Implementation of Uniform Interpolation Algorithms

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

October, 2020

Dedication

To my family.

Acknowledgments

TODO.

Implementation of Uniform Interpolation Algorithms

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 the implementation for the uniform interpolation problem of the following theories: (quantifier-free) equalitiy with uninterpreted functions (EUF), unit two variable per inequality (UTVPI), and theoretical aspects for the combination of the two previous theories. The uniform interpolation algorithms implemented in this thesis were originally proposed in [27].

Refutational proof-based solutions are the usual approach of many interpolation algorithms [21, 36, 35]. The approach taken in [27] relies on quantifier-elimination heuristics to construct a uniform 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 [16] 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

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 catastrophe 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 considered practical. On the other hand, for certain applications these contributions are relevant and their proper use fit such work flows uniquely.

Formal methods aim to bring a unique combination of automation, rigor, and efficiency (whenever efficient algorithms exits for the verification task). This thesis discusses a particular problem in software verification known as *the interpolation 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 Background

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 in 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) of such software. In Chapter 2, an extensive review of the formal concept is provided.

Even though 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. Despite the fact that 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*) [12, 34, 25].

- 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 [26] is undecidable [46, 3].

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 refutation proof. The interpolant is constructed using a recursive function over the structure of the proof tree. In [35, 36] 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 satellites 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 interpolation algorithm for the respective theory. In [50] 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 refutation proof to compute interpolants using Pudlak's algorithm. The introduction of the class of *equality interpolating theories* is considered among the

[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.

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 interpolating 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 [21] the authors modify a resolution-based refutation 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 contain AB-mixed literals. This is pointed as an improvement to the approach followed in [50] which executes a similar idea but it is done progressively as the proof-tree is built and does not require *equality propagating* theory solvers [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 [47] where the authors use a linear-inequality solver to provide an interpolant for the theory of linear inequalities over the rational/real numbers ($LIA(\mathbb{Q})/LIA(\mathbb{R})$). Additionally, they integrate the procedure with a *hierarchical reasoning* approach in order to incorporate the signature of theory for (quantifier-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 [50] require that the theory solvers keep track of the interpolating-term and propagate this term whenever possible

- Chapters 3 and 4 explain implementation details of the uniform interpolating algorithms for the theories EUF, UTVPI. Chapter 5 discusses the implementation of a theory combination algorithm using the uniform interpolation algorithm from Chapter 3 and 4. These chapters share the same structure. They start with the algorithms used to solve the interpolation problem, discuss implementation details including diagrams of the architecture of the implemented system, and show a performance comparison 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 follows:

1. Implementation of the interpolation algorithm for the theory EUF proposed in [27].
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 [27].
4. Implementation of the combination procedure for interpolating algorithm proposed in [50] in order to combine the implementations of item 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 [37, 20].

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 [48]. 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 countable 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* $\psi \Box \varphi$ *are formulas where* $\Box \in \{\rightarrow, \wedge, \vee\}$

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

For notation purposes, we compactly represent a tuple $\langle x_1, \dots, x_n \rangle$ *of variables as* $\underline{x}$*. Abusing the notation, a formula of the form* $\forall \underline{x}.\phi(\underline{x})$ *(resp.* $\exists \underline{x}.\phi(\underline{x})$*) denotes the formula* $\forall x_1.\forall x_2 \dots \forall x_n.\phi(x_1, \dots, x_n)$ *(resp.* $\exists x_1.\exists x_2 \dots \exists x_n.\phi(x_1, \dots, x_n)$*)*

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* [20] (or interpretation [48]) 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* $c \in \mathfrak{C}$
- $f^{\mathfrak{J}} \in \{\mathfrak{A}^n \rightarrow \mathfrak{A}\}$ *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 $s : Vars \rightarrow \mathfrak{A}$ 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 $\models_{\mathfrak{I},s}$) as a free extension from atomic predicates (function application of predicates) to general formulas as described in [20]. 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 $\bar{s} : Terms \rightarrow A$ be defined recursively as follows:*

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

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

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:*

- $\models_{\mathfrak{I},s} P(t_1,\dots,t_n)$ *if and only if* $P^{\mathfrak{I}}(\bar{s}(t_1),\dots,\bar{s}(t_n))$
- $\models_{\mathfrak{I},s} \neg\psi$ *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* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \psi \rightarrow \varphi$ *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* $s_{x\mapsto d} : Vars \rightarrow \mathfrak{A}$ *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 satisfied 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)* $\forall x. \forall y. x = y \to y = x$
- *(Transitivity)* $\forall x. \forall y. \forall z. (x = y \land y = z) \to x = z$
- *(Congruence)* $\forall x_1 \ldots \forall x_n . \forall y_1 \ldots \forall y_n . (x_1 = y_1 \land \cdots \land x_n = y_n) \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

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 [50], 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 interpolation 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 *interpolation 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 [14], 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 [30].

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 [35]. 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) [31, 10, 36].

Relaxed definitions are considered to the interpolation problem when dealing with specific theories [50] in a way that interpreted function can be also part of the interpolant. The latter is justified since otherwise, many interpolation formulas might not exist in different theories or the interpolants obtained might not be relevant (for example, lisp programs). This is formalized as follows:

Definition 2.3.1. *[50] 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-specific *interpolant.*

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

2.3.2 Uniform Interpolant

A uniform interpolant is a particular kind of interpolant for an inconsistent pair of formulas. Introduced in [45] as a construction to provide an interpretation for second order intuitionistic propositional logic IpC^2 [2] using intuitionistic propositional logic IpC. Our notion of uniform interpolant is taken from [24] where the authors provide the following definition:

Definition 2.3.2. *Fix a theory T and an existential formula $\exists \underline{e}.\phi(\underline{e}, \underline{z})$; call* residue *of $\exists \underline{e}.\phi(\underline{e}, \underline{z})$ the following set of quantifier-free formulae:*

$$Res(\exists \underline{e}.\phi(\underline{e}, \underline{z})) = \{\theta(\underline{z}, \underline{y}) | T \models \exists \underline{e}.\phi(\underline{e}, \underline{z}) \rightarrow \theta(\underline{z}, \underline{y})\} = \{\theta(\underline{z}, \underline{y}) | T \models \phi(\underline{e}, \underline{z}) \rightarrow \theta(\underline{z}, \underline{y})\}$$

A quantifier-free formula $\psi(\underline{y})$ is said to be a T-uniform interpolant *of $\exists \underline{e}.\phi(\underline{e}, \underline{z})$*

[2]I.e. IpC^2 quantifies over propositional variables.

if and only if $\psi(\underline{y}) \in Res(\exists \underline{e} \phi(\underline{e}, \underline{y}))$ *and* $\psi(\underline{z})$ *implies (modulo T) all the other formulae in* $Res(\exists \underline{e} \phi(\underline{e}, \underline{y}))$.

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 [5].

Regarding the decidability of the theories mentioned in Section 2.2, it is known that EUF is undecidable [5], and the theory of ordered commutative rings is undecidable when the structure uses integers as the domain and the semantics of the arithmetical operations [20] (on the other hand, this theory can be decidable if we keep the same structure but use the reals as the domain [20]). 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 [41, 19, 33].

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 [29, 39] have made

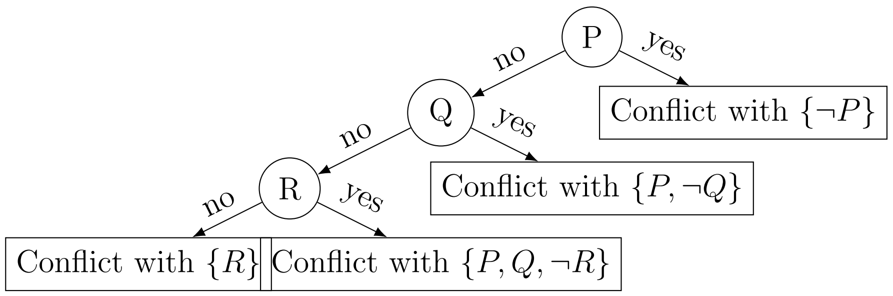

P
Q
R
no
yes
no
yes
no
yes
Conflict with $\{\neg P\}$
Conflict with $\{P, \neg Q\}$
Conflict with $\{R\}$
Conflict with $\{P, Q, \neg R\}$

Figure 2.1: Example of DPLL execution on $\{\{\neg P\}, \{P, Q, \neg R\}, \{R\}, \{P, \neg Q\}\}$

possible to use satisfiability algorithms to solve real-world problems in verification [3].

The DPLL algorithm [15] (and 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 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 [32]. If we are given a boolean combination of formulas from any theory that is capable of deciding the satisfiability of any

[3]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 solved in less that $\mathcal{O}(2^n)$ steps [29].

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

Figure 2.2: Example of resolution proof

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 [50] 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 [13], it is just sufficient 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 sub-terms in the formulas ψ and $u = v$.

The authors in [19, 41] 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 [19] 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 [19] 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 provide the FIND, MERGE operations.

In [44], 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 [44] 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 for merging the two classes into one. Hence, we can recursively recover the rest of the explanation by recursively traversing the sub-paths found.

Additionally, the authors in [44] extended the Congruence Closure algorithm [43] 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 [19]. The latter combines the traditional *pending* and *combine* list into one single list, hence removing the initial *combination* loop in the algorithm in [19].

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 [23] 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 [18]; and *equational propagation* (congruence closure for grounded equations) to update the state of a Union-Find data structure [22] that keeps the minimal equivalence relation defined by grounded equations in the input Horn clauses.

The author in [23] defined two variations of his algorithms by adapting the Congruence Closure algorithms in [19, 41]. 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 [40] the authors defined a procedure to achieve the above problem. The key idea is to *purify* the sub-formulas by including additional constant symbols equating sub-terms 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 integers is not convex. To see the latter consider* $1 \leq x \land x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x \lor 2 = x$. *However, it is not the case that* $1 \leq x \land x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x$ *nor* $1 \leq x \land x \leq 2 \models_{UTVPI(\mathbb{Z})} 2 = x$.

An interpolation combination framework as proposed in [50] 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 working 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 except for the SAT/SMT algorithms. For the latter, PicoSAT/TraceCheck[2] and Z3 [16] 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 the proof format output by PicoSAT/TraceCheck.

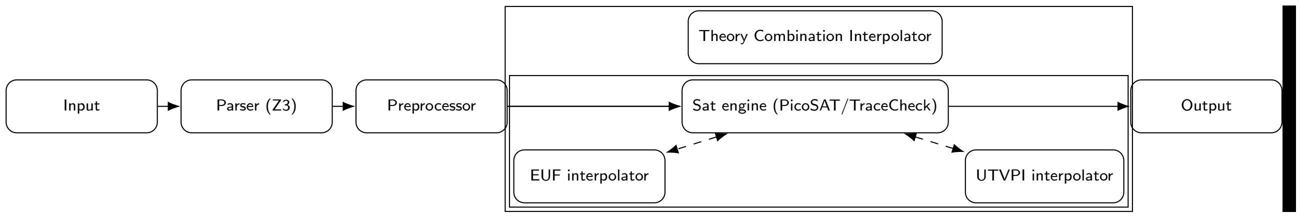


Figure 2.3: General System Diagram

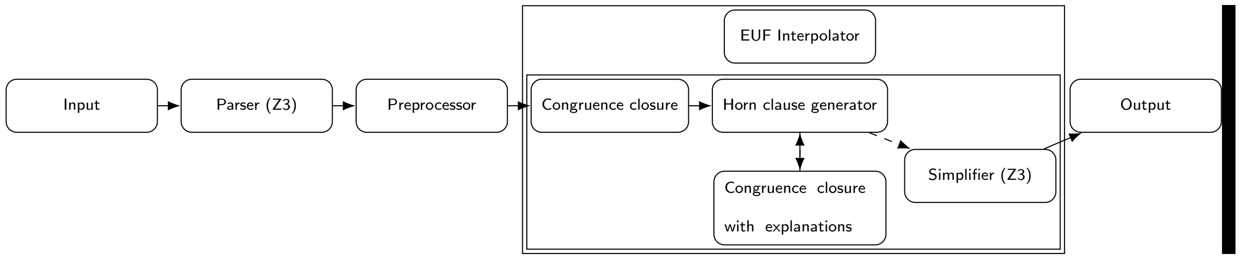


Figure 2.4: EUF Interpolator Diagram

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 software read interpolant formulas based on the order of appearance in a conjunction [36]. 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

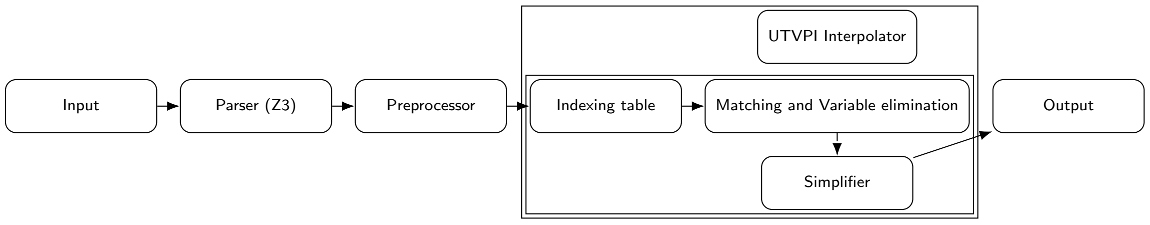


Figure 2.5: UTVPI Interpolator Diagram

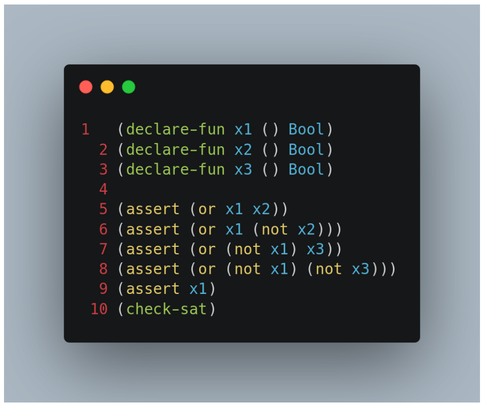
which prefixes the names of uninterpreted symbols with the strings a_, b_, c_ to indicate that the symbol name is either an A-local, B-local, or common symbol respectively. We extended Z3's API with functions that test if a formula is A-local, B-local, AB-pure, AB-common based of the definitions in [50] 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 criterion 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 PicoSAT/TraceCheck Proof Format

We used PicoSAT/TraceCheck to reconstruct a resolution-based proof necessary for Pudlak's algorithm in the interpolation combination component. Given that Z3 provides a user-friendly proof-producing API [17], 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 parser that generates a pdf compiled 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 [4].

[4]Z3 uses the term monotonicity instead of congruence



```
(declare-fun x1 () Bool)
(declare-fun x2 () Bool)
(declare-fun x3 () Bool)

(assert (or x1 x2))
(assert (or x1 (not x2)))
(assert (or (not x1) x3))
(assert (or (not x1) (not x3)))
(assert x1)
(check-sat)
```

Figure 2.6: Problematic SMT query for resolution proofs

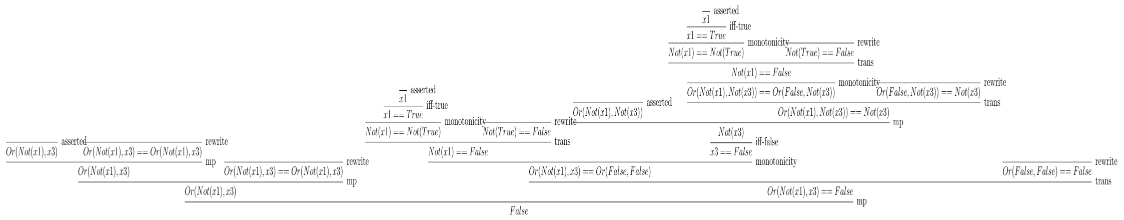


Figure 2.7: Z3 proof of figure 2.6

Thus, we opted to use the PicoSAT SAT solver which implements the DPLL algorithm. PicoSAT, as many SAT solvers, take as input a DIMACS cnf file which consists of a straightforward language to denote clauses. (Optional) Lines starting with the character 'c' denote comments, and a single line starting the string 'p cnf' followed by two positive integers denote the number of variables and number of clauses respectively. The next lines encode clauses. Each line is a list of integers numbers terminating with 0. Positive number encode non-negated literals whose identifier is the number, and negative numbers encode the negated literal whose identifier is the absolute value of the number.

The proof format produced by PicoSAT consists of an ordered list of clauses which are either from the input problem (facts) or learned during the search. Each line is encoded following the DIMACS format, but extended to include an identifying number at the beginning of each line (which identifies the clause), and a second list of integers after the terminating zero of the clause which denotes the non-ordered list of clause identifiers involved in a hyper resolution step; the resulting clause obtained from this step is precisely the clauses encoded by the line. An additional zero is used to terminate the second list of integers.

