Implementation of Interpolation Algorithms for Software Verification

by

José Abel Castellanos Joo

B.Tech., Universidad de las Américas Puebla, 2015

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2020

Dedication

To my family.

Acknowledgments

TODO.

Implementation of Interpolation Algorithms for Software Verification

by

José Abel Castellanos Joo

B.Tech., Universidad de las Américas Puebla, 2015

M.S., Computer Science, University of New Mexico, 2020

Abstract

This thesis discusses theoretical aspects and an implementation for the interpolation problem of the following theories: (quantifier-free) equalitiy with uninterpreted functions (EUF), unit two variable per inequality (UTVPI), and the combination of the two previous theories. The interpolation algorithms implemented in this thesis are originally proposed in [22].

Refutational proof-based solutions are the usual approach of many interpolation algorithms [17, 30, 29]. The approach taken in [22] relies on quantifier-elimination heuristics to construct an interpolant using one of the two formulas involved in the interpolation problem. The latter makes possible to study the complexity of the algorithms obtained. On the other hand, the combination method implemented in this thesis uses a Nelson-Oppen framework, thus we still require for this particular situation a refutational proof in order to guide the construction of the interpolant for the combined theory.

The implementation uses Z3 [11] for parsing purposes and satisfiability checking in the combination component of the thesis. Minor modifications were applied to

the Z3's enode data structure in order to label and distinguish formulas efficiently (i.e. distinguish A-part, B-part). Thus, the project can easily be integrated to the Z3 solver to extend its functionality for verification purposes using the Z3 plug-in module.

Contents

List of Figures

List of Tables

Glossary

TODO.

Chapter 1

Introduction

Modern society is witness of the impact computer software has done in recent years. The benefits of this massive automation is endless. On the other hand, when software fails, it becomes a catostrophe ranging from economic loss to threats for human life.

Due to strict and ambitious agendas, many software products are shipped with unseen and unintentional bugs which might potentially put at risk people's life like critical systems. Several approaches have been used to improve software quality. However, many of these approaches offer partial coverage or might take an abyssal amount of human effort to provide such solutions. Thus, these approaches cannot be consideral practical. On the other hand, for certain applications these contributions are relevant and their proper use fit such workflows uniquely.

Formal methods aim to bring a unique combination of automation, rigor, and efficiency (whenever effecient algorithms exits for the verification task). This thesis discusses a particular problem in software verification known as *the interpolantion problem* for the theories of the quantifier-free fragment of equality with uninterpreted functions (EUF) and unit two variable per inequality (UTVPI) and their combination. These two theories have been studied extensively and researcher have found

several applications for them.

1.1 Backgroud

An interpolant of a pair of logical formulas (α, β) is a logical formula γ such that α implies γ, $\beta \wedge \gamma$ is logically inconsistent, and γ only has common symbols of α and β. Informally this means that the interpolant 'belongs' to the consequence of α, and 'avoids' being part of the consequence of β. Not surprisingly, this intuition is behind many software verification routines where the first formula models a desirable state/property (termination, correctness) of a computer program, and the second formula models the set of undesirable states (non-termination, errors, crashes, etc) of such software. In Chapter 2, an extensive review of the formal concept is provided.

Eventhough interpolants are not a direct concern in verification problems, these problems are found in the core algorithms of the following two applications:

- Refinement of abstract models: In order to improve coverage and decrease the complexity in verification problems, abstract interpretation has become a proper technique to accomplish the latter together with model checking. Eventhough the methodology provides sound results, it is certainly not complete. Additionally, several abstractions do not capture the semantical meaning of programs due to the *over-approximation* approach. Hence, interpolants are used to strength predicate abstractions by using interpolants constructed from valid traces in the abstract model but not valid in the actual model (*spurious counterexamples*) [7, 28, 20].

- Invariant generation: following the same idea as in the previous case, *if a fix point is obtained in the refinement process*, we can obtain a logical invariant

of computer programs [1] Situations where this happens might be due to the finiteness of possible states in the program. It is worth mentioning that the invariant problem as stated in [21] is undecidable [38, 2].

1.2 Related work

There are several interpolation algorithms for the theories involved in the thesis work. The approaches can be classified into the following categories:

- Proof-based approach: This category relies on the availability of a refutational proof. The interpolant is constructed using a recursive function over the structure of the proof tree. In [29, 30] the author defines an interpolation calculus. This particular approach uses a proof tree produced by the SMT solver Z3 and does not need to modify any of Z3's internal mechanisms. Among the advantages of this approach is that theory combination is given for free since the SMT solver takes care of this problem. On the other hand, Z3's satellities theories are not sufficiently integrated with its proof-producing mechanism. Hence, one can find $\mathcal{T}-lemmas$ as black-boxes which introduces incompleteness in the interpolation calculus. For completeness, these lemmas are solved separately by another interpolantion algorithm for the respective theory. In [42] the authors provide a Nelson-Oppen framework to compute interpolants. For the convex-case, the approach only exchange equalities as required by the Nelson-Oppen framework. For the non-convex case, the authors require a resolution-based refutational proof to compute interpolants using Pudlak's algorithm. The introduction of the class of *equality interpolanting theories* is considered among

[1] The interpolation generation approach discussed can be understood as a *lazy framework* similarly to SAT/SMT algorithms. The former is about the production of interpolants, the latter is for assignments/models respectively. Both *block/learn* the formulas in order to find their results.

the most relevant contributions of the paper by the verification community. This is property about theories which states that if a theory is capable of proving a mixed equality $a = b$ (an equality which contains symbols from the two formulas in the interpolanting problem) then it exists a common term in the language of the theory t (known as the interpolating-term) such that the theory can prove $a = t$ and $t = b$. The property facilities formula-splitting for interpolation purposes. In [17] the authors modify a resolution-based refutational proof by introducing common-terms in the proof in order to produce interpolants in what is called colorable-proofs, which are proof trees which do not contained AB-mixed literals. This is pointed as an improvement to the approach followed in [42] which executes a similar idea but done progressively as the proof-tree is built and does not require the theories solver to be *equality propagating* [2]. However, this results is not generalizable to non-convex theories due to internal constraints.

- Reduction-based approach: This method transforms the interpolation problem into a query for some solver related to the theory. An example of this approach can be found in [39] where the authors use a linear-inequality solver to provide an interpolant fot the theory of linear inequalities over the rational/real numbers ($LIA(\mathbb{Q})/LIA(\mathbb{R})$). Additionally, they integrate the procedure with a *hierarchial reasoning* approach in order to incorporate the signature of theory for (quantifer-free) equalities with uninterpreted functions.

1.3 Outline of the thesis

- Chapter 2 provides an extensive background of fundamentals ideas, definitions and decision procedures used in the thesis work.

[2]The authors in [42] require that the theory solvers keep track of the interpolating-term and propagate this term whenever possible

- Chapters 3, 4, and 5 explain implentation details of the interpolating algorithms for the theories EUF, UTVPI, and their combination respectively. These chapters share the same structure. They start with the algorithms used to solve the interpolation problem, discuss about implementation details including diagrams of the architecture of the implemented system, and show a performance comparision with the iZ3 interpolation tool available in the SMT solver Z3 until version 4.7.0.

1.4 Contributions

The contributions of the thesis can be summarized as follow:

1. Implementation of the interpolation algorithm for the theory EUF proposed in [22].
2. Formulation and implementation of a new procedure for checking unsatisfiability of grounded equations in Horn clauses using a congruence closure algorithm with explanations used in the implementation of item 1.
3. Implementation of the interpolation algorithm for the theory UTVPI proposed in [22].
4. Implementation of the combination procedure for interpolating algorithm proposed in [42] in order to combine the implementations of items 1. and item 3.

Chapter 2

Preliminaries

This chapter discusses basic concepts from first-order logic that are used in the rest of this thesis. We will pay particular attention to their language and semantics since these are fundamental concepts necessary to understand the algorithmic constructions to compute interpolants. For a comprehensive treatment on the topic, the reader is suggested the following references [31, 15].

