\,p \quad \overline{o}s}{rs}\,o \tag{1}$$

The highlighted context can be reduced via an application of $R2$ as follows:

$$\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{p\overline{q}}}{\mathbf{p}}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{or}\,p \quad \overline{o}s}{rs}\,o \tag{2}$$

The proof has become illegal as the literal o is now not introduced by any clause. Since a stronger conclusion ($p \subset po$) has been derived, o is now redundant and it can be eliminated all the way down to the global root or up to the point it is reintroduced by some other resolution step. In this example o can be safely removed together with the last resolution step which also becomes redundant. The resulting legal (and stronger) proof becomes:

$$\dfrac{\dfrac{pq \quad p\overline{q}}{p}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{r}\,p \tag{3}$$

At this stage no other R rule can be directly applied to the proof.

Rule $S2$ does not perform any simplification on its own, however it is still used in our framework. Its contribution is to produce a "shuffling" effect in the proof, in order to create more chances for the R rules to be applied.

Consider again our running example. $S2$ can be applied as follows:

$$\dfrac{\dfrac{pq \quad p\overline{q}}{\mathbf{p}}\,q \quad \dfrac{\mathbf{qr} \quad \mathbf{\overline{pq}}}{\mathbf{\overline{p}r}}\,q}{\mathbf{r}}\,p \tag{4}$$

$$\dfrac{\mathbf{qr} \quad \dfrac{\dfrac{p\overline{q} \quad pq}{\mathbf{p}}\,q \quad \mathbf{\overline{pq}}}{\mathbf{\overline{q}}}\,p}{\mathbf{r}}\,q \tag{5}$$

$S2$ has now exposed a new redundancy involving the variable q. The proof can be readily simplified by means of an application of $R2'$:

$$\dfrac{qr \qquad \dfrac{\dfrac{\mathbf{p\overline{q}} \qquad \mathbf{pq}}{\mathbf{p}}\,q \qquad \mathbf{\overline{pq}}}{\mathbf{\overline{q}}}\,p}{r}\,q \tag{6}$$

$$\dfrac{qr \qquad \dfrac{\mathbf{p\overline{q}} \qquad \mathbf{\overline{pq}}}{\mathbf{\overline{q}}}\,p}{r}\,q \tag{7}$$

As discussed in §5.2.3, the rewriting framework defined by our rules allows the flexibility of choosing a strategy and a termination criterion for their application.

A simple strategy is to eagerly apply the R rules until possible, shuffle the proof by means of $S2$ with the purpose of disclosing other redundancies, and then apply the R rules again, in an iterative fashion. However there is usually a very large number of contexts where $S2$ could be applied, and it is computationally expensive to predict whether one or a chain of $S2$ applications would eventually lead to the creation of contexts for an R rule.

For efficiency reasons, we rely on the meta-algorithm described in Algorithm 7, for a particular instantiation of the ApplyRule method. Algorithm 7 does a single traversal of the proof, performing shuffling and compression; it is run multiple times, setting a number of traversals to be performed and a timeout as termination criteria (whichever is reached first). The resulting regularization procedure is *ReduceAnd-Expose*, listed as Algorithm 8.

```
Input: A legal proof, timelimit: timeout, numtrav: number of transformation
       traversals, an instantiation of ApplyRule
Output: A legal proof
begin
    for i=1 to numtrav do
        TransformAndReconstruct(ApplyRule)
        if timelimit is reached then
            break
    end
end
```

Algorithm 8: ReduceAndExpose.

5.3.2.2 The RecyclePivots Approach

The RecyclePivots algorithm was introduced in [BIFH+08] as a linear-time technique to perform a partial regularization of resolution proofs.

RecyclePivots is based on analyzing the paths of a proof, focusing on the pivots involved in the resolution steps; if a pivot is resolved upon more than once on a path (which implies that the pivot variable is introduced and then removed multiple times), the resolution step closest to the root is kept, while the others are simplified away.

We illustrate this approach by means of an example. Consider the leftmost path of proof (1). Variable p is used twice as pivot. The topmost resolution step is redundant as it resolves upon p, which is reintroduced in a subsequent step (curly brackets denote the set RL of removable literals, see later).

$$\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{\overline{p}o}}{\mathbf{qo}\ \{\overline{p},\overline{q}\}}\,p \qquad \mathbf{p\overline{q}}}{\mathbf{po}\ \{\overline{p}\}}\,q \qquad \dfrac{qo \quad \overline{pq}}{\overline{p}o}\,q}{o}\,p \tag{1}$$

Regularization can be achieved by eliminating the topmost resolution step and by adjusting the proof accordingly. The resulting proof is shown below.

$$\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{p\overline{q}}}{\mathbf{p}}\,q \qquad \dfrac{qo \quad \overline{pq}}{\overline{p}o}\,q}{o}\,p \tag{2}$$

Algorithm 9 shows the recursive version of RecyclePivots (RP in the following). It is based on a depth-first visit of the proof, from the root to the leaves. It starts from the global root, having as input a set of *removable literals RL* (initially empty). The removable literals are essentially the (partial) collection of pivot literals encountered during the bottom-up exploration of a path. If the pivot variable of a resolution step under consideration is in RL (lines 15 and 18), then the resolution step is redundant and one of the antecedents may be removed from the proof. The resulting proof is illegal and has to be reconstructed into a legal one, which can be done in linear time, as shown in [BIFH+08].

Note that in the case of resolution proof trees, the outcome of the algorithm is a regular proof. For arbitrary resolution proof DAGs the algorithm is executed in a limited form (when nodes with multiple children are detected) precisely by resetting RL (line 10); therefore the result is not necessarily a regular proof.

```
Input: A node n, a set of removable literals RL
1  begin
2      if n is visited then
3          return
4      else
5          Mark n as visited
6          if n is a leaf then
7              return
8          else
9              if n has more than one child then
10                 RL ← ∅
11             p ← piv(n)
12             if p ∉ RL and p̄ ∉ RL then
13                 RecyclePivots(n⁺,RL ∪ {p̄})
14                 RecyclePivots(n⁻,RL ∪ {p})
15             else if p ∈ RL then
16                 n⁺ ← null
17                 RecyclePivots(n⁻,RL)
18             else if p̄ ∈ RL then
19                 n⁻ ← null
20                 RecyclePivots(n⁺,RL)
21 end
```

Algorithm 9: RecyclePivots(n,RL).

5.3.2.3 RecyclePivotsWithIntersection

The aforementioned limitation is due to the same circumstance that restricts the application of rules in the Local Transformation Framework, as discussed in §5.2.1. The set of removable literals of a node is computed for a particular path from the root to the node (which is enough in presence of proof trees), but does not take into account the existence of other possible paths to that node. Thus, suppose a node n with pivot p is replaced by one of its parents (let us say n^+) during the reconstruction phase, and $C(n^+) \nsubseteq C(n)$; then, it might happen that some of the literals in $C(n^+) \setminus C(n)$ are not resolved upon along *all* paths from n to the root, and are thus propagated to the root, making the proof illegal.

In order to address this issue, the authors of [FMP11] extend RP by proposing RecyclePivotsWithIntersection (RPI), an iterative version of which is illustrated in Algorithm 10. RPI refines RP by keeping track for each node n of the set of pivot

literals $RL(n)$ which get resolved upon along *all* paths from n to the root.

The computation of RL in the two approaches is represented in Figure 5.7 and Figure 5.8. RP and RPI behave in the same way whenever a node n has only one child. In case n has no children, i.e., it is the root, RPI takes into account the possibility for the root to be an arbitrary clause (rather than only $\perp$, as in refutations) and sets RL to include all variables of $C(n)$; it is equivalent to having a path from n to $\perp$ where all variables of $C(n)$ are resolved upon. The major difference between RP and RPI is in the way a node n with multiple children is handled: RP sets $RL(n)$ to $\emptyset$, while RPI sets $RL(n)$ to the intersection $\bigcap(RL(m_i) \cup q_i)$ of the removable literals sets of its children, augmented with the pivots of the resolution steps of which the children are resolvents.

$RL(n) = \emptyset$ $\begin{tikzcd} & n \arrow[dl, "p_1"'] \arrow[dr, "p_k"] & \\ m_1 & \cdots & m_k \end{tikzcd}$	$RL(n) = (RL(m) \cup \{q\})$ $q = \begin{cases} p & \text{if } \overline{p} \in C(n) \\ \overline{p} & \text{if } p \in C(n) \end{cases}$ $\begin{tikzcd} n \arrow[d, "p"'] \\ m \end{tikzcd}$	$RL(n) = \emptyset$ n

Figure 5.7. Computation of RL in RecyclePivots.

$RL(n) = \bigcap(RL(m_i) \cup \{q_i\})$ $q_i = \begin{cases} p_i & \text{if } \overline{p_i} \in C(n) \\ \overline{p_i} & \text{if } p_i \in C(n) \end{cases}$ $\begin{tikzcd} & n \arrow[dl, "p_1"'] \arrow[dr, "p_k"] & \\ m_1 & \cdots & m_k \end{tikzcd}$	$RL(n) = (RL(m) \cup \{q\})$ $q = \begin{cases} p & \text{if } \overline{p} \in C(n) \\ \overline{p} & \text{if } p \in C(n) \end{cases}$ $\begin{tikzcd} n \arrow[d, "p"'] \\ m \end{tikzcd}$	$RL(n) = \bigcup\{q_i\}$ $q_i = \begin{cases} p_i & \text{if } \overline{p_i} \in C(n) \\ \overline{p_i} & \text{if } p_i \in C(n) \end{cases}$ n

Figure 5.8. Computation of RL in RecyclePivotsWithIntersection.

RPI starts in Algorithm 10 by computing a topological sorting of the nodes (line 2), from the root to the leaves. $RL(root)$ is computed as the set of literals in the root clause; for any other node n, $RL(n)$ is initialized and then iteratively refined each time one of its children is visited. Similarly to RecyclePivots, whenever visiting an

```
Input: A legal proof
Input: A proof to be reconstructed
Data: TS: nodes topological sorting vector, RL: vector of sets of removable
      literals
1  begin
2      TS ← topological_sorting_bottom_up(proof)
3      foreach n ∈ TS do
4          if n is not a leaf then
5              if n is the root then
6                  RL(n) ← {p̄_i}_{p_i ∈ C(n)}
7              else
8                  p ← piv(n)
9                  if p ∈ RL(n) then
10                     n⁺ ← null
11                     if n⁻ not seen yet then
12                         RL(n⁻) ← RL(n)
13                         Mark n⁻ as seen
14                     else RL(n⁻) ← RL(n⁻) ∩ RL(n)
15                 else if p̄ ∈ RL(n) then
16                     n⁻ ← null
17                     if n⁺ not seen yet then
18                         RL(n⁺) ← RL(n)
19                         Mark n⁺ as seen
20                     else RL(n⁺) ← RL(n⁺) ∩ RL(n)
21                 else if p ∉ RL(n) and p̄ ∉ RL(n) then
22                     if n⁻ not seen yet then
23                         RL(n⁻) ← (RL(n) ∪ {p})
24                         Mark n⁻ as seen
25                     else RL(n⁻) ← RL(n⁻) ∩ (RL(n) ∪ {p})
26                     if n⁺ not seen yet then
27                         RL(n⁺) ← (RL(n) ∪ {p̄})
28                         Mark n⁺ as seen
29                     else RL(n⁺) ← RL(n⁺) ∩ (RL(n) ∪ {p̄})
30     end
31 end
```

Algorithm 10: RecyclePivotsWithIntersection.

inner node n, if $piv(n)$ appears in $RL(n)$ then the resolution step is redundant and can be simplified away (lines 9-14, 15-20); in that case, $RL(n)$ is propagated to a parent of n without the addition of $piv(n)$.

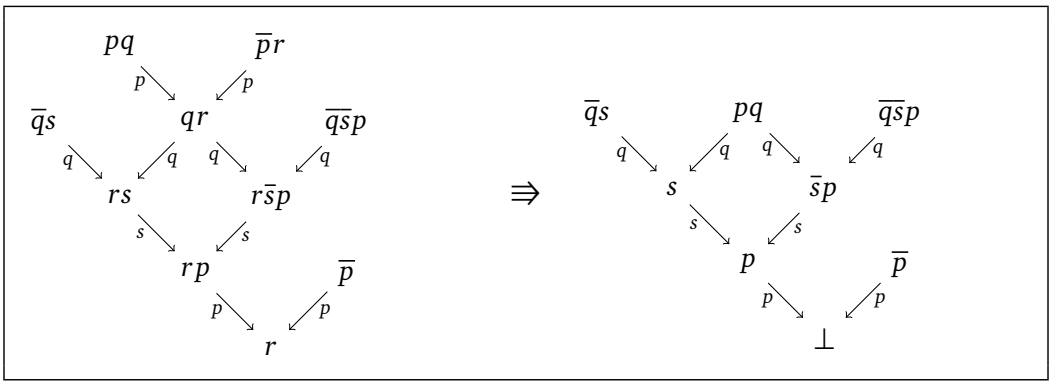


Figure 5.9. Compression of a proof by means of RecyclePivotsWithIntersection.

Figure 5.9 shows the effect of RPI on a small proof where RP cannot achieve any compression: RP sets $RL(qr) = \emptyset$ since qr has two children, while RPI sets $RL(qr) = \{\overline{r}, \overline{p}, \overline{q}\}$ and consequently simplifies the uppermost resolution step, since it is able to detect that p is resolved upon along both paths from qr to the root.

