JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering and Natural Sciences

EUF-Proofs for SMT4J

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the academic degree

Diplom-Ingenieur

in the Master's Program

COMPUTER SCIENCE

Submitted by

Katalin Fazekas

At the

Institute for Formal Models and Verification

Advisor

Univ.-Prof. Dr. Armin Biere

Co-advisor

Assist.-Prof. Dr. Martina Seidl

Linz, August 2015

Abstract

SMT (Satisfiability Modulo Theories) Solvers are considered as a new, promising generation of decision engines in numerous software and hardware verification tools due to their support of various decision procedures for theories beyond propositional logic. One general theory that is supported by most SMT Solvers is specialized in equalities over uninterpreted functions (EUF). In this theory one can decide whether an equality is a logical consequence (in first-order logic with equality) of a conjunction of equalities.

Since SMT Solvers are fairly complex systems with a large, frequently growing code base, the objective to ensure their correctness is highly challenging. Therefore unsatisfiable core production i.e., the extraction of an unsatisfiable subformula which is as small as possible, and detailed proof generation on the level of theories are desirable functions of an SMT Solver. The constructed unsatisfiable cores can be exploited to build stronger theory lemmas which reduce the search space considerably. The generated proofs can be verified automatically by external tools to achieve higher level of confidence about the correctness of the solver.

SMT4J (SMT for Java) is a simple, java-based lazy SMT Solver that neither supported the generation of proofs nor exploited the power of unsatisfiable cores for pruning the search space. The main contribution of this thesis is the extension of the EUF theory solver of SMT4J with unsatisfiable core generation capability based on a congruence closure algorithm. Furthermore, this work extends the theory solver with detailed proof production functionality in order to ensure the quality of the produced results. In extensive experiments considerable run time improvements can be observed if the unsatisfiable cores are used. Furthermore, it is shown that proof generation can be efficiently integrated into the solving process.

Zusammenfassung

In modernen Software- und Hardware Verifikationsframeworks werden SMT Solver heutzutage als eine neue, vielversprechende Generation von Entscheidungswerkzeugen eingesetzt, deren Ausdruckstärke in Form von Theorien gesteuert werden kann. Eine allgemeine Theorie, die üblicherweise von jedem SMT Solver unterstützt wird, ermöglicht logisches Schließen bezüglich der Gleichheit über uninterpretierten Funktionen (engl. equalities over uninterpreted functions [EUF]). Mittels dieser Theorie kann entschieden werden, ob die Gleichheit zweier Ausdrücke eine logische Konsequenz (in Prädikatenlogik mit Gleichheit) von einer Verknüpfung von Gleichheiten anderer Ausdrücke ist.

Da SMT Solver hochoptimierte Systeme mit umfangreichem, häufig stark wachsendem Quellcode sind, ist das Sicherstellen ihrer Korrektheit eine große Herausforderung. Eine detaillierte Beweisgenerierung auf der Ebene von Theorien ist daher eine wünschenswerte Funktionalität eines SMT Solver. Die erzeugten Beweise können von externen Tools automatisch verifiziert werden und das Vertrauen der Anwender und Anwenderinnen in den Solver steigt. Ein weiteres nützliches Feature ist die Extraktion von unerfüllbaren Kernen. Hierbei wird aus einer Menge von unerfüllbaren Einschränkungen eine möglichst kleine Teilmenge extrahiert, die immer noch unerfüllbar ist.

SMT4J (SMT für Java) ist ein einfacher, Java basierter SMT Solver, der weder Beweisgenerierung unterstützte noch die Stärke von unerfüllbaren Kernen zur Einschränkung des Suchraumes nutzte. Der wichtigste Beitrag dieser Masterarbeit ist die Erweiterung des EUF Theorie-Solvers von SMT4J um die Fähigkeit unerfüllbare Kerne zu erzeugen. Hierfür wird ein Congruence-Closure Algorithmus eingesetzt. Weiters erweitert diese Arbeit den Theorie-Solver um detaillierte Beweiserstellungsfunktionalitäten, welche die Qualität der erzeugten Ergebnisse sicher stellt. Umfangreiche Tests zeigen eine erhebliche Verbesserung der Laufzeit, wenn unerfüllbare Kerne verwendet werden. Ferner wird gezeigt, dass die Beweisgenerierung effizient in den Beweisprozess integriert werden kann.