2.1 First-Order Predicate Logic

2.1.1 Language

A language is a collection of symbols of different sorts equipped with a rule of composition that effectively tells us how to recognize elements that belong to the language [40]. In particular, a first-order language is a language that expresses boolean combinations of predicates using terms (constant symbols and function applications). In mathematical terms,

Definition 2.1.1. *A first-order language (also denoted signature) is a triple* $\langle \mathfrak{C}, \mathfrak{P}, \mathfrak{F} \rangle$

of non-logical symbols where:

- $\mathfrak{C}$ *is a collection of constant symbols*
- $\mathfrak{P}$ *is a collection of n-place predicate symbols*
- $\mathfrak{F}$ *is a collection of n-place function symbols*

including logical symbols like quantifiers(universal ($\forall$)*, existential*($\exists$)*) logical symbols like parenthesis, propositional connectives (implication* ($\rightarrow$)*, conjunction* ($\wedge$)*, disjunction* ($\vee$)*, negation* ($\neg$)*), and a countably number of variables* Vars *(i.e.* $\text{Vars} = \{v_1, v_2, \dots\}$*).*

The rules of composition distinguish two objects, terms *and* formulas*, which are defined recursively as follows:*

- *Any variable symbol or constant symbol is a term.*
- *If* $t_1, \dots, t_n$ *are terms and* f *is an n-ary function symbol, then* $f(t_1, \dots, t_n)$ *is also a term.*
- *If* $t_1, \dots, t_n$ *are terms and* P *is an n-ary predicate symbol, then* $P(t_1, \dots, t_n)$ *is formula.*
- *If* x *is a variable and* ψ, φ *are formulas, then* $\neg\psi, \forall x.\psi, \exists x.\psi$ *and* $\psi \square \varphi$ *are formulas where* $\square \in \{\rightarrow, \wedge, \vee\}$

No other expression in the language can be considered terms nor formulas if such expressions are not obtained by the previous rules.

2.1.2 Semantics

In order to define a notion of truth in a first-order language is necessary to associate for each non-logical symbol (since logical symbols have established semantics from propositional logic) a denotation or mathematical object and an *assignment* to the collection of variables. The two previous components are part of an *structure* [15] (or interpretation [40]) for a first-order language.

Definition 2.1.2. *Given a first-order language $\mathfrak{L}$, an interpretation $\mathfrak{I}$ is a pair $(\mathfrak{A}, \mathfrak{J})$, where $\mathfrak{A}$ is a non-empty domain (set of elements) and $\mathfrak{J}$ is a map that associates to the non-logical symbols from $\mathfrak{L}$ the following elements:*

- $c^{\mathfrak{J}} \in \mathfrak{A}$ *for each* $c \in \mathfrak{C}$
- $f^{\mathfrak{J}} \in \{\mathfrak{A}^n \to \mathfrak{A}\}$ *for each n-ary function symbol* $f \in \mathfrak{F}$
- $P^{\mathfrak{J}} \in \{\mathfrak{A}^n\}$ *for each n-ary predicate symbol* $P \in \mathfrak{P}$

An assignment $s : Vars \to \mathfrak{A}$ is a map between Vars to elements from the domain of the interpretation.

With the definition of interpretation $\mathfrak{I}$ and assignment s, we can recursively define a notion of *satisfiability* (denoted by the symbol $\models_{\mathfrak{J},s}$) as a free extension from atomic predicates (function application of predicates) to general formulas as described in [15]. For the latter, we need to extend the assignment function to all terms in the language.

Definition 2.1.3. *Let $\mathfrak{I} = (\mathfrak{A}, \mathfrak{J})$ be an interpretation and s an assignment for a given language, Let $\bar{s} : Terms \to A$ be defined recursively as follows:*

- $\bar{s}(c) = c^{\mathfrak{J}}$

- $\bar{s}(f(t_1, \ldots, t_n)) = f^{\mathfrak{J}}(\bar{s}(t_1), \ldots, \bar{s}(t_n))$

Notice that the extension of s depends on the interpretation used.

Definition 2.1.4. *Given an interpretation $\mathfrak{I} = (\mathfrak{A}, \mathfrak{J})$, an assignment s, and ψ a formula, we define $\mathfrak{I} \models_s \psi$ (read ψ is satisfiable under interpretation $\mathfrak{I}$ and assignment s) recursively as follows:*

- $\models_{\mathfrak{I},s} P(t_1, \ldots, t_n)$ *if and only if* $P^{\mathfrak{J}}(\bar{s}(t_1), \ldots, \bar{s}(t_n))$
- $\models_{\mathfrak{I},s} \neg\psi$ *if and only if it is not the case that* $\models_{\mathfrak{I},s} \psi$
- $\models_{\mathfrak{I},s} \psi \wedge \varphi$ *if and only if* $\models_{\mathfrak{I},s} \psi$ *and* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \psi \vee \varphi$ *if and only if* $\models_{\mathfrak{I},s} \psi$ *or* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \psi \to \varphi$ *if and only if* $\models_{\mathfrak{I},s} \neg\psi$ *or* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \forall x.\psi$ *if and only if for every* $d \in \mathfrak{A}, \models_{\mathfrak{I},s_{x\mapsto d}} \psi$, *where* $s_{x\mapsto d} : Vars \to \mathfrak{A}$ *is reduct of* s *under* $Vars \setminus \{x\}$ *and* $s_{x\mapsto d}(x) = d$
- $\models_{\mathfrak{I},s} \exists x.\psi$ *if and only if exits* $d \in \mathfrak{A}, \models_{\mathfrak{I},s_{x\mapsto d}} \psi$, *where* $s_{x\mapsto d}$ *is defined as in the previous item.*

If an interpretation and assignment satisfies a formula, then we say that the interpretation and the assignment are a model for the respective formula. A collection of formulas are satified by an interpretation and assignment if these model each formula in the collection.

A formula ψ is said to be a valid *formula of the interpretation $\mathfrak{I}$ when $\models_{\mathfrak{I},s} \psi$ for all possible assignments s.*

Additionally, if all the models $(\mathfrak{I}, s)$ in a language of a collection of formulas Γ satisfy a formula ψ, then we say that Γ logically implies ψ (written $\Gamma \models \psi$). For the latter, ψ is said to be a valid *formula of the model $(\mathfrak{I}, s)$.*

2.2 Mathematical Theories

A theory $\mathcal{T}$ is a collection of formulas that are closed under logical implication, i.e. if $\mathcal{T} \models \psi$ then $\psi \in \mathcal{T}$. This concept is quite relevant for our thesis work since we will focus on two theories, the quantifier-free fragment of the theory of equality with uninterpreted functions (EUF), and the theory of unit two variable per inequality (UTVPI).

For some theories it is enough to provide a collection of formulas (known as the axioms of the theory). For the case of the theories of interest for the thesis, the axiomatization if the following:

2.2.1 Equality with uninterpreted functions

Definition 2.2.1. *Let $\mathfrak{L}_{EUF} = \{\{\}, \{=\}, \{f_1, \ldots, f_n\}\}$ be the language of EUF. The axioms of the theory are:*

- *(Reflexivity)* $\forall x.x = x$
- *(Symmetry)* $\forall x.\forall y.x = y \rightarrow y = x$
- *(Transitivity)* $\forall x.\forall y.\forall z.(x = y \wedge y = z) \rightarrow x = z$
- *(Congruence)* $\forall x_1 \ldots \forall x_n.\forall y_1 \ldots \forall y_n.(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

We notice that the congruence axiom is not a first-order logic axiom, but rather an axiom-scheme since it is necessary to *instantiate* such axiom for every arity of the function symbols in a given language.

2.2.2 Ordered commutative rings

In order to describe the UTVPI theory we will first introduce the language and theory of an ordered commutative ring.

Definition 2.2.2. *Let* $\mathfrak{L}_{Ord-R} = \{, \{0, 1\}, \{=, \leq\}, \{+, -, *\}, \}$ *be the language of an ordered commutative ring* R*. The axioms of the theory are:*

- $\forall x.\forall y.\forall z.x + (y + z) = (x + y) + z$
- $\forall x.\forall y.x + y = y + x$
- $\forall x.x + 0 = x$
- $\forall x.x + (-x) = 0$
- $\forall x.\forall y.\forall z.x * (y * z) = (x * y) * z$
- $\forall x.x * 1 = x$
- $\forall x.\forall y.x * y = y * x$
- $\forall x.\forall y.\forall z.x * (y + z) = x * y + x * z$
- $\forall x.\forall y.\forall z.(y + z) * x = y * x + z * x$
- $\forall x.\forall y.\forall z.x \leq y \rightarrow x + z \leq y + z$
- $\forall x.\forall y.(0 \leq x \wedge 0 \leq y) \rightarrow 0 \leq x * y.$
- $0 \neq 1 \wedge 0 \leq 1$

Section 2.4 discusses computability aspects for the theories of interest that are relevant for verification.