5.3.2.4 RecyclePivots and the Local Transformation Framework

RecyclePivots (as well as its refinement RecyclePivotsWithIntersection) and ReduceAndExpose both aim at compressing a proof by identifying and removing pivot redundancies along paths from the root to the leaves. The main difference between the two approaches is that RecyclePivots operates on a *global perspective* without changing the topology of the proof (i.e., no shuffling), while ReduceAndExpose operates on *local contexts* and allows the topology to change. Both approaches have advantages and disadvantages.

Operating on a global perspective without modifying the topology allows a one-pass visit and compression of the proof. Maintaining a fixed topology, however, may prevent the disclosure of hidden redundancies. For instance the application of RecyclePivots to the example of §5.3.2.1 would have stopped to step (3), since no more redundant pivots can be found along a path (the proof is regular). The local contexts instead have to be gathered and considered multiple times. On the other hand, the ability of ReduceAndExpose to change the topology allows more redundancies to be exposed.

Another advantage of RecyclePivots is that it can eliminate redundancies that are separated by many resolution steps. The R rewriting rules instead are applicable only

when there is a reintroduction of a certain variable immediately after a resolution step upon it. Such configurations, when not present in the proof, can be produced by means of applications of the $S2$ rule.

The ability of the Local Transformation Framework to disclose redundancies and the effectiveness of RecyclePivots at removing them can be combined in a simple hybrid approach, shown in Algorithm 11.

```
Input: A legal proof, numloop: number of global iterations, numtrav:
       number of transformation traversals for each global iteration,
       timelimit: timeout, an instantiation of ApplyRule
Output: A legal proof
1 begin
2     timeslot = timelimit/numloop
3     for i=1 to numloop do
4         RecyclePivots(root,∅)
5         // RPtime is the time taken by RecyclePivots in the last call
6         ReduceAndExpose(timeslot − RPtime,numtrav,ApplyRule)
7     end
8 end
```

Algorithm 11: RP + RE.

The algorithm takes as input an overall time limit, a number of *global iterations* and a number of transformation traversals for ReduceAndExpose. The time limit and the amount of global iterations determine the execution time available to ReduceAndExpose during each iteration. ReduceAndExpose and RecyclePivots are run one after the other by Algorithm 11, alternately modifying the topology to expose redundancies and simplifying them away.

A similar, but more efficient algorithm can be obtained by simply replacing the call to RecyclePivots with a call to RecyclePivotsWithIntersection.

5.3.3 Proof Compactness

The focus of this section is the notion of compactness as introduced in §5.3.1: a proof is compact whenever different nodes are labeled with different clauses, that is $C(n_i) = C(n_j) \Rightarrow i = j$ for any i, j. We first present an algorithm to address redundancies related to the presence of multiple occurrences of a same unit clause in a proof. Then we illustrate a technique based on a form of structural hashing, which makes a proof more compact by identifying and merging nodes having exactly the

same pair of parents. We conclude by showing how to combine these procedures with the Local Transformation Framework.

5.3.3.1 Unit Clauses-Based Simplification

The simplification of a proof by exploiting the presence of unit clauses has already been addressed in the literature in [FMP11] and [BIFH+08]. The two works pursue different goals. The *RecycleUnits* algorithm from [BIFH+08] uses learned unit clauses to rewrite subproofs that were derived before learning them. On the other hand, the *LowerUnits* algorithm from [FMP11] collects unit clauses and reinserts them at the level of the global root, thus removing redundancies due to multiple resolution steps on the same unit clauses.

Following the idea of [FMP11], we present *PushdownUnits*, listed as Algorithm 12. First, the algorithm traverses a proof in a top-down manner, detaching and collecting subproofs rooted in unit clauses, while at the same time reconstructing the proof to keep it legal (based on the schema of Algorithm 7); then, (some of) these subproofs are attached back at the end of the proof, adding new resolution steps. PushdownUnits improves over LowerUnits by performing unit collection and proof reconstruction in a single pass.

The algorithm works as follows. The proof is traversed according to a topological order. When a node n is visited s.t. $C(n)$ is the resolvent of a sound resolution step with pivot p, its parents are examined. Assume n^+ is a unit clause, that is $C(n^+) = p$; then n is replaced by the other parent n^- and n^+ is added to the set of unit clauses CU.

This transformation phase might add extra literals EL to the original global root r; if this is the case, the necessary resolution steps to make the proof legal are added at the end, starting from r. The nodes previously collected are taken into account one by one; for each m, if $C(m) = s$ and $\bar{s}$ is one of the extra literals EL, then a new resolution step is added and its resolvent becomes the new root.

Note that not necessarily all these nodes will be added back to the proof. Multiple nodes might be labeled by the same literal, in which case the correspondent variable will be used only once as pivot. Also, a collected literal which was an antecedent of some resolution step might have been anyway resolved upon again along all paths from that resolution step to the global root; if so, it does not appear in the set of extra literals. The subproofs rooted in these unnecessary nodes can be (partially) pruned away to further compress the proof.

```
Input: A legal proof
Output: A legal proof
Data: TS: nodes topological sorting vector, CU: collected units set, EL: set
      of extra literals appearing in the global root
1  begin
2      TS ← topological_sorting_top_down(proof)
3      r ← global root
4      foreach n ∈ TS do
5          if n is not a leaf then
6              p ← piv(n)
7              if p̄ ∈ C(n⁻) and p ∈ C(n⁺) then
8                  C(n) ← Res_p(C(n⁻), C(n⁺))
9                  if C(n⁺) = p then
10                     Substitute n with n⁻
11                     CU ← CU ∪ {n⁺}
12                 else if C(n⁻) = p̄ then
13                     Substitute n with n⁺
14                     CU ← CU ∪ {n⁻}
15             else if piv(n) ∉ C(n⁻) and piv(n) ∈ C(n⁺) then
16                 Substitute n with n⁻
17             else if piv(n) ∈ C(n⁻) and piv(n) ∉ C(n⁺) then
18                 Substitute n with n⁺
19             else if piv(n) ∉ C(n⁻) and piv(n) ∉ C(n⁺) then
20                 Heuristically choose a parent, substitute n with it;
21     end
22     EL ← extra literals of C(r)
23     foreach m ∈ CU do
24         s ← C(m)
25         if s̄ ∈ EL then
26             Add a new node o s.t. C(o) = Res_v(s)(C(r), C(m))
27             r ← o
28     end
29 end
```

Algorithm 12: PushdownUnits.

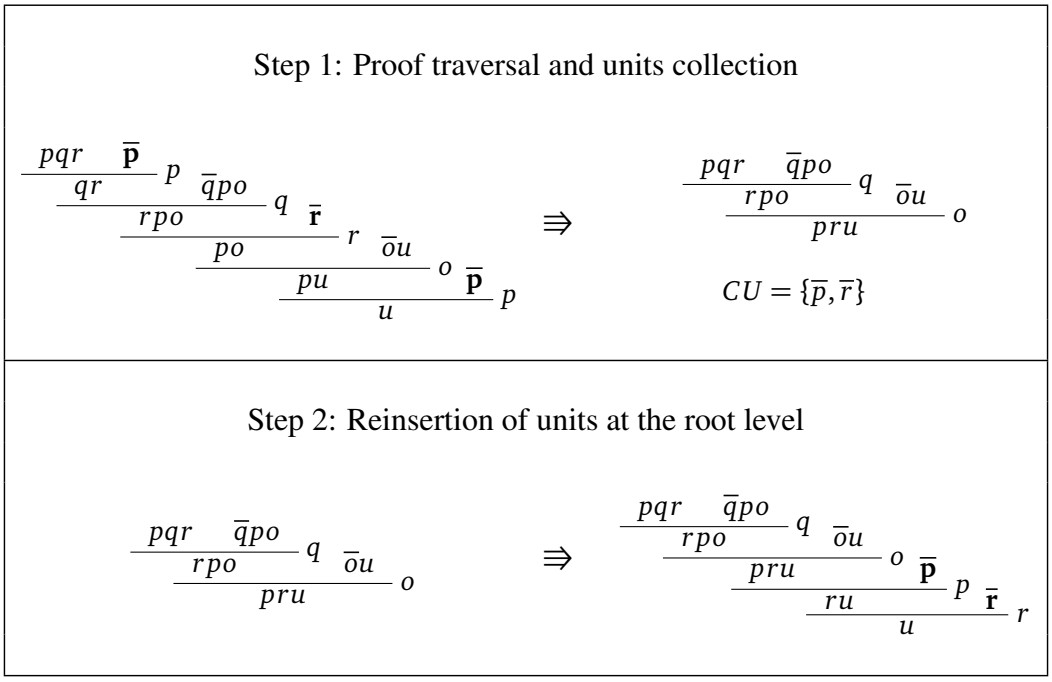


Figure 5.10. Example of application of PushdownUnits. Note that the lowest occurrence of $\overline{p}$ is not added back to the proof.

5.3.3.2 Structural Hashing

The work of [Cot10] proposes an algorithm based on a form of *structural hashing*; it explicitly takes into account how resolution proofs are obtained in CDCL SAT-solvers from a sequence of subproofs deriving learnt clauses, and keeps a hash map which stores for each derived clause its pair of antecedents. While building the global proof from the sequence of subproofs, whenever a clause would be added, if its pair of antecedents is already in the hash map, then the existing clause is used.

Taking inspiration from the idea at the base of this technique, we present a post-processing compression algorithm, *StructuralHashing*, which aims at improving the compactness of a proof. StructuralHashing is illustrated in Algorithm 13.

The proof is traversed in topological order. When a node n is visited, the algorithm first checks whether its antecedents are already in the hash map; if so, another node m with the same parents has been seen before. In that case, n is replaced by m and the children of n are assigned to m. The use of a topological visit guarantees the soundness of the algorithm: it is safe to replace the subproof rooted in n with that rooted in m since either (i) m is an ancestor of n (and the subproof rooted in m is contained in the subproof rooted in n) or (ii) m and n are not on a same path to

```
Input: A legal proof
Output: A legal proof
Data: TS: nodes topological sorting vector, HM: hash map associating a
      node to its pair of parents
1  begin
2      TS ← topological_sorting_top_down(proof)
3      foreach n ∈ TS do
4          if n is not a leaf then
5              if < n+, n− > ∈ HM then
6                  m ← HM(< n+, n− >)
7                  Replace n with m
8                  Assign n children to m
9              else
10                 HM(< n+, n− >) ← n
11     end
12 end
```

Algorithm 13: StructuralHashing.

the global root, so m is not involved in the derivation of n.

Note that StructuralHashing does not guarantee a completely compact proof; if two nodes n_1, n_2 have the same parents, then $C(n_1) = C(n_2)$, but the converse is not necessarily true. A complete but more computationally expensive technique might consist in employing a hash map to associate clauses with nodes (rather than pairs of nodes with nodes as done in StructuralHashing), based on a function that derives map keys from the clauses content; an implementation of this technique can be found in [ske].

5.3.3.3 StructuralHashing and the Local Transformation Framework

StructuralHashing is a one-pass compression technique, like RecyclePivots and RecyclePivotsWithIntersection. Nevertheless, it is still possible to exploit the Local Transformation Framework in order to disclose new redundancies and remove them, in an iterative manner. We illustrate this approach in Algorithm 14.

5.3.3.4 A Synergic Algorithm

It is possible to combine the compression techniques illustrated so far as shown in Algorithm 15, exploiting their individual features for a synergistic effect. The com-

```
Input: A legal proof, numloop: number of global iterations, numtrav: number
       of transformation traversals for each global iteration, timelimit:
       timeout, an instantiation of ApplyRule
Output: A legal proof
1 begin
2     timeslot=timelimit/numloop
3     for i=1 to numloop do
4         StructuralHashing()
5         // SHtime is the time taken by StructuralHashing in the last call
6         ReduceAndExpose(timeslot−SHtime,numtrav,ApplyRule)
7     end
8 end
```

Algorithm 14: SH + RE.

bined approach executes the algorithms sequentially for a given number of *global iterations*. Note that PushdownUnits is kept outside of the loop: in our experience, SH, RPI and RE are unlikely to introduce unit clauses in the proofs, thus for efficiency PushdownUnits is run only once before the main loop.

```
Input: A legal proof, numloop: number of global iterations, numtrav: number
       of transformation traversals for each global iteration, timelimit:
       timeout, an instantiation of ApplyRule
Output: A legal proof
1 begin
2     timeslot=timelimit/numloop
3     PushdownUnits()
4     for i=1 to numloop do
5         StructuralHashing()
6         RecyclePivotsWithIntersection()
7         // SHtime and RPItime are the time taken by StructuralHashing
          // and RecyclePivotsWithIntersection in the last call
8         ReduceAndExpose(timeslot−SHtime−RPItime,numtrav,ApplyRule)
9     end
10 end
```

Algorithm 15: PU + SH + RPI + RE.

The overall complexity of the combined algorithm is parametric in the number of global iterations and actual transformation traversals (also depending on the

specified time limit).

PushdownUnits performs a topological visit of the proof, collecting unit clauses and adding them back at the level of the global root; the complexity is $O(|V|+|E|)$, linear in the size of the resolution proof DAG.

Complexity is $O(|V|+|E|)$ also for StructuralHashing, which traverses the proof once, making use of an hash table to detect the existence of multiple nodes with the same resolvents.

An iterative implementation of RecyclePivotsWithIntersection consists of a bottom-up scan of the proof, while computing the sets of removable literals and pruning branches, followed by a reconstruction phase; the complexity is again $O(|V|+|E|)$.