Example 2.5.0.1. *The following extended proof trace is obtained after running the PicoSAT solver on the cnf file encoding the formula in example 2.1:*

$$\begin{array}{cccccc}
1 & -1 & 0 & 0 & & \\
2 & -3 & 2 & 1 & 0 & 0 \\
3 & 3 & 0 & 0 & & \\
4 & 1 & -2 & 0 & 0 & \\
6 & -2 & 0 & 1 & 4 & 0 \\
7 & 0 & 1 & 6 & 2 & 3
\end{array}$$

Figure 2.8: Example of extended proof trace by PicoSAT for problem 2.1

As we can notice, the first four lines encode the original input clauses; 1 denotes variable P, 2 denotes variable Q, and 3 denotes variable R. Also, these lines do not include any clause identifier in the second list of number since they are facts. The last two lines denote clauses with identifier 6 and 7. Clause 6 is $\neg Q$, which is obtained after the hyper resolution step using clauses 1 and 4. The last line does not contain elements in the first list since this encodes the contradiction clause.

From the previous example we can observe the list of clause identifiers for the hyper resolution step are not ordered in the sense that we cannot produce multiple resolution steps from the representation. In order to obtain the latter one can implement a unit propagation algorithm and discover such ordering.

TraceCheck is a tool incorporated in the previous version of PicoSAT [42] to verify proof traces. It is also able to obtain a pure resolution proof. The presentation of the proof trace is restricted to contain only two clause identifiers in the second list for each clause.

Example 2.5.0.2. *The following extended proof trace is obtained after running the TraceCheck on the previous proof trace:*

```
3   3   0   0
2   1   2  -3  0  0
4   1  -2   0  0
1  -1   0   0
6  -2   0   1  4  0
7   2  -3   0  1  2  0
8  -3   0   7  6  0
9   0   8   3  0
```

Figure 2.9: Example of extended proof trace by PicoSAT for problem 2.1

Chapter 3

Uniform 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 [7], 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 an EUF solver.

Previous work addressing the interpolation problem for EUF has involved techniques ranging from interpolant-extraction from refutation proof trees [35, 36, 49], and colored congruence closure graphs [21]. 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 [20].

3.1 Kapur's Uniform Interpolation Generation Algorithm for EUF

Kapur's interpolation algorithm for the EUF theory uses quantifier-elimination techniques to remove symbols in the first formula of an inconsistent pair of formulas 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 known as uncommon symbols. In preparation to discuss Kapur's algorithm we need to provide the following definitions.

Definition 3.1.1. *Let f be an $n-ary$ function symbol and $a_1, \ldots, a_n, b$ terms from the EUF language. We say*

$$f(a_1, \ldots, a_n) = b$$

is an $f-equation$ if the terms $a_1, \ldots, a_n, b$ are constants in the EUF language. We refer to the outermost symbol of the f-equation *as the function symbol appearing in such $f-equation$.*

$f-equations$ are used in Kapur's algorithm to simplify the structure of terms and in Phase II to expose hidden arguments and eliminate uncommon function symbols.

As part of the input of Kapur's algorithm, there is a set of uncommon symbols given directly or computed from the inconsistent pair of formulas by inspection. Using these symbols we can define the following recursive definition of *uncommon terms*:

Definition 3.1.2. *A term t in the EUF language is* uncommon *if:*

- *t is an uncommon constant*

- *t is a function application of the form $f(t_1, \dots, t_n)$ where either f is an uncommon symbol or any t_i where $1 \leq i \leq n$ is an uncommon term*

Similarly, we can extend homomorphically the notion of uncommon terms to uncommon predicates following a similar construction.

- *Let t_1, t_2 be terms in the EUF language. $t_1 = t_2$, $t_1 \neq t_2$ are uncommon predicates if either t_1 or t_2 are uncommon terms.*
- *Let ψ, φ be predicates in the EUF language. $\psi \star \varphi$ are uncommon predicates if either ψ or φ are uncommon predicates where $\star \in \{\wedge, \vee, \rightarrow\}$. $\neg\psi$ is an uncommon predicate if ψ is an uncommon predicate.*

Regarding extensions of the language with new constants, the *uncommon* property is preserved under equalities. Formally, we mean the following:

Definition 3.1.3. *Let $\mathfrak{L}$ be a EUF language and $\mathfrak{a}$ a constant symbol not belonging to $\mathfrak{L}$. We say $\mathfrak{a}$ is a* common constant under the theory $\mathcal{T}$ in the extended language $\mathfrak{L} \cup \{\mathfrak{a}\}$ *if there exists a common term t in the language $\mathfrak{L}$ such that $\models_{\mathcal{T}} t = \mathfrak{a}$, otherwise $\mathfrak{a}$ is an* uncommon constant*.*

Since congruence closure algorithms are relevant to Kapur's algorithm, we introduce the following definition for notation purposes:

Definition 3.1.4. *Let $\mathcal{E}$ be an equivalence relation between grounded terms of some language $\mathfrak{A}$. The function $repr_{\mathcal{E}} : \mathfrak{A} \rightarrow \mathfrak{A}, repr_{\mathcal{E}} : a \mapsto b$ where b is the representative element in $\mathcal{E}$ for a.*

The interpolating formula produced by Kapur's algorithm is a conjunction of equations and Horn clauses. A Horn clause if a disjunction of literals which contain at most one non-negated literal. Due to its relevance in the procedure in the

EUF theory, for a Horn clause h we denote $antecedent(h)$ to be the conjunction of disequations in the disjunction of h and $head(h)$ to be either the equation in the disjunction if such is present in h or the particle $\perp$ otherwise.

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

- **Flattening:** For each sub-term t in the input formula assign a fresh unique constant $\mathfrak{a}_t$. Additionally, for each sub-term t generate new equations of the form:

  - $c = \mathfrak{a}_c$, if t is a constant c
  - $f(\mathfrak{a}_{t_1}, \ldots, \mathfrak{a}_{t_n}) = \mathfrak{a}_{f(t_1,\ldots,t_n)}$, if t is a function application of the form $f(t_1, \ldots, t_n)$

  Clearly, we can see that this step generates $f - equations$.

- **Elimination of uncommon terms using congruence closure:** This step builds an equivalence relation $\mathcal{E}$ of the $f - equations$ introduced in the Flattening step using a congruence closure algorithm such that the representatives are common terms whenever possible. Uncommon terms appearing in the current conjunction of equations are replaced by their representatives.

- **Horn clause generation by exposure:** This step produces for all pairs of $f - equations$ $(f(\mathfrak{a}_1, \ldots, \mathfrak{a}_n) = \mathfrak{c}, f(\mathfrak{b}_1, \ldots, \mathfrak{b}_n) = \mathfrak{d})$ Horn clauses of the form $\bigwedge_{i=1}^{n}(repr_{\mathcal{E}}(\mathfrak{a}_i) = repr_{\mathcal{E}}(\mathfrak{b}_i)) \rightarrow repr_{\mathcal{E}}(\mathfrak{c}) = repr_{\mathcal{E}}(\mathfrak{d})$ when any of the two following situations happen [1]:

  - The outermost symbol of the $f - equations$ is an uncommon symbol.

[1] Trivial equations in the antecedent of a Horn clause are removed; if the head equation of Horn clause produced in this step is trivial then such Horn clause is discarded

- There is at least one constant argument in any of the $f-equations$ that is an uncommon constant.

- **Conditional elimination:** We identify the Horn clauses $h := \bigwedge_i (c_i = d_i) \to a = b$ that have *common antecedents* and uncommon head equations. We perform the following procedure:

  - if a and b are both uncommon terms: replace the equation $a = b$ appearing in the antecedents of all the current Horn clauses by $antecedent(h)$.
  - if either a is common and b uncommon: replace b by a in all the current Horn clauses $h^{'}$ and append $antecedent(h)$ to $antecedent(h^{'})$.
  - if either a is uncommon and b common: Proceed similarly as in the previous case.

  We repeat this step until we cannot produce any new Horn clauses.

- **Conditional replacement:** For each Horn clause of the form $\bigwedge_i (a_i = b_i) \to u = c$ where the antecedent is common, the term u in its head equation is an uncommon term, and the term c is a common term, replace every instance of u appearing in each $f-equation$ by c to generate Horn clauses with antecedent $\bigwedge_i a_i = b_i$.

  Return the conjunction of formulas obtained as the interpolant.

If the user is not interested in an explicit interpolant, we can present a **lazy/pseudo interpolant** which is an ordered sequence of the original equations together with the Horn clauses produced in Phase II using an appropriate order between the uncommon terms. In order to provide a description of a procedure for the latter we need to introduce the following definitions:

Definition 3.1.5. *Let $\succ$ be a partial order between terms such that $a \succ b$ whenever a is common and b is uncommon. A* dependency pair *for a horn clause $h := \bigwedge_i (a_i =$*

$b_i) \to c = d$ is a pair $(min(c, d, \succ), \{max(c, d, \succ)\} \cup \{u | u$ is an uncommon term appearing in antecedent(h) $\})$. The first element of a dependency pair is denoted as the target of h and the second element the source of h.

A valid dependency pair *is a dependency pair for some Horn clause h which its target is not included in its source.*

We can notice from definition 3.1.5 than even for equations, its source is never empty.

Definition 3.1.6. *Let $\succ$ be a partial order between terms such that $a \succ b$ whenever a is common and b is uncommon. Given a set of Horn clauses H, a* dependency graph for H *is the directed graph $G_H = (V, E)$ where*

- $V := \{(target_i, source_i) | (target_i, source_i)$ *is a valid dependency pair from* $H\}$
- $E := \{(target_i, source_i) \to (target_j, source_j) | \{target_i\} \cup source_i \subseteq source_j\}$

Using *valid dependency pairs* of the Horn clauses produced in Phase II we can construct an acyclic directed graph as shown by the following theorem:

Theorem 3.1.1. *For any set of Horn clauses H, its dependency graph never contains a cycle between its nodes.*

Proof. Suppose there exists a sequence of n nodes such that $(target_1, source_1) \to \cdots \to (target_n, source_n) \to (target_1, source_1)$. Since $\subseteq$ is transitive we can conclude that $target_1 \in source_1$, which leads to a contradiction since all the nodes in a dependency graph are valid dependency pairs. □

Thus, given a set of Horn clauses H, we can compute its dependency graph and use a topological sort algorithm to produce the ordered sequence required in the lazy interpolant representation. Lazy interpolants avoid the possible exponential size of the formal 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 sub-formula 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.

In order to efficiently work with Horn clauses during and after the Conditional elimination step in Kapur's algorithm, [28] introduced *conditional congruence closure* as an extension of the congruence closure generated by a conjunction of equalities. This structure includes Horn clauses in the set of consequences of the theory induced by the input formulas, allowing the membership checking of Horn clauses as well.

Definition 3.1.7. *Let S be a set of equations in the EUF language $\mathfrak{L}$, and $CC(S)$ the set of consequences of S using congruence closure. Then the* conditional congruence closure of S*, abbreviated as $CCC(S)$, is defined as follows:*

$$H \to a = b \in CCC(S) \text{ if and only if } a = b \in CC(S \cup H)$$

where H is a conjunction of equations and a, b terms in $\mathfrak{L}$.

3.2 Implementation

The description of the interpolation algorithm presented in the previous section suggests a straight forward implementation of the first two stages using well-known algorithms [41, 19] and data structures from the SMT solver Z3 [16] to represent elements from the EUF language. One particular change was required in the congruence closure algorithm since Kapur's algorithm keeps common terms as representatives whenever common terms belong to a partition of terms induced by the equivalence relation.

The algorithm in [19] uses a union-find data structure to encode the equivalence classes of the nodes in the abstract syntax tree of the input formula with a 'modify the smaller subtree' strategy. This means that when two nodes u, v in the abstract syntax tree are meant to be merged, the representative of the new combined equivalence class is the node which has a bigger number of predecessors nodes in the abstract syntax tree pointing to the equivalence class of the node. The idea was to update the least amount of nodes that possibly can change representatives due to the most recent merge operation of the equivalence classes and congruence.

Notation 3.2.1. *Given a theory $\mathcal{E}$ and a term u in the language of $\mathcal{E}$, we can indicate by $[u]$ the equivalence class induced by $\mathcal{E}$, i.e. $[u]_{\mathcal{E}} = \{v \in TERMS \mid \models_{\mathcal{E}} u = v\}$.* [2]

Our algorithm uses a difference partial order to maintain common terms as representatives of the equivalence classes. The non-reflexive relation $\succ_{common}$ [3] is defined for all nodes u, v in the abstract syntax tree of terms as:

$$u \succ_{common} v = \begin{cases} |list(u)| > |list(v)| & \text{if } (u \text{ is a common term} \Leftrightarrow \\ & v \text{ is a common term}) \\ \text{u is a common term} & \text{otherwise} \end{cases}$$

where $list(u) = \{f(u_1, \ldots, u_n) \in \text{TERMS} \mid \exists i \in \{1, \ldots, n\} \text{ such that } u_i \in [u]\}$

In the next section we will discuss the changes proposed to Phase III in Kapur's algorithm.

[2]If the theory is clear from context, the notation $[u]$ denotes the equivalence class of u.

[3]The reflexive relation $u \succeq_{common} v$ is defined as $u = v \vee u \succ_{common} v$, where the equality between nodes is defined as $|list(u)| = |list(v)| \wedge u$ is a common term $\iff v$ is a common term.

3.2.1 New conditional elimination step in Kapur's algorithm

The modification of Phase III implemented in this thesis work combines and extends the algorithms and data structures introduced in [23, 44]. The algorithm in [23] is a direct extension of [18] which adapts a congruence closure algorithm from [41, 19] in order to update the union-find data structure maintaining the equivalence relation between all the sub-terms in the input formula. The implementation of the congruence closure algorithm in [44] extends the usual $Find, Merge$ operations on the union-find data structure with the $Explain$ operator, which accomplishes the following:

> $Explain(e, e^{'})$: if a sequence U of unions of pairs (previous Merge operations) $(e_1, e_1^{'}), \ldots, (e_p, e_p^{'})$ has taken place, it returns a subset E of U if $(e, e^{'})$ belongs to the equivalence relation generated by E and it returns $\perp$ otherwise.

If the EUF signature of the input equations does not contain functional symbols, the $Explain$ operation from [44] returns the minimal subset of E. Nonetheless, such assertion is not guaranteed when the signature includes functional symbols.

The motivation behind the combination of Gallier's data structure and the congruence closure algorithm with explanations is twofold:

1. First, we want to introduce common equations from the antecedents of the Horn clauses obtained during Phase II to a conditional equivalence relation as well as updating the conditional equivalence relation structure by using the equation propagation mechanism from the congruence closure algorithm and the implicational propagation component from Gallier's structure.

2. Additionally, we want to find all the common Horn clauses provable from the original input of equations. The Explain operator in [44] is recursively

used to construct the antecedent of such Horn clauses during the conditional replacement step. The Explain operator traverses a proof-tree data structure containing the nodes that were used to combine equivalence classes in the underlying union-find data structure. Thus, the MERGE operation in [23] is required to update the proof tree [4] from [44] as well.

The thesis work accomplishes the previous points by implementing the following:

1. In addition to the parsing procedure and initialization of the Gallier data structure, we assert into the union-find data structure every common equation in the antecedent of a Horn clause.

2. The implemented C++ class for the congruence closure with explanation [5] includes a pointer as data member to the class implementation for the Gallier data structure in order to propagate the equational information achieved during merges and updates due to congruence. The modified algorithms appear below in pseudo-code notation:

3. We implemented the *ExtendedExplain* procedure, which given as input an equation $a = b$ and a conditional equivalence relation $\mathcal{E}$ obtained from a set of equations and a set of Horn clauses in an EUF language $\mathfrak{L}$, it returns a list C of common equations in $\mathfrak{L}$ such that $\models_{\mathcal{E}} (\bigwedge_{h \in C} h) \rightarrow a = b$ if the equation $a = b$ is belongs to the conditional equivalence relation $\mathcal{E}$; and it returns an empty list otherwise. The pseudo-code for the ExtendedExplain procedure is shown below:

Using the modified Gallier algorithm for Unsatisfiability testing of grounded Horn clauses endowed with the congruence closure with explanations procedure we compute the conditional congruence closure (Phase III of Kapur's algorithm) induced by a set of Horn clauses H as follows:

[4]This structure is used in order to retrieve explanations.
[5]A fragment of the actual code is shown at Section 6.3.