2.3 Interpolants

Following the notation in [42], we denote $\mathcal{V}(\psi)$ to be the set of non-logical symbols, variables and constants of formula ψ. Given an instance for the interpolation problem (A, B) [1], we distinguish the following categories:

- ψ is *A-local* if $\mathcal{V}(\psi) \in \mathcal{V}(A) \setminus \mathcal{V}(B)$
- ψ is *B-local* if $\mathcal{V}(\psi) \in \mathcal{V}(B) \setminus \mathcal{V}(A)$
- ψ is *AB-common* if $\mathcal{V}(\psi) \in \mathcal{V}(A) \cap \mathcal{V}(B)$
- ψ is *AB-pure* when either $\mathcal{V}(\psi) \subseteq \mathcal{V}(A)$ or $\mathcal{V}(\psi) \subseteq \mathcal{V}(B)$, otherwise ψ is *AB-mixed*

Example 2.3.0.1. *Consider the following interpolation pair:* $(f(a+2)+1 = c+1 \wedge f(a+2) = 0, f(c) \leq b \wedge b < f(0))$. *With respect to the previous interpolation pair, we can tell that:*

- *The formula* $f(a+2) = c$ *is AB-pure but not A-local nor B-local nor AB-common*
- *The formula* $\neg(a \leq f(f(b)+1))$ *is an AB-mixed literal*
- *The formula* $a + 1 = 1$ *is A-local.*
- *The formula* $c + 1 = 1$ *is AB-pure but not AB-common.*
- *The formula* $c = 0$ *is AB-common.*
- *In general,* $AB - common$ *formulas are not* $AB - pure$ *formulas.*

[1] For the rest of the thesis, we will denote the first formula of an interpolantion problem as the A-part and the second component as the B-part

2.3.1 Craig interpolation theorem

Let α, β, γ be logical formulas in a given theory. If $\models_{\mathcal{T}} \alpha \rightarrow \beta$, we say that γ is an interpolant for the interpolation pair (α, β) if the following conditions are met:

- $\models_{\mathcal{T}} \alpha \rightarrow \gamma$
- $\models_{\mathcal{T}} \gamma \rightarrow \beta$
- Every non-logical symbol in γ occurs both in α and β.

The *interpolantion problem* can be stated naturally as follows: given two logical formulas α, β such that $\models_{\mathcal{T}} \alpha \rightarrow \beta$, find the interpolant for the pair (α, β).

In his celebrated result [9], Craig proved that for every pair (α, β) of formulas in first-order logic such that $\models \alpha \rightarrow \beta$, an interpolation formula exists. Nonetheless, there are many logics and theories that this result does not hold [24].

Usually, we see the interpolation problem defined differently in the literature, where it is considered β' to be $\neg\beta$ and the problem requires that the pair (α, β') is mutually contradictory (unsatisfiable). This definition was popularized by McMillan [29]. This shift of attention explains partially the further development in interpolation generation algorithms since many of these relied on SMT solvers that provided refutation proofs in order to (re)construct interpolants for different theories (and their combination) [25, 5, 30].

Relaxed definitions are considered to the interpolation problem when dealing with specific theories [42] in a way that interpreted function can be also part of the interpolant. The latter is justified since otherwise, many interpolation formulas might not exists in different theories or the interpolants obtained might not be relevant (for example, lisp programs). This is formalize as follows:

Definition 2.3.1. *[42] Let $\mathcal{T}$ be a first-order theory of a signature Σ and let $\mathcal{L}$ be the class of quantifier-free Σ formulas. Let $\Sigma_{\mathcal{T}} \subseteq \Sigma$ denote a designated set of interpreted symbols in $\mathcal{T}$. Let A, B be formulas in $\mathcal{L}$ such that $A \wedge B \models_{\mathcal{T}} \bot$. A* theory-specific *interpolant for (A, B) in $\mathcal{T}$ is a formula I in $\mathcal{L}$ such that $A \models_{\mathcal{T}} I$, $B \wedge I \models_{\mathcal{T}} \bot$, and I refers only to AB-common symbols and symbols in $\Sigma_{\mathcal{T}}$.*

Example 2.3.1.1. *In example 2.3.0.1 we can tell $c + 1 = 1$ is not an interpolant simplify because the symbol 1 only appears on the A-part. However, if $\Sigma_{\mathcal{LIA}(\mathbb{Z})}$ contains the interpreted symbols of $LIA(\mathbb{Z})$ (i.e. $+, *, 0, 1, 2, \dots$), then $c+1 = 1$ becomes a* theory-spefic *interpolant.*

Notice that $c = 0$ is an interpolant even if the set of interpreted symbols used for interpolation is empty.

2.4 Decision Procedures

Given a theory $\mathcal{T}$ in a formula ψ in the language of the theory, is it possible to know $\models_{\mathcal{T}} \psi$? The last question is known as the verification of the decision problem for the respective theory $\mathcal{T}$. This question has been studied extensively for many theories of interest [3].

Regarding the decidability of the theories mentioned in Section 2.2, it is known that EUF is undecidable [3], and the theory of ordered commutative rings is undecidable when the structure uses integers as the domain and the semantics of the arithmetical operations [15] (on the other hand, this theory can be decidable if we keep the same structure but use the reals as the domain [15]). Nonetheless, the quantifier-free fragment of EUF and the restriction imposed in the decision problem for the UTVPI theory allow efficient algorithms to decide validity and satisfiability in their respective theories [35, 14, 27].

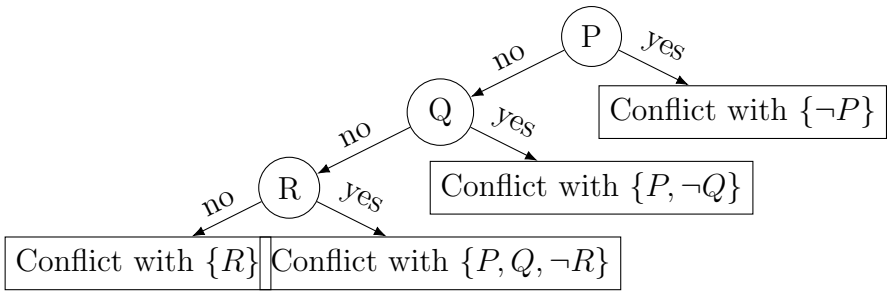


Figure 2.1: DPLL execution on the clauses $\{\{\neg P\}, \{P, Q, \neg R\}, \{R\}, \{P, \neg Q\}\}$

In the rest of this section we review some decision problems and provide references to their respective decision procedures used in the implementation work of the thesis.

2.4.1 Satisfiability and Satisfiability Modulo Theories

The satisfiability problem consists on finding a propositional assignment for a propositional formula. This problem is at the core level of complexity theory, defining an important class of problems known as NP, which is a class whose algorithms seem to be intractable. Developments in algorithms and heuristics [23, 33] have made possible to use satisfiability algorithms to solve real-world problems in verification [2].

The DPLL algorithm [10] (an other extensions) is the frequent algorithm found in many SAT solvers. Fundamentally, it is a search-based algorithm which implements of operators (decide, unit-propagation, backtrack) to find a satisfiable assignment. If the algorithm is not able to find a satisfying assignment for a formula, then it is possible to extract a *resolution proof* based on the traces of the search operations.

Example 2.4.0.1. *We can see that the following resolution proof resembles the structure of the DPLL execution on figure 2.1, i.e. if we rotate the proof-tree and mark*

[2]These advances do not provide an answer to the well-known P vs. NP problem. There are results indicating a class of problem instances for many of the SAT algorithms which cannot be solve in less that $\mathcal{O}(2^n)$ steps [23].

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg P \quad P \vee Q \vee \neg R}{Q \vee \neg R}\, res_P \qquad R}{Q}\, res_R \qquad P \vee \neg Q}{P}\, res_Q \qquad \neg P}{\bot}\, res_P$$

Figure 2.2: Example of resolution proof

the nodes by the pivots used we obtained a similar tree obtained by the execution of the DPLL algorithm. This fact becomes relevant becomes it enables us to construct a resolution proof from the traces of a SAT solver used in the theory combination part of the implementation.

We can extend this approach to work not only with propositional variables but with terms of more complex signatures [26]. If we are given a boolean combination of formulas from any theory that is capable of deciding the satisfiability of any conjunction of formulas in the theory, by using a *lazy framework* integration with a SAT solver is possible to find either a model or declare unsatisfiable such boolean combination as follows: (i) first abstract the literals in the boolean combination to (pseudo) boolean propositions; (ii) find a satisfying assignment (using a SAT solver) of the (pseudo) boolean propositions; (iii) using the theory solver, test if the collection of positive and negative literals induced by the pseudo boolean variables is satisfiable; (iv) if it is then declare the formula satisfiable, otherwise *learn* (or block) the pseudo boolean clause obtained (by negating the conjunction of boolean constraints) in the SAT solver and repeat from step (ii); (v) if it is not possible to find a satisfying assignment for the pseudo boolean variables, declare the original formula to be unsatisfiable.