Each execution of TransformAndReconstruct, on which ReduceAndExpose is based, computes a topological sorting of the nodes and traverses the proof top-down applying rewriting rules. If m transformation traversals are executed, the complexity of ReduceAndExpose is $O(m(|V|+|E|))$.

Note that PushdownUnits, RecyclePivotsWithIntersection, TransformAndReconstruct also perform operations at the level of clauses, checking the presence of pivots, identifying rule contexts, updating resolvents. These operations depend on the width of the involved clauses; in practice, this value is very small compared to the proof size, and the complexity can be considered $O(|V|+|E|)$.

Finally, if n global iterations are carried out, the total complexity is $O(nm(|V|+|E|))$. There is a clear tradeoff between efficiency and compression. The higher the value of m is, the more redundancies are exposed and then removed; in practice, however, especially in case of large proofs, a complexity higher than linear cannot be afforded, so the nm factor should be kept constant in the size of the proofs.

Some heuristics on the application of the local rules in conjunction with RecyclePivots, RecyclePivotsWithIntersection and StructuralHashing have been proved particularly successful: we refer the reader to §5.3.4, §5.3.5 and §5.5 for details.

5.3.4 Experiments on SMT Benchmarks

As a first stage of experimentation, we carried out an evaluation of the three algorithms RecyclePivots (RP), ReduceAndExpose (RE), and their combination RP+RE. The algorithms were implemented inside the tool OpenSMT [BPST10], with proof-logging capabilities enabled.

We experimented on the set of unsatisfiable benchmarks taken from the SMT-LIB [RT06] from the categories QF_UF, QF_IDL, QF_LRA, QF_RDL. For these sets of benchmarks we noticed that the aforementioned compression techniques are very effective. We believe that the reason is connected with the fact that the introduction of theory lemmata in SMT is performed lazily: the delayed introduction of

clauses involved in the final proof may negatively impact the online proof construction in the SAT solver.

All the experiments were carried out on a 32-bit Ubuntu server featuring a Dual-Core 2GHz Opteron CPU and 4GB of memory; a timeout of 600 seconds and a memory threshold of 2GB (whatever is reached first) were put as limit to the executions [1].

Table 5.1. Results for SMT benchmarks. *#Bench* reports the number of benchmarks solved and processed within the time/memory constraints, *RedNodes%* and *RedEdges%* report the average compression in the number of nodes and edges of the proof graphs, and *RedCore%* reports the average compression in the unsatisfiable core size. *TranTime* is the average transformation time in seconds.

	#Bench	RedNodes%	RedEdges%	RedCore%	TranTime(s)
RP	1370	6.7	7.5	1.3	1.7

(a)

	#Bench		RedNodes%		RedEdges%		RedCore%		TranTime(s)	
Ratio	RE	RP+RE	RE	RP+RE	RE	RP+RE	RE	RP+RE	RE	RP+RE
0.01	1364	1366	2.7	8.9	3.8	10.7	0.2	1.4	3.5	3.4
0.025	1363	1366	3.8	9.8	5.1	11.9	0.3	1.5	3.6	3.6
0.05	1364	1366	4.9	10.7	6.5	13.0	0.4	1.6	4.3	4.1
0.075	1363	1366	5.7	11.4	7.6	13.8	0.5	1.7	4.8	4.5
0.1	1361	1364	6.2	11.8	8.3	14.4	0.6	1.7	5.3	5.0
0.25	1357	1359	8.4	13.6	11.0	16.6	0.9	1.9	8.2	7.6
0.5	1346	1348	10.4	15.0	13.3	18.4	1.1	2.0	12.1	11.5
0.75	1339	1341	11.5	16.0	14.7	19.5	1.2	2.1	15.8	15.1
1	1335	1337	12.4	16.7	15.7	20.4	1.3	2.2	19.4	18.8

(b)

The executions of RE and RP+RE are parameterized with a time threshold, which we set as a fraction of the time taken by the solver to solve the benchmarks: more difficult instances are likely to produce larger proofs, and therefore more time is necessary to achieve compression. Notice that, regardless of the ratio, RE and RP+RE both perform at least one complete transformation loop, which could result in an execution time slightly higher than expected for low ratios and small proofs.

[1] The full experimental data is available at `http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

Table 5.2. Results for SMT benchmarks. *MaxRedNodes%* and *MaxRedEdges%* are the maximum compression of nodes and edges achieved by the algorithms in the suite on individual benchmarks.

	MaxRedNodes%	MaxRedEdges%	MaxRedCore%
RP	65.1	68.9	39.1

(a)

	MaxRedNodes%		MaxRedEdges%		MaxRedCore%	
Ratio	RE	RP+RE	RE	RP+RE	RE	RP+RE
0.01	54.4	66.3	67.7	70.2	45.7	45.7
0.025	56.0	77.2	69.5	79.9	45.7	45.7
0.05	76.2	78.5	78.9	81.2	45.7	45.7
0.075	76.2	78.5	79.7	81.2	45.7	45.7
0.1	78.2	78.8	82.9	83.6	45.7	45.7
0.25	79.3	79.6	84.1	84.4	45.7	45.7
0.5	76.2	79.1	83.3	85.2	45.7	45.7
0.75	78.2	79.9	84.4	86.1	45.7	45.7
1	78.3	79.9	84.6	86.1	45.7	45.7

(b)

Table 5.1 shows the average proof compression after the application of the algorithms. Table 5.1a shows the compression obtained after the execution of RP. Table 5.1b instead shows the compression obtained with RE and RP+RE parameterized with a timeout (ratio · solving time). In the columns we report the compression in the number of nodes and edges, the compression of the unsatisfiable core, and the actual transformation time. Table 5.2 is organized as Table 5.1 except that it reports the best compression values obtained over all the benchmarks.

On a single run RP clearly achieves the best results for compression with respect to transformation time. To get the same effect on average on nodes and edges, for example, RE needs about 5 seconds and a ratio transformation time/solving time equal to 0.1, while RP needs less than 2 seconds. As for core compression, the ratio must grow up to 1. On the other hand, as already remarked, RP cannot be run more than once.

The combined approach RP+RE shows a performance which is indeed better than the other two algorithms taken individually. It is interesting to see that the global perspective adopted by RP gives an initial substantial advantage, which is

slowly but constantly reduced as more and more time is dedicated to local transformations and simplifications.

Table 5.2b displays some remarkable peaks of compression obtained with the RE and RP+RE approaches on the best individual instances. Interestingly we noticed that in some benchmarks, like *24.800.graph* of the QF_IDL suite, RP does not achieve any compression, due to the high amount of nodes with multiple resolvents present in its proof that forces RecyclePivots to keep resetting the removable literals set RL. RP+RE instead, even for a very small ratio (0.01), performs remarkably, yielding 47.6% compression for nodes, 49.7% for edges and 45.7% for core.

5.3.5 Experiments on SAT Benchmarks

A second stage of experimentation was preceded by an implementation of all the compression algorithms discussed so far (Algorithm 6 - Algorithm 15) within the tool PeRIPLO, presented in §3.6.1.

We evaluated the following algorithms: PushdownUnits (PU), RecyclePivotsWithIntersection (RPI), ReduceAndExpose (RE), StructuralHashing (SH) (Algorithms 12,9,8,13) and their combinations RPI+RE (Algorithm 11), SH+RE (Algorithm 14), PU+RPI+SH+RE (Algorithm 15); the evaluation was carried out on a set of purely propositional benchmarks from the SAT Challenge 2012 [SATa], the SATLIB benchmark suite [SATc] and the CMU collection [CMU].

First, a subset of *unsatisfiable* benchmarks was extracted from the SAT Challenge 2012 collection by running MiniSAT 2.2.0 alone with a timeout of 900 seconds and a memory threshold of 14GB; this resulted in 261 instances from the Application track and the Hard Combinatorial track. In addition to these, another 125 unsatisfiable instances were obtained from the SATLIB Benchmark Suite and the CMU collection, for a total of 386 instances [2].

The experiments were carried out on a 64-bit Ubuntu server featuring a Quad-Core 4GHz Xeon CPU and 16GB of memory; a timeout of 1200 seconds and a memory threshold of 14GB were put as limit to the executions. The PeRIPLO framework was able to handle proofs up to 30 million nodes, as in the case of the *rbcl_xits_07_UNSAT* instance from the Application track in the SAT Challenge 2012 collection.

Differently from the case of SMT benchmarks, we decided to specify as termination criterion an explicit amount of transformation traversals per global iteration, focusing on the dependency between proofs size and time taken by the algorithms

[2]The full experimental data is available at `http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

Table 5.3. Results for SAT benchmarks. *#Bench* reports the number of benchmarks solved and processed within the time/memory constraints, *RedNodes%* and *RedEdges%* report the average compression in the number of nodes and edges of the proof graphs, *RedCore%* the average compression in the unsatisfiable core size. *TranTime* is the average transformation time in seconds; *Ratio* is the ratio between transformation time and overall time.

	#Bench	RedNodes%	RedCore%	RedEdges%	TranTime(s)	Ratio
PU	200	1.81	0.00	2.18	5.44	0.09
SH	205	5.90	0.00	6.55	4.53	0.07
RPI	203	28.48	1.75	30.66	14.32	0.21
RE 3	203	4.16	0.09	4.85	24.84	0.31
RE 5	203	5.06	0.14	5.88	37.86	0.41
RE 10	202	6.11	0.17	7.08	67.09	0.56
PU+SH+RPI	196	32.81	1.47	35.70	18.66	0.27

(a)

RPI+RE	#Bench	RedNodes%	RedCore%	RedEdges%	TranTime(s)	Ratio
2,3	201	30.69	2.08	33.49	34.78	0.39
2,5	200	30.71	2.15	33.53	40.37	0.45
3,3	200	31.28	2.23	34.22	51.43	0.51
3,5	200	31.56	2.34	34.50	61.16	0.56

(b)

SH+RE	#Bench	RedNodes%	RedCore%	RedEdges%	TranTime(s)	Ratio
2,3	204	17.33	0.09	19.20	33.87	0.38
2,5	204	19.81	0.15	21.92	48.40	0.47
3,3	204	21.68	0.16	23.96	56.39	0.51
3,5	202	23.69	0.18	26.17	70.75	0.59

(c)

PU+SH+RPI+RE	#Bench	RedNodes%	RedCore%	RedEdges%	TranTime(s)	Ratio
2,3	195	39.46	1.89	43.34	35.23	0.44
2,5	195	40.46	1.93	44.49	38.49	0.46
3,3	195	41.68	2.06	45.86	47.41	0.51
3,5	195	42.41	2.05	46.71	52.91	0.54

(d)

to move over proofs and compress them.

Table 5.3 reports the performance of the compression techniques. Table 5.3a shows the results for the individual techniques PU, SH, RPI, RE, the latter tested for an increasing amount of transformation traversals (3, 5, 10), and the combination PU+SH+RPI without RE. Tables 5.3b, 5.3c, 5.3d respectively report on the combinations RPI+RE, SH+RE, PU+SH+RPI+RE: in the first column, a pair n, m indicates that n global iterations and m transformation traversals per global iteration were carried out.

RPI is clearly the most effective technique on a single run, as for compression and ratio transformation time / overall time. For this set of experiments we tuned RE focusing on its ability to disclose new redundancies, so we did not expect exceptional results when running the algorithm by itself; the performance of RE improves with the number of transformation traversals performed, but cannot match that of RPI.

On the other hand, the heuristics adopted in the application of the rewriting rules (see §5.5) have a major effect on SH, enhancing the amount of compression from about 6% to more than 20%.

The combined approaches naturally achieve better and better results as the number of global iterations and transformation traversals grows. In particular, Algorithm 15, which brings together the techniques for regularization, compactness and redundancies exposure, reaches a remarkable average compression level of 40%, surpassing (ratio being equal) all other combined approaches.

Table 5.4. Results for SAT benchmarks. *MaxRedNodes%* and *MaxRedEdges%* are the maximum compression of nodes and edges achieved by the PU+SH+RPI+RE combination on a single benchmark.

PU+SH+RPI+RE	MaxRedNodes%	MaxRedCore%	MaxRedEdges%
2,3	83.7	21.5	83.7
2,5	84.9	21.6	85.2
3,3	87.1	22.1	87.4
3,5	87.9	22.2	88.2

We report for completeness in Table 5.4 the maximum compression obtained by the PU+SH+RPI+RE combination on the best individual instances.

5.4 Proof Transformation for Interpolation

In §3.3.5 we discussed interpolation for first order theories, in the context of lazy SMT solving; we focused on the approaches of [YM05] and [CGS10], which integrate propositional interpolation systems with procedures that compute interpolants for conjunctions of atoms in a theory. Later, in §3.4.1, we introduced the notion of theory labeled interpolation system and showed how it generalizes both the systems of [YM05] and [CGS10].

All these frameworks suffer from a limitation: theory lemmata, appearing in refutations, must not contain AB-mixed predicates. However, several decision procedures defined for SMT solvers heavily rely on the creation of new predicates during the solving process. Examples are delayed theory combination (DTC) [BBC+05b], Ackermann's expansion [Ack54], lemmas on demand [dMR02] and splitting on demand [BNOT06] (see §5.4.2). All these methods may introduce new predicates, which can potentially be AB-mixed.

In the following we use A and B to denote two quantifier-free formulae in a theory $\mathcal{T}$, for which we would like to compute an interpolant. Theories of interest are equality with uninterpreted functions EUF, linear arithmetic over the rationals LRA and the integers LIA, the theory of arrays AX, or a combination of theories, such as $EUF \cup LRA$.