Algorithm 1 Modified Unsatisfiability Testing for Ground Horn Clauses

```
procedure SATISFIABLE(var H : Hornclause; var queue, combine: queuetype;
var GT(H) : Graph; var consistent : boolean)
    while queue not empty do
        node := pop(queue);
        for $clause_1$ in H[node].clauselist do
            if numargs[$clause_1$] = 0 then
                nextnode := poslitlist[$clause_1$];
                if $\neg$ H[nextnode].val then
                    if nextnode $\neq \bot$ then
                        queue := push(nextnode, queue);
                        H[nextnode].val := true;
                        u := left(H[nextnode].atom);
                        v := right(H[nextnode].atom);
                        if FIND(R, u) $\neq$ FIND(R, v) then
                            combine := push((u, v), combine);
                        end if
                    else
                        consistent := false;
                    end if
                end if
            end if
        end for
        if queue is empty then
            closure(combine, queue, R);
        end if
    end while
end procedure

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

Conditional Congruence Closure Algorithm:

Step 1. Let $\mathcal{E}$ be an empty equivalence class for all the terms in the term tree of the Horn clauses in H, i.e. there is an equivalence class for each term in H.

Step 2. Insert all the Horn clauses in H to the Gallier data structure and update $\mathcal{E}$ according to Gallier's algorithm.

Algorithm 2 Modified Congruence Closure with Explanation Algorithms - Merge

```
procedure MERGE(R : partition, u, v : node; queue, combine : queuetype)
    if $u$ and $v$ are constants $a$ and $b$ then
        add $a = b$ to Pending;
        Propagate();
    else                                  ▷ $u = v$ is of the form $apply(a_1, a_2) = a$
        if $Lookup(Representative(a_1), Representative(a_2))$ is some $apply(b_1, b_2) = b$ then
            add $(apply(a_1, a_2) = a, apply(b_1, b_2) = b)$ to Pending;
            Propagate();
        else
            set Lookup(Representative($a_1$), Representative($a_2$)) to $apply(a_1, a_2)$=$a$;
            add $apply(a_1, a_2)$=$a$ to UseList(Representative($a_1$)) and to UseList(Representative($a_2$));
        end if
    end if
end procedure
```

Step 3. For all the common equations $a = b$ in the Horn clauses $h \in H$, Merge a and b in $\mathcal{E}$. Update Gallier's data structure accordingly.

Step 4. Return $\mathcal{E}$ as the conditional congruence closure for H.

We notice that the algorithm proposed to compute the conditional congruence closure does not call the Explain operator of the Congruence Closure with Explanation since the latter will be used to construct common Horn clauses from induced conditional equivalence class. The Find and Merge operation takes $\mathcal{O}(1)$ amortized cost. Thus, the amount of work related to the congruence closure in the proposed algorithm take $\mathcal{O}(n \log n)$ time complexity where n denotes the number of equations in the graph term of the Horn clauses H. However, the amount of work required by the *unionupdate* operation of Gallier data structure is $\mathcal{O}((2m + n)\lfloor n \rfloor + 1)$ where m is the number of nodes in the graph representation of the Horn clauses H. In the worst-case scenario there can be at most $O(n^2)$ of these Horn clauses due to the Phase II of Kapur's algorithm. Therefore, the time complexity of the conditional congruence closure algorithm proposed is $\mathcal{O}((m^2 + n) \log n)$.

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($a_1$, $a_2$) = $a$, apply($b_1$, $b_2$) = $b$) from Pending
 4:         if Representative($a$) $\neq$ Representative($b$) and w.l.o.g. |ClassList(Representative($a$))| $\leq$ |ClassList(Representative($b$))| then
                $old_{repr_a}$ := Representative(a);
 6:             Insert edge $a \rightarrow b$ labelled with E into the proof forest;
                for each c in ClassList($old_{repr_a}$) do
 8:                 set Representative(c) to Representative(b)
                    move c from ClassList($old_{repr_a}$) 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($c_1$, $c_2$) = c in UseList($old_{repr_a}$) do
                    if Lookup(Representative($c_1$), Representative($c_2$)) is some apply($d_1$, $d_2$) = d then
22:                     add (apply($c_1$, $c_2$) = c, apply($d_1$, $d_2$) = d) to Pending;
                        remove apply($c_1$, $c_2$) = c from UseList($old_{repr_a}$);
24:                 else
                        set Lookup(Representative($c_1$), Representative($c_2$)) to apply($c_1$, $c_2$) = c;
26:                     move apply($c_1$, $c_2$) = c from UseList($old_{repr_a}$) to UseList(Representative(b));
                    end if
28:             end for
            end if
30:     end while
    end procedure
```

Remark: the only equations asserted into the conditional equivalence class of the algorithm above are only the common equations from antecedents in Horn clauses from H.

Algorithm 4 Auxiliary function - ExtendedExplain

```
    procedure ExtendedExplain( $t_1$ : TERMS, $t_2$ : TERMS, $H$ : Horn clauses, $\mathcal{E}$ : Conditional Equivalence Relation)
2:      if $Find(t_1, \mathcal{E}) \neq Find(t_2, \mathcal{E})$ then
            throw  Error: $t_1, t_2$ do not belong to the same equivalence class
4:      end if
        if $t_1.id() = t_2.id()$ then
6:          return $\{\}$
        end if
8:      $\{c, u\}$ = Explain($t_1$, $t_2$, $\mathcal{E}$) where $c$ is a list of common equation and $u$ is a list of uncommon equations
        return $c \cup \bigcup\{ExtendedExplain(a, b, H, \mathcal{E}) | consequent \in u, a = b \in antecedent, (\bigwedge antecedent) \to consequent \in H\}$
10: end procedure
```

3.2.2 Invariants of the proposed conditional elimination step

For the next lemmas, unless stated otherwise, let H be the Horn clauses obtained after executing Phase II of Kapur's algorithm, and $\mathcal{E}$ be the conditional equivalence class computed by 3.2.1 using the Horn clauses H.

We can prove that if the equation $t_1 = t_2$ belongs to the conditional equivalence class E, then ExtendedExplain returns a list of common equations C such that $\models_{\mathcal{E}} (\bigwedge_{h \in C} h) \to a = b$ using the axiomatization introduced in 2.2.1 as inference rules with the following lemmas:

Lemma 3.2.1. *Let $\mathcal{L}$ be some EUF language, and t_1, t_2 terms in $\mathcal{L}$ such that $\models_{\mathcal{E}}$ $t_1 = t_2$. If $\exists$ uncommon $a = b \in Explain(t_1, t_2)$, then $\exists h \in H$ such that h is of the form $\bigwedge_i (c_i = d_i) \to a = b$.*

Proof. By the definition of the Explain operator in 3.2.1, we see that the such operator will return a list of all the asserted equations in the conditional equivalence relation. By inspection of the proposed algorithm, assertions into the conditional equivalence relation only happen in two places: when an equation in the antecedent of a horn clause of H is common, and while updating the underlying union-find data structure when the *numargs* field of a Horn clause becomes zero. The latter assert the consequent of such Horn clause. Since u is not common, then such Horn clause should exist in H. □

Lemma 3.2.2. *Let $\mathcal{L}$ be some EUF language and t_1, t_2 terms in $\mathcal{L}$ such that $\models_{\mathcal{E}} t_1 = t_2$.*

The list of equations returned by ExtendedExplain$(t_1, t_2, H, \mathcal{E})$ contains only common equations.

Proof. If t_1, t_2 belong to the same equivalence relation, then there exists a closed derivation (proof tree) with $t_1 = t_2$ as root node and the asserted equations introduced by the modified algorithm as leaves.

The proof proceeds by induction on the complexity of the derivation:

- Case 1. The last inference rule in the proof tree was Reflexivity: The ExtendedExplain algorithm returns an empty list. Thus, the statement is vacuously true.
- Case 2. The last inference rule in the proof tree was Symmetry: Line 8 of ExtendedExplain returns a list of common equations c and a list of uncommon equations u. By Lemma 3.2.1 we see that there exists a Horn clause in H for every uncommon equation in u. Line 9 of ExtendedExplain applies recursively to each of these equations. Since these equations belong to the conditional equivalence relation, there exists a derivation for the latter. By the inductive invariant, ExtendedExplain should return a list of common equation for these subproofs. Thus, the final list contains only common equations.
- Case 3. The last inference rule in the proof tree was Transitivity: The proof follows similarly to the previous case applied to the two subproofs for the Transitivity rule.

- Case 4. The last inference rule in the proof tree was Congruence: The proof follows similarly to the previous case applied n times for each subproof in the Congruence rule.

$\square$

Corollary 3.2.2.1. *Let H be a set of Horn clauses produced by Phase II of Kapur's algorithm, $\mathcal{E}$ the equivalence relation obtained in Phase I of Kapur's algorithm, and $\mathcal{E}'$ the conditional equivalence relation obtained after the modified conditional elimination algorithm. Then $H \models_{\mathcal{E}'} a = b$ if and only if $\exists$ common $x \subseteq \bigcup_{h \in H}$ antecedent(h) such that $H \models_{\mathcal{E}} \bigwedge x \to a = b$.*

Example 3.2.0.1. *Let us consider the input equations $\{1.f(f(x)) = f(x)$, $2.x = f(f(x)), 3.f(a) \neq a\}$ with the set of symbols to eliminate to be $\{f\}$. Flattening introduces the following new equisatisfiable equations: $\{4.f(x) = e_1, 5.f(e_1) = e_2, 5.f(a) = e_3, 6.x = e_2, 7.e_2 = e_1, 8.e_3 = a \to \bot\}$. Phase II introduces the following Horn clauses H: $\{9.x = e_1 \to e_1 = e_2, 10.x = a \to x = e_3, 11.e_1 = a \to e_2 = e_3\}$.*

Our algorithm notices that Horn clause 10. contains only common equations in the antecedent. Hence, $x = a$ is added to the conditional congruence closure $\mathcal{E}$. With the latter $x = e_3$ is also added by the numargs *of Horn clause 10 reaches 0. By transitivity, our algorithm includes $e_3 = a$ since both $x = e_3, x = a$ belong to $\mathcal{E}$. The last equation decreases the* numargs *entry of the Horn clause 8. Thus, $\bot$ in $\mathcal{E}$.*

Using ExtendedExplain, we can compute the common antecedent x for the Horn clause in Corollary 3.2.2.1 using ExtendedExplain as follows:

- *ExtendedExplain*$(e_3 = a) \to \bot$
- $\bigwedge\{$*ExtendedExplain*$(x = e_3)$, *ExtendedExplain*$(x = a)\} \to \bot$, *since $x = e_3$ was added to $\mathcal{E}$ by the congruence closure, $x = a$ was added to $\mathcal{E}$ at the beginning*

of the algorithm because it is a common equation in the antecedent of a Horn clause in H, and $Explain(e_3 = a) = \{x = e_3, x = a\}$

- $\bigwedge(\{x = a\} \cup \{x = a\}) \to \bot$, *since* $Explain(x = e_3) = \{x = a\}$
- $x = a \to \bot$, *after simplification*

3.2.3 New conditional replacement step in Kapur's algorithm

Once the conditional equivalence relation is built after the execution of the previous step, we can compute conditional replacements as follows. For the latter, we will require the following auxiliary functions:

Algorithm 5 Auxiliary function - Candidates

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

Algorithm 6 Auxiliary function - allCandidates

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

By inspection is easy to notice that *Candidates* and *allCandidates* return a set of common terms and a set of sets of common terms respectively.

The proposed conditional replacement step produces common Horn clauses from previous uncommon equations and uncommon Horn clauses obtained in Phase II of Kapur's algorithm. The pseudo-code of the algorithms to process the original equations are shown below:

Algorithm 7 Conditional Replacement - Part 1

```
procedure CONDITIONAL REPLACEMENT(z3::expr const & x, z3::expr const & y, H : Horn clauses, $\mathcal{E}$ : conditional equivalence relation )
    if x is constant and y is constant then
        for $\sigma_x$ in Candidates(x) do
            for $\sigma_y$ in Candidates(y) do
                horn_clause.add(ExtendedExplain(x, $\sigma_x$, H, $\mathcal{E}$) + ExtendedExplain(y, $\sigma_y$, H, $\mathcal{E}$), $\sigma_x = \sigma_y$)
            end for
        end for
    end if
    if x is constant and y is of the form $f_y(t_1^{'}, \dots, t_{k_2}^{'})$ then
        for $\sigma_x$ in Candidates(x) do
            for $\sigma_{f_y}$ in Candidates($f_y(t_1^{'}, \dots, t_{k_2}^{'})$) do
                horn_clause.add(ExtendedExplain(x, $\sigma_x$, H, $\mathcal{E}$) + ExtendedExplain($f_y(t_1^{'}, \dots, t_{k_2}^{'}$, H, $\mathcal{E}$), $\sigma_y$), $\sigma_x = \sigma_{f_y}$)
            end for
            for $arguments_{f_y}$ in CartesianProd(AllCandidates($f_y(t_1^{'}, \dots, t_{k_2}^{'})$)) do
                horn_clause.add(ExtendedExplain(x, $\sigma_x$, H, $\mathcal{E}$) + $\sum_{i=1}^{k_2}$ ExtendedExplain($t_i^{'}$, $arguments_{f_y}[i]$, H, $\mathcal{E}$), $\sigma_x = f_y(arguments_{f_y})$)
            end for
        end for
    end if
    if x is of the form $f_x(t_1, \dots, t_{k_1})$ and y is a constant then
        return CONDITIONAL ELIMINATION(y, x);
    end if
end procedure
```

Lemma 3.2.3. *Let* $Eqs := \{f(\mathfrak{a}_{i,1}, \dots, \mathfrak{a}_{i,n}) = \mathfrak{a}_i\}$ *be the set of equations produced in Phase I of Kapur's algorithms, H the set of Horn clauses obtained in Phase II, and* $\mathcal{E}^{'}$ *the conditional equivalence relation obtained by the proposed conditional replacement step. Executing the proposed conditional replacement step on* $Eqs, H, \mathcal{E}$ *produces common Horn clauses.*

Proof. By inspecting the conditional replacement algorithm in 3.2.3, we noticed the

Algorithm 8 Conditional Replacment - Part 2

```
    procedure CONDITIONAL REPLACEMENT(z3::expr const & x, z3::expr const & y, H : Horn clauses, $\mathcal{E}$ : conditional equivalence relation)
 2:     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 $\sigma_{f_x}$ in Candidates($f_x(t_1, \ldots, t_{k_1})$) do
 4:             for $\sigma_{f_y}$ in Candidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) do
                    horn_clause.add(ExtendedExplain($f_x(t_1, \ldots, t_{k_1})$, $\sigma_{f_x}$, H, $\mathcal{E}$) + ExtendedExplain($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$, $\sigma_y$, H, $\mathcal{E}$), $\sigma_{f_x} = \sigma_{f_y}$)
 6:             end for
                for $arguments_{f_y}$ in CartesianProd(AllCandidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) do
 8:                 horn_clause.add(ExtendedExplain($f_x(t_1, \ldots, t_{k_1})$, $\sigma_{f_x}$, H, $\mathcal{E}$) + $\sum_{i=1}^{k_2}$ ExtendedExplain($t_i^{'}$, $arguments_{f_y}[i]$, H, $\mathcal{E}$), $\sigma_{f_x} = f_y(arguments_{f_y})$)
                end for
10:         end for
            for $arguments_{f_x}$ in CartesianProd(AllCandidates($f_x(t_1, \ldots, t_{k_1})$)) do
12:             for $\sigma_{f_y}$ in Candidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) do
                    horn_clause.add($\sum_{i=1}^{k_1}$ ExtendedExplain($t_i$, $arguments_{f_x}[i]$, H, $\mathcal{E}$) + ExtendedExplain($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$, $\sigma_y$, H, $\mathcal{E}$), $f_x(arguments_{f_x}) = \sigma_{f_y}$)
14:             end for
                for $arguments_{f_y}$ in CartesianProd(AllCandidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) do
16:                 horn_clause.add($\sum_{i=1}^{k_1}$ ExtendedExplain($t_i$, $arguments_{f_x}[i]$, H, $\mathcal{E}$) + $\sum_{i=1}^{k_2}$ ExtendedExplain($t_i^{'}$, $arguments_{f_y}[i]$, H, $\mathcal{E}$), $f_x(arguments_{f_x}) = f_y(arguments_{f_y})$)
                end for
18:         end for
        end if
20: end procedure
```

antecedents of the Horn clauses constructed are obtained using the ExtendedExplain procedure, which from lemma 3.2.2 it produces a list of only common equations.

Additionally, the equations in the consequent part of the Horn clauses produced in the conditional replacement procedure are of the form:

- $\sigma_x = \sigma_y$, where $\exists t_1, t_2 \in TERMS$ such that $\sigma_x \in Candidates(t_1)$ and $\sigma_y \in Candidates(t_2)$
- $\sigma_x = f(arguments_f)$, where $\exists t, f(t_1, \ldots, t_n) \in TERMS$ such that $\sigma_x \in Candidates(t)$ and $f(arguments_f) \in \{f(\hat{t_1}, \ldots, \hat{t_n})|$ f is a common functional symbol , $\hat{t_i} \in$

$Candidates(t_i)$ for $1 \leq i \leq n\}$

- $f_x(arguments_x) = f_y(arguments_y)$, using the same definition for $f(arguments_f)$ from previous item

It is easy to see that the above equations are common if any of the involved sets are not empty. Therefore, the algorithm only produces common Horn clauses. □

In order to process Horn clauses obtained from Phase II of Kapur's algorithm, we apply the conditional replacement step for equations over the equations in the consequent of all the Horn clauses which *numargs* entries are equal to 0.