Contents

Chapter 1

Introduction

Formal verification processes of hardware and software systems attempt to decide whether a formal description of a system satisfies a list of constraints or not. Many successful approaches recast the specifications and the constraints into a satisfiability problem of a suitable logic, in order to increase the automation level of the verification process. Decision procedures for propositional satisfiability, or shortly SAT, are widely used for this purpose. Current state-of-the-art SAT Solvers are highly accurate and efficient in their field, therefore it is a widespread practice to utilize them as decision engines. Nevertheless, in many situations the verification conditions may not be expressible in an efficient way confined only to propositional logic and therefore tools for more expressive logics are necessary in the verification process. SMT – Satisfiability Modulo Theories – Solvers attempt to meet this claim by the support of various decision procedures for theories beyond sentential logic. Due to their extended expressiveness, SMT Solvers are recently considered as the new generation of decision engines in numerous software and hardware verification tools. An extensive description of SMT solvers and their application domains can be found for instance in [9].

The range of the supported background theories of an SMT Solver may vary from solver to solver, but there is a continuously growing core of theories that is covered by most of the recent SMT Solvers. Some of these most common supported theories are for example linear arithmetic [20], the theory of arrays and bit-vectors [13] and the theory of equalities over uninterpreted functions (abbreviated as EUF). In the so-called *lazy* SMT Solvers the support of a background theory $\mathcal{T}$ is usually materialized as a specialized submodule integrated with an efficient SAT solver. These sub-modules, that are called as *theory solvers*, implement specialized decision procedures and data structures in order to decide, as efficient as possible, the satisfiability of a conjunction of $\mathcal{T}$-related literals.

In the context of this thesis, the focus is mainly restricted to the theory of equalities over uninterpreted functions. This theory is frequently used to model overly complex systems on a higher level of abstraction. More precisely, in this theory the complex and precise details of e.g. a data structure are represented simply by functions without interpretation. This disregard of information does not influence the unsatisfiability of a logical formula, therefore it is a comfortable way to simplify a formal specification. Some domains of application where equalities over uninterpreted functions are frequently exploited as best-practice are for instance automated hardware verification [14] and translation validation [33] tasks. The implementation of an EUF theory solver in a lazy SMT Solver usually builds on a congruence closure based algorithm in order to decide whether an equality is a logical consequence (in first-order logic with equality) of a conjunction of equalities.

Since SMT Solvers are frequently used in verification processes, it is not surprising that some kind of assurance of their correctness is highly desired, actually, the correctness of an SMT Solver is essential. By detailed model generation capability, the work of an SMT

Solver is verifiable for satisfiable problem instances almost without any additional effort. Nevertheless, the trustworthiness of an SMT Solver about unsatisfiable problem instances is not that obviously checkable. Among others, this reason stresses the importance of producing independently verifiable results for unsatisfiable instances. Therefore, several SMT Solvers derive an inconsistency from the input problem during the solving process. Then, the proof of unsatisfiability is an independently verifiable object. On the level of theory solvers, a proof can be a small unsatisfiable subset – also called core – of the accepted theory literals, or even a detailed deduction of the inconsistency from this core, derived by the rules of the corresponding theory. On the level of the entire SMT Solver, a proof usually describes the unsatisfiability of the propositional representation of the input formula, where the derivation is typically based on some form of resolution.

The verification method of a produced proof is another relevant question related to SMT Solvers. The unsatisfiable core of a problem instance can for example be checked by another SMT Solver. Nevertheless, a trustworthy tool for verification purposes is a more attractive option. A more detailed proof production requires collateral works from an SMT Solver, but simplifies the checking process and a simpler process can be accomplished by less code, namely with less error proneness. So far, there is no strong consensus in the SMT solver community on the expected format of unsatisfiability proofs and thus there are no laid down principles about their verification process neither.

SMT4J (SMT for Java) is a simple, java-based lazy SMT Solver that is under development by the Institute for Formal Models and Verification (FMV) of the Johannes Kepler University Linz. At present, the solver supports satisfiability decisions with respect to a few background theories (without quantifiers) and is able to produce a trace of its SAT computations for unsatisfiable problem instances. The employed searching strategy of SMT4J demands unsatisfiable core production from each theory solver, in order to narrow down the search space of possible satisfying assignments.