This section shows how to compute an AB-pure proof from an AB-mixed one but without interfering with the internals of the SMT solver; our technique applies to any approach that requires the addition of AB-mixed predicates (see §5.4.2 for a set of examples). We illustrate how to employ the Local Transformation Framework to effectively modify the proofs, in such a way that the $\mathcal{T}$-LISs can be applied; in this way, it is possible to achieve a complete decoupling between the solving phase and the interpolant generation phase.

A sketch of the approach is depicted in Figure 5.11. The idea is to move all AB-mixed predicates (in grey) toward the leaves of the proof (Figure 5.11b) within maximal AB-mixed subproofs.

Definition 5.4.1 (AB-Mixed Subproof)**.** Given a resolution proof P, an *AB-mixed subproof* is a subproof P' of P rooted in a clause C, whose intermediate pivots are all AB-mixed predicates. P' is *maximal* if C does not contain AB-mixed predicates.

When dealing with a background theory $\mathcal{T}$ we note the following fact: if P' is a maximal AB-mixed subproof rooted in a clause C, then *C is a valid theory lemma for $\mathcal{T}$*.

This observation derives from Definition 5.4.1 and from the fact that (i) AB-mixed predicates can only appear in theory lemmata (as they do not appear in the

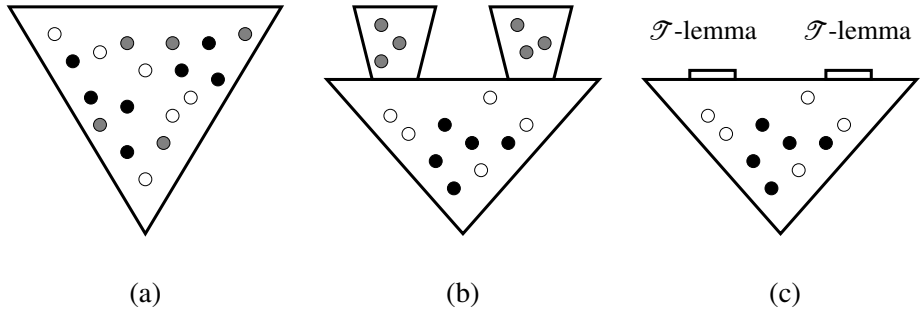


Figure 5.11. An overview of our approach. (a) is the proof generated by the SMT solver. White points represent A-local predicates, black points represent B-local predicates, grey points represent AB-mixed predicates. (b) AB-mixed predicates are confined inside AB-mixed trees. (c) AB-mixed trees are removed and their roots are valid theory lemmata in $\mathscr{T}$.

original formula) and (ii) a resolution step over two theory lemmata generates another theory lemma.

Once AB-mixed maximal subproofs are formed, it is possible to replace them with their root clauses (Figure 5.11c). The obtained proof is now free of AB-mixed predicates and can be used to derive an interpolant by means of a $\mathscr{T}$-LIS, provided that an interpolant generating procedure is available for the theory $\mathscr{T}$.

The crucial part of our approach is an algorithm for proof transformation. It relies on the Local Transformation Framework discussed in §5.2. An ad-hoc application of the rules can be used to transform a proof P into a proof P', where all AB-mixed variables are confined in AB-mixed subproofs. Each rewriting rule can effectively swap two pivots p and q in the resolution proof, or perform simplifications, depending on the particular context.

In the following, to facilitate the understanding of the algorithm, we will call AB-mixed and AB-pure predicates *light* and *heavy* respectively. The rules are applied when a light predicate is below a heavy predicate in the proof graph. The effect of an exhaustive application of the rules is to lift light predicates over heavy predicates as bubbles in water.

5.4.1 Pivot Reordering Algorithms

The Local Transformation Framework can be effectively employed to perform a *local reordering* of the pivots. Each rule in Figure 5.5 either swaps the position of two pivots ($S1, S2, R2$), or it eliminates at least one pivot ($R1, R2', R3$). This feature can be used to create an application strategy aimed at sorting the pivots in a proof

P, by transforming it into a proof P' such that all light variables are moved above heavy variables.

In order to achieve this goal it is sufficient to consider only *unordered* contexts, i.e. those in which $v(t)$ is a light variable and $v(s)$ is a heavy variable. Therefore a simple non-deterministic algorithm can be derived as Algorithm 16.

```
Input: A legal proof
Output: A legal proof without unordered contexts
Data: U: set of unordered contexts
begin
    Determine U, e.g. through a visit of the proof
    while U ≠ ∅ do
        Choose a context in U
        Apply the associated rule, and SubsumptionPropagation if necessary
        Update U
    end
end
```

Algorithm 16: PivotReordering.

The algorithm terminates: note in fact that each iteration strictly decreases the distance of an occurrence of a heavy pivot w.r.t. the global root, until no more unordered contexts are left.

```
Input: A left context lc, a right context rc
begin
    if lc is ordered and rc is unordered then
        Apply rule for rc
    else if lc is unordered and rc is ordered then
        Apply rule for lc
    else if lc is unordered and rc is unordered then
        Heuristically choose between lc and rc and apply rule
end
```

Algorithm 17: ApplyRuleForPivotReordering.

A more efficient choice is to make use of Algorithm 7 TransformAndReconstruct, by instantiating the ApplyRule method so that it systematically pushes light variables above heavy ones; a possible instantiation is shown in Algorithm 17. An algorithm for pivot reordering would then consist of a number of consecutive runs of

TransformAndReconstruct, stopping when no more unordered contexts are found: Algorithm 18, *PivotReordering2*, implements this approach.

```
Input: A legal proof
Output: A legal proof without unordered contexts
1 begin
2     while unordered contexts are found do
3         TransformAndReconstruct(ApplyRuleForPivotReordering)
4     end
5 end
```

Algorithm 18: PivotReordering2.

5.4.2 SMT Solving and AB-Mixed Predicates

In this section we show a number of techniques currently employed in state-of-the-art SMT solvers that can potentially introduce *AB*-mixed predicates during the solving phase. If these predicates become part of the proof of unsatisfiability, the proof reordering algorithms described in §5.4.1 can be applied to produce an *AB*-pure proof.

5.4.2.1 Theory Reduction Techniques

Let $\mathcal{T}_k$ and $\mathcal{T}_j$ be two decidable theories such that $\mathcal{T}_k$ is weaker (less expressive) than $\mathcal{T}_j$. Given a $\mathcal{T}_j$-formula φ, and a decision procedure SMT($\mathcal{T}_k$) for quantifier-free formulae in $\mathcal{T}_k$, it is often possible to obtain a decision procedure SMT($\mathcal{T}_j$) for quantifier-free formulae in $\mathcal{T}_j$ by augmenting φ with a finite set of $\mathcal{T}_k$-lemma ψ. These lemmata (or axioms) explicitly encode the necessary knowledge such that $\mathcal{T}_k \models \varphi \wedge \psi$ if and only if $\mathcal{T}_j \models \varphi$. Therefore a simple decision procedure for $\mathcal{T}_j$ is as described by Algorithm 19.

```
Input: φ for T_j
1 begin
2     ψ = generateLemmata(φ)
3     return SMT(T_k)(φ ∧ ψ)
4 end
```

Algorithm 19: A reduction approach for SMT($\mathcal{T}_j$).

In practice the lemmata generation function can be made lazy by plugging it inside the SMT solver directly; this paradigm is known as lemma on demand [dMR02] or splitting on demand [BNOT06]. We show some reduction techniques as follows.

Reduction of AX to EUF. We consider the case where $\mathcal{T}_k = EUF$, the theory of equality with uninterpreted functions, and $\mathcal{T}_j = AX$, the theory of arrays with extensionality. The axioms of EUF are the ones of equality (reflexivity, symmetry, and transitivity) plus the congruence axioms $\forall x, y\,(x = y \rightarrow f(x) = f(y))$, for any functional symbol of the language.

The theory of arrays AX is instead axiomatized by:

$$\forall x, y, z \qquad rd(wr(x, y, z), y) = z \tag{5.1}$$

$$\forall x, y, w, z \qquad y = w \vee rd(wr(x, y, z), w) = rd(x, w) \tag{5.2}$$

$$\forall x, z \qquad x = z \longleftrightarrow (\forall y.\ rd(x, y) = rd(z, y)) \tag{5.3}$$

State-of-the-art approaches for AX implemented in SMT solvers [BB08, dMB09, GKF08, BNO+08] are all based on reduction to EUF. Instances of the axioms of AX are added to the formula in a lazy manner until either the formula is proven unsatisfiable or saturation is reached. The addition of new lemmata may require the creation of AB-mixed predicates when a partitioned formula is considered.

Example 5.4.1. Let $\varphi = A \wedge B$, where A is $c = wr(d, i, e)$, and B is $rd(c, j) \neq rd(d, j) \wedge rd(c, k) \neq rd(d, k) \wedge j \neq k$. Symbols $\{i, e\}$ are A-local, $\{j, k\}$ are B-local, and $\{c, d\}$ are AB-common. To prove φ unsatisfiable with a reduction to EUF, we need to instantiate axiom (5.2) twice, so that ψ is $(i = j \vee rd(wr(d, i, e), j) = rd(d, j)) \wedge (i = k \vee rd(wr(d, i, e), k) = rd(d, k))$. Notice that we introduced four AB-mixed predicates. Now we can send $\varphi \wedge \psi$ to an SMT solver for EUF to produce the proof of unsatisfiability. Figure 5.12 shows a possible resolution proof generated by the SMT solver, and how it can be transformed into a proof without AB-mixed predicates.

Reduction of LIA to LRA. Decision procedures for LIA (linear integer arithmetic) often rely on iterated calls to a decision procedure for LRA (linear rational arithmetic). An example is the method of *branch-and-bound*: given a feasible rational region R for $\vec{d} = (d_1, \ldots, d_n)$, and a non-integer point $\vec{c} \in R$ for $\vec{d}$, then one step of branch-and-bound generates the two subproblems $R \cup \{d_i \leq \lfloor c_i \rfloor\}$ and $R \cup \{d_i \geq \lceil c_i \rceil\}$. These are again recursively explored until an integer point $\vec{c}$ is found.

Note that the splitting on the bounds can be delegated to the propositional engine by adding the lemma $((d_i \leq \lfloor c_i \rfloor) \vee (d_i \geq \lceil c_i \rceil))$. In order to obtain a faster convergence of the algorithm, it is possible to split on *cuts*, i.e. linear constraints, rather

Id	Clauses	Prop. abstract.
1	$c = wr(d,i,e)$	p_1
2	$rd(c,j) \neq rd(d,j)$	$\overline{p_2}$
3	$rd(c,k) \neq rd(d,k)$	$\overline{p_3}$
4	$j \neq k$	$\overline{p_4}$
5	$(i = j \vee rd(wr(d,i,e),j) = rd(d,j))$	$p_5\, p_6$
6	$(i = k \vee rd(wr(d,i,e),k) = rd(d,k))$	$p_7\, p_8$
7	$(c \neq wr(d,i,e) \vee rd(c,j) = rd(d,j) \vee rd(wr(d,i,e),j) \neq rd(d,j))$	$\overline{p_1}\, p_2\, \overline{p_6}$
8	$(c \neq wr(d,i,e) \vee rd(c,k) = rd(d,k) \vee rd(wr(d,i,e),k) \neq rd(d,k))$	$\overline{p_1}\, p_3\, \overline{p_8}$
9	$(j = k \vee i \neq j \vee i \neq k)$	$p_4\, \overline{p_5}\, \overline{p_7}$

$$
\begin{array}{l}
\underline{p_7 p_8 \quad \overline{p_1} p_3 \overline{p_8}} \\
\quad \underline{\overline{p_1} p_3 p_7 \qquad p_4 \overline{p_5} \overline{p_7}} \\
\quad \mathbf{p_4}\ \underline{\overline{\mathbf{p_1}}\mathbf{p_3}\mathbf{p_4}\overline{\mathbf{p_5}} \qquad \overline{\mathbf{p_4}} \quad p_5 p_6 \quad \overline{p_1} p_2 \overline{p_6}} \\
\quad\quad \mathbf{p_5}\ \underline{\overline{\mathbf{p_1}}\mathbf{p_3}\overline{\mathbf{p_5}} \qquad \overline{\mathbf{p_1}}\mathbf{p_2}\mathbf{p_5}} \\
\quad\quad\quad \underline{\overline{\mathbf{p_1}}\mathbf{p_2}\mathbf{p_3} \qquad \overline{p_3}} \\
\quad\quad\quad\quad \underline{\overline{p_1} p_2 \qquad p_1} \\
\quad\quad\quad\quad\quad \underline{p_2 \qquad \overline{p_2}} \\
\quad\quad\quad\quad\quad\quad \bot
\end{array}
$$

(a)

$$
\begin{array}{l}
\underline{p_7 p_8 \quad \overline{p_1} p_3 \overline{p_8}} \\
\quad \underline{\overline{p_1} p_3 p_7 \qquad p_4 \overline{p_5} \overline{p_7}} \quad \underline{p_5 p_6 \quad \overline{p_1} p_2 \overline{p_6}} \\
\quad \mathbf{p_5}\ \underline{\overline{\mathbf{p_1}}\mathbf{p_3}\mathbf{p_4}\overline{\mathbf{p_5}} \qquad \overline{\mathbf{p_1}}\mathbf{p_2}\mathbf{p_5}} \\
\quad\quad \mathbf{p_4}\ \underline{\overline{\mathbf{p_1}}\mathbf{p_2}\mathbf{p_3}\mathbf{p_4} \qquad \overline{\mathbf{p_4}}} \\
\quad\quad\quad \underline{\overline{\mathbf{p_1}}\mathbf{p_2}\mathbf{p_3} \qquad \overline{p_3}} \\
\quad\quad\quad\quad \underline{\overline{p_1} p_2 \qquad p_1} \\
\quad\quad\quad\quad\quad \underline{p_2 \qquad \overline{p_2}} \\
\quad\quad\quad\quad\quad\quad \bot
\end{array}
$$

(b)

Figure 5.12. Clauses from Example 5.4.1. $\varphi = \{1,2,3,4\}$, $\psi = \{5,6\}$. Clauses 7-9 are theory lemmata discovered by the EUF solver. (a) is a possible proof obtained by the SMT solver (for EUF) on $\varphi \wedge \psi$. (b) is a proof after swapping p_4 and p_5 by means of rule $S2$; in the resulting proof all mixed literals (p_5-p_8) appear in the upper part of the proof in an AB-mixed proof subtree. The root of the AB-mixed subtree $\overline{p_1} p_2 p_3 p_4$ is a valid theory lemma in AX.

than on simple bounds. However cuts may add AB-mixed predicates if A-local and B-local variables are mixed into the same cut.