These algorithms are used in the last section of the thesis work. The implementation for the interpolation combination method in [42] requires a resolution-based proof in order to compute partial interpolants by integrating Pudlak's algorithm.

2.4.2 Congruence Closure

The congruence closure problem consists of given a conjunction of equalities and disequalities ψ determine if an equality $u = v$ follows from the consequence generated by $\models_{EUF} \psi$.

As noted in [8], it is just succiffient to compute the minimal relation containing the initial relation defined by the equalities in ψ closed under reflexivity, symmetry, transitivity and congruence considering all the subterms in the formulas ψ and $u = v$.

The authors in [14, 35] independently formulated an optimized version of the algorithm afore mentioned. Their key observation was to introduce a list of pointers keeping track of the antecedents nodes in the abstract syntax tree induced by the formulas. The latter allows a fast signature checking in order to determine if two nodes are equivalent under the equivalence relation of the formulas. The algorithm in [14] has better runtime complexity $\mathcal{O}(n \log n)$ since it also implements a 'modify the smaller half' (using the union-find data structure). The congruence closure algorithm in [14] has a runtime complexity of $\mathcal{O}(n^2)$. Nonetheless, the authors reported no significant advantage of the first approach because, for verification purposes, the list of antecedents is usually small. Both approaches provides the FIND, MERGE operations.

In [37], the authors introduced a Union-Find data structure that supports the additional Explanation operation. This operation receives as input an equation between constants. If the input equation is a consequence of the current equivalence relation defined in the Union Find data structure, the Explanation operation returns the minimal sequence of equations used to build such equivalence relation, otherwise it returns 'Not provable'. A proper implementation of this algorithm extends the traditional Union-Find data structure with a *proof-forest*, which consists of an additional representation of the underlying equivalence relation that does not compress

paths whenever a call to the Find operation is made. For efficient reasons, the Find operation uses the path compression and weighted union.

The main observation in [37] is that, in order to recover an explanation between two terms, by traversing the path between the two nodes in the proof tree, the last edge in the path guarantees to be part of the explanation. Intuitively, this follows because only the last Union operation was responsible of merging the two classes into one. Hence, we can recursively recover the rest of the explanation by recursively traversing the subpaths found.

Additionally, the authors in [37] extended the Congruence Closure algorithm [36] using the above data structure to provide Explanations for the theory of EUF. The congruence closure algorithm is a simplification of the congruence closure algorithm in [14]. The latter combines the traditional *pending* and *combine* list into one single list, hence removing the initial *combination* loop in the algorithm in [14].

The implementation work utilizes the latter congruence closure with explanations for the interpolation algorithm of the theory EUF. The idea was to use the explanation operator to construct uncommon-free Horn clauses.

2.4.3 Satisfiability of Horn clauses with ground equations

In [19] it was proposed an algorithm for testing the unsatisfiability of ground Horn clauses with equality. The main idea was to interleave two algorithms: *implicational propagation* (propositional satisfiability of Horn clauses) that updates the truth value of equations in the antecedent of the input Horn clauses [13]; and *equational propagation* (congruence closure for grounded equations) to update the state of a Union-Find data structure [18] that keeps the minimal equivalence relation defined by grounded equations in the input Horn clauses.

The author in [19] defined two variations of his algorithms by adapting the Congruence Closure algorithms in [14, 35]. Additionally, modifications in the data structures used by the original algorithms were needed to make the interleaving mechanism more efficient.

Our implementation uses the equality propagation mechanism in the algorithm proposed by Gallier when we have to deal with Horn clauses with ground equations. In addition, we also needed to design some modifications of the original formulation so it can integrate with the congruence closure with explanation algorithm mentioned in the previous section.

2.4.4 Nelson-Oppen framework for theory and interpolation combination

The theory combination problem consists on taking a formula from the union of two (or more) disjoint languages and tell if such formula is satisfiable or not in the combined theory, i.e. a theory resulting after putting together two (or more) axiomatizations.

In [34] the authors defined a procedure to achieve the above problem. The key idea is to *purify* (or separate) the subformulas by including additional constant symbols equating subterms such that the resulting formula can be splitted into components of the appropriate language for each theory solvers to work with. The separation naturally will hide relevant information to the solvers and they might not be able to decide satisfiability correctly. The authors noticed that to solve the above problem it is enough to share disjunction of equalities between the combined theories of shared terms. In addition, they proved that some theories have the following property:

Definition 2.4.1. *Let $\mathcal{T}$ be a theory. We say that $\mathcal{T}$ is a* convex theory *if a finite conjunction of formulas in $\mathcal{T}$ $\psi = \bigwedge_{i=1}^{m} \psi_i$ satisfies $\psi \models_{\mathcal{T}} \bigvee_{j=1}^{n} x_j = y_j$, then exists*

$k \in \{1, \ldots, n\}$ *such that* $\psi \models_{\mathcal{T}} x_k = y_k$.

Hence, it is important to detect whether the theories involved are convex or not since this can improve performance since convex theories do not need to share disjunctions of equalities as mentioned before (since all these disjunctions imply a single equality).

Example 2.4.1.1.

- *The conjunctive fragment of equality logic is convex since it can always decide the membership of an equation in the equivalence relation.*
- *The theory of UTVPI over the intergers is not convex. To see the latter consider* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x \vee 2 = x$. *However, it is not the case that* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x$ *nor* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 2 = x$.

An interpolation combination framework as proposed in [42] follow the same idea towards theory combination. Inductively, they define *partial interpolants* for each shared equality/disjunction of equalities until some theory reaches the unsatisfiable state, which is expected since an interpolant is a pair a mutually contradicting formulas.

This framework was implemented in the thesis work. This framework was chosen in particular since it allows to work with non-convex theories (in our case for the theory of UTVPI over $\mathbb{Z}$).

2.5 General system description

The algorithms implemented in this thesis used the C++ programming language. The overall architecture of the system is the following:

All the decision procedures mentioned in this chapter were implemented with the exception of the SAT/SMT algorithms. For the latter, zChaff [33] and Z3 [11]

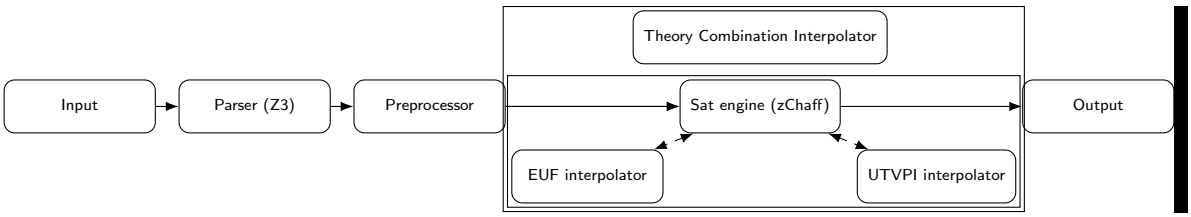


Figure 2.3: General System Diagram

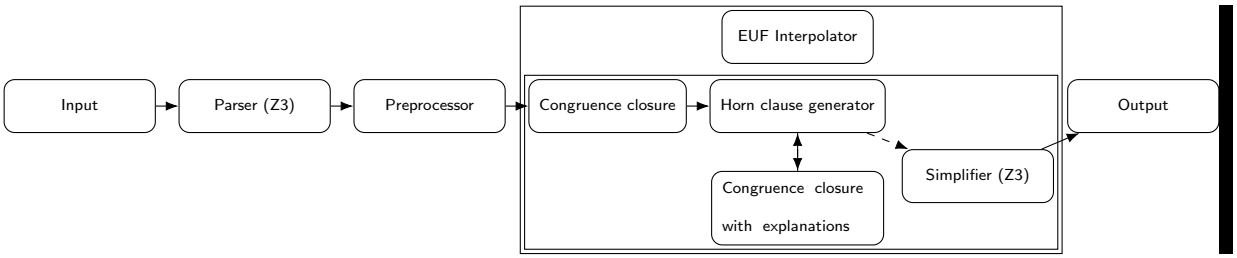


Figure 2.4: EUF Interpolator Diagram

were chosen as the libraries to work with these algorithms. The rest of this section discusses some minor modifications implemented in the above mentioned Z3 and zChaff libraries.