The main contribution of this thesis is the extension of the EUF theory solver of SMT4J with unsatisfiable core generation capability based on the congruence closure algorithm introduced by Robert Nieuwenhuis and Albert Oliveras in [29]. Furthermore, this work complements the theory solver with detailed proof production functionalities in order to ensure the quality of the produced results. The thesis includes an EUF theory specific fuzzer tool to support the exhaustive testing of the theory solver and an exceedingly simple proof checker that serves for verification purposes. Moreover, this thesis attempts to answer the following questions:

- Is the pure lemmas on demand approach adequately efficient for a competitive SMT Solver?
- Is it possible to develop an EUF theory solver that efficiently provides exhaustive unsatisfiable core generation?
- Can the use of all unsatisfiable cores enhance the performance of the lemmas on demand approach on formulas with EUF as background theory?
- How can the correctness of a theory solver be verified?
- Can the unsatisfiable core producing algorithm be modified to efficiently generate, besides the core, a detailed deduction of each found contradiction?
- How can the correctness of a generated deduction be verified?

The thesis is structured as follows. Chapter 2 defines the relevant concepts and notation of the further chapters and describes briefly the most important data structure of the implementation. Chapter 3 presents the proof generation abilities of some recently popular

SMT Solvers and introduces the certificates produced by the EUF theory solver of SMT4J. Chapter 4, as the core of the thesis, describes in details the proof generation method of the EUF theory solver in the SMT4J framework. The subsequent chapter demonstrates possible solutions for proof verification and defines the main properties of the EUF Proof Checker. The evaluation of the execution experiences are given in Chapter 6. Chapter 7 concludes the thesis and contains suggestions for future developments.

Chapter 2

Preliminaries

2.1 Formal Preliminaries

The satisfiability problems considered in this thesis are formulated in a fragment of many sorted first-order logic with equality, therefore this section provides a short overview of the syntax and semantics of this logic and introduces the notation used throughout the thesis.

2.1.1 Syntax

The alphabet of a many sorted first-order language contains logical and non-logical symbols with auxiliary delimiters. The logical symbols are the standard propositional connectives $\{\neg, \wedge, \vee, \supset\}$, propositional constants $\{\top, \bot\}$, the quantifiers $\{\forall, \exists\}$, and the variables. The parentheses and the comma are used for separating the symbols. The logical and the delimiter symbols are part of every first-order language. The non-logical symbols, however, are always given by a signature and they define a particular first-order language. The definitions of the non-logical part of a many sorted first-order language are as follows (based on [42]).

Definition 2.1 (Alphabet of non-logical symbols)**.** The set of the non-logical symbols of a many-sorted first-order language are defined as a quadruple $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$, where

1. $\mathcal{S}$ is a non-empty set of sorts,
2. $\mathcal{P}$ is a non-empty set of predicate symbols,
3. $\mathcal{F}$ is a set of function symbols, and
4. $\mathcal{C}$ is a set of constant symbols.

Sorts are denoted with π (possibly with subscripts). To each sort $\pi \in \mathcal{S}$ belongs a countably infinite set of variables $\mathcal{V}^\pi = \{v_1^\pi, v_2^\pi, ...\}$.

Definition 2.2 (Signature)**.** Let $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$ be an alphabet as defined above. Then a signature Σ over $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$ is a triplet $\langle \Sigma^\mathcal{F}, \Sigma^\mathcal{P}, \Sigma^\mathcal{C} \rangle$, where

1. $\Sigma^\mathcal{P}$ assigns to each predicate symbol $P \in \mathcal{P}$ an ordered sequence of sorts $(\pi_1, \pi_2, ..., \pi_k)$, where $k \geq 0$ and $\pi_1, \pi_2, ..., \pi_k \in \mathcal{S}$,
2. $\Sigma^\mathcal{F}$ assigns to each function symbol $f \in \mathcal{F}$ an ordered sequence of sorts $(\pi_1, \pi_2, ..., \pi_k, \pi)$, where $k > 0$ and $\pi_1, \pi_2, ..., \pi_k, \pi \in \mathcal{S}$ and
3. $\Sigma^\mathcal{C}$ assigns to each constant symbol $c \in \mathcal{C}$ a sort π, where $\pi \in \mathcal{S}$.