Example 5.4.2. Let $\varphi = A \wedge B$ in LIA, where A is $5d - e \leq 1 \wedge e - 5d \leq -1$, and B is $5b - e \leq -2 \wedge e - 5b \leq 3$. Setting ψ as the axiom $(d - b \leq 0) \vee (d - b \geq 1)$ (which contains two AB-mixed literals) is sufficient for $\varphi \wedge \psi$ to be proven unsatisfiable by a solver for LRA, by discovering two additional theory lemmata $((5d - e \not\leq 1) \vee (e - 5b \not\leq 3) \vee (d - b \leq 0))$ and $((5d - e \not\leq -1) \vee (e - 5z \not\leq -2) \vee (d - b \geq 1))$.

Ackermann's Expansion. When $\mathcal{T}_j$ is a combination of theories of the form $EUF \cup \mathcal{T}_k$, Ackermann's expansion [Ack54] can be used to reduce the reasoning from $\mathcal{T}_j$ to $\mathcal{T}_k$. The idea is to use as ψ the exhaustive instantiation of the congruence axiom $\forall x, y\ (x = y \rightarrow f(x) = f(y))$ for all pairs of uninterpreted constants appearing in uninterpreted functional symbols and all uninterpreted functional symbols f in φ. This instantiation generates AB-mixed predicates when x is instantiated with an A-local symbol and y with a B-local one.

Example 5.4.3. Let $\mathcal{T}_k = LRA$. Let $\varphi = A \wedge B$, where A is $(a = j + k \wedge f(a) = c)$, and B is $(b = j + k \wedge f(b) = d \wedge c \neq d)$. Setting ψ as the axiom $(a \neq b) \vee (f(a) = f(b)$ is sufficient for LRA to detect the unsatisfiability of $\varphi \wedge \psi$, by discovering two additional theory lemmata $((f(a) \neq f(b)) \vee (f(a) \neq c) \vee (f(b) \neq d) \vee (c \neq d))$ and $((a \neq j + k) \vee (b \neq j + k) \vee (a = b))$.

5.4.2.2 Theory Combination via DTC

A generic framework for theory combination was introduced by Nelson and Oppen in [NO79]. We recall it briefly as follows.

Given two signature-disjoint and stably-infinite theories $\mathcal{T}_1$ and $\mathcal{T}_2$, a decision procedure for a conjunction of constraints in the combined theory $\mathcal{T}_1 \cup \mathcal{T}_2$ can be obtained from the decision procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$. First, the formula φ is *flattened*, i.e. auxiliary uninterpreted constants are introduced to separate terms that contain both symbols of $\mathcal{T}_1$ and $\mathcal{T}_2$. Then the idea is that the two theory solvers for $\mathcal{T}_1$ and $\mathcal{T}_2$ are forced to exhaustively exchange *interface equalities* i.e. equalities between the constants that appear both in constraints of $\mathcal{T}_1$ and $\mathcal{T}_2$ after flattening[3].

Delayed theory combination (DTC) implements a non-deterministic version of the Nelson-Oppen framework, in which interface equalities are not exchanged by the deciders directly, but they are guessed by the SAT solver. With DTC it is possible to

[3]Note that in practice flattening can be avoided. For instance in Example 5.4.4 we do not perform any flattening.

achieve a higher level of modularity w.r.t. the classical Nelson-Oppen framework. DTC is currently implemented (with some variations) in most state-of-the-art SMT solvers.

If no *AB*-mixed interface equality is generated, an interpolant can be derived with the methods already present in the literature; otherwise our method can be applied to reorder the proof, as an alternative to the techniques described in [CGS08, GKT09].

Example 5.4.4. Consider again φ of Example 5.4.3. Since a, b, $f(a)$, $f(b)$ appear in constraints of both theories, we need to generate two interface equalities $a = b$ and $f(a) = f(b)$. The guessing of their polarity is delegated to the SAT solver. The SMT solver will detect unsatisfiability after the EUF solver discovers the two theory lemmata $((a \neq b) \vee (f(a) = f(b))$ and $((f(a) \neq f(b)) \vee (f(a) \neq c) \vee (f(b) \neq d) \vee (c \neq d))$ and the *LRA* solver discovers the theory lemma $((a \neq j + k) \vee (b \neq j + k) \vee (a = b))$.

5.4.3 Experiments on SMT Benchmarks

For the purpose of this experimentation we chose to focus on one particular application among those of §5.4.2, namely Ackermann's Expansion for Theory Combination.

We evaluated the proof transformation technique on the set of QF_UFIDL formulae from the SMT-LIB [RT06] (QF_UFIDL refers to the combined theory $EUF \cup IDL$). The suite contains 319 unsatisfiable instances. Each formula was split in half to obtain an artificial interpolation problem (in the same fashion as [CGS08]).

The pivot reordering algorithm Algorithm 18 was realized by means of the Local Transformation Framework and implemented in OpenSMT [BPST10]. Proof manipulation was applied when the proof contained *AB*-mixed predicates, in order to lift them up inside *AB*-maximal subproofs and replace them with their roots.

We ran the experiments on a 32-bit Ubuntu server equipped with Dual-Core 2GHz Opteron 2212 CPU and 4GB of memory. The benchmarks were executed with a timeout of 60 minutes and a memory threshold of 2GB (whatever was reached first): 172 instances, of which 82 proofs contained *AB*-mixed predicates[4], were successfully handled within these limits. We have reported the cost of the transformation and its effect on the proofs; the results are summarized in Table 5.5. We grouped benchmarks together following the original classification used in SMT-LIB and provided average values for each group[5].

[4]Note that in some cases *AB*-mixed predicates were produced during the search, but they did not appear in the proof.

[5]The full experimental data is available at `http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

Table 5.5. The effect of proof transformation on QF_UFIDL benchmarks summarized per group: *#Bench* is the number of benchmarks in a group, *#AB* is the average number of AB-mixed predicates in a proof, *Time%* is the average time overhead induced by transformation, *Nodes%* and *Edges%* represent the average difference in the proof size as a result of transformation.

Group	#Bench	*#AB*	Time%	Nodes%	Edges%
RDS	2	7	84	-16	-19
EufLaArithmetic	2	74	18	187	193
pete	15	20	16	66	68
pete2	52	13	6	73	80
uclid	11	12	29	87	90
Overall	82	16	13	74	79

The results in Table 5.5 demonstrate that our proof transformation technique induces, on average, about 13% overhead with respect to plain solving time. The average increase in size is around 74%, but not all the instances experienced a growth; we observed in fact that in 42 out of 82 benchmarks the transformed proof was smaller than the original one both in the number of nodes and edges. Overall it is important to point out that the creation of new nodes due to the application of the S rules did not entail any exponential blow-up in the size of the proofs during the transformation process.

Another interesting result to report is the fact that only 45% of the proofs contained AB-mixed predicates and, consequently, required transformation. This is another motivation for using off-the-shelf algorithms for SMT solvers and have the proof transformed in a second stage, rather than tweaking (and potentially slowing down) the solver to generate clean proofs upfront.

5.4.4 Pivot Reordering for Propositional Interpolation

This section concludes our discussion on interpolation by moving back from the context of SMT to that of SAT. We complete the analysis begun by the authors of [JM05] and illustrate how, in the case of purely propositional refutations, a transformation technique can be devised to generate interpolants directly in conjunctive or disjunctive normal form.

The table below recalls McMillan's interpolation system Itp_M for propositional logic, introduced in §3.3.4.1. Algorithm 18, PivotReordering2, can be employed to restructure a refutation so that Itp_M generates an interpolant in CNF. It is sufficient

McMillan's interpolation system Itp_M.

Leaf:	$C\,[I]$
	$I = \begin{cases} C\|_{AB} & \text{if } C \in A \\ \top & \text{if } C \in B \end{cases}$
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
	$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \\ I^+ \wedge I^- & \text{if } p \in B \text{ or } p \in AB \end{cases}$

in fact to modify the definition of light and heavy predicates given in §5.4, so that a context is considered unordered whenever $v(t)$ is local to A (*light*) and $v(s)$ is a propositional variable in B or in AB (*heavy*). Effect of the refutation transformation is to push up light variables, so that, along every path from the leaves to the root, light variables appear before heavy variables.

We need to show that this condition is sufficient in order for Itp_M to produce an interpolant in CNF.

Theorem 5.4.1. *Assume a refutation* Π *without unordered contexts.* Itp_M *generates an interpolant in CNF from* Π.

Proof by structural induction. **Base case**. The partial interpolant for a leaf labeled by a clause C is either $\top$ or $C|_{AB}$, so it is in CNF.

Inductive step. Given an inner node n and the associated pivot $p = piv(n)$, assume the partial interpolants I^+ and I^- for $C(n^+) = C_1 \vee p$ and $C(n^-) = C_2 \vee \overline{p}$ are in CNF. We have four possibilities:

- Case 1: I^+ and I^- are both in clausal form; then either n^+, n^- are leaves or they are inner nodes with light pivot variables. p can be either light or heavy: in the first case I is itself a clause, in the second case I is a conjunction of clauses, so it is in CNF.

- Case 2: I^+ is a clause, I^- is a conjunction of at least two clauses; then n^+ can be either a leaf or an inner node with a light pivot, but I^- must be an inner node with a heavy pivot (due to $\wedge$ being the main connective of I^-). Since Π does not have unordered contexts, p must be a heavy variable, thus $I = I^+ \wedge I^-$ is in CNF.

- Case 3: I^+ is a conjunction of at least two clauses, I^- is a clause. Symmetric to Case 2.

- Case 4: Both I^+ and I^- are a conjunction of at least two clauses. As for Case 2 and Case 3.

□

A similar argumentation holds for the generation of interpolants in disjunctive normal form. Let us consider the interpolation system dual to McMillan's, $Itp_{M'}$:

McMillan′'s interpolation system $Itp_{M'}$.

Leaf:	$C\,[I]$
$I = \begin{cases} \bot & \text{if } C \in A \\ \neg C\|_{AB} & \text{if } C \in B \end{cases}$	
Inner node:	$\dfrac{C^+ \vee p\,[I^+] \qquad C^- \vee \overline{p}\,[I^-]}{C^+ \vee C^-\,[I]}$
$I = \begin{cases} I^+ \vee I^- & \text{if } p \in A \text{ or } p \in AB \\ I^+ \wedge I^- & \text{if } p \in B \end{cases}$	

Algorithm 18 can be employed to transform the refutation; in this case a context is unordered if $v(t)$ is a variable local to B (*light*) and $v(s)$ is a variable local to A or common (*heavy*). The effect of pushing up light variables is that, during the construction of the interpolant, the connective $\wedge$ will be introduced before $\vee$ along each path, so that the resulting interpolant will be in disjunctive normal form (note that the partial interpolant of a leaf is already in DNF, being a conjunction of literals).

We can thus state the following theorem:

Theorem 5.4.2. *Assume a refutation* Π *without unordered contexts.* $Itp_{M'}$ *generates an interpolant in DNF from* Π.

As already pointed out in [JM05], the price to pay for a complete transformation might be an exponential increase of the refutation size, due to the node duplications necessary to apply rules $S1, S2, R2$ to contexts where C_4 has multiple children (see Figure 5.5). A feasible compromise consists in performing a partial CNFization or DNFization by limiting the application of such rules to when C_4 has a single child; in this case, the refutation growth depends only on the application of rule $S1$, and the increase is maintained linear.

5.5 Heuristics for the Proof Transformation Algorithms

In this section we discuss some of the heuristics implemented in OpenSMT and PeRIPLO to guide the application of the Local Transformation Framework rules

and the reconstruction of proofs, with reference to compression (§5.3) and pivot reordering for interpolation (§5.4).

Some of the algorithms presented so far (Algorithms 6,7,12) need to handle the presence of resolution steps which are not valid anymore since the pivot is missing from both antecedents; in that case, the resolvent node n must be replaced by either parent. A heuristics which has been proven useful for determining the replacing parent is the following. If one of the parents (let us say n^+) has only n as child, then n is replaced by n^-; since n^+ loses its only child, then (part of) the subproof rooted in n^+ gets detached from the global proof, yielding a simplification of the proof itself. If both parents have more than one child, then the parent labeled by the smaller clause is the one that replaces n, aiming at increasing the amount of simplifications performed while moving down to the global root.