Lemma 3.2.4. *Let H be the set of Horn clauses obtained at the end of Phase II of Kapur's algorithm with numargs entry equal to 0 and $\mathcal{E}'$ the associated conditional equivalence relation obtained at the end of the conditional elimination procedure. If $h \in H$ is of the form $\bigwedge_i (a_i = b_i) \rightarrow c = d$, then the Horn clauses $\{\bigwedge_i ExtendedExplain(a_i, b_i, H, \mathcal{E}') \rightarrow h' | h' \in ConditionalReplacement(c, d, H, \mathcal{E}')\}$ are common Horn clauses.*

Proof. We observe that if the *numargs* entry of a Horn clause $h \in H$ is equal to 0, then all the equations in the antecedent of h belong to the conditional equivalence relation. Thus, there exists a proof in $\mathcal{E}'$ for each of these equations. From the latter, the consequent of h also belongs to the conditional equivalence relation. Using Lemmas 3.2.2 and 3.2.3 we can conclude that the Horn clause in the statement is a common Horn clause. □

3.2.4 An additional application: checking membership of ground Horn clauses with Explanations

By adding common equations from the antecedent of the Horn clauses during the conditional elimination step we are able to extend an equivalence relation to a conditional equivalence relation that includes the set of consequences of the aforementioned common equations. In a more general setting, we can specify an arbitrary set of equations, or property, that are meant to enlarge the theory to a conditional equivalence relation for specific purposes. To reuse the ExtendedExplain procedure and associated lemmas, we replace the *common* predicate for any predicate of interest.

In particular, given an equivalence relation $\mathcal{E}$, a set of Horn clauses H, and Horn clause h of the form $\bigwedge(a_i = b_i) \to c = d$, we can find an explanation to the query $H \models_{\mathcal{E}} h$ by conditionally extending H with the equations from the antecedent of h and then retrieving an explanation of the membership of the consequent of h in the conditional equivalence relation.

Lemma 3.2.5. *Let H be a set of Horn clauses, h a Horn clause of the form $(\bigwedge_i a_i = b_i) \to c = d$, $\mathcal{E}$ an equivalence relation between terms of a language in EUF, and $\mathcal{E}'$ the associated conditional equivalence relation obtained from extending $\mathcal{E}$ with the equations in the antecedent of h.*

If there exists an explanation $\mathfrak{p} := \{\mathfrak{a}_1 = \mathfrak{b}_1, \ldots, \mathfrak{a}_n = \mathfrak{b}_n\}$ for the query $H \models_{\mathcal{E}'} c = d$, then the explanation $\mathfrak{p}'$ for the query $H \models_{\mathcal{E}} (\bigwedge_i a_i = b_i) \to c = d$ is the subset of $\mathfrak{p}$ that does not contain equations from $\bigcup_{i=1}\{a_i = b_i\}$, i.e. the explanation $\mathfrak{p}'$ does not contain the equations from the antecedent of the Horn clause h.

Proof. If there exists an explanation $\mathfrak{p}$ for $H \models_{\mathcal{E}'} c = d$ then there exists a proof tree with $c = d$ as root node and the equations from $\mathfrak{p}$ as leaves. From lemma 3.2.1 and the definition 3.2.1 we notice that $\mathfrak{p}$ only contains equations derived from $\mathcal{E}$ and the antecedent of h.

Using the *discharge rule* from basic propositional logic, i.e. $\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \to B}$, then we can obtain a proof tree with leaves only containing terms that do not belong to the antecedent of h.

Hence, the new proof tree will have $(\bigwedge_{i=1}^{m} a_i^{'} = b_i^{'}) \to c = d$ as root node and derived equations from $\mathcal{E}$ as leaves, where $\{a_i^{'} = b_i^{'}\} \subseteq \mathit{antecedent}(h)$ for each $1 \leq i \leq m$. Thus, $\models \mathit{antecedent}(h) \to a_i^{'} = b_i^{'}$ for each $1 \leq i \leq m$. Let $\mathfrak{p}^{'}$ be $\mathfrak{p} \backslash \bigcup_{i=1}^{m} \{a_i = b_i\}$. Therefore, by transitivity, we have that $\mathfrak{p}^{'}$ is an explanation in $\mathcal{E}$ for $H \models_{\mathcal{E}} h$.

□

The time complexity of this algorithm follows the reasoning as the complexity analysis for the conditional congruence closure algorithm previously proposed. If n is the number of terms in graph of terms and m is the number of nodes in the graph representation of the Horn clauses H, then the algorithm runs in $\mathcal{O}((m + n) \log n)$ time.

3.3 Evaluation

3.3.1 Detailed evaluation of examples

In this section we discuss in full detail the execution trace of the implementation of some examples.

Symbol elimination example

Let us consider the following example from [27] $\alpha_1 = \{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$ with the set of symbols to eliminate $U_1 = \{u\}$. The implementation produces the following trace (slightly modified for presentation purposes) in order to compute the interpolant of $\alpha_1; U_1$:

```
Before conditionalEliminationEqs
Horn clauses produced
0. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
1. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
2. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
3. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
4. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
5. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
6. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
7. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
8. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
9. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
Number of horn clauses: 10
Executing conditionalElimination
After conditionalEliminationEqs/Before conditionalEliminationHcs
Horn clauses produced
0. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
1. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
2. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
3. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
4. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
5. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
6. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
7. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
8. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
9. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
10. 0x5618a6ca55a0 (Leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
11. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
12. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
13. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
14. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
Number of horn clauses: 15
Executing conditionalEliminationfor Horn clauses
After conditionalEliminationHcs
Horn clauses produced
0. 0x5618a6cabb10 (Leader) (= c_y1 c_y2) and (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_s1 c_s2)
1. 0x5618a6c9fac0 (Leader) (= c_y1 c_y2) and (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
2. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
3. 0x5618a6cab830 (Leader) (= c_z2 c_y1) and (= c_z1 c_y1) -> (= c_s1 c_s2)
4. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
5. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
6. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
7. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
8. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
9. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
```

```
10. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
11. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
12. 0x5618a6ca65b0 (Leader) (= c_z1 c_y2) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
13. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
14. 0x5618a6ca55a0 (Leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
15. 0x5618a6ca6300 (Leader) (= c_z2 c_y2) and (= c_z1 c_y2) -> (= c_s1 c_s2)
16. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
17. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
18. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
19. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
Number of horn clauses: 20
Horn clauses produced
0. 0x5618a6cabb10 (Not leader) (= c_y1 c_y2) and (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_s1 c_s2)
1. 0x5618a6c9fac0 (Not leader) (= c_y1 c_y2) and (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
2. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
3. 0x5618a6cab830 (Not leader) (= c_z2 c_y1) and (= c_z1 c_y1) -> (= c_s1 c_s2)
4. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
5. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
6. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
7. 0x5618a6ca65b0 (Not leader) (= c_z1 c_y2) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
8. 0x5618a6ca55a0 (Not leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
9. 0x5618a6ca6300 (Not leader) (= c_z2 c_y2) and (= c_z1 c_y2) -> (= c_s1 c_s2)
10. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
Number of horn clauses: 11
Interpolant:
(ast-vector
  (=> (and (= z1 y1) (= z1 y2) (= z1 y1) (= z2 y1)) (= t (f s1 s2)))
  (=> (= z1 z2) (= s1 s2))
  (=> (and (= z2 y1) (= z1 y2) (= z1 y1) (= z2 y1)) (= t (f s2 s2)))
  (=> (and (= z2 y1) (= z1 y2)) (= t (f s2 s1)))
  (=> (and (= z1 y1) (= z1 y2)) (= t (f s1 s1))))
```

The interface offered by SMT solvers with interpolation features usually provide the conventional A-part, B-part format. In order to compare the results with the implementation two instances were tested, so we can obtain interpolants from both systems:

- Problem instance: A-part : $\{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$; B-part : $\{z_1 = z_2, s_1 \neq s_2\}$
  - Z3: (or (= s2 s1) (not (= z2 z1)))
  - Mathsat: (not (and (= z1 z2) (not (= s1 s2))))
  - Our implementation: ($\rightarrow$ (= z1 z2) (= s1 s2))

- Problem instance: A-part : $\{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$; B-part : $\{z_1 = y_1, z_1 = y_2, f(s_1, s_1) \neq t\}$
  - Z3: (or (= (f s1 s1) t) (not (= z1 y1)) (not (= y2 y1)))
  - Mathsat: (not (and (not (= t (f s1 s1))) (and (= z1 y1) (= z1 y2))))
  - Our implementation: (→ (and (= y1 y2) (= z1 y2) (= z1 y2)) (= t (f s1 s1))))

Clearly, the interpolant obtained by just eliminating the symbol v implies the outputs produced by Z3 and Mathsat for the previous problem instances.

Simple example with dis-equality

Let us consider another example from [27] $\alpha_2 = \{f(x_1) \neq f(x_2)\}$ with the set of symbols to eliminate $U_2 = \{f\}$. The implementation produces the following trace for $\alpha_2; U_2$:

```
Before conditionalEliminationEqs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
Executing conditionalElimination
After conditionalEliminationEqs/Before conditionalEliminationHcs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
Executing conditionalEliminationfor Horn clauses
After conditionalEliminationHcs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
Horn clauses produced
0. 0x5648882deea0 (Leader) (= c_x2 c_x1) -> false
Number of horn clauses: 1
(ast-vector
  (=> (= x2 x1) false))
```

To compare our result with Z3 and Mathsat we included the B-part formula to be $\{x_1 = x_2\}$. The interpolants obtained by these systems were the same, which was (not (= x1 x2)).

3.3.2 Performance comparison with iZ3 and MathSat

This section discusses a benchmark of interpolant generation for the EUF theory, which it will allow us to test our implementation and contrast the execution time with other interpolant generation algorithms from Z3 and Mathsat.

Benchmark description

The benchmark uses the following parameters:

- i stands for the number of constants
- j stands for the number of function symbols with arity between 2 and 3
- k stands for limit for random terms to consider in the problem
- n stands for the equations/dis-equations in the A-part

The benchmark generates a pair of two unsatisfiable formulas in the EUF language from a fixed theory with the following parameters:

$$S = \{c_1, \ldots, c_i, f_1, \ldots, f_j\}$$

where i, j are random integer numbers. Using the S, we enumerate the grounded terms G in S and assign a natural number to each number denoting its position in the enumeration.

We generate n random equations/dis-equations from the signature S such that this collection of formulas is consistent for the $A - part$ of the input problem. The latter is implemented using a $z3 :: solver$ to ensure this condition. The equations/dis-equations are of the form:

$$position(k_1, S) = position(k_2, S) \text{ , or } position(k_1, S) \neq position(k_2, S)$$

where $position(k, S)$ denotes the k^{th} element in G. The integers k_1, k_2 are chosen uniformly at random from a distribution of integer values $\{0, \ldots, k\}$, where k is a parameter of the benchmark.

Next we randomly generate a second set of consistent equations/dis-equations (B-Part) until the $A-part$ and the $B-part$ are inconsistent using a second $z3 :: solver$.

Experimental results

We designed this problem because it is not trivial to compute a uniform/interpolant due to randomness of the problem. This problem was executed 100 times with parameters (i = 10, j = 5, k = 100, n = 10) and (i = 20, j = 10, k = 100, n = 40) using a computer desktop equipped with an Intel i7-9700 @ 4.70 GHz processor and 16 GB of RAM.

The following graph reports the time needed by our implementation, iZ3, and the interpolation generation algorithm from Mathsat. It is well known that both iZ3 and Mathsat does not compute uniform interpolants. Regardless, the benchmark was used with the purpose to compare their execution time on normal interpolants.

The time was measured using a bash script which takes the difference of the output produced by the UNIX utility $date + `\%s.\%N'$ at the beginning and at the end of the execution of the tested algorithms.

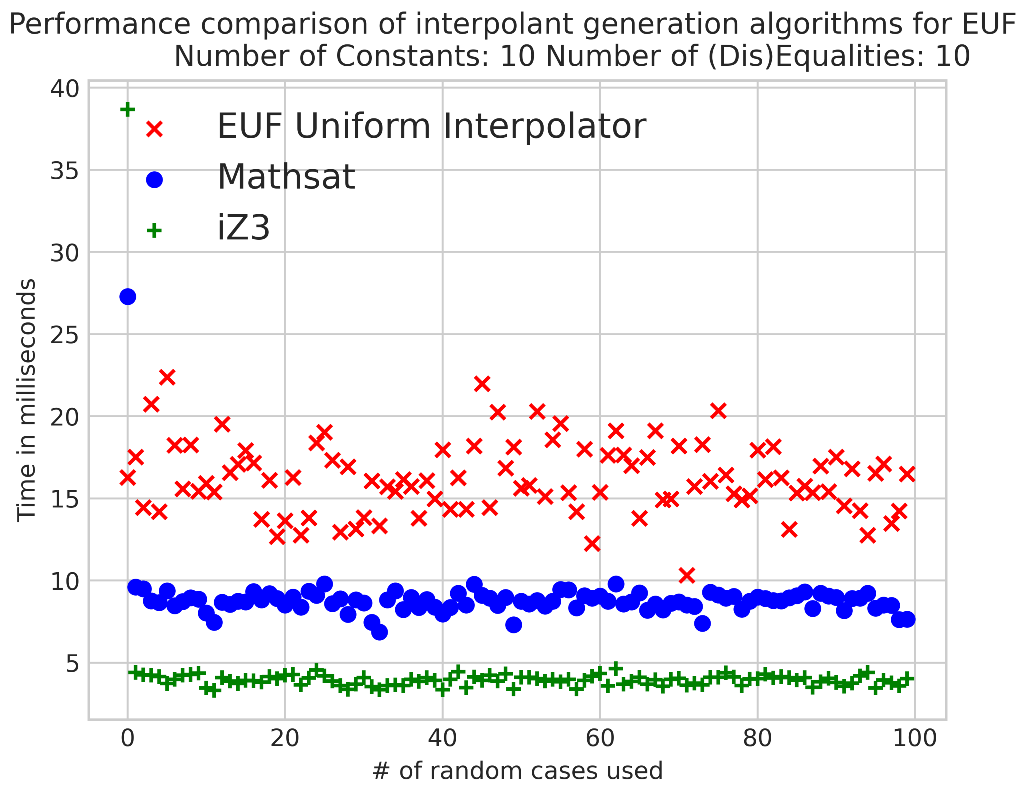


Figure 3.1: Performance comparison graph of EUF interpolant generation algorithms for the benchmark (i = 10, j = 5, k = 100, n = 10)

3.4 Conclusions

This chapter discussed the new approach for uniform interpolant generation introduced by Prof. Kapur for the EUF theory. We proposed a modification of his algorithm and proved its correctness. The implementation and testing work confirms that the approach produces stronger interpolants compared to other interpolant generation algorithms (iZ3, Mathsat).

The performance comparison section indicates that of our implementation appears to be slower than iZ3 and Mathsat. On the other hand, it is important to

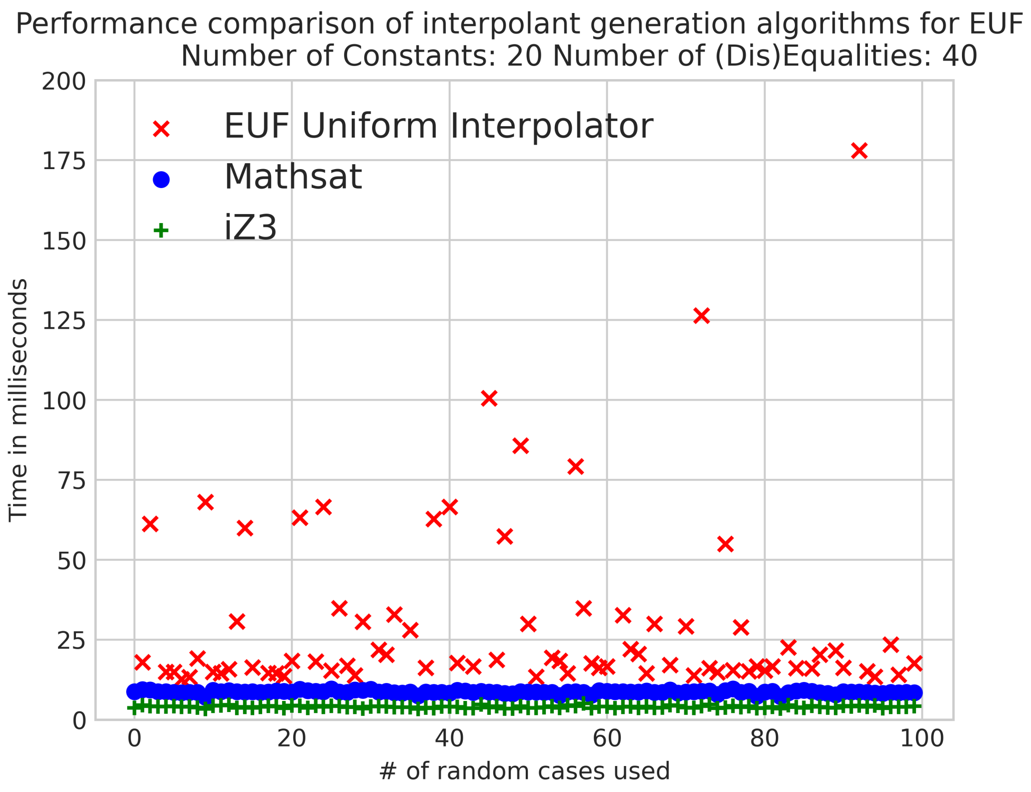


Figure 3.2: Performance comparison graph of EUF interpolant generation algorithms for the benchmark (i = 20, j = 10, k = 100, n = 40)

highlight that the algorithm have different output specification since Prof. Kapur algorithm is able to find a uniform interpolant.