2.5.1 Minor modifications to Z3

Z3 standard input is SMTLib2 [1]. This grammar does not provide a standard specification regarding a suitable format to work with interpolants. Interpolation

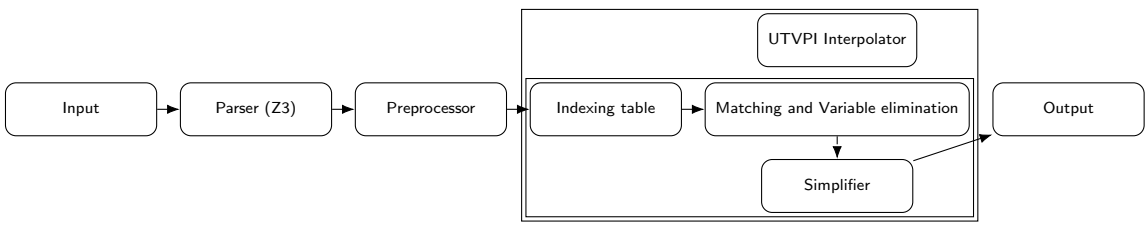


Figure 2.5: UTVPI Interpolator Diagram

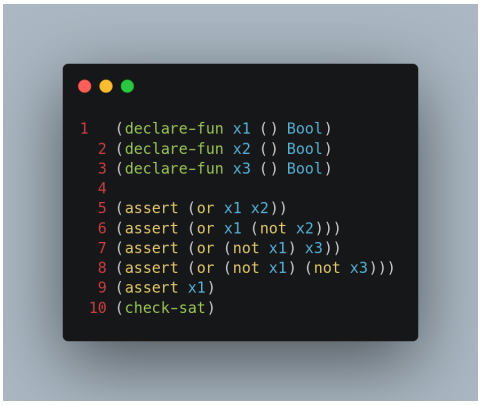
software read interpolant formulas based on the order of appereance in a conjunction [30]. In our case we require two conjuncts of conjunctions of literals in the EUF theory, UTVPI theory or combined theory.

As we can notice in figures 2.3, 2.4, and 2.5, there is preprocessor component which prefixes the names of uninterpreted symbols with the strings a_, b_, c_ to indicate that the symbol name is either an A-local, B-local, or common symbol respectively. We extended Z3's API with functions that test if a formula is A-local, B-local, AB-pure, AB-common based of the definitions in [42] because it is necessary to constantly check this conditions for splitting purposes. Another reason for the latter is justified because the implemented congruence closure algorithm takes as an additional criteria to maintain as representative term an AB-common term. A similar change was implemented in the congruence closure implementation of Z3. Nonetheless, it was irrelevant since Z3's internal structure separates the abstract syntax tree, which is part of its API with the enode data structures, which does not allow the super to modify it. This is the reason why it was not possible to work directly with Z3 congruence closure implementation and a separate implementation was necessary.

2.5.2 Minor modifications to zChaff

We used zChaff to reconstruct a resolution-based proof necessary for Pudlak's algorithm in the interpolation combination componente. Given that Z3 provides a user friendly proof-producing API [12], why did the implementation work require another SAT solver to obtain the resolution-proof?

There are several reasons for the latter. First, the author of the thesis was not able to find an appropriate configuration of parameters for the SMT solver to provide such proofs. In order to grasp an idea of the latter, it was implemented a Z3 proof



```
1  (declare-fun x1 () Bool)
2  (declare-fun x2 () Bool)
3  (declare-fun x3 () Bool)
4
5  (assert (or x1 x2))
6  (assert (or x1 (not x2)))
7  (assert (or (not x1) x3))
8  (assert (or (not x1) (not x3)))
9  (assert x1)
10 (check-sat)
```

Figure 2.6: Problematic SMT query for resolution proofs

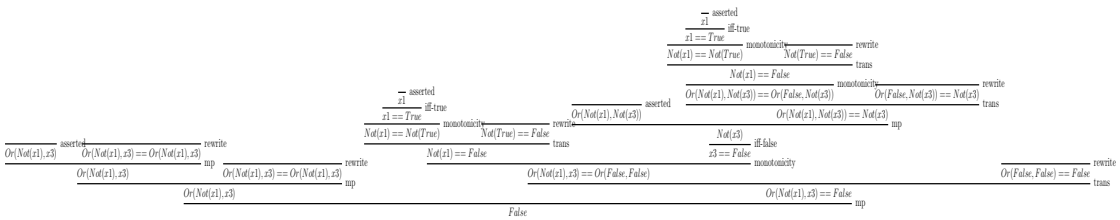


Figure 2.7: Z3 proof of figure 2.6

parser that render a pdf rendered by LaTeX. Many examples indicated that Z3 selects more convenient theories to work with some problems. For instance, the formula in pure propositional logic shown in figure 2.6 used proof rules from EUF [3].

Thus, we opted to use the zChaff SAT solver which implements the DPLL algorithm. The minor modifications to the software was to include the pivots in the proof trace zChaff produces. Many SAT solvers produce more sophisticated proofs in formats like DRAT, DRUP, etc. The latter require to implement unit-propagation to construct a resolution-base proof and hence the implementation might unintentionally slow the performance of the solver during the proof-checking step.

[3] Z3 uses the term monotonicity instead of congruence

Chapter 3

Interpolation algorithm for the theory of EUF

Interpolation algorithms for the theory of equality with uninterpreted functions are relevant as the core component of verification algorithms. Many useful techniques in software engineering like bounded/unbounded model checking and invariant generation benefit directly from this technique. In [4], the authors introduced a methodology to debug/verify the control logic of pipelined microprocessors by encoding its specification and a logical formula denoting the implementation of the circuit into a EUF solver.

Previous work addressing the interpolation problem for EUF has involved techniques ranging from interpolant-extraction from refutation proof trees [29, 30, 41], and colored congruence closure graphs [17]. Kapur's algorithm uses a different approach by using approximated quantifier-elimination, a procedure that given a formula, it produces a logically equivalent formula without a variable in particular [15].

3.1 Algorithm

Kapur's interpolation algorithm for the EUF theory uses quantifier-elimination techniques to remove symbols in the first formula of an interpolation problem instance that are not common with the second formula of the latter. Hence, the input for this algorithm is a conjunction of equalities in the EUF theory and a set of symbols to eliminate, also unknown as uncommon symbols.

The steps/ideas in Kapur's algorithm for interpolant generation for the EUF theory are the following:

- **Elimination of uncommon terms using congruence closure**. This step builds an equivalence relation using the input of the algorithm such that the representatives are common terms whenever possible. Let *answer* be a variable denoting the conjunction of all the input formulas which uncommon subterms are replaced by their representatives. If all the representatives in the equivalence relation are common terms, return *answer* as the interpolant. Otherwise, continue with the following step.

- **Horn clause generation by exposure**. This step uses Ackermann's reduction [26] to produce Horn clauses to eliminate uncommon terms identifying two cases:

  - The term is uncommon because the function symbol is uncommon.
  - The term is uncommon because at least one of its arguments is an uncommon term.

  Conjunct to *answer* the conjunction of all these Horn Clauses. If all the Horn Clauses above are common, return *answer* as the interpolant. Otherwise, continue with the following step.

- **Conditional elimination**. Since some of the Horn clauses produced by the previous step are not common, we identify the Horn clauses that have *common antecedents* and head equation with at least one uncommon term. For each of these Horn clauses, replace the uncommon term in its head by the common term in its head in the rest of the Horn Clauses, and include its antecedent to the antecedent of such Horn clauses. We can see that this step reduces the number of uncommon terms in the equalities of the Horn Clauses. We repeat this step until it cannot be performed. At the end, we take only the set of Horn Clauses that have common antecedents and have one uncommon term in its head. The call these Horn clauses *useful Horn clauses*. Continue with the next step.

- **Conditional replacement**. Using the *useful Horn clauses* generated in the previous step, update *answer* to be the formula resulting after *conditionally replacing* uncommon terms in each equation of *answer* by an appropriate common term in the head of a *useful Horn clauses*. To be more precise, let $\bigwedge_i a_i = b_i \rightarrow u \mapsto c$ be a useful Horn clause, where the antecedent is a conjunction of common grounded equations, u is an uncommon term, and c is a common term. Then for every instance of u in each equation of *answer*, conditionally replace u by c under $\bigwedge_i a_i = b_i$. We notice that equations in *answer* of the previous step will become Horn Clauses with less uncommon terms. For completeness, we perform these replacements zero or more times (up to the maximal number of instances per equation) in order to leave space for other *useful Horn Clauses* to replace the uncommon term in their heads as well. Remove all the literals in the current *answer* that contain uncommon terms and return this as the interpolant.