As far as the heuristics for the application of rewriting rules are concerned, the ApplyRule method adheres to some general lines. Whenever a choice is possible between a left and a right context, a precedence order is respected: ($X > Y$ means: the application of X is preferred over that of Y):

$$R3 > \{R2', R1\} > R2 > S1' > S2 > S1$$

The compression rules R have always priority over the shuffling rules S, $R3$ being the favorite, followed by $R2'$ and $R1$. Among the S rules, $S1'$ is able to perform a local simplification, which makes it preferred to $S2$ and especially to $S1$, which increases the size of the proof; between equal S rules, the one which does not involve a node duplication (see Figure 5.5) is chosen.

Additional constraints depend on the actual goal of the transformation. If the aim is pivot reordering, the constraints are as illustrated in Algorithm 18, with ties broken according to the general lines given above. If the aim is compression, then $S1$ is never applied, since it increases the size of the proof and it is not apparent at the time of its application whether it would bring benefits in a second moment, neither are applied $R2, S1', S2$ if they involve a duplication. A strategy which proved successful in the application of S rules is to push up nodes with multiple resolvents whenever possible, with the aim of improving the effect of RecyclePivots and RecyclePivotsWithIntersection; interestingly, this technique shows as a side effect the disclosure of redundancies which can effectively be taken care of by StructuralHashing.

These heuristics have been discovered through experimentation and have been adopted due to their practical usefulness for compression, in a setting where the large size of proofs allows only a few traversals (and thus a limited application of rules) by means of ReduceAndExpose, and where the creation of new nodes should be avoided; it is thus unlikely that, arbitrarily increasing the number of traversals,

they would expose and remove all pivots redundancies. A more thorough, although practically infeasible, approach could rely on keeping track of all contexts and associated rules in a proof Π. Since the S rules are revertible, an equivalence relation $=_S$ could be defined among proofs so that $\Pi =_S \Pi'$ if Π' can be obtained from Π (and vice versa) by means of a sequence of applications of S rules. A backtracking-based algorithm could be employed to systematically visit equivalence classes of proofs, and to move from an equivalence class to another thanks to the application of an R rule.

5.6 Related Work

Various proof manipulation techniques have been developed in the last years, the main goal being compression.

[Amj07] proposes an algorithm based on heuristically reordering the resolution steps that form a proof, trying to identify a subset of the resolution steps that is still sufficient to derive the empty clause. The approach relies on using an additional graph-like data structure to keep track of how literals of opposite polarity are propagated from the leaves through the proof and then resolved upon.

[Sin07] explicitly assumes a CDCL context, where a resolution-based SAT solver generates a sequence of derivations called *proof chains*, combined in a second moment to create the overall proof. The author presents an algorithm that works at the level of proof chains, aiming at identifying and merging shared substructures to generate a smaller proof.

[Amj08] further develops this approach, adopting a representation of resolution proofs that allows the use of efficient algorithms and data structures for substring matching; this feature is exploited to perform memoization of proofs by detecting and reusing common subproofs.

[Cot10] introduces two compression methods. The first one is based on a form of structural hashing, where each inner node in a proof graph is associated with its pair of antecedents in a hash map. The compression algorithm traverses the sequence of proof chains while updating the hash map, and adds a resolution step to the overall proof only if it does not already exist. The second one consists of a rewriting procedure that, given in input a proof and a heuristically chosen propositional variable p, transforms the proof so that the last resolution step is on p; this might result in a smaller proof.

[BIFH+08] presents a technique that exploits learned unit clauses to rewrite subproofs that were derived before learning them. The authors also propose a compression algorithm (*RecyclePivots*) that searches for resolution steps on the same pivot

along paths from leaves to the root in a proof. If a pivot is resolved upon more than once on a path (which implies that the pivot variable is introduced and then removed multiple times), the resolution step closest to the root is kept, while the others are simplified away. The algorithm is effective on resolution proof trees, but can be applied only in a limited form to resolution proof DAGs, due to the possible presence of multiple paths from a node to the root.

This restriction is relaxed in the work of [FMP11], that extends the algorithm of [BIFH$^+$08] into *RecyclePivotsWithIntersection* to keep track, for each node, of the literals which get resolved upon along *all* paths from the node to the root. [FMP11] also presents an algorithm that traverses a proof, collecting unit clauses and reinserting them at the level of the global root, thus removing redundancies due to multiple resolution steps on the same unit clauses; this technique is later generalized in [BP13] to lowering subproofs rooted in non-unit clauses.

[Gup12] builds upon [BIFH$^+$08] in developing three variants of *RecyclePivots* tailored to resolution proof DAGs. The first one is based on the observation that the set of literals which get resolved in a proof upon along all paths from the node to the root must be a superset of the clause associated to the node, if the root corresponds to the empty clause. The second and third ones actually correspond respectively to *RecyclePivotsWithIntersection* and to a parametric version of it where the computation of the set of literals is limited to nodes with up to a certain amount of children.

Our set of compression techniques has been illustrated with reference to [Sin07], [BIFH$^+$08] and [FMP11] in §5.3.

Besides compression, a second area of application of proof manipulation has been interpolation, both in the propositional and in the first order settings.

[DKPW08] introduces a global transformation framework for interpolation to reorder the resolution steps in a proof with respect to a given partial order among pivots; compression is shown to be a side effect for some benchmarks. Compared to [DKPW08], our approach works locally, and leaves more freedom in choosing the strategies for rule applications. Also our target is not directly computing interpolants, but rather rewriting the proof in such a way that existing techniques can be applied.

The same authors focus in [DKPW10] on the concept of strength of an interpolant. They present an analysis of existing propositional interpolation algorithms, together with a method to combine them in order to obtain weaker or stronger interpolants from a same proof of unsatisfiability. They also address the use and the limitations of the local transformation rules of [JM05]. The rewriting rules corresponding to $S1$ and $S2$ in the Local Transformation Framework (§5.2) were first introduced in [JM05] and further examined in [DKPW10] as a way to modify a proof to obtain stronger or weaker interpolants, once fixed the interpolation algorithm; we

devised the remaining rules after an exhaustive analysis of the possible proof contexts. [JM05] also discusses the application of $S1$ and $S2$ to generate interpolants in conjunctive normal form; however, not all the contexts are taken into account, and, as pointed out in [DKPW10], the contexts for $S1$ and $S2$ are not correctly identified.

Note that $S1$ and $S2$ have also a counterpart in Gentzen's sequent calculus system LK [Gen35]: $S1$ corresponds to swapping applications of the structural cut and contraction rules, while $S2$ is one of the rank reduction rules.

Interpolation for first order theories in presence of AB-mixed predicates is addressed in [CGS08], only for the case of DTC, by tweaking the decision heuristics of the solver, in such a way that it guarantees that the produced proof can be handled with known methods. In particular the authors define a notion of ie-*local proofs*, and they show how to compute interpolants for this class of proofs, and how to adapt an SMT solver to produce only ie-local proofs. [GKT09] the relaxes the constraint on generating ie-local proofs by introducing the notion of *almost-colorable* proofs. We argue that our technique is simpler and more flexible, as different strategies can be derived with different applications of our local transformation rules. Our method is also more general, since it applies not only to theory combination but to any approach that requires the addition of AB-mixed predicates (see §5.4.2).

More recently, a tailored interpolation algorithm has been proposed in [CHN13] for the combined theory of linear arithmetic and uninterpreted functions; it has the notable feature of allowing the presence of mixed predicates, thus making proof manipulation not necessary anymore.

5.7 Summary and Future Developments

In this chapter we have presented a proof transformation framework based on a set of local rewriting rules and shown how it can be applied to the tasks of proof compression and pivot reordering.

As for compression, we discussed how rules that effectively simplify the proof can be interleaved with rules that locally perturbate the topology, in order to create new opportunities for compression. We identified two kinds of redundancies in proofs, related to the notions of regularity and compactness, and presented and compared a number of algorithms to address them, moving from existing techniques in the literature. Individual algorithms, as well as their combinations, were implemented and tested over a collection of benchmarks both from SAT and SMT libraries, showing remarkable levels of compression in the proof size.

As for pivot reordering, we described how to employ the rewriting rules to isolate and remove AB-mixed predicates, in such a way that standard procedures for

interpolation in SMT can be applied. The approach enables the use of off-the-shelf techniques for SMT solvers that are likely to introduce AB-mixed predicates, such as Ackermann's expansion, lemma on demand, splitting on demand and DTC. We showed by means of experiments that our rules can effectively transform the proofs without generating any exponential growth in their size. Finally, we explored a form of interaction between LISs and proof manipulation by providing algorithms to reorder resolution steps in a propositional proof to guarantee the generation of interpolants in conjunctive or disjunctive normal form.

Future work on proof compression can concern, on the theoretical side, the identification of new types of redundancies and the development of algorithms aimed at removing them; on the practical side, their implementation in PeRIPLO and experimentation on families of benchmarks of different origin.

An interesting topic to investigate is the relationship and the mutual dependence between proof manipulation and interpolation systems. We have shown for example how to perform ad-hoc transformations to allow Itp_M and $Itp_{M'}$ to respectively yield interpolants in CNF and DNF. The rules $S1$ and $S2$ can be employed to strengthen or weaken an interpolant, given a LIS Itp_L and a refutation Π; it might be thus possible to formally extend the lattice of labeled interpolation systems by examining the family of interpolants that can be generated from a Itp_L by systematically using $S1, S2$ on Π.

Another aspect to consider is the applicability of proof manipulation to collectives. In Chapter 4 we discussed how to generate a family of interpolants satisfying a certain property from the same proof; it remains to understand whether and how the results presented hold if the proof undergoes changes, even just local ones due to a restricted application of $S1, S2$.

Chapter 6

Conclusions

Symbolic model checking has been considerably enhanced with the introduction of Craig interpolation as a means for overapproximation. Interpolants can be efficiently computed from proofs of unsatisfiability; multiple interpolants can be generated from the same proof, and proof themselves can be transformed, thus allowing additional interpolants.

Interpolants are not unique, and it is a fundamental problem to determine what features determine their quality, and how these features are connected with the effectiveness in the verification.

This thesis addresses the problem by characterizing aspects that make interpolants good, and by developing new techniques that guide the generation of interpolants in order to improve the performance of model checking approaches. The main research contributions are summarized in the following.

A DPLL-Stochastic Algorithm for Satisfiability. In §2.2 we addressed the challenge of developing new, effective, DPLL-stochastic algorithms by presenting an approach to the satisfiability problem based on the stochastic technique known as cross-entropy method. In our framework the propositional model is extended to a multi-valued setting, a probability space is induced over the enlarged search space and a performance function is defined that correlates with the likelihood of the input formula to be satisfiable. We implemented our variant of the cross-entropy method in a C++ tool named CROiSSANT. CROiSSANT is employed as a preprocessor to a DPLL-based solver, to identify the areas of the space of all possible assignments that are more likely to contain a satisfying assignment; this information is in turn given to the solver to suggest variables assignments during the search. We conducted experiments on different sets of benchmarks, using CROiSSANT in combination with the state-of-the-art solver MiniSAT 2.2.0; the results show an improvement both in

running times and number of solved instances, providing evidence that our framework is a sound basis for further research on cross-entropy based SAT solvers.

Systematic Generation of Interpolants of Different Strength and Structure. In §3.2 we discussed and justified the adoption of logical strength and structure as parameters with respect to which the quality of interpolants can be evaluated. Based on that, we contributed to a theoretical formalization of a generic interpolation approach, introducing a new parametric interpolation framework for arbitrary theories and inference systems, which is able to compute interpolants of different structure and strength, with or without quantifiers, from the same proof. We described the framework in relation with well-known interpolation algorithms, that respectively address local proofs in first order logic and the propositional hyper-resolution system, and showed that they can be regarded as instantiations of our method.

Impact of Interpolants Size and Strength in Model Checking. In §3.6 we addressed the key problem of generating effective interpolants in verification, by assessing the impact of size and logical strength in SAT-based bounded model checking. We took into account two BMC applications which use interpolation to generate function summaries: (i) verification of a C program incrementally with respect to different properties, and (ii) incremental verification of different versions of a C program with respect to a fixed set of properties. We integrated the PeRIPLO framework with the model checkers FunFrog and eVolCheck, which respectively implement (i) and (ii), to drive interpolation by manipulating the proofs from which the interpolants are computed and by generating interpolants of different strength. We provided solid experimental evidence that compact interpolants improve the verification performance in the two applications. We also carried out a first systematic evaluation of the impact of strength in a specific verification domain, showing that different applications can benefit from interpolants of different strength: specifically, stronger and weaker interpolants are respectively desirable in (i) and (ii).

Formal Analysis of Interpolation Properties. In Chapter 4 we discussed how many verification techniques require the generation of collections of interdependent interpolants, which need to satisfy particular properties in order to make verification possible. We identified and uniformly presented the most common among these properties, systematically analyzing the relationships among them and showing that they form a hierarchy. In doing so, we extended the traditional setting of a single interpolation system to families of systems, thus giving the flexibility to choose different systems for computing different interpolants in a collection. The generality of

these results derives from their being independent of the concrete systems adopted to generate the interpolants.

Interpolation Properties and Interpolant Strength. The propositional labeled interpolation systems are a suitable instrument for our research on the strength of interpolants, since they consist of a parametric framework which allows to systematically generate interpolants of different strength from the same proof. For this reason, in §4.3 we addressed the interpolation properties and the relationships among them in the context of LISs, formally proving both sufficient and necessary conditions for a family of LISs and for a single LIS (including the well-known McMillan and Pudlák's interpolation systems) to enjoy each property. These results allow for the systematic study of how interpolants strength affects state-of-the-art model checking techniques, while preserving their soundness.