Each non-logical symbol can be associated with a non-negative number, which is the so-called *arity* of the symbol.

Definition 2.3 (Arity of symbols)**.** Let $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$ be an alphabet and $\Sigma = \langle \Sigma^{\mathcal{F}}, \Sigma^{\mathcal{P}}, \Sigma^{\mathcal{C}} \rangle$ a signature defined over this alphabet as defined above. Then the arity function *arity* : $\Sigma^{\mathcal{F}} \cup \Sigma^{\mathcal{P}} \cup \Sigma^{\mathcal{C}} \to \mathbb{N}$ is defined as follows:

$$arity(\Sigma(s)) = \begin{cases} 0 & \text{if } s \in \mathcal{C} \\ k & \text{if } s \in \mathcal{F} \text{ and } \Sigma^{\mathcal{F}}(s) = (\pi_1, \pi_2, ..., \pi_k, \pi) \\ k & \text{if } s \in \mathcal{P} \text{ and } \Sigma^{\mathcal{P}}(s) = (\pi_1, \pi_2, ..., \pi_k) \end{cases}$$

A function symbol f (resp. predicate symbol P) with n as arity is called *n-ary function* (resp. *n-ary predicate*) (when n is one, then *unary function* (resp. *unary predicate*), when n is two, then *binary function* (resp. *binary predicate*), and so on). In this work constant symbols are denoted with a, b, c, d, e (possibly with subscripts) and the letters f, g, h usually denote the function symbols of $\mathcal{F}$, while the letters P, Q, R denote predicate symbols.

Definition 2.4 (Terms over Σ)**.** Let Σ be a signature over an alphabet $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$. Then the set of the terms over the signature Σ is the smallest set such that:

1. Each variable v^{π} of sort $\pi \in \mathcal{S}$ is a term of sort π.
2. Each constant $c \in \mathcal{C}$ of sort $\pi \in \mathcal{S}$ is a term of sort π.
3. If $f \in \mathcal{F}$ is a function symbol of signature $(\pi_1, \pi_2, ..., \pi_k, \pi)$, and $t_1, t_2, ..., t_k$ are respectively terms of sorts $\pi_1, \pi_2, ..., \pi_k$, then $f(t_1, t_2, ..., t_k)$ is a term of sort π.

Definition 2.5 (Formulas over Σ)**.** Let Σ be a signature over an alphabet $\langle \mathcal{S}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$. Then the set of first-order formulas of $\mathcal{L}_{\Sigma}$ is the smallest set such that:

1. If $P \in \mathcal{P}$ is a predicate symbol of signature $(\pi_1, \pi_2, ..., \pi_k)$, and $t_1, t_2, ..., t_k$ are respectively terms of sorts $\pi_1, \pi_2, ..., \pi_k$, then $P(t_1, t_2, ..., t_k)$ is a formula.
2. If φ is a formula, then $\neg\varphi$ is a formula.
3. If φ_1 and φ_2 are formulas and $\circ$ is a binary logical connective symbol, then $(\varphi_1 \circ \varphi_2)$ is a formula.
4. If φ is a formula, Q is a quantifier and x is an arbitrary variable, then $Qx\varphi$ is a formula.

A language whose expressions (formulas and terms) are built up from the elements of a signature Σ by using the rules defined above is denoted as $\mathcal{L}_{\Sigma}$. The first-order formulas built up from the first rule of Definition 2.5 are called *atomic formulas*, while the formulas which contain quantifiers as described in the last rule of Definition 2.5 are called *quantified formulas*. The formulas of $\mathcal{L}_{\Sigma}$ in this thesis are denoted by upper or lower Greek letters, while the letters s, t and u denote arbitrary terms. A given occurrence of a variable x in a formula φ is called *bounded*, when it is in the scope of a quantifier that names it. A given occurrence of a variable x is *free*, when it is not bounded. A term is called *ground term*, when it does not contain any variables. A first-order formula is called *sentence* or *closed*, when it does not contain any free variables. As a technical convenience, in the thesis all the variables will be treated as constants and the signatures are always extended implicitly by these introduced constant symbols. Therefore in the following, each term is a ground term. The *size* (number of symbols) of an arbitrary ground term t is denoted by $|t|$: $|c| = 1$ when $c \in \mathcal{C}$ is a constant term and $|f(t_1, t_2, ..., t_n)| = 1 + |t_1| + |t_2| + ... + |t_n|$ when $f \in \mathcal{F}$ is a non-constant term.