There are several places where the current implementation can be improved. Currently, the data structures for flattening introduce additional constants that might not be needed since the implementation used a recursive scan of the arguments of each expression without checking for term sharing. Moreover, the explanation mechanism might be improved/changed by the proof producing mechanism already available in Z3.

Chapter 4

Uniform Interpolation algorithm for UTVPI Formulas

This theory appears heavily in formal methods dealing with abstract domains, introduced in [38], which have been useful for efficient linear invariant discovery, safety analysis, and static analysis of programs. 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 [33] and as well as for interpolation [11] of the theory.

The algorithm in [27] follows a similar approach to the uniform 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 which idea combines the reduction of UTVPI formulas to difference logic [38] and a cycle detection of maximal size [11].

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

4.1 Kapur's Uniform Interpolant Algorithm for the UTVPI theory

The algorithm proposed [27] uses inference rules to obtain the normal form of the conjunction of inequalities of octagon formulas as well as eliminating the uncommon variables from the A-part of the input formula. The rules are the following:

$$\frac{ax + ax \leq c \qquad a \in \{-1, 0, 1\} \text{ and } x \in Vars}{ax \leq \lfloor \frac{c}{2} \rfloor} \text{ Normalize}$$

$$\frac{s_1x_1 + s_2x_2 \leq c_1 \qquad -s_2x_2 + s_3x_3 \leq c_2}{s_1x_1 + s_3x_3 \leq c_1 + c_2} \text{ Elim}$$

The algorithm normalizes the inequalities at the beginning as a preprocessing step and applies the Normalize and 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 match detection (similar to detecting pivots for resolution in SAT) is important for an efficient implementation.

4.2 Implementation

The signature used by the implementation to encode UTVPI formulas is $\{x_0, x_1, \ldots, x_n, +, -, \leq\}$, thus all the variables are indexed by a natural number. The variable x_0 is a *dummy variable* that acts as a place-holder for 0.

The thesis work introduces the following data structures in order to obtain an efficient implementation:

- Indexing data structure which encodes inequalities of the input formula using natural numbers. The latter is obtained by introducing an effective enumeration on the UTVPI terms in a given UTVPI signature. The latter is represented with a function $Position :$ UTVPI term $\to \mathbb{N}$.

- Array of numbers *Bounds* indexed by the numeral representation of the inequalities representing the minimum bound of the encoded inequality, i.e. given an UTVPI inequality $a_1x_1 + a_2x_2 \leq c$, then $Bounds[Position(a_1x_1 + a_2x_2)] = c$
- Data structure to keep track of the signs of variables to be eliminated in the inequalities for efficient matching.
- Data structure to represent an UTVPI term in normal form endowed with addition and subtraction operations and the *Position* function mentioned in the first item.

4.2.1 Normal Forms and Ordering of UTVPI terms

Normal forms helps us to avoid the ambiguity introduced by the commutativity of addition and neutrality of adding by 0. Also, normal forms allow the data structures mentioned above to keep track of less UTVPI terms. The normal form of a UTVPI term is simply defined by the following function:

$$norm(\pm x_m \pm x_n) = \begin{cases} \pm x_{max(m,n)} \pm x_{min(m,n)} & \text{if } m \neq n \\ \pm x_n + x_0 & otherwise \end{cases}$$

In order to obtain a bijection between UTVPI terms and natural numbers, first we define an ordering on the normal forms of UTVPI terms. We encode the UTVPI term $\pm x_m \pm x_n$ using the point $(\pm m, \pm n) \in \mathbb{Z}^2$ and let *TermToPoint* be a map such that $\pm x_m \pm x_n \mapsto (\pm m, \pm n)$. We define the following orderings relevant to the UTVPI term ordering:

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 UTVPI 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. $\prec_t$) of UTVPI terms are the following:*

- $x_0 + x_0$
- $-x_1 + x_0, x_1 + x_0$
- $-x_2 + x_0, -x_2 - x_1, -x_2 + x_1, x_2 + x_0, x_2 - x_1, x_2 + x_1$
- $-x_3 + x_0, -x_3 - x_1, -x_3 + x_1, -x_3 - x_2, -x_3 + x_2, x_3 + x_0, x_3 - x_1, x_3 + x_1, x_3 - x_2, x_3 + x_2,$

4.2.2 Bijection between normalized UTVPI terms and natural numbers

Let T be the set of normalized UTVPI terms. Using the previous ordering $\succ_t$ on T makes clear that, since $|T| \leq |\mathbb{Z}^2|$, then T is countable and thus there exists a bijection between T and $\mathbb{N}$. In order to construct an explicit bijection, we notice the following facts.

We can arrange ordered partitions P of T such that two normalized UTVPI terms $\pm x_{m_1} \pm x_{m_2}$ and $\pm x_{n_1} \pm x_{n_2}$ belong to the same partition if $|m_1| = |m_2|$.

The example 4.2.1.1 arranged the normalized UTVPI terms into this ordered partitions listing them by items. We notice that each partition contains $2(1+2(i-1))$ elements except the partition containing $x_0 + x_0$ using the following lemma.

Lemma 4.2.1. *The i^{th} ordered partition of P contains $2(1 + 2(i - 1))$ elements.*

Proof. Let A be the i^{th} ordered partition from P. We notice that A contains two disjoint subsets with the same number of elements. The first one is $\{-x_i + x_0\} \cup \{-x_i \pm x_j | 1 \leq j < i\}$ and the second one is $\{x_i + x_0\} \cup \{x_i \pm x_j | 1 \leq j < i\}$. The cardinality of each of these subsets of A is $1 + 2(i - 1)$. Thus, $|A| = 2(1 + 2(i - 1))$. Hence, the assertion holds true. $\square$

Using the previous lemma, it is easy to see that the first element of the i^{th} ordered partition is $2(i-1)^2+1$ by taking the sums of all the cardinalities of all the previous ordered partitions and including the first ordered partition $\{x_0 + x_0\}$. Let t be the n^{th} normalized UTVPI term. The latter means that t belongs to the $\lfloor\sqrt{\frac{n-1}{2}} + 1\rfloor^{th}$ ordered partition. Let $i = \lfloor\sqrt{\frac{n-1}{2}} + 1\rfloor$. Hence, t is of the form $\pm x_i \pm x_j$ for some $j < i$. To find the signs of and index j of t, we can check if $d := n - 2(i - 1)^2 - 1$ (the distance between the t and the first element of its ordered partition) if greater than or equal to $1 + 2(i - 1)$. The last condition just detects if t belongs to the second half or to the first half of the i^{th} ordered partition, which will indicate the first sign of t. The second sign and index j can be obtained by checking the parity of d and by computing $(d - 1)/2$. The formal algorithm for this construction is shown in Algorithm 9 of this subsection.

Similarly, given a normalized UTVPI term $\pm x_m \pm x_n$ we can compute its position in the enumeration using the signs and index information using the previous formulas. The formal algorithm for this construction is shown in Algorithm 10 of this subsection.

This explicit bijection allows us to implement a data structure *Bounds* based on an array of integers extended with $\pm\infty$ to encode upper bounds indexed by the natural number representing the normalized UTVPI term from an UTVPI inequality. For initialization purposes, all the entries in this vector are set to $+\infty$ and these values are updated accordingly to keep track of the minimum possible value for the inequality after the application of the inference rules mentioned at the introduction of the section.

Example 4.2.1.2. *Let us consider the input formula* $\alpha_1 = \{x_1 - x_2 \geq -4, -x_2 - x_3 \geq 5, x_2 + x_5 \geq 4, x_2 + x_4 \geq -3\}$

The normalized input is $\{-x_2 + x_1 \geq -4, -x_3 - x_2 \geq 5, x_5 + x_2 \geq 4, x_4 + x_2 \geq -3\}$.

With the above information, the data structure Bounds initially contains the following entries:

$Bounds[5] = -4,$ *since* $position(-x_2 + x_1) = 5$

$Bounds[12] = 5,$ *since* $position(-x_3 - x_2) = 12$

$Bounds[46] = 4,$ *since* $position(x_5 + x_2) = 46$

$Bounds[30] = -3,$ *since* $position(x_4 + x_2) = 30$

$Bounds[i] = +\infty,$ *where* $i \in \mathbb{N} \setminus \{5, 12, 46, 30\}$

4.3 Evaluation

4.3.1 Detailed evaluation of examples

Let us consider the following example: $\alpha_1 = \{x_1 - x_2 \geq -4, -x_2 - x_3 \geq 5, x_2 + x_6 \geq 4, x_2 + x_5 \geq -3\}; \beta_1 = \{-x_1 + x_3 \geq -2, -x_4 - x_6 \geq 0, -x_5 + x_4 \geq 0\}$.

Algorithm 9 UTVPI constructor

```
procedure UTVPI CONSTRUCTOR(position : integer)
    coefficient1 = 0
    coefficient2 = 0
    varindex1 = 0
    varindex2 = 0
    if position = 0 then
        return
    end if
    varindex1 = $\sqrt{\frac{position-1}{2}} + 1$
    initial_group_position = $2 * (varindex1 - 1)^2 + 1$
    half_size_group = $2 * varindex1 - 1$
    if position ≤ initial_group_position + half_size_group then
        coefficient1 = -1
        if position = initial_group_position then
            coefficient2 = 0
            varindex2 = 0
            return
        end if
        separation = position - initial_group_position
        varindex2 = $\frac{separation-1}{2} + 1$
        if mod separation 2 = 0 then
            coefficient2 = 1
            return
        end if
        coefficient2 = -1
        return
    end if
    coefficient1 = 1
    if position = initial_group_position + half_size_group + 1 then
        coefficient2 = 0
        varindex2 = 0
        return
    end if
    separation = position - initial_group_position - half_size_group - 1
    varindex2 = $\frac{separation-1}{2} + 1$
    if mod separation 2 = 0 then
        coefficient2 = 1
        return
    end if
    coefficient2 = -1
    return
end procedure
```

The output obtained by our implementation is (and (≤ (- (- x6) x1) 0) (≤ (+ (- x6) x3) (- 9)) (≤ (- (- x5) x1) 7) (≤ (+ (- x5) x3) (- 2)))) and produced the following trace for this problem:

Algorithm 10 UTVPI position

```
procedure UTVPI POSITION( $s_1 x_{m_1} + s_2 x_{m_2}$ : UTVPI term)
    initial_group_position = $2 * (m_1 - 1)^2 + 1$
    if $s_1 = -1$ then
        sign_a_offset = 0
    else
        if $s_1 = 0$ then
            return 0
        else
            if $s_1 = 1$ then
                sign_a_offset = $2 * (m_1 - 1) + 1$
            end if
        end if
    end if
    if $s_2 = -1$ then
        sign_b_offset = $1 + 2 * (m_2 - 1)$
    else
        if $s_2 = 0$ then
            sign_b_offset = 0
        else
            if $s_2 = 1$ then
                sign_b_offset = $2 * m_2$
            end if
        end if
    end if
    return initial_group_position + sign_a_offset + sign_b_offset
end procedure
```

```
Processing
(+ (- c_x1) a_x2)
Updating structure with
x_2 - x_1 <= 4
Processing
(+ a_x2 c_x3)
Updating structure with
x_3 + x_2 <= -5
Processing
(- (- a_x2) c_x6)
Updating structure with
- x_4 - x_2 <= -4
Processing
(- (- a_x2) c_x5)
Updating structure with
- x_5 - x_2 <= 3
Removing this var: x_0
Removing this var: x_1
Removing this var: x_2
Reducing x_2 - x_1 and - x_4 - x_2
Result: - x_4 - x_1
Reducing x_2 - x_1 and - x_5 - x_2
Result: - x_5 - x_1
Reducing x_3 + x_2 and - x_4 - x_2
Result: - x_4 + x_3
```

```
Reducing x_3 + x_2 and - x_5 - x_2
Result: - x_5 + x_3
Removing this var: x_3
Removing this var: x_4
Removing this var: x_5
Interpolant:
(ast-vector
  (<= (- (- x6) x1) 0)
  (<= (+ (- x6) x3) (- 9))
  (<= (- (- x5) x1) 7)
  (<= (+ (- x5) x3) (- 2)))
```

The Z3 SMT solver provides mechanisms to modify the arithmetic engine and several plug-ins to specialize algorithm for specific theories. It contains a proper specialization to work with UTVPI queries for satisfiability checking. However, the interpolation APIs do not include mechanisms to specialize the interpolation algorithm for UTPVI formulas, i.e. Z3 does produce interpolant in the UTVPI theory. Thus, the interpolant obtained by Z3 for the above problem is : (and ($\leq$ 9 (+ (* (-1) x3) (* 2 x6) x1)) ($\leq$ (- 5) (+ (* (- 1) x3) (* 2 x5) x1))). We can notice by the coefficients in the result that the interpolant is not a UTVPI formula, thus Z3 must have reduced the problem to linear integer arithmetic.

The result obtained by Mathsat is (and ($\leq$ (- 5) (+ x1 (+ (* (- 1) x3) (* 2 x5)))) ($\leq$ 9 (+ x1 (+ (* (- 1) x3) (* 2 x6))))) which is the same as Z3 modulo the commutativity of the additions in the expression. Despite this difference, the following query to Z3 verifies that the interpolation produced by our implementation implies the interpolation produced by the SMT solver above mentioned; at the same time, the interpolant produced by the SMT solver does not imply our interpolant.

```
(declare-fun x1 () Int)
(declare-fun x3 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)

(define-fun implementation_interpolant () Bool
(and
(<= (- (- x6) x1) 0)
(<= (+ (- x6) x3) (- 9))
(<= (- (- x5) x1) 7)
(<= (+ (- x5) x3) (- 2))
)
)

(define-fun z3_interpolant () Bool
(and
(<= 9 (+ (* (- 1) x3) (* 2 x6) x1))
(<= (- 5) (+ (* (- 1) x3) (* 2 x5) x1))
)
)

(push)
(assert (not (implies  z3_interpolant implementation_interpolant)))
(check-sat) ;; This check returns sat, i.e. z3_interpolant does not imply implementation_interpolant
(pop)
(push)
(assert (not (implies implementation_interpolant z3_interpolant)))
(check-sat) ;; This check returns unsat, i.e. implementation_interpolant implies z3_interpolant
(pop)
```

For the next example, let us consider $\alpha_2 = \{-x_2 - x_1 + 3 \geq 0, x_1 + x_3 + 1 \geq 0, -x_3 - x_4 - 6 \geq 0, x_5 + x_4 + 1 \geq 0\}$; $\beta_2 = \{x_2 + x_3 + 3 \geq 0, x_6 - x_5 - 1 \geq 0, x_4 - x_6 + 4 \geq 0\}$.

Our implementation produced the following output (and (≤ (+ (- x3) x2) 4) (≤ (+ x4 x3) (- 6)) (≤ (- (- x5) x4) 1)) and obtained the following trace:

```
Processing
(+ c_x2 a_x1)
Updating structure with
x_2 + x_1 <= 3
Processing
(- (- c_x3) a_x1)
Updating structure with
- x_3 - x_2 <= 1
Processing
(+ c_x4 c_x3)
Updating structure with
x_4 + x_3 <= -6
Processing
(- (- c_x5) c_x4)
Updating structure with
- x_5 - x_4 <= 1
Removing this var: x_0
Removing this var: x_1
Removing this var: x_2
Reducing x_2 + x_1 and - x_3 - x_2
Result: - x_3 + x_1
Removing this var: x_3
Removing this var: x_4
Removing this var: x_5
(ast-vector
  (<= (+ (- x3) x2) 4)
  (<= (+ x4 x3) (- 6))
```

```
 (<= (- (- x5) x4) 1))
```

The interpolant obtained by Z3 is (and (≤ (- 4) (+ x3 (* (- 1) x2))) (≤ (+ x3 x4) (- 6)) (≥ (+ x4 x5) (- 1))); and the interpolant produced by Mathsat is (≤ (+ x2 (+ x3 (+ x4 (* (- 1) x5)))) (- 7)). Using Z3, we were able to verify that:

- the interpolantion obtained by our implementation is equivalent to the interpolant obtained by Z3.
- the interpolant obtained by our implementation implies the interpolant obtained by Mathsat, but the converse does not hold.