If the user is not interested in an explicit interpolant, we can present the *usable Horn clauses* in a proper order such that the replacements can be done without

exponentially increasing the size of the interpolant. This representation is useful because it provides a more compact representation of the interpolant that the user might be able quicker to obtain. Additionally, the user might be just interested in a particular subformula of the interpolant, so the latter representation offers such feature.

This algorithm allows a flexible implementation which can lead several optimizations based on the nature and applications of the interpolant.

3.2 Implementation

The description of the interpolation algorithm presented in the previous section suggests an straight forward implementation of the first two stages. This thesis work considers the following implementation for the rest of the stages of the algorithm:

3.2.1 New optimized conditional elimination step in Kapur's algorithm

First, we explain a high level ideal on how we improve the *conditional elimination* step in Kapur's algorithm. We notice that this step *propagates equationally* the head equations of grounded Horn clauses with common antecedents. Initially we employ the unsatisfiability algorithm for Horn clauses to achieve such propagation. However, the original algorithm will not be enough because it will only propagate the head equation when all the antecedents have truth value equal to true. To fix that problem, we modify two steps in Gallier's algorithms:

- When we build the data structure *numargs* that keeps track of the number of unproven equations in the antecedent of each Horn clause, we change this

number by the number of unproven uncommon equations in the antecedent of each Horn clause. This will be useful because we only introduce head equations intro the queue data structure in Gallier's algorithm when all the antecedents are true. With this modification, our algorithm introduces head equations when all the antecedent equations are common. Additionally the algorithm can still update correctly the truth value of common equations, but these are not relevant for our propagation purposes.

- To guarantee that *numargs* keeps the right number of uncommon equations yet to be proven, we also modify the update mechanism for *numargs* in the main while loop of the algorithm. The original algorithm reduces by one the corresponding entry in *numargs* whenever a recently popped element from the queue matches the antecedent of a Horn clause. We only decrease this value if such popped equation is uncommon. This prevents the algorithm from accidentally reducing the number of uncommon equations yet to be proven, which can cause that we propagate the uncommon head equation when the antecedent of a Horn clause only consists of common equations.

At the end of this algorithm we can identify *usable Horn clauses* by checking the Horn clauses with *numargs* entries equal to 0. Nonetheless, these Horn clauses are not the desired *usable Horn clauses* because the unsatisfiability testing algorithm did not update the antecedents of the Horn clauses. The main difficulty to design a data structure for the latter to work inside the unsatisfiability testing algorithm was the queue data structure only adds grounded equation whenever the truth value of the literal changes to true, which happens during *equational propagation* or during the *implicational propagation* steps. For the *implicational propagation* the task is easy because we can know the clause where the just new proven ground equation comes, but it cannot be the same situation for the *equational propagation* since this step relies on congruence closure.

To remedy this issue, we equip our congruence closure algorithm with the Explanation operator, so we can recover the grounded equations needed to entail any particular grounded equation. Additionally, this will require a data structure to maintain the Horn clauses for each grounded equation that it is the head equation of. With the latter we can recover the Horn Clauses where each grounded equation came from to update the antecedents and obtain *usable Horn clauses*.

The algorithm appears below in pseudo-code notation:

3.2.2 Ground Horn Clauses with Explanations

We notice that, by removing our changes to the unsatisfiability testing for grounded Horn clauses regarding uncommon symbols, we effectively combine the congruence closure with explanations to the original unsatisfiability testing algorithm. With the latter, we can query the membership of a Horn clauses in a given user-defined theory and additionally obtain a proof of the latter. This approach works by introducing the antecedent equations of a grounded Horn clause as part of the user-defined theory in order to prove its head equation. By the Deduction Theorem [31], we can recover a proof of the original queried Horn clause by removing the antecedent equations appearing the proof given by the Explain operation.

3.2.3 New optimized conditional replacement step in Kapur's algorithm

Once the conditional congruence closure data structure is built after the execution of the previous step, we are ready to compute conditional eliminations as follows. For the latter, we will require the following auxiliary functions:

3.3 Evaluation

TODO: keep working here.

Algorithm 1 Modified Unsatisfiability Testing for Ground Horn Clauses

```
 1: procedure SATISFIABLE(var H : Hornclause; var queue, combine: queuetype;
    var GT(H) : Graph; var consistent : boolean)
 2:     while queue not empty and consistent do
 3:         node := pop(queue);
 4:         for clause1 in H[node].clauselist do
 5:             if ¬ clause1.isCommon() then
 6:                 numargs[clause1] := numargs[clause1] - 1
 7:             end if
 8:             if numargs[clause1] = 0 then
 9:                 nextnode := poslitlist[clause1];
10:                 if ¬ H[nextnode].val then
11:                     if nextnode ≠ ⊥ then
12:                         queue := push(nextnode, queue);
13:                         H[nextnode].val := true;
14:                         u := left(H[nextnode].atom);
15:                         v := right(H[nextnode].atom);
16:                         if FIND(R, u) ≠ FIND(R, v) then
17:                             combine := push((u, v), combine);
18:                         end if
19:                     else
20:                         consistent := false;
21:                     end if
22:                 end if
23:             end if
24:         end for
25:         if queue is empty and consistent then
26:             closure(combine, queue, R);
27:         end if
28:     end while
29: end procedure

30: procedure CLOSURE(var combine, queue : queuetype; var R : partition)
31:     while combine is not empty do
32:         (u, v) = pop(combine)
33:         MERGE(R, u, v, queue)
34:     end while
35: end procedure
```

Algorithm 2 Modified Congruence Closure with Explanation Algorithms - Merge

```
    procedure MERGE(R : partition, u, v : node; queue, combine : queuetype)
 2:     if u and v are constants a and b then
            add a = b to Pending;
 4:         Propagate();
        else                                          ▷ u=v is of the form apply(a1, a2)=a
 6:         if Lookup(Representative(a1), Representative(a2)) is some apply(b1, b2)=b then
                add (apply(a1, a2)=a, apply(b1, b2) = b) to Pending;
 8:             Propagate();
            else
10:             set Lookup(Representative(a1), Representative(a2)) to apply(a1, a2)=a;
                add apply(a1, a2)=a to UseList(Representative(a1)) and to UseList(Representative(a2));
12:         end if
        end if
14: end procedure
```

Algorithm 3 Modified Congruence Closure with Explanation Algorithms - Propagate

```
    procedure PROPAGATE( )
 2:     while Pending is non-empty do
            Remove E of the form a=b or (apply(a1, a2) = a, apply(b1, b2) = b) from
    Pending
 4:         if Representative(a) ≠ Representative(b) and w.l.o.g.
    |ClassList(Representative(a))| ≤ |ClassList(Representative(b))| then
                oldReprA := Representative(a);
 6:             Insert edge a → b labelled with E into the proof forest;
                for each c in ClassList(oldReprA) do
 8:                 set Representative(c) to Representative(b)
                    move c from ClassList(oldReprA) to ClassList(Representative(b))
10:                 for each pointer L in ClassList(u) do
                        if H[L].val = false then
12:                         set the field H[L].lclass or H[L].rclass pointed to by p to
    Representative(b)
                            if H[L].lclass = H[L].rclass then
14:                             queue := push(L, queue);
                                H[L].val := true
16:                         end if
                        end if
18:                 end for
                end for
20:             for each apply(c1, c2) = c in UseList(oldReprA) do
                    if Lookup(Representative(c1), Representative(c2)) is some ap-
    ply(d1, d2) = d then
22:                     add (apply(c1, c2) = c, apply(d1, d2) = d) to Pending;
                        remove apply(c1, c2) = c from UseList(oldReprA);
24:                 else
                        set Lookup(Representative(c1), Representative(c2)) to ap-
    ply(c1, c2) = c;
26:                     move apply(c1, c2) = c from UseList(oldReprA) to
    UseList(Representative(b));
                    end if
28:             end for
            end if
30:     end while
    end procedure
```

Algorithm 4 Auxiliary function - Candidates

```
    procedure CANDIDATES(z3::expr const & t)
 2:     if t is common then
            return {t}
 4:     else
            return {t'|t' ∈ Class(t), t' is common}
 6:     end if
    end procedure
```