A *literal* is an atomic formula (*positive literal*) or its negation (*negative literal*) and denoted as l (possibly with subscripts). A *clause* is a disjunction of literals: $l_1 \vee l_2 \vee ... \vee l_n$. The empty disjunction of literals (*empty clause*) is denoted by $\perp$. A formula in conjunctive normal form (*CNF formula*) is a conjunction of zero or more clauses: $c_1 \wedge c_2 \wedge ... \wedge c_n$. An empty CNF formula is denoted by $\top$. A clause that contains at most one positive literal is called *Horn clause*. A *quantifier-free formula* is a formula in which no quantifier occurs.

In the context of the thesis, all formulas are quantifier-free and the alphabets are implicitly extended for each sort $\pi \in \mathcal{S}$ with a binary equality predicate symbol $=_\pi$ of signature (π, π).

2.1.2 Semantics

To define the meaning of the expressions of a language $\mathcal{L}_\Sigma$, first of all, the universe of the variables and the factual values of the constant symbols has to be given. Furthermore, it has to be fixed which operations and relations are represented by the function and predicate symbols. The establishment of this information is an interpretation of the language. The formal definitions are as follows (based on [42]).

Definition 2.6 (Domain of interpretation)**.** Let Σ be a signature as defined above. Then the *universe* or *domain* – denoted by $\mathcal{U}$ – of the interpretation is defined by

$$\mathcal{U} = \bigcup_{\pi \in \mathcal{S}} \mathcal{U}_\pi$$

Definition 2.7 (Interpretation of Σ)**.** Let Σ be a signature as defined above. Then the interpretation of Σ is a $\mathcal{I} = \langle \mathcal{I}_\mathcal{S}, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P}, \mathcal{I}_\mathcal{C} \rangle$ quadruple of functions, where

1. $\mathcal{I}_\mathcal{S} : \pi \mapsto \mathcal{U}_\pi$ assigns to each sort $\pi \in \mathcal{S}$ a non-empty set $\mathcal{U}_\pi$ of elements of sort π.
2. $\mathcal{I}_\mathcal{C} : c \mapsto c^\mathcal{I}$ assigns to each constant symbol $c \in \mathcal{C}$ of sort π an element $c^\mathcal{I} \in \mathcal{U}_\pi$.
3. $\mathcal{I}_\mathcal{F} : f \mapsto f^\mathcal{I}$ assigns to each function symbol $f \in \mathcal{F}$ of signature $(\pi_1, \pi_2, ..., \pi_k, \pi)$ a function $f^\mathcal{I} : \mathcal{U}_{\pi_1} \times \mathcal{U}_{\pi_2} \times ... \times \mathcal{U}_{\pi_k} \to \mathcal{U}_\pi$, and
4. $\mathcal{I}_\mathcal{P} : P \mapsto P^\mathcal{I}$ assigns to each predicate symbol $P \in \mathcal{P}$ of signature $(\pi_1, \pi_2, ..., \pi_k)$ a logical predicate $P^\mathcal{I} : \mathcal{U}_{\pi_1} \times \mathcal{U}_{\pi_2} \times ... \times \mathcal{U}_{\pi_k} \to \{true, false\}$.

When the meaning and the universe of the non-logical symbols of the language is established by an interpretation, the meaning of the expressions built up from these symbols can be defined based on this interpretation. The interpretation is also called *structure*, when it is over an empty set of variables. In that case, the assignment of a formula under that interpretation does not depend on the variables, therefore the definitions of the semantics can be simplified by ignoring the variable-evaluation dependency.

Definition 2.8 (Semantics of terms)**.** Let $\mathcal{I}$ be an interpretation of a signature Σ as defined above. Then the value of a term t of sort π under the interpretation $\mathcal{I}$ – denoted by $|t|^\mathcal{I}$ – is defined as follows.