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)

(define-fun implementation_interpolant () Bool
(and
(<= (+ (- x3) x2) 4)
(<= (+ x4 x3) (- 6))
(<= (- (- x5) x4) 1))
)

(define-fun z3_interpolant () Bool
(and
(<= (- 4) (+ x3 (* (- 1) x2)))
(<= (+ x3 x4) (- 6))
(>= (+ x4 x5) (- 1)))
)

(define-fun mathsat_interpolant () Bool
(<= (+ x2 (+ x3 (+ x4 (* (- 1) x5)))) (- 7))
)

(push)
(assert (not (iff z3_interpolant implementation_interpolant)))
(check-sat) ;; This check returns unsat, i.e. z3_interpolant is equivalent to implementation_interpolant
(pop)

(push)
(assert (not (implies  mathsat_interpolant implementation_interpolant)))
(check-sat) ;; This check returns sat, i.e. mathsat_interpolant does not imply implementation_interpolant
(pop)

(push)
```

```
(assert (not (implies implementation_interpolant mathsat_interpolant)))
(check-sat) ;; This check returns unsat, i.e. implementation_interpolant implies mathsat_interpolant
(pop)
```

4.3.2 Performance comparison with iZ3 and MathSat

This section discusses a benchmark of interpolation generation for the UTVPI theory. Similarly to the previous chapter, we compare the performance of our tool with respect to the iZ3 and Mathsat.

Benchmark description

The benchmark contains the following parameters:

- l stands for a random number of the max value of the bounds
- m stands for the number of variables allowed in the random signature
- n stands for the number of inequalities of the form: $a_1 x_{k_1} + a_2 x_{k_2} \leq c$ or $a_1 x_{k_1} - a_2 x_{k_2} \leq c$ where a_1, a_2 are chosen uniformly at random from the set of elements $\{-1, 0, 1\}$, k_1, k_2 are chosen uniformly at random from the set of elements $\{1, \dots, n\}$, and c is chosen uniformly at random from the set $\{-l, \dots, l\}$

We constructed a pair of inconsistent formulas $(A - part, B - part)$ using an identical construction to the benchmark proposed in the previous chapter, using two $z3 :: solver$ in order to maintain each $A - part$ and $B - part$ consistent but inconsistent with each other.

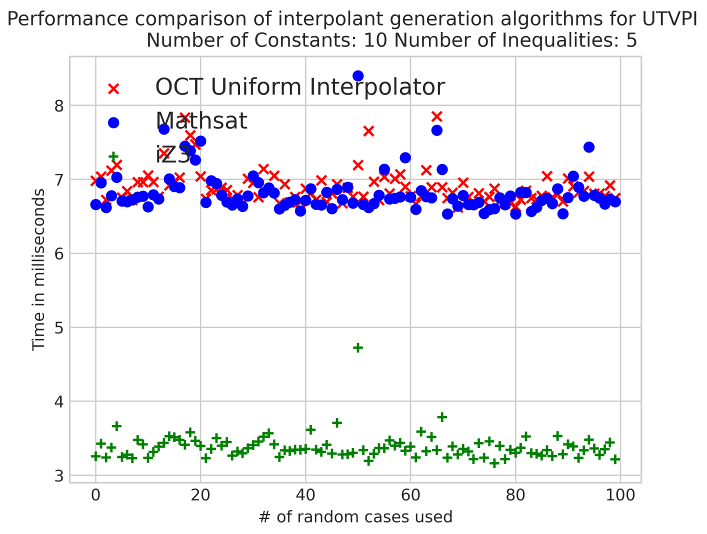


Figure 4.1: Performance comparison graph of UTVPI interpolant generation algorithms for the benchmark (l = 1000, m = 10, n = 5)

Experimental results

We designed the problem because the randomness makes it difficult to come up with trivial solutions. This problem was executed 100 and 10000 times using the parameters $(l = 1000, m = 10, n = 5)$ and $(l = 10, m = 10, n = 20)$ respectively.

The following graph reports the time needed by our implementation, iZ3, and the interpolation generation algorithm from Mathsat. It is well known that both iZ3 and Mathsat does not compute uniform interpolants for UTVPI. Regardless, the benchmark was used with the purpose to compare their execution time on normal interpolants.

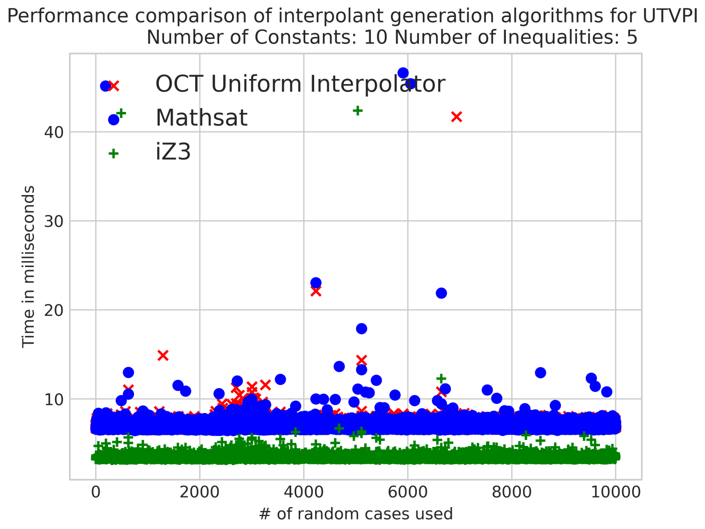


Figure 4.2: Performance comparison graph of UTVPI interpolant generation algorithms for the benchmark (l = 1000, m = 10, n = 5)

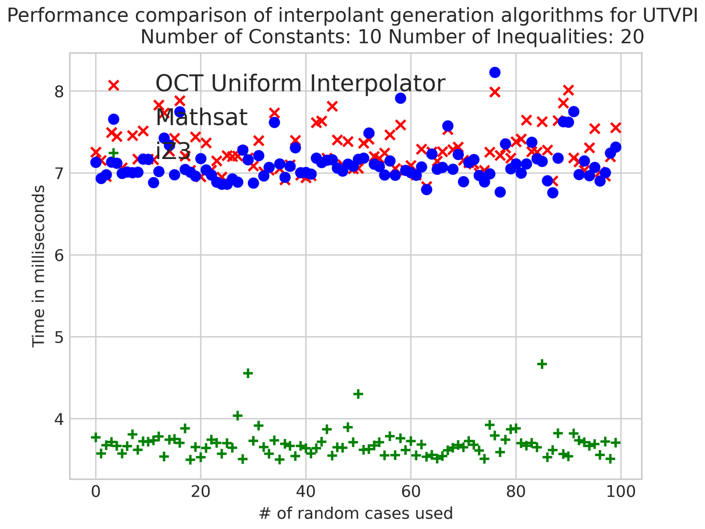


Figure 4.3: Performance comparison graph of UTVPI interpolant generation algorithms for the benchmark (l = 1000, m = 10, n = 20)

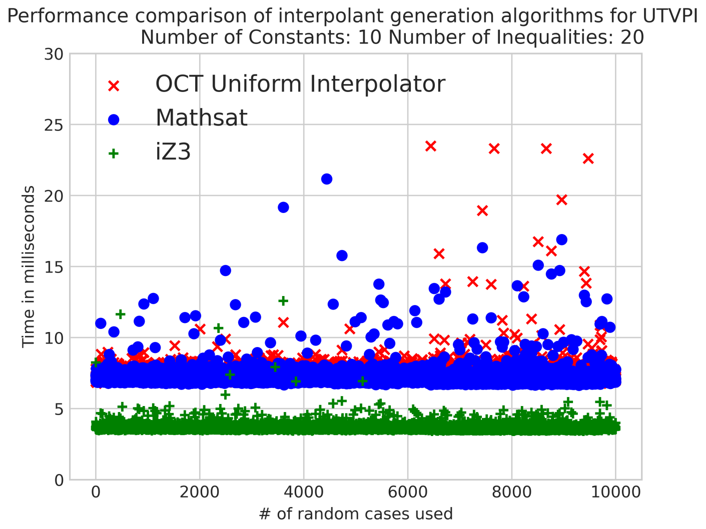


Figure 4.4: Performance comparison graph of UTVPI interpolant generation algorithms for the benchmark (l = 1000, m = 10, n = 20)

4.4 Conclusions

This chapter presented the approach by Prof. Kapur for computing the uniform interpolant of an inconsistent pair of formulas in the UTVPI theory. Several examples and testing of the implementation indicated that the output produced by Kapur's algorithm produces a stronger interpolant than the other ones produced by iZ3 and Mathsat. The performance comparison showed a closer difference in times compared to the performance comparison obtained for the EUF theory.

In order to improve the current implementation it will be interesting to explore the use of heuristics to determine the order in which the uncommon variable should be eliminated. The current implementation uses a lexicographic approach based on the occurrence of the symbols in the input formula. Different heuristic can consider the number of occurrences of the uncommon symbols in different inequalities, a topological order based on the dependency of the linear combinations of the inequalities, etc.

Chapter 5

Interpolation algorithm for the theory combination of EUF and UTVPI

Theory combination techniques involve reusing the algorithms for verification problems of the theories involved by either purifying terms over the theories, or reducing the original problem into a base theory, or a combination of these two. Usually the approaches following the first approach aforementioned rely on a Nelson-Oppen framework [50, 4, 6]

Despite possibly more efficient approaches using model-based theory combination techniques [1] [4], the latter requires operations which many SMT solvers do not provide proper API to implement these. Hence, the approach used in the thesis work follows [50] since it does not require extensive modification to the decision procedures used. Many of the necessary modifications were implemented on top of Z3 and

[1]An interesting property of model-based theory combination is that it is not needed to propagate disjunctions, even if the theories are not convex. The reason is due to the *disjunctiviness* of the problem is handled *lazily* by the SAT solver.

PicoSAT/TraceCheck.

The propagation of new equalities and disjunction of equalities requires the additional step to split these formulas into the correct part of the interpolant pair. Among the major contributions of [50] was the introduction of the class of equality interpolanting theories.

Definition 5.0.1. *A theory $\mathcal{T}$ is* equality interpolanting *if for every A, B in $\mathcal{T}$ and every AB-mixed equality $a = b$ such that $A \wedge B \models_{\mathcal{T}} a = b$, there exists a term t in $\mathcal{T}$ (called interpolanting term) such that $A \wedge B \models_{\mathcal{T}} a = t$ and $A \wedge B \models_{\mathcal{T}} b = t$.*

The relevance of the existance of the interpolating term for a deduced AB-mixed equality becomes relevant in the context of splitting a formula into a suitable A-part and B-part respectively.

Deciding where to include AB-common terms to either the A-part or the B-part of the interpolantion pair affects the final result since the interpolant will be *closer* to the A-part or to the B-part respectively. The authors in [50] include AB-common terms to the B-part. However, the implementation work includes AB-common terms to the A-part due to the interest in uniform interpolants.

5.1 Yorsh-Musuvathi Interpolation Combination Framework

The algorithm extends a Nelson-Oppen framework which allows the Yorsh-Musuvathi framework to produce an Interpolant for combined theories using Interpolant generation algorithm for each of the theories involved. In order to integrate interpolants to the latter, the algorithm includes a *partial interpolant* every time a disjunction of equalities (conflict clause) is propagated. These *partial interpolants* are computed from an unsatisfiability proof obtained by including the negation of the disjunction to the formula using the following definition:

Definition 5.1.1. *[50] Let $\langle A, B\rangle$ be a pair of clause sets such that $A \wedge B \vdash \bot$. Let $\mathcal{T}$ be a proof of unsatisfiability of $A \wedge B$. The propositional formula $p(c)$ for a clause c in $\mathcal{T}$ is defined by induction on the proof structure:*

- *if c is one of the input clauses then*
  - *if $c \in A$, then $p(c) := \bot$*
  - *if $c \in B$, then $p(c) := \top$*
- *otherwise, c is a result of resolution, i.e. let c_1, c_2 be two clauses of the form $x \vee c_1^{'}$, $\neg x \vee c_2^{'}$ respectively. The partial interpolant for c is defined as follows:*
  - *if $x \in A$ and $x \notin B$ (x is A-local), then $p(c) := p(c_1) \vee p(c_2)$*
  - *if $x \notin A$ and $x \in B$ (x is B-local), then $p(c) := p(c_1) \wedge p(c_2)$*
  - *otherwise (x is AB-common), then $p(c) := (x \vee p(c_1)) \wedge (\neg x \vee p(c_2))$*

5.2 Implementation

The thesis implementation uses the Yorsh-Musuvathi framework to reuse the uniform interpolant generation algorithms for the EUF and UTVPI theory. The implementation maintains a map data structure that keeps track of the *partial interpolants*. This ensures that the base case for the above formula $p(c)$ is replaced by previous clauses as required in [50].

Since introducting negations is necessary to compute partial interpolants, we noticed the following interaction with the theories involved in the thesis work:

- EUF case: negations of literals in this theory are just dis-equalities, which the interpolation algorithm implemented handles as Horn clauses with a false head term.
- UTVPI case: negations of literals in this theory are either dis-equalities or strict inequalities. The dis-equalities are purified an appended to the EUF component; strict inequalities of the form $x > y$ are replaced by non-strict inequalities of the form $x \geq y + 1$.

The main loop of the procedure is shown below:

5.3 Evaluation

5.3.1 Detailed evaluation of a complete example

A simple example

Let us consider the following example: $\alpha = \{f(x_1) = 0, x_1 = a, y_1 \leq a\}; \beta = \{x_1 \leq b, y_1 = b, f(y_1) \neq 0\}$. The implementation produces the following interpolant [2]:

$((x_1 \neq y_1 \vee f(y_1) = 0 \vee (f(x_1) = 0 \wedge ((x_1 = y_1 \vee (-x_1 + y_1 \leq 0 \wedge x_1 - y_1 \leq -1) \vee y_1 - x_1 \leq -1) \wedge (x_1 \neq y_1)))) \wedge (x_1 = y_1 \vee (((x_1 = y_1) \vee (-x_1 + y_1 \leq 0 \wedge x_1 - y_1 \leq -1) \vee y_1 - x_1 \leq -1) \wedge x_1 \neq y_1)))$

The following SMT query verifies that the previous result obtained is an interpolant of the input formula:

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
```

[2]A trace of the execution can be found at the Appendix section since it is considerably large to include it in this section

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 $\psi()$
 4:     $\psi.init()$
        while true do
 6:         if $T_1 \models_{EUF} \perp$ then
                return $T_1$
 8:         end if
            if $T_2 \models_{UTVPI} \perp$ then
10:             return $T_2$
            end if
12:         if $T_1 \models_{EUF} \psi.current()$ then
                if $T_2 \models_{UTVPI} \psi.current()$ then
14:                 continue
                else
16:                 append $\psi.current()$ to $T_2$
                    $\psi.init()$
18:             end if
            else
20:             if $T_2 \models_{UTVPI} \psi.current()$ then
                    continue
22:             else
                    append $\psi.current()$ to $T_2$
24:                 $\psi.init()$
                end if
26:         end if
            $UpdatePartialInterpolant(\psi.current())$
28:         $\psi.next()$
        end while
30: end procedure
```

```
(declare-fun x3 () Int)
(declare-fun y1 () Int)
(declare-fun y2 () Int)
(declare-fun y3 () Int)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)

(define-fun part_a () Bool
(and
(= (f x1) 0)
(= x1 a)
(<= y1 a)
))

(define-fun part_b () Bool
(and
(<= x1 b)
(= y1 b)
(distinct (f y1) 0)
))

(define-fun my_interpolant () Bool

(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1)))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
)

(push)
;; This returns unsat, which verifies that the input
;; is an inconsistent pair of formulas
(assert (and part_a part_b))
(check-sat)
(pop)
(push)
;; This returns unsat, which verifies that the ouput
;; is implied by the A-part
(assert (not (implies part_a my_interpolant)))
(check-sat)
(pop)
(push)
;; This returns unsat, which verifies that the ouput
;; is inconsistent with the B-part
(assert (and my_interpolant part_b))
(check-sat)
(pop)
```

For this example these are the interpolants reported by Z3 and Mathsat respectively:

- Z3 interpolant: (and ($\geq$ (+ x1 (* (- 1) y1)) 0) (= (f x1) 0))
- Mathsat interpolant: (or (= (f y1) 0) ($\leq$ 1 (+ x1 (* (- 1) y1))))

The following SMT query verifies the strength relation between the interpolants produced:

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
```

```
(declare-fun x3 () Int)
(declare-fun y1 () Int)
(declare-fun y2 () Int)
(declare-fun y3 () Int)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)

(define-fun my_interpolant () Bool

(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1)))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
)

(define-fun z3_interpolant () Bool
(and (>= (+ x1 (* (- 1) y1)) 0) (= (f x1) 0))
)

(define-fun mathsat_interpolant () Bool
(or (= (f y1) 0) (<= 1 (+ x1 (* (- 1) y1))))
)

(push)
;; The following returns sat, which means
;; that my_interpolant does not imply z3_interpolant
(assert (not (implies my_interpolant z3_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that z3_interpolant implies my_interpolant
(assert (not (implies z3_interpolant my_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that my_interpolant implies math_interpolant
(assert (not (implies my_interpolant mathsat_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns sat, which means
;; that mathsat_interpolant does not imply my_interpolant
(assert (not (implies mathsat_interpolant my_interpolant )))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that z3_interpolant implies math_interpolant
(assert (not (implies z3_interpolant mathsat_interpolant)))
(check-sat)
(pop)
(push)
```

```
;; The following returns sat, which means
;; that mathsat_interpolant does not imply z3_interpolant
(assert (not (implies mathsat_interpolant z3_interpolant )))
(check-sat)
(pop)
```

5.4 A uniform interpolant generation algorithm for the combined theory of EUF and UTVPI

This section introduces a uniform interpolant generation algorithm for the aforementioned theories. The algorithm is a tableux-like algorithm which specifies formula state and reduction rules. It extends the approach in [24] by incorporating additional structure to handle UTVPI formulas and more reduction rules to handle the Normalize and Elim rules introduced in the uniform interpolant generation algorithm for UTVPI theory in Chapter 4.