Algorithm 5 Auxiliary function - Auxiliar explain

```
    procedure EXPLAIN(z3::expr const & t1, z3::expr const & t2)
 2:     z3::expr_vector ans;
        if t1.id() equals t2.id() then
 4:         return ans;
        end if
 6:     auto partial_explain = hsat.equiv_class.explain(t1, t2);
        for (auto const & element : partial_explain) do
 8:         if element is common then
                ans.push_back(element);
10:        else
                auto const & entry = hsat.head_term_indexer.find(equation.id());
12:             if entry equals hsat.head_term_indexer.end() then
                    if equation is common then
14:                     ans.push_back(equation);
                    end if
16:             else
                    for (auto const & hsat_equation : entry → second → getAntecedent()) do
18:                     ans.push_back(hsat_equation);
                    end for
20:             end if
            end if
22:    end for
    end procedure
```

Algorithm 6 Auxiliary function - allCandidates

```
    procedure ALLCANDIDATES(z3::expr const & t)
 2:     if t has f-symbol uncommon then
            return {{}};
 4:     end if
        if t has f-symbol common and is of the form f(t_1, ..., t_n) then
 6:         return {candidates(t_1), ..., candidates(t_n)};
        end if
 8:     if t is a constant then
            undefined
10:    end if
    end procedure
```

Algorithm 7 Conditional Elimination - Part 1

```
    procedure CONDITIONAL ELIMINATION(z3::expr const & x, z3::expr const &
    y)
 2:     if x is constant and y is constant then
            for σ_x in CANDIDATES(x) do
 4:             for σ_y in CANDIDATES(y) do
                    horn_clause.add(EXPLAIN(x, σ_x) + EXPLAIN(y, σ_y), σ_x = σ_y)
 6:             end for
            end for
 8:     end if
        if x is constant and y is of the form f_y(t'_1, ..., t'_{k_2}) then
10:         for σ_x in CANDIDATES(x) do
                for σ_{f_y} in CANDIDATES(f_y(t'_1, ..., t'_{k_2})) do
12:                 horn_clause.add(EXPLAIN(x, σ_x) + EXPLAIN(f_y(t'_1, ..., t'_{k_2})),
    σ_y), σ_x = σ_{f_y}
                end for
14:             for arguments_{f_y} in CARTESIANPROD(ALLCANDIDATES(f_y(t'_1, ..., t'_{k_2})))
    do
                    horn_clause.add(EXPLAIN(x, σ_x) + Σ_{i=1}^{k_2} EXPLAIN(t'_i,
    arguments_{f_y}[i]), σ_x = f_y(arguments_{f_y}))
16:             end for
            end for
18:     end if
        if x is of the form f_x(t_1, ..., t_{k_1}) and y is a constant then
20:         return CONDITIONAL ELIMINATION(y, x);
        end if
22: end procedure
```

Algorithm 8 Conditional Elimination - Part 2

procedure CONDITIONAL ELIMINATION(z3::expr const & x, z3::expr const & y)
  if x is of the form $f_x(t_1, \ldots, t_{k_1})$ and y is of the form $f_y(t_1^{'}, \ldots, t_{k_2}^{'})$ **then**
    for σ_{f_x} in CANDIDATES($f_x(t_1, \ldots, t_{k_1})$) **do**
      for σ_{f_y} in CANDIDATES($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) **do**
        horn_clause.add(EXPLAIN($f_x(t_1, \ldots, t_{k_1})$, σ_{f_x}) + EXPLAIN($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$), σ_y), $\sigma_{f_x} = \sigma_{f_y}$
      end for
      for $arguments_{f_y}$ in CARTESIANPROD(ALLCANDIDATES($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) **do**
        horn_clause.add(EXPLAIN($f_x(t_1, \ldots, t_{k_1})$, σ_{f_x}) + $\sum_{i=1}^{k_2}$ EXPLAIN($t_i^{'}$, $arguments_{f_y}[i]$), $\sigma_{f_x} = f_y(arguments_{f_y})$)
      end for
    end for
    for $arguments_{f_x}$ in CARTESIANPROD(ALLCANDIDATES($f_x(t_1, \ldots, t_{k_1})$)) **do**
      for σ_{f_y} in CANDIDATES($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) **do**
        horn_clause.add($\sum_{i=1}^{k_1}$ EXPLAIN(t_i, $arguments_{f_x}[i]$) + EXPLAIN($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$), σ_y), $f_x(arguments_{f_x}) = \sigma_{f_y}$
      end for
      for $arguments_{f_y}$ in CARTESIANPROD(ALLCANDIDATES($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) **do**
        horn_clause.add($\sum_{i=1}^{k_1}$ EXPLAIN(t_i, $arguments_{f_x}[i]$) + $\sum_{i=1}^{k_2}$ EXPLAIN($t_i^{'}$, $arguments_{f_y}[i]$), $f_x(arguments_{f_x}) = f_y(arguments_{f_y})$)
      end for
    end for
  end if
end procedure

Chapter 4

Interpolation algorithm for UTVPI Formulas

This theory appears heavily in formal methods dealing with abstract domains introduced in [32]. The decision problem consists of checking the satisfiability of a particular fragment of $LIA(\mathbb{Z})$. The fragment consists on conjunctions of inequalities with at most two variables which integers coefficients are restricted to $\{-1, 0, 1\}$. Efficient algorithms are found in the literature for both the satisfiability problem [27] as well as for interpolation [6] of the theory. Eventhough this fragment looks severely constrained, it has been used for a range of applications where problems can be modelled using this particular kind of literals.

The algorithm in [22] follows a similar approach to the interpolation algorithm for EUF in the sense that the attention is given to one of the formulas in the interpolation pair [1] Other approaches towards interpolation follow graph-based algorithms. The idea combines the reduction of UTVPI formulas to difference logic [32] and a cycle detection of maximal size [6].

[1] This implementation uses the first formula of the pair.

4.1 Algorithm

The algorithm proposed [22] uses inference rules to close the relation and eliminate the uncommon variables from the A-part of the input formula. The rules are the following:

$$\frac{s_1 x_1 + s_2 x_2 \leq c \qquad s_1 = s_2 \in \{-1, 0, 1\} \text{ and } x_1 = x_2 \in Vars}{s_1 x_1 \leq \lfloor \frac{c}{2} \rfloor} \text{ Normalize}$$

$$\frac{s_1 x_1 + s_2 x_2 \leq c_1 \qquad -s_2 x_2 + s_3 x_3 \leq c_2}{s_1 x_1 + s_3 x_3 \leq c_1 + c_2} \text{ Elim}$$

The algorithm normalizes the inequalities at the beginning as a preprocessing step and applies the Elim rule whenever it is possible until no more uncommon variables remain in the input formula. Hence, having an efficient representation of the inequalities and detecting matches (like pivots for resolution steps) is important for an efficient implementation. To achieve this goal we implemented to an encoding of the inequalities using natural numbers, an array of numbers indexed by the numeral representation of the inequalities which keeps track of the minimum bound of the encoded inequality, and a data structure to keep track of the signs of variables in the inequalities for efficient matching.

4.2 Implementation

In order to obtain a bijection between UTVPI inequalities and natural numbers, first we define an ordering on the inequalities and notice some invariants of the latter.

We encode the term $\pm x_m \pm x_n$ using the point $(\pm m, \pm n) \in \mathbb{Z}^2$. Let $TermToPoint$ be the map that $\pm x_m \pm x_n \mapsto (\pm m, \pm n)$.The variable x_0 is a *dummy variable* that acts as a place holder for 0. Additionally, we will restrict the terms/pairs such that

the absolute value of the first index variable is strictly greater than the absolute value of the second index variable [2] since addition is commutative, except for the point (0, 0) which encodes the inequality with no variables.

We define the following orderings relevant for the terms of the form $\pm x_m \pm x_n$.

Definition 4.2.1. *Let $\succ_m$ be an ordering on the integers such that $a \succ_m b$ if and only $|a| > |b|$ or ($|a| = |b|$ and $a > b$) where $>$ is the standard ordering on integers.*

Let $\succ_p$ be an ordering on pair of integers such that $(m_1, n_1) \succ_p (m_2, n_2)$ if and only if $m_1 \succ_m m_2$ or ($m_1 = m_2$ and $n_1 \succ_m n_2$).