1. If $c \in \mathcal{C}$ is a constant symbol of sort π, then $|c|^\mathcal{I}$ is the element $c^\mathcal{I} \in \mathcal{U}_\pi$,
2. If $t_1, t_2, ..., t_k$ are terms respectively of sorts $\pi_1, \pi_2, ..., \pi_k$, and their values under $\mathcal{I}$ are respectively the elements $|t_1|^\mathcal{I} \in \mathcal{U}_{\pi_1}, |t_2|^\mathcal{I} \in \mathcal{U}_{\pi_2}, ..., |t_k|^\mathcal{I} \in \mathcal{U}_{\pi_k}$, then $|f(t_1, t_2, ..., t_k)|^\mathcal{I}$, the value under $\mathcal{I}$ of a function symbol $f \in \mathcal{F}$ of signature $(\pi_1, \pi_2, ..., \pi_k, \pi)$, is the element $f^\mathcal{I}(|t_1|^\mathcal{I}, |t_2|^\mathcal{I}, ..., |t_k|^\mathcal{I}) \in \mathcal{U}_\pi$.

Definition 2.9 (Semantics of $\mathcal{L}_\Sigma$)**.** Let $\mathcal{I}$ be an interpretation of a quantifier and variable free language $\mathcal{L}_\Sigma$. Then the truth value of a formula φ of $\mathcal{L}_\Sigma$ under the $\mathcal{I}$ interpretation – denoted by $|\varphi|^\mathcal{I}$ – is as follows:

1. $|P(t_1, t_2, ..., t_k)|^\mathcal{I} \rightleftharpoons \begin{cases} true & \text{if } P^\mathcal{I}(|t_1|^\mathcal{I}, |t_2|^\mathcal{I}, ..., |t_k|^\mathcal{I}) = true \\ false & \text{otherwise} \end{cases}$
2. $|s = t|^\mathcal{I} \rightleftharpoons |s|^\mathcal{I} = |t|^\mathcal{I}$
3. if φ has the form of $\neg\varphi_1$, then $|\varphi|^\mathcal{I} \rightleftharpoons \neg|\varphi_1|^\mathcal{I}$,
4. if φ has the form of $\varphi_1 \wedge \varphi_2$, then $|\varphi|^\mathcal{I} \rightleftharpoons |\varphi_1|^\mathcal{I} \wedge |\varphi_2|^\mathcal{I}$,
5. if φ has the form of $\varphi_1 \vee \varphi_2$, then $|\varphi|^\mathcal{I} \rightleftharpoons |\varphi_1|^\mathcal{I} \vee |\varphi_2|^\mathcal{I}$,
6. if φ has the form of $\varphi_1 \supset \varphi_2$, then $|\varphi|^\mathcal{I} \rightleftharpoons |\varphi_1|^\mathcal{I} \supset |\varphi_2|^\mathcal{I}$.

The interpretation of the propositional constants follows directly, namely $\top$ represents *true* in every interpretation, while $\bot$ means *false*. Let φ be a variable and quantifier free formula of $\mathcal{L}_\Sigma$. When $|\varphi|^{\mathcal{I}} = true$ holds under an interpretation $\mathcal{I}$, then $\mathcal{I}$ *satisfies* φ and it is denoted as $\mathcal{I} \models \varphi$. In that case, φ is called *satisfiable*, and $\mathcal{I}$ is called a *model* of φ. If all interpretations of $\mathcal{L}_\Sigma$ satisfy φ, then φ is *valid* and denoted as $\models \varphi$. A valid formula is also called *tautology*. When there is no interpretation that satisfies φ, then φ is called *unsatisfiable*. Any inconsistent subset of the clauses of an unsatisfiable CNF formula is called *unsatisfiable core (UC)*. The unsatisfiable core is called *minimal unsatisfiable core*, when by removing any one of the clauses, it turns into satisfiable. Sometimes, the unsatisfiable core of a formula is also called *conflict set*.

The interpretation of an atomic equality formula is based on the identity relation of the arguments, i.e., it is evaluated to true, if and only if a and b are identical. The further details of these atoms and their interpretation will be in the context of a background *theory* clarified during the next section. A theory is defined as follows (based on [42]).

Definition 2.10 (Axiomatic theory)**.** Let $\mathcal{L}_\Sigma$ be a first-order language and $\mathcal{A}$ be a set of closed formulas of $\mathcal{L}_\Sigma$. Then the pair $\mathcal{T} = \langle \mathcal{L}_\Sigma, \mathcal{A} \rangle$ is an axiomatic theory and the formulas of $\mathcal{A}$ are the non-logical axioms of the theory $\mathcal{T}$.