5.4.1 A tableux-like uniform interpolant generation algorithm for EUF

Let us review the uniform interpolant generation algorithm for EUF from [24]. The algorithm follows a similar preprocessing step as the uniform interpolation algorithm for EUF discussed in Chapter 3. The 'formula state' encodes a triplet $\langle \delta(\underline{y}, \underline{z}), \phi(\underline{y}, \underline{z}), \psi(\underline{e}, \underline{y}, \underline{z}) \rangle$ of the resulting formula after preprocessing with the following meaning:

- $\underline{e}$ encodes information about the variables to be eliminated.
- $\underline{y}$ encodes information about the variables to be eliminated than became common variables since the algorithm detected an equality of a variable in this

category with a common term.

- $\underline{z}$ encodes information about the common variables. It is worth mentioning that each $\underline{\text{e}}$, $\underline{\text{y}}$, and $\underline{\text{z}}$ are indexed terms which encode an ordering relation between them.
- $\delta(\underline{y}, \underline{z})$ stands for the formulas that provide an explicit definition for variables in $\underline{y}$ using a DAG-representation.
- $\phi(\underline{y}, \underline{z})$ stands for the formulas that do not contain variables to eliminate.
- $\psi(\underline{e}, \underline{y}, \underline{z})$ stands for the formulas that contain variables to eliminate.

Some definitions are needed before discussing the tableux rules in [24].

Definition 5.4.1. *A term t(resp. literal L) is $\underline{\text{e}}$-free when there is no occurrence of variables to eliminate in t (resp. L).*

Two flat terms t, u of the form

$t := f(a_1, \ldots, a_n), u := f(b_1, \ldots, b_n)$

are said to be compatible *if and only if every $i = 1, \ldots, n$, either a_i is identical to b_i or both a_i and b_i are $\underline{\text{e}}$-free.*

The difference set of two compatible terms *like above is the set of dis-equalities $a_i \neq b_i$ such that a_i is not identical to b_i.*

The algorithm in [24] provides the following tableux rules:

1. Simplification rules:
   - 1.0) If an atom like $t = t$ belong to ψ, just remove it; if a literal like $t \neq t$ occurs somewhere, delete ψ, replace ϕ with $\bot$ and stop.

- 1.i) If t is not a variable to eliminate and ψ contains both $t = a$ and $t = b$, remove the latter and replace it with $b = a$.
- 1.ii) If ψ contains $e_i = e_j$ with $i > j$, remove it and replace every e_i in δ, ϕ, ψ by e_j.

2. DAG update rule: if ψ contains $e_i = t(\underline{y}, \underline{z})$, remove it, rename every e_i in δ, ϕ, ψ as y_j (for fresh y_j) and add $y_j = t(\underline{y}, \underline{z})$ to $\delta(\underline{y}, \underline{z})$.

3. e-Free Literal rule: if ψ contains a literal $L(\underline{y}, \underline{z})$ move it to $\phi(\underline{y}, \underline{z})$.

4. If ψ contains a pair of atoms $t = a$ and $u = b$, where t and u are compatible flat terms, and not dis-equality from the difference set of t, u belongs to ϕ, then non-deterministically apply one of the following alternatives:
   - 4.0) Remove from ψ the atom $f(b_1, \ldots, b_n) = b$, add to ψ the atom $b = a$ and add to ϕ all the equalities $a_i = b_i$ such that $a_i \neq b_i$ is in the difference set of t, u;
   - 4.1) Add to ϕ one of the dis-equalities form the difference set of t, u.

5.4.2 Our proposed uniform interpolant generation algorithm for EUF + UTVPI

Our algorithm first purifies a given satisfiable input formula and uses an extended tableux-like algorithm using additional rules to deal with UTVPI formulas. If the input formula is not satisfiable, then return $\perp$ as result. We use the data structures discussed in Chapter 4 to deal with UTVPI formulas, hence we keep a normal form representation of the UTVPI inequalities in the formula state inside the proposed tableux-like algorithm. The previous rules of the uniform interpolant generation algorithm for EUF involving propagation of equations suits the UTVPI component as well. The additional rules are the following:

5. Eliminate uncommon UTVPI terms: if there are UTVPI inequalities of the form $a_i x + a_j e_j \leq k_1$ and $a_k y - a_j e_j \leq k_2$ in ψ, then introduce to ϕ the UTVPI inequality $a_i x + a_k y \leq k_1 + k_2$.

6. Normalize UTVPI inequalities: if there is a UTVPI inequality of the form $a_i x + a_i x \leq k$ in the formula state, then remove it and insert to ψ the UTVPI inequality $a_i x \leq \lfloor k/2 \rfloor$

7. Normalize bounds: if there are two UTVPI inequalities of the form $a_i x + a_j y \leq k_1$, $a_i x + a_j y \leq k_2$ in the formula state with $\{k_1, k_2\} \in \mathbb{N}$, then remove them both and insert to ψ the UTVPI inequality $a_i x + a_j y \leq min(k_1, k_2)$

8. Propagate fully bounded uncommon UTVPI inequalities: if there are two UTVPI inequalities in ψ of the form $a_i e_i + a_j e_j \leq k_1$ and -$a_i e_i - a_j e_j \leq k_2$, where $\{a_i, a_j\} \subseteq \{1, -1\}$ and $i > j$ then non-deterministically apply the following rule:

   – Remove both $a_i e_i + a_j e_j \leq k_1$ and $-a_i e_i - a_j e_j \leq k_2$ from ψ and replacing every e_i by $e_j = l + a_j a_i e_i$ where $l \in \{a_j k_2, a_j k_2 + 1, \ldots, a_j k_1 - 1, a_j k_1\}$.

9. Propagate fully bounded uncommon UTVPI terms: if there are two UTVPI inequalities in ψ of the form $a_i e_i + a_j e_j \leq k_1$ and -$a_i e_i - a_j e_j \leq k_2$, where $\{a_i, a_j\} \subseteq \{1, -1\}$ and $i > j$ then non-deterministically apply the following rule:

   – Remove both $a_i e_i + a_j e_j \leq k_1$ and $-a_i e_i - a_j e_j \leq k_2$ from ψ and replacing every e_i by $e_j = l + a_j a_i e_i$ where $l \in \{a_j k_2, a_j k_2 + 1, \ldots, a_j k_1 - 1, a_j k_1\}$.

Remark : Notice that if no transformation applies to the formula state, we obtain the same the following types of formulas for the EUF component according to [24]

- ψ only contains dis-equalities of the kind $e_i \neq a$ and equalities of the kind $f(a_1, \ldots, a_n) = a$. If ψ contains equalities of the kind $f(a_1, \ldots, a_n) = a$ it means that at least one a_i must belong to $\underline{e}$. If ψ contains two equalities of the form $f(a_1, \ldots, a_n) = a$ and $f(b_1, \ldots, b_n) = b$ then it means $f(a_1, \ldots, a_n)$ and $f(b_1, \ldots, b_n)$ are incompatible or $a_i \neq b_i$ belongs to ϕ.

Similarly, the following types of formulas for the UTVPI component exists in the formula state if no transformation apply: bounded only by an upper bound, bounded only by a lower bound, and fully Bounded with $\pm$ infinity. The latter follows directly from the normal form representation of UTVPI inequalities induced by the data structures introduced in Chapter 4. Otherwise, rules 8 and 9 should have applied.

5.4.3 Termination, Correctness and Completeness of the proposed algorithm

Lemma 5.4.1. *The non-deterministic procedure presented above always terminates.*

Proof. Following a similar proof as in [24] it is enough to show every branch of the algorithm terminates. The rules involving the EUF component are cover in [24]. For the rest of the rules involving UTVPI expressions, we first notice that there is a $\mathcal{O}(n^2)$ bound over all the UTVPI expressions for UTVPI signature S where n is the number of distinct variables in S. The rules relax the bounds of all the bounds in the inequalities, which does not induce a negative cycle, otherwise the input formula could not be satisfiable. $\square$

Before proving the correctness of the tableux algorithm for the theory combination of EUF and UTVPI we first state an important result relating the existence of uniform interpolant in a concept denoted extensions.

Lemma 5.4.2. *[8] Let T be a theory. A formula $\psi(\underline{y})$ is a uniform interpolant in T of $\exists \underline{e}.\phi(\underline{e}, \underline{y})$ if and only if it satisfies the following two conditions:*

i) $T \models \forall \underline{y}.((\exists \underline{e}.\phi(\underline{e}, \underline{y})) \rightarrow \psi(\underline{y}))$

ii) for every model $\mathcal{M}$ of T, for every tuple of elements $\underline{a}$ from the support of $\mathcal{M}$ such that $\mathcal{M} \models \psi(\underline{a})$ it is possible to find another model $\mathcal{N}$ of T such that M embeds into $\mathcal{N}$ and $\mathcal{N} \models \exists \underline{e}.\psi(\underline{e}, \underline{a})$

Theorem 5.4.3. *Let us apply the proposed tableux algorithm for uniform interpolant generation for EUF + UTVPI to the satisfiable input formula $\exists \underline{e}.\phi(\underline{e}, \underline{z})$ and that the algorithm terminates with its branches in the formula states*

$$\langle \delta_1(\underline{y_1}, \underline{z}), \phi_1(\underline{y_1}, \underline{z}), \psi_1(\underline{e_1}, \underline{y_1}, \underline{z})\rangle, \ldots, \langle \delta_k(\underline{y_k}, \underline{z}), \phi_k(\underline{y_k}, \underline{z}), \psi_k(\underline{e_k}, \underline{y_k}, \underline{z})\rangle$$

Then the uniform interpolant of $\exists \underline{e}.\phi(\underline{e}, \underline{z})$ in $EUF + UTVPI$ is the DAG-unravelling [3] of the formula

$$\bigvee_{i=1}^{k} \exists \underline{y_i}.(\delta_i(\underline{y_i}, \underline{z}) \wedge \phi_i(\underline{y_i}, \underline{z}))$$

Proof. The proof follows the similar style of the proof of Theorem 5.1 in [24]. Let α be $\bigvee_{i=1}^{k} \exists \underline{y_i}.(\delta_i(\underline{y_i}, \underline{z}) \wedge \phi_i(\underline{y_i}, \underline{z}))$. First, we notice that it is enough to prove the two items 5.4.2 to show our assertion. The first item follows since α was obtained from the rules in the proposed algorithm.

To address the second item of the lemma, given a model $\mathcal{M}$ such that $\mathcal{M} \models \alpha$, finding an isomorphic embedding $\mathcal{N}$ such that $\mathcal{N} \models \exists \underline{e}.\phi(\underline{e}, \underline{z})$ is equivalent , using Robinson Lemma [9], to prove that there is a model for $\Delta(\mathcal{M}) \cup \phi \cup \{a = \mathcal{I}(a)\}_{a \in \underline{y} \cup \underline{z}}$.

In [24], we can obtain a model for the EUF component using the model induced by the canonical form obtained by orienting the equations in ψ. In other to obtain

[3] Unravelling means that the explicit recursive substitution of the each of the y_i variables are substituted in $\phi_i(\underline{y_i}, \underline{z})$ using the equations in $\delta_i(\underline{y_i}, \underline{z})$.

a model for the UTVPI component, we noticed that we need to find a model for the three kinds of UTVPI inequalities ψ. According to our remark 5.4.2 there are three kind.

For the kind of UTVPI inequalities where are fully bounded by infinity we can take any element in the natural numerals to model these UTVPI expressions since those inequalities are trivially true.

For the other two kinds, it is easy two see that these equalities are satisfiable since they were obtained by the rules in the tableux algorithm on the purified input formula. Since the input formula is satisfiable and purification preserves satisfiability, then it cannot be the case that the two kind of UTVPI inequality are unsatisfiable.

With the latter, we can find a model $\mathcal{N}$ satisfying $\exists \underline{e}.\psi(\underline{e}, \underline{y}, \underline{z})$ with the desired conditions. Therefore, α is an uniform interpolant for $\exists \underline{e}.\phi(\underline{e}, \underline{z})$.

□

5.5 Conclusions

This chapter showed the implementation of the interpolation combination algorithm in [50] and the necessary changes for the EUF and UTVPI algorithms presented in previous chapters.

In addition, a new algorithm for the uniform interpolant generation for the combined theory of EUF + UTVPI was introduced. The implementation of this algorithm is will be considered in the future if the rules 8 and 9 can be replaced for propagation rules that avoid splitting.

Chapter 6

Future Work

Regarding implementation work, there are several improvements that were not explored in the thesis work since many of them do not change the overall time complexity of the implementation, but these consume additional memory resources. These improvements are the following:

Major improvements:

- Improve hash function for terms: during testing the congruence closure algorithm, it was evident that dealing with numerous terms, collisions happened to the point that some terms were merged since the signature function indicated to do so, but this merges should not happen. Intrinsically, the current implementation relies on the hash function provided by Z3, which might not be optimized for many terms or it might be that their decision procedures encode internal information differently. Nonetheless, for a typical verification problem the implementation will not find any problems since these issues were noticed while dealing with instance problems involving graph terms of more than 1000000 nodes.

- Regarding the theory combination of EUF and UTPVI, it will be interesting to explore a more specific interpolation combination approach. The thesis culminated with a uniform interpolant algorithm for the combine theory, but the author agrees a better job can be done to avoid propagating disjunctions of the equalities by extending the signature, consequently the interpolation procedures, for both of the theories involved or propagate these disjunctions as using a compact representation.
- Currently the implementation for combined EUF and UTVPI theory does not incrementally check the consistency of the current formula in the main loop of the implementation 5.2. Changing the latter to an incremental approach will maintain lemmas so the computation of intermediate results will not be performed repetitively.

Minor improvements:

- The space needed to encode curry nodes *can be significantly 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 curryfication of all the terms in the arguments of function applications.

Appendix A: additional proofs

6.1 Theorems about Interpolants

6.1.1 Interpolants are closed under conjunction and disjunction

Theorem 6.1.1. *If I_1, I_2 are interpolants of $\langle A, B \rangle$, then $I_1 \wedge I_2, I_1 \vee I_2$ are also interpolants of $\langle A, B \rangle$.*

Proof. We notice that $vars(I_1 \wedge I_2) \subseteq vars(A) \cap vars(B)$ and $vars(I_1 \vee I_2) \subseteq vars(A) \cap vars(B)$, otherwise I_1, I_2 couldn't be interpolants.

Since I_1, I_2 are interpolants, we have that $A \vdash I_1$, $B \wedge I_1 \vdash \bot$ and $A \vdash I_2$, $B \wedge I_2 \vdash \bot$

Here are formal proofs for $A \vdash I_1 \wedge I_2$ and $B \wedge I_1 \wedge I_2 \vdash \bot$:

$$\begin{array}{c} A \qquad A \\ \vdots \qquad \vdots \\ I_1 \qquad I_2 \\ \hline I_1 \wedge I_2 \end{array} \qquad\qquad \begin{array}{c} B \wedge I_1 \wedge I_2 \\ \hline B \wedge I_1 \\ \vdots \\ \bot \end{array}$$

Here are formal proofs for $A \vdash I_1 \vee I_2$ and $B \wedge (I_1 \vee I_2) \vdash \bot$:

$$\dfrac{\begin{matrix} A \\ \vdots \\ I_1 \end{matrix}}{I_1 \vee I_2} \qquad \dfrac{\begin{matrix} B \wedge (I_1 \vee I_2) \\ \vdots\, * \\ (B \wedge I_1) \vee (B \wedge I_2) \end{matrix} \quad \begin{matrix} \overline{B \wedge I_1} \\ \vdots \\ \bot \end{matrix} \quad \begin{matrix} \overline{B \wedge I_2} \\ \vdots \\ \bot \end{matrix}}{\bot}$$

where $*$ is any proof applying the distributivity property of the conjunction symbol over the disjunction symbol.

$\square$

6.1.2 Interpolants distribute conjunctions over disjunctions in the A-part

Theorem 6.1.2. *Let $\mathcal{F}_i$ be a set of formulas and I_i an interpolant for each $\langle A \wedge \mathcal{F}_i, B \rangle$ respectively. Then $\bigvee_i I_i$ is an interpolant for $\langle A \wedge (\bigvee_i \mathcal{F}_i), B \rangle$*

Proof. Let $A_i = A \wedge \mathcal{F}_i$ and $\hat{A} = A \wedge (\bigvee_i \mathcal{F}_i)$.

We see that $vars(I_i) \subseteq vars(A_i) \subseteq vars(\hat{A})$, hence $vars(\bigvee_i I_i) = \bigcup_i vars(I_i) \subseteq \bigcup_i vars(A_i) \subseteq vars(\hat{A})$. Similarly we can prove that $vars(\bigvee_i I_i) \subseteq vars(B)$. Thus, $vars(\bigvee_i I_i) \subseteq vars(A \wedge (\bigvee_i \mathcal{F}_i)) \cap vars(B)$.