Let $\succ_t$ be an ordering on terms of the form $\pm x_m \pm x_n$ such that $t_1 \succ_t t_2$ if and only if $TermToPoint(t_1) \succ_p TermToPoint(t_2)$

Example 4.2.1.1. *The first 32 elements (in ascending order w.r.t. $\succ_t$) of UTVPI inequalities [3] are the following:*

$x_0 + x_0 \leq b_0$

$-x_1 + x_0 \leq b_1, x_1 + x_0 \leq b_2$

$-x_2+x_0 \leq b_3, -x_2-x_1 \leq b_4, -x_2+x_1 \leq b_5, x_2+x_0 \leq b_6, x_2-x_1 \leq b_7, x_2+x_1 \leq b_8$

$-x_3 + x_0 \leq b_9, -x_3 - x_1 \leq b_{10}, -x_3 + x_1 \leq b_{11}, -x_3 - x_2 \leq b_{12}, -x_3 + x_2 \leq b_{13}, x_3 + x_0 \leq b_{14}, x_3 - x_1 \leq b_{15}, x_3 + x_1 \leq b_{16}, x_3 - x_2 \leq b_{17}, x_3 + x_2 \leq b_{18}$

From the example, we notice that we can group/order the inequalities by groups using the first index. The first element of the i^{th} group corresponds to the $2(i-1)^2+1$ element in the $\succ_t$ order. The observation follows from an inductive argument since there are $2(1 + 2(i - 1))$ elements in the i^{th} group. It is also straight forward to

[2] This condition also avoids the problem of keeping track of variable duplication in the inequality.

[3] For readability purposes we include the bound for the UTVPI term

find the position of the first element in the second half of any group. With the above information it is possible to find the map between UTVPI terms and naturals numbers.

This bijection allows us to implement a data structure based on a vector of integers extended with $\pm\infty$ which encodes the upper bounds for the i^{th} inequality present in the input formula. For initialization purpose all the entries in this vector are set to ∞ and these values are updated accordingly to keep track to the minimum possible value for the inequality after the application of the inference rules mentioned at the introduction of the section.

4.3 Evaluation

TODO.

Algorithm 9 UTVPI constructor

```
    procedure UTVPI CONSTRUCTOR(position : integer)
 2:     coefficient1 = 0
        coefficient2 = 0
 4:     varindex1 = 0
        varindex2 = 0
 6:     if position = 0 then
            return
 8:     end if
        varindex1 = sqrt((position−1)/2) + 1
10:     initial_group_position = 2 ∗ (varindex1 − 1)^2 + 1
        half_size_group = 2 ∗ varindex1 − 1
12:     if position ≤ initial_group_position + half_size_group then
            coefficient1 = -1
14:         if position = initial_group_position then
                coefficient2 = 0
16:             varindex2 = 0
                return
18:         end if
            separation = position - initial_group_position
20:         varindex2 = (separation−1)/2 + 1
            if mod separation 2 = 0 then
22:             coefficient2 = 1
                return
24:         end if
            coefficient2 = -1
26:         return
        end if
28:     coefficient1 = 1
        if position = initial_group_position + half_size_group + 1 then
30:         coefficient2 = 0
            varindex2 = 0
32:         return
        end if
34:     separation = position - initial_group_position - half_size_group - 1
        varindex2 = (separation−1)/2 + 1
36:     if mod separation 2 = 0 then
            coefficient2 = 1
38:         return
        end if
40:     coefficient2 = -1
        return
42: end procedure
```

Algorithm 10 UTVPI position

```
    procedure UTVPI POSITION( s_1 x_{m_1} + s_2 x_{m_2} : UTVPI term)
 2:     initial_group_position = 2 * (m_1 - 1)^2 + 1
        if s_1 = -1 then
 4:         sign_a_offset = 0
        else
 6:         if s_1 = 0 then
                return 0
 8:         else
                if s_1 = 1 then
10:                 sign_a_offset = 2 * (m_1 - 1) + 1
                end if
12:         end if
        end if
14:     if s_2 = -1 then
            sign_b_offset = 1 + 2 * (m_2 - 1)
16:     else
            if s_2 = 0 then
18:             sign_b_offset = 0
            else
20:             if s_2 = 1 then
                    sign_b_offset = 2 * m_2
22:             end if
            end if
24:     end if
        return initial_group_position + sign_a_offset + sign_b_offset
26: end procedure
```

Chapter 5

Interpolation algorithm for the theory combination of EUF and UTVPI

Theory combination is an approach in order to reuse the veriffication algorithms for the theories used in a problem.

5.1 Algorithm

TODO. Discuss relevance of SAT solver and resolution proof. TODO. Mention issues with the negation for literals in UTVPI due to partial interpolantion.

5.2 Implementation

TODO.

Algorithm 11 Nelson-Oppen Propagation

```
    procedure NELSON-OPPEN PROPAGATION ( z3::expr_vector const & part_A,
    z3::expr_vector const & part_B )
 2:     T_1, T_2 = Purify(part_A, part_B)
        DisjunctionEqualitiesIterator ψ()
 4:     ψ.init()
        while true do
 6:         if T_1 ⊨_EUF ⊥ then
                return T_1
 8:         end if
            if T_2 ⊨_UTVPI ⊥ then
10:             return T_2
            end if
12:         if T_1 ⊨_EUF ψ.current() then
                if T_2 ⊨_UTVPI ψ.current() then
14:                 continue
                else
16:                 append ψ.current() to T_2
                    ψ.init()
18:             end if
            else
20:             if T_2 ⊨_UTVPI ψ.current() then
                    continue
22:             else
                    append ψ.current() to T_2
24:                 ψ.init()
                end if
26:         end if
            ψ.next()
28:     end while
    end procedure
```

5.3 Evaluation

TODO.

Chapter 6

Future Work

Regarding implemenation work, the are several improvements that were not explored in the thesis work since many of them do not change the overal complexity of the implementation, but definitely these consume more resources. These particular improvements are the following:

- Improve hash function for terms: during testing the congruence closure algorithm, it was evident that dealing with a large of terms, collisions happened to the point that some terms were merged since the signature function indicated to do so, but this should not happen. Nonetheless, for a typical verification problem the implementation will not find any problems.

- The space needed to encode curry nodes *can be significally reduced* up to an order of $\mathcal{O}(n)$. The reason for this current allocation schema is that it exists a double bonding effect while performing curryification of all the terms in the arguments of function applications.

- Replace zChaff with SAT solvers with more permissive license: while modifying the code of zChaff the author noticed a request in the README file of zChaff

that its license does not allow distribution of the sofware. However, the software can be used for research and non-exclusive personal use. I think any extension of MiniSat [16] with proof-logging will suffice for our purposes. This task should not be too complicated since the current implementation does not uses any API of the zChaff inside the implementation.

Regarding potential extension of the systems, it will interesting to explore a more specific interpolation combination approach as in [39]. The Nelson-Oppen framework seems to be adequate if the goal is to combine several theories or be as general as possible. The reason why the hierarchical reasoning approach was not implemented was because the authors did not make any claim about the possibility to use non-convex theories. Perhaps, this is not a real limitation of the approach.

Appendices

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[2] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Trans. Comput. Logic*, 2(1):1–11, January 2001.

[3] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Universitext. Springer Berlin Heidelberg, 2001.

[4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification*, pages 68–80, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[5] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Proof tree preserving interpolation. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–138, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Interpolant generation for utvpi. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 167–182, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[8] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, USA, 1969.

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

[10] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations and z3, 01 2008.

[13] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267 – 284, 1984.

[14] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[15] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[16] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[17] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstić, and Cesare Tinelli. Ground interpolation for the theory of equality. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964.

[19] Jean H. Gallier. Fast algorithms for testing unsatisfiability of ground horn clauses with equations. *Journal of Symbolic Computation*, 4(2):233 – 254, 1987.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, January 2004.

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

[22] Deepak Kapur. A new algorithm for computing (strongest) interpolants over quantifier-free theory of equality over uninterpreted symbols. *Manuscript*, 2017.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 1st edition, 2015.

[24] Yuichi Komori. Logics without craig's interpolation property. *Proc. Japan Acad. Ser. A Math. Sci.*, 54(2):46–48, 1978.

[25] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 199–213, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[26] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[27] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for utvpi constraints. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, pages 168–183, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[28] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[29] K. L. McMillan. An interpolating theorem prover. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[30] Kenneth McMillan. Interpolants from z3 proofs. In *Formal Methods in Computer-Aided Design*, October 2011.

[31] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall-CRC, 5th edition, 2009.

[32] Antoine Miné. The octagon abstract domain. *CoRR*, abs/cs/0703084, 2007.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.

[34] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

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

[36] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 78–90, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[37] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications*, pages 453–468, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[38] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[39] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 346–362, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[40] Dirk van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994.

[41] Georg Weissenbacher. Interpolant strength revisited. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 312–326, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 353–368, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.