Extension of the Labeled Interpolation Systems to SMT. In order to extend the applicability of our results from SAT- to SMT-based verification, we investigated in §3.4 and in §4.4 the interpolation properties in the ambit of satisfiability modulo theories; we presented the theory labeled interpolation systems, an extension of the labeled interpolation systems to first order theories that generalizes standard approaches to SMT interpolation, and analyzed how the constraints for some of the properties change in presence of a first order theory.

Proof Compression Framework. In §5.3 we acknowledged the impact the size of resolution proofs can have on the techniques that rely on them, for example in interpolation-based verification, and the convenience of compressing them. We gave our contribution to the area of proof compression by developing an approach based on local transformation rules; it is a post-processing method, with the benefit of being independent from the way the resolution proofs are produced. We discussed how rules that directly perform simplifications can be interleaved with rules that locally perturbate the topology, in order to create new opportunities for compression. We identified two kinds of redundancies, related to the notions of regularity and compactness, and presented and compared a number of algorithms to address them, moving from existing techniques in the literature. Individual algorithms, as well as their combinations, were implemented in the tools OpenSMT and PeRIPLO and tested over a collection of benchmarks both from the SAT and SMT world, showing remarkable levels of compression in the proofs size.

Proof Manipulation to Enable Interpolation in SMT. Standard approaches to SMT interpolation suffer from a significant limitation, since they need to assume the absence of mixed predicates in the resolution proofs, while there is in fact a number of techniques used in SMT solvers (Ackermann's expansion, lemma on demand, splitting on demand, delayed theory combination) that might require the addition of mixed predicates to the input formula before or during solving time. The solution we presented in §5.4 is to rewrite the proof of unsatisfiability by means of local rules, in such a way that mixed predicates are isolated and then removed from the proof. Our approach makes interpolation flexible and modular: first, off-the-shelf SMT solvers are run on the input formula; then, the proof is transformed to get rid of the mixed predicates; finally, standard interpolation algorithm for SMT are applied. We demonstrated that our method is sound and showed by means of experiments, carried out within the OpenSMT framework, that it is also practically efficient.

Tools for Proof Manipulation And Interpolation. A first contribution consists in the realization within the SMT solver OpenSMT, developed by our Formal Verification Group at USI, of a framework for proof compression and interpolation, which implements the labeled interpolation systems and a partial extension to satisfiability modulo theories.

The main tool contribution is the creation of PeRIPLO, Proof tRansformer and Interpolator for Propositional LOgic, built on the state-of-the-art SAT solver MiniSAT 2.2.0. PeRIPLO contains a more efficient implementation of the labeled interpolation systems and of the compression techniques of OpenSMT, as well as new algorithms for proof compression and manipulation, both from the literature and novel. PeRIPLO is also able to produce individual interpolants and collections of interpolants, with reference to various interpolation properties and in accordance with the constraints imposed by the properties on the labeled interpolation systems.

PeRIPLO is illustrated in §3.6.1 and in Chapter 5, and it is available at:

`http://verify.inf.usi.ch/content/periplo`

The tools executables, as well as the full data related to the experiments reported in this thesis, can be found at:

`http://verify.inf.usi.ch/sites/default/files/Rollini-phddissertationmaterial.tar.gz`

Bibliography

[ABE00] P.A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *TACAS*, pages 411–425, 2000.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*, volume 12 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1954.

[ACKS02] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded Model Checking for Timed Systems. In *FORTE*, pages 243–259. Springer, 2002.

[AGC12] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *VMCAI*, pages 39–55. Springer, 2012.

[AKRR05] G. Alon, D.P. Kroese, T. Raviv, and R.Y. Rubinstein. Application of the Cross-Entropy Method to the Buffer Allocation Problem in a Simulation-Based Environment. *Annals of Operations Research*, 134(1):137–151, 2005.

[ALMS09a] G. Audemard, J.M. Lagniez, B. Mazure, and L. Sais. Integrating Conflict Driven Clause Learning to Local Search. In *LSCS*, pages 55–68, 2009.

[ALMS09b] G. Audemard, J.M. Lagniez, B. Mazure, and L. Sais. Learning in Local Search. In *ICTAI*, pages 417–424, 2009.

[AM03] N. Amla and K.L. McMillan. Automatic Abstraction Without Counterexamples. In *TACAS*, pages 2–17. Springer, 2003.

[AM05] E. Amir and S. McIlraith. Partition-Based Logical Reasoning for First-Order and Propositional Theories. *Artificial Intelligence*, 162(1-2):49–88, 2005.

[AM13] A. Albarghouthi and K. L. McMillan. Beautiful Interpolants. In *CAV*, pages 313–329. Springer, 2013.

[Amj07] H. Amjad. Compressing Propositional Refutations. *Electronic Notes in Theoretical Computer Science*, 185:3–15, 2007.

[Amj08] H. Amjad. Data Compression for Proof Replay. *Journal of Automated Reasoning*, 41(3-4):193–218, 2008.

[AMP09] A. Armando, J. Mantovani, and L. Platania. Bounded Model Checking of Software Using SMT Solvers instead of SAT Solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[AS09] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, pages 399–404, 2009.

[BB08] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. SMT/BPR, pages 6–11, 2008.

[BBC+05a] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35(1-3):265–293, 2005.

[BBC+05b] M. Bozzano, R. Bruttomesso, A. Cimatti, T.A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *CAV*, pages 335–349. Springer, 2005.

[BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *DAC*, pages 317–320, 1999.

[BCC+03] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207. Springer, 1999.

[BCF+07] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *CAV*, pages 547–560. Springer, 2007.

[BCF+08] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, pages 299–303. Springer, 2008.

[BCF+09] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):63–99, 2009.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BFG+05] C.W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *CAV*, pages 291–295. Springer, 2005.

[BHG09] A. Balint, M. Henn, and O. Gableske. A Novel Approach to Combine a SLS- and a DPLL-Solver for the Satisfiability Problem. In *SAT*, pages 284–297. Springer, 2009.

[BHMW09] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie13] A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *SAT Competition*, page 51, 2013.

[BIFH+08] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-Time Reductions of Resolution Proofs. In *HVC*, pages 114–128. Springer, 2008.

[Bjø10] Nikolaj Bjørner. Linear Quantifier Elimination as an Abstract Decision Procedure. In *IJCAR*, pages 316–330, 2010.

[BKK11] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, pages 191–207, 2011.

[BKRW11] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. *Journal of Automated Reasoning*, 47(4):341–367, 2011.

[BKS04] P. Beame, H.A. Kautz, and A. Sabharwal. Towards Understanding and Harnessing the Potential of Clause Learning. *Journal of Artificial Intelligence Research*, 22(1):319–351, 2004.

[BLM01] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In *CAV*, pages 454–464. Springer, 2001.

[BNO+08] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. A Write-Based Solver for SAT Modulo the Theory of Arrays. In *FMCAD*, pages 101–108, 2008.

[BNOT06] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In *LPAR*, pages 512–526. Springer, 2006.

[BP13] J. Boudou and B.W. Paleo. Compression of Propositional Resolution Proofs by Lowering Subproofs. In *TABLEAUX*, 2013.

[BPST10] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *TACAS*, pages 150–153. Springer, 2010.

[BR13] J.D. Backes and M.D. Riedel. Using Cubes of Non-State Variables with Property Directed Reachability. In *DATE*, pages 807–810, 2013.

[Bra11] A.R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87. Springer, 2011.

[Bra12] A.R. Bradley. Understanding IC3. In *SAT*, pages 1–14. Springer, 2012.

[BRST10] R. Bruttomesso, S.F. Rollini, N. Sharygina, and A. Tsitovich. Flexible Interpolation with Local Proof Transformations. In *ICCAD*, pages 770–777, 2010.

[Bru03] R. Bruni. Approximating Minimal Unsatisfiable Subformulae by Means of Adaptive Core Search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[BS96] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[CFGN07] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-Entropy Based Testing. In *FMCAD*, pages 101–108, 2007.

[CFGN09] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-Entropy-Based Replay of Concurrent Programs. In *FASE*, pages 201–215. Springer, 2009.

[CG12] A. Cimatti and A. Griggio. Software Model Checking via IC3. In *CAV*, pages 277–293, 2012.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169. Springer, 2000.

[CGS07] A. Cimatti, A. Griggio, and R. Sebastiani. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In *SAT*, pages 334–339. Springer, 2007.

[CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *TACAS*, pages 397–412. Springer, 2008.

[CGS09] A. Cimatti, A. Griggio, and R. Sebastiani. Interpolant Generation for UTVPI. In *CADE*, pages 167–182. Springer, 2009.

[CGS10] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.

[CHN13] J. Christ, J. Hoenicke, and A. Nutz. Proof Tree Preserving Interpolation. In *TACAS*, pages 124–138. Springer, 2013.

[CIM12] H. Chockler, A. Ivrii, and A. Matsliah. Computing Interpolants without Proofs. In *HVC*, 2012.

[CIM+13] H. Chockler, A. Ivrii, A. Matsliah, S.F. Rollini, and N. Sharygina. Using Cross-Entropy for Satisfiability. In *SAC*, pages 1196–1203, 2013.

[CJK07] A. Costa, O.D. Jones, and D. Kroese. Convergence Properties of the Cross-Entropy Method for Discrete Optimization. *Operations Research Letters*, 35(5):573–580, 2007.

[CLO07] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2007.

[CLV13] G. Cabodi, C. Lolacono, and D. Vendraminetto. Optimization Techniques for Craig Interpolant Compaction in Unbounded Model Checking. In *DATE*, pages 1417–1422, 2013.

[CMU] CMU BMC benchmarks. `http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html`.

[Col75] G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Automata Theory and Formal Languages*, pages 134–183. Springer, 1975.

[Coo71] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *STOC*, pages 151–158. ACM Press, 1971.

[Coo72] D.C. Cooper. Theorem Proving in Arithmetic without Multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

[Cot10] S. Cotton. Two Techniques for Minimizing Resolution Proofs. In *SAT*, pages 306–312. Springer, 2010.

[Cra57a] W. Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[Cra57b] W. Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[Cra93] J.M. Crawford. Solving Satisfiability Problems Using a Combination of Systematic and Local Search. In *2nd DIMACS Challenge*, pages 1–7, 1993.

[CVC] CVC4. `http://cvc4.cs.nyu.edu/web/`.

[DGS93] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical Support for Modularisation. In *Logical Environments*, pages 83–130, 1993.

[DHN06] N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In *SAT*, pages 36–41. Springer, 2006.

[DKPW08] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Restructuring Resolution Refutations for Interpolation. Technical report, ETH, Zürich, 2008.

[DKPW10] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant Strength. In *VMCAI*, pages 129–145. Springer, 2010.

[DLL62] M. Davis, G. Logemann, and D.W. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[dMB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer, 2008.

[dMB09] L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *FMCAD*, pages 45–52, 2009.

[dMR02] L. de Moura and H. Rue. Lemmas on Demand for Satisfiability Solvers. In *SAT*, pages 244–251, 2002.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[Dub02] U. Dubin. The Cross-Entropy Method for Combinatorial Optimization with Applications. Master Thesis, The Technion, Haifa, 2002.

[EKS08] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. *Journal on Satisfiability*, 5:27–56, 2008.

[ES04] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518. Springer, 2004.

[FF04] B. Ferris and J. Froehlich. WalkSAT as an Informed Heuristic to DPLL in SAT Solving. Technical report, University of Washington, Seattle, 2004.

[FGG+09] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground Interpolation for the Theory of Equality. In *TACAS*, pages 413–427. Springer, 2009.

[FH07] L. Fang and M.S. Hsiao. A New Hybrid Solution to Boost SAT Solver Performance. In *DATE*, pages 1307–1313, 2007.

[FMM+06] P. Fontaine, J.Y Marion, S. Merz, L.P. Nieto, and A. Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In *TACAS*, pages 167–181. Springer, 2006.

[FMP11] P. Fontaine, S. Merz, and B.W. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In *CADE*, pages 237–251. Springer, 2011.

[FR75] J. Ferrante and C. Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM Journal on Computing*, 4(1):69–76, 1975.

[FR04] H. Fang and W. Ruml. Complete Local Search for Propositional Satisfiability. In *AAAI*, pages 161–166, 2004.

[FSS13] G. Fedyukovich, O. Sery, and N. Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *TACAS*, pages 292–307. Springer, 2013.

[GD07] V. Ganesh and D.L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, pages 519–531. Springer, 2007.

[Gen35] G. Gentzen. Untersuchungen über das Logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[GGA04] M.K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-Based Unbounded Symbolic Model Checking Using Circuit Cofactoring. In *ICCAD*, pages 510–517, 2004.

[GKF08] A. Goel, S. Krstić, and A. Fuchs. Deciding Array Formulas with Frugal Axiom Instantiation. In *SMT/BPR*, pages 12–17, 2008.

[GKSS08] C.P. Gomes, H.A. Kautz, A. Sabharwal, and B. Selman. Satisfiability Solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[GKT09] A. Goel, S. Krstić, and C. Tinelli. Ground Interpolation for Combined Theories. In *CADE*, pages 183–198. Springer, 2009.

[GLS12] A. Griggio, T.T.H. Le, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. *Logical Methods in Computer Science*, 8(3):2–31, 2012.

[GLST05] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-Guided Underapproximation-Widening for Multi-Process Systems. In *POPL*, pages 122–131, 2005.

[GM12] E. Goldberg and P. Manolios. Quantifier Elimination by Dependency Sequents. In *FMCAD*, pages 34–44, 2012.