To prove $\hat{A} \vdash \bigvee_i I_i$: We notice that $\hat{A} = \bigvee_i A_i$. From the latter and using the generalized version of the disjunction elimination rule in logic, i.e.

$$\dfrac{\alpha_1 \vee \cdots \vee \alpha_n \quad \dfrac{\begin{matrix} \overline{\alpha_1} \\ \vdots \\ I_1 \end{matrix}}{I_1 \vee \cdots \vee I_n} \quad \ldots \quad \dfrac{\begin{matrix} \overline{\alpha_n} \\ \vdots \\ I_n \end{matrix}}{I_1 \vee \cdots \vee I_n}}{I_1 \vee \cdots \vee I_n}$$

and distributing disjunctions over conjunctions in $\bigvee_i A_i$ the statement holds.

To prove $B \wedge \bigvee_i I_i \vdash \bot$: Since each $B \wedge I_i \vdash \bot$, by using the generalized version of the disjunction elimination rule from above, the result holds.

Therefore, $\bigvee_i I_i$ is an interpolant for $\langle A \wedge (\bigvee_i \mathcal{F}_i), B\rangle$.

$\square$

Appendix B: additional traces

6.2 Trace for example in theory combination chapter

```
Part a
EUF-component
(= (c_f c_x1) c_oct_1)
(= c_x1 a_a)
Octagon-component
(= 0 c_oct_1)
(<= c_y1 a_a)
Part b
EUF-component
(= c_y1 b_b)
(distinct (c_f c_y1) c_oct_2)
Octagon-component
(<= c_x1 b_b)
(= 0 c_oct_2)
Shared variables
(ast-vector
  c_x1
  a_a
  c_y1
  b_b
  c_oct_1
  c_oct_2)
(= 0 c_oct_1)
(<= c_y1 a_a)
(<= c_x1 b_b)
(= 0 c_oct_2)
(= (c_f c_x1) c_oct_1)
(= c_x1 a_a)
(= c_y1 b_b)
(not (= (c_f c_y1) c_oct_2))

Disjunction implied in EUF: (= c_x1 a_a)
Partial interpolant already computed
```

```
Disjunction implied in EUF: (= c_y1 b_b)
Partial interpolant already computed

Disjunction implied in OCT: (= c_x1 c_y1)
Clause Id: 2 (Fact) Predicate: (<= c_y1 a_a) Interpolant(old): false
Clause Id: 3 (Fact) Predicate: (<= c_x1 b_b) Interpolant(old): true
Clause Id: 5 (Fact) Predicate: (= c_x1 a_a) Interpolant(old): false
Clause Id: 6 (Fact) Predicate: (= c_y1 b_b) Interpolant(old): true
Clause Id: 7 (Fact) Predicate: (not (= c_x1 c_y1)) Interpolant(new): false
Clause Id: 8 (Conflict Clause) Predicate: (or (not (<= c_y1 a_a))
    (not (<= c_x1 b_b))
    (not (= c_x1 a_a))
    (not (= c_y1 b_b))
    (= c_x1 c_y1))
Inside partialInterpolantConflict
Case OCT
(ast-vector
  (<= c_y1 a_a)
  (= c_x1 a_a)
  (not (= c_x1 c_y1)))
(ast-vector
  (<= c_x1 b_b)
  (= c_y1 b_b))
DNF: (let ((a!1 (and (<= (- c_y1 a_a) 0)
               (<= (- c_x1 a_a) 0)
               (<= (+ (- c_x1) a_a) 0)
               (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (- c_y1 a_a) 0)
               (<= (- c_x1 a_a) 0)
               (<= (+ (- c_x1) a_a) 0)
               (<= (+ (- c_x1) c_y1) (- 1)))))
  (or a!1 a!2))
Input for OctagonInterpolant (ast-vector
  (<= (- c_y1 a_a) 0)
  (<= (- c_x1 a_a) 0)
  (<= (+ (- c_x1) a_a) 0)
  (<= (- c_x1 c_y1) (- 1)))
Input for OctagonInterpolant (ast-vector
  (<= (- c_y1 a_a) 0)
  (<= (- c_x1 a_a) 0)
  (<= (+ (- c_x1) a_a) 0)
  (<= (+ (- c_x1) c_y1) (- 1)))
Theory-specific interpolant: (let ((a!1 (and (<= (+ (- c_x1) c_y1) 0) (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (+ (- c_x1) c_y1) (- 1)))))
  (or a!1 a!2))
Interpolant for OCT: (let ((a!1 (and (<= (+ (- c_x1) c_y1) 0) (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (+ (- c_x1) c_y1) (- 1)))))
  (and (or a!1 a!2 false false false) true true))
Interpolant((from conflict)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 9 (Derived(2,8)) Predicate: (or (not (<= c_x1 b_b)) (not (= c_x1 a_a)) (not (= c_y1 b_b)) (= c_x1 c_y1)) Pivot: (<= c_y1 a_a)█
Pivot is A-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or false a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 10 (Derived(9,3)) Predicate: (or (= c_x1 c_y1) (not (= c_x1 a_a)) (not (= c_y1 b_b))) Pivot: (<= c_x1 b_b)
Pivot is B-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 true)))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 11 (Derived(10,5)) Predicate: (or (= c_x1 c_y1) (not (= c_y1 b_b))) Pivot: (= c_x1 a_a)
Pivot is A-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)) false))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 12 (Derived(11,6)) Predicate: (= c_x1 c_y1) Pivot: (= c_y1 b_b)
Pivot is B-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 true)))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 13 (Derived(12,7)) Predicate: false Pivot: (= c_x1 c_y1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (or (not (= c_x1 c_y1)) false))))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))
Final interpolant for conflict clause: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))

Disjunction implied in OCT: (= c_oct_1 c_oct_2)
Clause Id: 1 (Fact) Predicate: (= 0 c_oct_1) Interpolant(old): false
Clause Id: 4 (Fact) Predicate: (= 0 c_oct_2) Interpolant(old): true
Clause Id: 7 (Fact) Predicate: (not (= c_oct_1 c_oct_2)) Interpolant(new): false
Clause Id: 8 (Conflict Clause) Predicate: (or (not (= 0 c_oct_1)) (not (= 0 c_oct_2)) (= c_oct_1 c_oct_2))
Inside partialInterpolantConflict
Case OCT
-------It was sat!
(ast-vector
  (= 0 c_oct_1)
  (not (= c_oct_1 c_oct_2)))
(ast-vector
  (= 0 c_oct_2))
DNF: (let ((a!1 (and (<= c_oct_1 0)
               (<= (- c_oct_1) 0)
```

```
                (<= (+ (- c_oct_1) c_oct_2) (- 1)))))
  (or (and (<= c_oct_1 0) (<= (- c_oct_1) 0) (<= (- c_oct_1 c_oct_2) (- 1)))
      a!1))
Input for OctagonInterpolant (ast-vector
  (<= c_oct_1 0)
  (<= (- c_oct_1) 0)
  (<= (- c_oct_1 c_oct_2) (- 1)))
Input for OctagonInterpolant (ast-vector
  (<= c_oct_1 0)
  (<= (- c_oct_1) 0)
  (<= (+ (- c_oct_1) c_oct_2) (- 1)))
Theory-specific interpolant: (let ((a!1 (and (<= (- c_oct_1) 0)
                (<= c_oct_1 0)
                (<= (+ (- c_oct_2) c_oct_1) (- 1)))))
  (or a!1
      (and (<= (- c_oct_1) 0) (<= c_oct_1 0) (<= (- c_oct_2 c_oct_1) (- 1)))))
Interpolant for OCT: (let ((a!1 (and (<= (- c_oct_1) 0)
                (<= c_oct_1 0)
                (<= (+ (- c_oct_2) c_oct_1) (- 1)))))
(let ((a!2 (or a!1
               (and (<= (- c_oct_1) 0)
                    (<= c_oct_1 0)
                    (<= (- c_oct_2 c_oct_1) (- 1)))
               false
               false)))
  (and a!2 true)))
Interpolant((from conflict)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (or a!1 a!2))
Clause Id: 9 (Derived(1,8)) Predicate: (or (not (= 0 c_oct_2)) (= c_oct_1 c_oct_2)) Pivot: (= 0 c_oct_1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (and (or (= 0 c_oct_1) false) (or (not (= 0 c_oct_1)) a!1 a!2)))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
Clause Id: 10 (Derived(9,4)) Predicate: (= c_oct_1 c_oct_2) Pivot: (= 0 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!4 (or (not (not (= 0 c_oct_2))) true)))
```

```
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (not (= 0 c_oct_2)) a!3) a!4)))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (or (not (= 0 c_oct_2)) a!3)))
Clause Id: 11 (Derived(10,7)) Predicate: false Pivot: (= c_oct_1 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (or (not (= c_oct_1 c_oct_2)) false))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
Final interpolant for conflict clause: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
EUF solver found a contradiction
(ast-vector
  (= (c_f c_x1) c_oct_1)
  (= c_x1 a_a)
  (= c_y1 b_b)
  (not (= (c_f c_y1) c_oct_2))
  (= c_x1 c_y1)
  (= c_oct_1 c_oct_2))
Clause Id: 1 (Fact) Predicate: (= (c_f c_x1) c_oct_1) Interpolant(old): false
Clause Id: 4 (Fact) Predicate: (not (= (c_f c_y1) c_oct_2)) Interpolant(old): true
Clause Id: 5 (Fact) Predicate: (= c_x1 c_y1) Interpolant(old): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))
Clause Id: 6 (Fact) Predicate: (= c_oct_1 c_oct_2) Interpolant(old): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
Clause Id: 7 (Conflict Clause) Predicate: (or (not (= (c_f c_x1) c_oct_1))
    (= (c_f c_y1) c_oct_2)
    (not (= c_x1 c_y1))
    (not (= c_oct_1 c_oct_2)))
Inside partialInterpolantConflict
Case EUF
Part a: (ast-vector
  (= (c_f c_x1) c_oct_1)
  (= c_x1 c_y1)
  (= c_oct_1 c_oct_2))
Part b: (ast-vector
  (not (= (c_f c_y1) c_oct_2)))
Theory-specific Interpolant for EUF: (and (= c_oct_1 (c_f c_x1)))
Interpolant for EUF: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1))))
      (a!3 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!4 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
      (a!5 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!3 a!4))))
(let ((a!6 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!5)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or (and (= c_oct_1 (c_f c_x1)))
               false
               (and a!2 (not (= c_x1 c_y1)))
               a!6)))
  (and a!7 true)))))
Interpolant((from conflict)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
  (or a!4 (= c_oct_1 (c_f c_x1)) (and a!6 (not (= c_x1 c_y1)))))))
Clause Id: 8 (Derived(1,7)) Predicate: (or (= (c_f c_y1) c_oct_2) (not (= c_x1 c_y1)) (not (= c_oct_1 c_oct_2))) Pivot: (= (c_f c_x1) c_oct_1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or (not (= (c_f c_x1) c_oct_1))
               a!4
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (and (or (= (c_f c_x1) c_oct_1) false) a!7)))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (and (= (c_f c_x1) c_oct_1) a!7)))))
Clause Id: 9 (Derived(8,4)) Predicate: (or (not (= c_oct_1 c_oct_2)) (not (= c_x1 c_y1))) Pivot: (= (c_f c_y1) c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1))))
      (a!9 (or (not (= (c_f c_y1) c_oct_2)) true)))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2) (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
```

```
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (or (= (c_f c_y1) c_oct_2) (and (= (c_f c_x1) c_oct_1) a!7))))))
Clause Id: 10 (Derived(9,5)) Predicate: (not (= c_oct_1 c_oct_2)) Pivot: (= c_x1 c_y1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (not (not (= c_x1 c_y1))) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (not (= c_x1 c_y1))
               (= (c_f c_y1) c_oct_2)
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Clause Id: 11 (Derived(10,6)) Predicate: false Pivot: (= c_oct_1 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1)))))
      (a!10 (or (not (not (= c_oct_1 c_oct_2))) a!4)))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9)) a!10))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9))
       (or (= c_oct_1 c_oct_2) a!4)))))))
Final interpolant for conflict clause: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9))
       (or (= c_oct_1 c_oct_2) a!4)))))))
-> Final Interpolant: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!3 (or (not (= (c_f c_x1) 0))
               (= 0 (c_f c_x1))
               (and a!2 (not (= c_x1 c_y1)))))
      (a!5 (or (= c_x1 c_y1) (and a!2 (not (= c_x1 c_y1))))))
(let ((a!4 (or (not (= c_x1 c_y1)) (= (c_f c_y1) 0) (and (= (c_f c_x1) 0) a!3))))
  (and a!4 a!5)))))
Interpolant:
(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1)))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
rm -rf tests/*.o tests/basic_test
```

Appendix C: additional implementation code of relevant data structures

6.3 Congruence Closure with Explanation operation header

```
class Hornsat;

class CongruenceClosureExplain : public CongruenceClosure {

  Hornsat * hsat;

  PendingElements pending_elements;
  PendingPointers equations_to_merge;
  PendingPointers pending_to_propagate;

  FactoryCurryNodes const & factory_curry_nodes;

  LookupTable lookup_table;
```

```
UseList     use_list;

void pushPending(PendingPointers &, const PendingElement &);
void merge();
void merge(EquationCurryNodes const &);
void propagate();
void propagateAux(CurryNode const &, CurryNode const &,
   EqClass, EqClass, PendingElement const &);

EqClass          highestNode(EqClass, UnionFind &);
EqClass          nearestCommonAncestor(EqClass, EqClass,
   UnionFind &);
PendingPointers explain(EqClass, EqClass);
void             explainAlongPath(EqClass, EqClass, UnionFind
    &, ExplainEquations &, PendingPointers &);
std::ostream &  giveExplanation(std::ostream &, EqClass,
   EqClass);

public:
CongruenceClosureExplain(Hornsat *, CongruenceClosureExplain
    const &, UnionFindExplain &);
CongruenceClosureExplain(Z3Subterms const &,
   UnionFindExplain &, FactoryCurryNodes &, IdsToMerge const
    &);
~CongruenceClosureExplain();

bool areSameClass(EqClass, EqClass);
bool areSameClass(z3::expr const &, z3::expr const &);

EqClass  constantId(EqClass);
```

```
  EqClass  find(EqClass);
  z3::expr z3Repr(z3::expr const &);

  void merge(EqClass, EqClass);
  void merge(z3::expr const &, z3::expr const &);

  PendingPointers explain(z3::expr const &, z3::expr const &);
  std::ostream &  giveExplanation(std::ostream &, z3::expr
     const &, z3::expr const &);

  z3::expr_vector z3Explain(z3::expr const &, z3::expr const
     &);
  std::ostream &  z3Explanation(std::ostream &, const z3::expr
      &, const z3::expr &);

  friend std::ostream & operator << (std::ostream &, const
     CongruenceClosureExplain &);
};
```

6.4 Hornsat (Gallier's data structure) header

```
class Hornsat {

  friend class EUFInterpolant;

  unsigned num_hcs, num_literals;
  // This structure is only used in our approach
  // for conditional-elimination
  std::unordered_map<unsigned, HornClause *> head_term_indexer
     ;

  UnionFindExplain          ufe;
  CongruenceClosureExplain equiv_classes;

  std::vector<Literal>   list_of_literals;
  ClassList              class_list;
  std::vector<unsigned>  num_args;
  std::vector<LiteralId> pos_lit_list;
  // 'facts' is a queue of all the (temporary)
  // literals that have value true
  std::queue<LiteralId>  facts;
  std::queue<TermIdPair> to_combine;

  bool consistent;

  void satisfiable();
  void closure();

 public:
  Hornsat(CongruenceClosureExplain &, HornClauses const &);
  ~Hornsat();

  void build(CongruenceClosureExplain &, HornClauses const &);
  bool isConsistent() const ;
  void unionupdate(LiteralId, LiteralId);
  friend std::ostream & operator << (std::ostream &, Hornsat
     const &);
};
```

Bibliography

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

[2] Armin Biere. Picosat essentials. *JSAT*, 4:75–97, 05 2008.

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

[4] Maria Paola Bonacina and Moa Johansson. On interpolation in decision procedures. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 1–16, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

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

[6] Roberto Bruttomesso, Silvio Ghilardi, and Silvio Ranise. Quantifier-free interpolation in combinations of equality interpolating theories. *ACM Trans. Comput. Logic*, 15(1), March 2014.

[7] 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.

[8] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Model completeness, covers and superposition. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 142–160, Cham, 2019. Springer International Publishing.

[9] C.C. Chang and H.J. Keisler. *Model Theory: Third Edition.* Dover Books on Mathematics. Dover Publications, 2013.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.

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

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

[21] 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.

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

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

[24] Silvio Ghilardi, Alessandro Gianola, and Deepak Kapur. Compactly representing uniform interpolants for euf using (conditional) dags, 2020.

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

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

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

[28] Deepak Kapur. Conditional congruence closure over uninterpreted and interpreted symbols. *Journal of Systems Science and Complexity*, 32:317–355, 02 2019.

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

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

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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.

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

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

[42] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[43] 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.

[44] 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.

[45] Andrew M. Pitts. On an interpretation of second order quantification in first order intuitionistic propositional logic. *J. Symbolic Logic*, 57(1):33–52, 03 1992.

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

[47] 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.

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

[49] 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.

[50] 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.