[GMP07] E. Grégoire, B. Mazure, and C. Piette. Local-Search Extraction of MUSes. *Constraints*, 12(3):325–344, 2007.

[GNZ05] S. Ghilardi, E. Nicolini, and D. Zucchelli. A Comprehensive Framework for Combined Decision Procedures. In *FroCoS*, pages 1–30. Springer, 2005.

[Gri11] A. Griggio. Effective Word-Level Interpolation for Software Verification. In *FMCAD*, pages 28–36, 2011.

[GRS13] A. Gurfinkel, S.F. Rollini, and N. Sharygina. Interpolation Properties and SAT-Based Model Checking. In *ATVA*, pages 255–271, 2013.

[GS97] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83. Springer, 1997.

[GSMT98] C.P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. In *AIPS*, pages 208–213, 1998.

[Gup12] A. Gupta. Improved Single Pass Algorithms for Resolution Proof Reduction. In *ATVA*, pages 107–121. Springer, 2012.

[GW93] I.P. Gent and T. Walsh. Towards an Understanding of Hill-Climbing Procedures for SAT. In *AAAI*, pages 28–33, 1993.

[HB12] K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *SAT*, pages 157–171. Springer, 2012.

[HBS13] Z. Hassan, A.R. Bradley, and F. Somenzi. Better Generalization in IC3. In *FMCAD*, pages 157–164, 2013.

[HHP10] M. Heizmann, J. Hoenicke, and A. Podelski. Nested Interpolants. In *POPL*, pages 471–482, 2010.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from Proofs. In *POPL*, pages 232–244, 2004.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.

[HK05] E.A. Hirsch and A. Kojevnikov. UnitWalk: A New SAT Solver that Uses Local Search Guided by Unit Clause Elimination. *Annals of Mathematics and Artificial Intelligence*, 43:91–111, 2005.

[HKV12] K. Hoder, L. Kovács, and A. Voronkov. Playing in the Grey Area of Proofs. In *POPL*, pages 259–272, 2012.

[HLDV06] D. Habet, C.M. Li, L. Devendeville, and M. Vasquez. A Hybrid Approach for SAT. In *CP*, pages 172–184. Springer, 2006.

[Hoo99] H.H. Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *AAAI/IAAI*, pages 661–666, 1999.

[HS05] H.H. Hoos and T. Stützle. *Stochastic Local Search. Foundations and Applications*. Morgan Kaufmann, 2005.

[Hua95] G. Huang. Constructing Craig Interpolation Formulas. In *COCOON*, pages 181–190. Springer, 1995.

[Hua05] J. Huang. MUP: a Minimal Unsatisfiability Prover. In *ASP-DAC*, pages 432–437, 2005.

[JC11] A.K. John and S. Chakraborty. A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations. In *CAV*, pages 486–503. Springer, 2011.

[JC13] A.K. John and S. Chakraborty. Extending Quantifier Elimination to Linear Inequalities on Bit-Vectors. In *TACAS*, pages 78–92. Springer, 2013.

[JCG08] H. Jain, E.M. Clarke, and O. Grumberg. Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In *CAV*, pages 254–267. Springer, 2008.

[JD08] T.A. Junttila and J. Dubrovin. Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking. In *LPAR*, pages 290–304, 2008.

[JM05] R. Jhala and K.L. McMillan. Interpolant-Based Transition Relation Approximation. In *CAV*, pages 39–51. Springer, 2005.

[JM06] R. Jhala and K.L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, pages 459–473. Springer, 2006.

[JO02] N. Jussien and L. Olivier. Local Search with Constraint Propagation and Conflict-Based Heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[JS97] R.J. Bayardo Jr and R. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *AAAI/IAAI*, pages 203–208, 1997.

[JS05] H. Jin and F. Somenzi. Prime Clauses for Fast Enumeration of Satisfying Assignments to Boolean Circuits. In *DAC*, pages 750–753, 2005.

[JW90] R.G. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[KK02] J.M. Keith and D.P. Kroese. Rare Event Simulation and Combinatorial Optimization Using Cross Entropy: Sequence Alignment by Rare Event Simulation. In *WSC*, pages 320–327. ACM, 2002.

[KL51] S. Kullback and R.A. Leibler. On Information and Sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KLR10] D. Kroening, J. Leroux, and P. Rümmer. Interpolating Quantifier-Free Presburger Arithmetic. In *LPAR*, pages 489–503. Springer, 2010.

[KMZ06] D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for Data Structures. In *SIGSOFT FSE*, pages 105–116, 2006.

[Kra97] J. Krajíček. Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, 1997.

[KRS13] L. Kovács, S.F. Rollini, and N. Sharygina. A Parametric Interpolation Framework for First-Order Theories. In *MICAI*, pages 24–40, 2013.

[KS08] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Theoretical Computer Science. Springer, 2008.

[KTB11] D.P. Kroese, T. Taimre, and Z.I. Botev. *Handbook of Monte Carlo Methods*. Probability and Statistics. Wiley, 2011.

[KV09] L. Kovács and A. Voronkov. Interpolation and Symbol Elimination. In *CADE*, pages 199–213. Springer, 2009.

[KW07] D. Kroening and G. Weissenbacher. Lifting Propositional Interpolants to the Word-Level. In *FMCAD*, pages 85–89, 2007.

[KW09] D. Kroening and G. Weissenbacher. An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions. In *HVC*, pages 150–168. Springer, 2009.

[LMS04] I. Lynce and J. Marques-Silva. On Computing Minimum Unsatisfiable Cores. In *SAT*, pages 305–310, 2004.

[LMS08] F. Letombe and J. Marques-Silva. Improvements to Hybrid Incremental SAT Algorithms. In *SAT*, pages 168–181. Springer, 2008.

[LNO06] S.K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, pages 424–437, 2006.

[LT08] C. Lynch and Y. Tang. Interpolants for Linear Arithmetic in SMT. In *ATVA*, pages 156–170. Springer, 2008.

[LW93] R. Loos and V. Weispfenning. Applying Linear Quantifier Elimination. *The Computer Journal*, 36(5):450–462, 1993.

[Mar02] L. Margolin. Cross-Entropy Method for Combinatorial Optimization. Master Thesis, The Technion, Haifa, 2002.

[Mar05] L. Margolin. On the Convergence of the Cross-Entropy Method. *Annals of Operations Research*, 134(1):201–214, 2005.

[McM92] K.L. McMillan. *Symbolic Model Checking. An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1992.

[McM02] K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *CAV*, pages 250–264, 2002.

[McM03] K.L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13. Springer, 2003.

[McM04a] K.L. McMillan. An Interpolating Theorem Prover. In *TACAS*, pages 16–30, 2004.

[McM04b] K.L. McMillan. Applications of Craig Interpolation to Model Checking. In *CSL*, pages 22–23. Springer, 2004.

[McM06] K.L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, pages 123–136. Springer, 2006.

[McM11] K.L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, pages 19–27, 2011.

[MLA+05] M. Mneimneh, I. Lynce, Z. Andraus, J. Marques-Silva, and K. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas . In *SAT*, pages 467–474, 2005.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[Mon10] D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, pages 585–599. Springer, 2010.

[MR13] K.L. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses Using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[MS99] J. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *EPIA*, pages 62–74. Springer, 1999.

[MSG98] B. Mazure, L. Sais, and E. Grégoire. Boosting Complete Techniques Thanks to Local Search Methods. *Annals of Mathematics and Artificial Intelligence*, 22(3-4):319–331, 1998.

[MSK97] D.A. McAllester, B. Selman, and H.A. Kautz. Evidence for Invariants in Local Search. In *AAAI/IAAI*, pages 321–326, 1997.

[MSS96] J. Marques-Silva and K.A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, 1996.

[MZ06] P. Manolios and Y. Zhang. Implementing Survey Propagation on Graphics Processing Units. In *SAT*, pages 311–324. Springer, 2006.

[Nec97] G.C. Necula. Proof-Carrying Code. In *POPL*, pages 106–119, 1997.

[Nip08] T. Nipkow. Linear Quantifier Elimination. In *IJCAR*, pages 18–33, 2008.

[NO79] G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[NO80] G. Nelson and D.C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[OMA+04] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, and I.L. Markov. AMUSE: A Minimally-Unsatisfiable Subformula Extractor. In *DAC*, pages 518–523, 2004.

[Opp80] D.C. Oppen. Complexity, Convexity and Combinations of Theories. *Theoretical Computer Science*, 12:291–302, 1980.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT*, pages 294–299, 2007.

[PICW04] G. Parthasarathy, M.K. Iyer, K.T. Cheng, and L.C. Wang. An Efficient Finite-Domain Constraint Solver for Circuits. In *DAC*, pages 212–217, 2004.

[Pud97] P. Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[Pug91] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*, pages 4–13, 1991.

[QS82] J. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, pages 337–351, 1982.

[RAF+13] S.F. Rollini, L. Alt, G. Fedyukovich, A. Hyvärinen, and N. Sharygina. PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification. In *LPAR*, pages 683–693, 2013.

[RBS10] S.F. Rollini, R. Bruttomesso, and N. Sharygina. An Efficient and Flexible Approach to Resolution Proof Reduction. In *HVC*, pages 182–196, 2010.

[RBST] S.F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich. Resolution Proof Transformation for Compression and Interpolation. In *http://arxiv.org/abs/1307.2028*.

[RD03] S. Ranise and D. Déharbe. Applying Light-Weight Theorem Proving to Debugging and Verifying Pointer Programs. *ENTCS*, 86(1):105–119, 2003.

[RK04] R.Y. Rubinstein and D.P. Kroese. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Information Science and Statistics. Springer, 2004.

[RN10] S.J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, 2010.

[Rol] S.F. Rollini. Proof tRansformer and Interpolator for Propositional LOgic (PeRIPLO). `http://verify.inf.usi.ch/content/periplo`.

[RSS07] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, pages 346–362. Springer, 2007.

[RSS12] S.F. Rollini, O. Sery, and N. Sharygina. Leveraging Interpolant Strength in Model Checking. In *CAV*, pages 193–209. Springer, 2012.

[RT06] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smtlib.org`, 2006.

[Rub97] R.Y. Rubinstein. Optimization of Computer Simulation Models with Rare Events. *European Journal on Operations Research*, 99(1):89–112, 1997.

[Rub02] R.Y. Rubinstein. The Cross-Entropy Method and Rare-Events for Maximal Cut and Bipartition Problems. *ACM Transactions on Modeling and Computer Simulation*, 12(1):27–53, 2002.

[SATa] SAT Challenge 2012. `http://baldur.iti.kit.edu/SAT-Challenge-2012/`.

[SATb] SAT Competitions. `http://www.satcompetition.org/`.

[SATc] SATLIB benchmarks. `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

[SB11] F. Somenzi and A.R. Bradley. IC3: Where Monolithic and Incremental Meet. In *FMCAD*, pages 3–8, 2011.

[Seb07] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability*, 3:144–224, 2007.

[SFS11] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-Based Function Summaries in Bounded Model Checking. In *HVC*, pages 160–175. Springer, 2011.

[SFS12a] O. Sery, G. Fedyukovich, and N. Sharygina. FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization. In *ATVA*, pages 203–207. Springer, 2012.

[SFS12b] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental Upgrade Checking by Means of Interpolation-Based Function Summaries. In *FMCAD*, pages 114–121, 2012.

[Sho84] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, 1984.

[Sin07] C. Sinz. Compressing Propositional Proofs by Common Subproof Extraction. In *EUROCAST*, pages 547–555, 2007.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI*, pages 337–343, 1994.

[SKC95] B. Selman, H.A. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. In *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.

[ske] Skeptic proof theory library. `https://github.com/Paradoxika/Skeptik`.

[SKK03] C. Sinz, A. Kaiser, and W. Küchlin. Formal Methods for the Validation of Automotive Product Configuration Data. *AI EDAM*, 17(1):75–97, 2003.

[SKM97] B. Selman, H.A. Kautz, and D.A. McAllester. Ten Challenges in Propositional Reasoning and Search. In *IJCAI*, volume 1, pages 50–54, 1997.

[SLM92] B. Selman, H.J. Levesque, and D.G. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI*, pages 440–446, 1992.

[SNA12] R. Sharma, A.V. Nori, and A. Aiken. Interpolants as Classifiers. In *CAV*, pages 71–87. Springer, 2012.

[SS77] R.M. Stallman and G.J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[SS01] D. Schuurmans and F. Southey. Local Search Characteristics of Incomplete SAT Procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[SSJ+03] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdir. Debugging Overconstrained Declarative Models Using Unsatisfiable Cores. In *ASE*, pages 94–105, 2003.

[Tse68] G.S. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13, 1968.

[VG09] Y. Vizel and O. Grumberg. Interpolation-Sequence Based Model Checking. In *FMCAD*, pages 1–8, 2009.

[WA09] T. Weber and H. Amjad. Efficiently Checking Propositional Refutations in HOL Theorem Provers. *Journal of Applied Logic*, 7(1):26–40, 2009.

[WBCG00] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In *CAV*, pages 124–138, 2000.

[Wei12] G. Weissenbacher. Interpolant Strength Revisited. In *SAT*, pages 312–326. Springer, 2012.

[YM05] G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *CADE*, pages 353–368. Springer, 2005.

[Zad88] L.A. Zadeh. Fuzzy Logic. *IEEE Computer*, 21(4):83–93, 1988.

[ZM03a] L. Zhang and S. Malik. Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formulas. In *SAT*, pages 239–249, 2003.

[ZM03b] L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *DATE*, pages 880–885, 2003.