Definition 2.11 (Model of theory)**.** An interpretation $\mathcal{I}$ of a first-order language $\mathcal{L}_\Sigma$ is a model of the theory $\mathcal{T} = \langle \mathcal{L}_\Sigma, \mathcal{A} \rangle$, if for all $A \in \mathcal{A}$ formulas $|\mathcal{A}|^{\mathcal{I}} = true$ holds.

The models of a theory are also called $\mathcal{T}$-interpretations. A formula φ is satisfiable in a theory, if there is a model of the theory, that satisfies also the formula, i.e., if there is an interpretation $\mathcal{I}$, that satisfies all the axioms of the theory and the formula. Note, that the axioms of a theory usually involve quantifiers. As the definition of axioms is the only place in the thesis where quantifiers are involved, the concise definition of their semantics is not introduced, for precise details see e.g. [21, Chapter 5.3].

2.2 Equalities over Uninterpreted Functions

SMT Solvers decide the satisfiability of various first-order formulas with respect to some background theories. These background theories, as was mentioned earlier, tighten the class of the models for those interpretations, which satisfy the axioms of the theory. The most general background theory, denoted as $\mathcal{T}_\varepsilon$, is specialized in equalities over uninterpreted functions (EUF). An example formula of this theory is

$$x = y \land y = z \land f(z) = t \supset f(x) = t.$$

This formula can be translated to the following question: Is $f(x) = t$ a logical consequence (in first-order logic with equality) of the equalities $x = y, y = z$ and $f(z) = t$? In other words, in the theory $\mathcal{T}_\varepsilon$, one can decide whether an equality is a consequence of other equalities.

The theory does not impose any kind of restriction on the interpretation of the non-logical symbols. Moreover, it ignores the semantics of the symbols (except the equality symbol). This is why this theory is frequently used as a background theory, for example during automated hardware verification tasks [14] and translation validation processes [33]. When the concrete complex details of the functionality is not in focus, uninterpreted functions are used to abstract away these details. If a formula in equality logic with uninterpreted functions is found to be unsatisfiable, it is unsatisfiable with interpreted functions as well.

In the context of the thesis, the EUF theory handles equalities and disequalities between ground terms of the same sort as atomic formulas. Predicate symbols are treated as uninterpreted logical functions over ground terms. The equalities are always written in an infix way, e.g. $s = t$ or $s \neq t$. Therefore the symbol $=$ can denote the equality predicate symbol in the logic and the equality at the meta-level as well, nevertheless, the meaning should be always clear from the context. A ground term of $\mathcal{T}_\varepsilon$ is either a constant symbol $c \in \mathcal{C}$ or an uninterpreted function symbol $f \in \mathcal{F}$. The set of ground terms over $\mathcal{C} \cup \mathcal{F}$ – denoted as $\mathcal{G}$ – is the smallest set containing all constant symbols such that whenever $f \in \mathcal{F}$ is a function symbol of signature $(\pi_1, \pi_2, ..., \pi_k, \pi)$ and $t_1, t_2, ..., t_n \in \mathcal{G}$ are terms respectively of sorts $\pi_1, \pi_2, ..., \pi_k$, then $f(t_1, t_2, ..., t_k) \in \mathcal{G}$. A $\mathcal{T}_\varepsilon$*-literal* is either an atomic formula (equality or disequality) or its negation in the theory. The conjunctive formulas will be denoted as a set of literals $\{l_1, l_2, ...l_n\}$. The satisfiability of a conjunction of ground equalities and disequalities in $\mathcal{T}_\varepsilon$ is decidable in polynomial time by using a *congruence closure procedure* [1, p. 24–33].

In the EUF theory, the equality symbol represents a binary monotonic equivalence relation over $\mathcal{G}$. The properties of the equality predicate can be described in an axiomatic way.

Definition 2.12 (Axioms of equality)**.** The theory axioms of $\mathcal{T}_\varepsilon$ are defined as follows:

$$\forall t(t = t) \tag{2.1}$$

$$\forall s \forall t(s = t \supset t = s) \tag{2.2}$$

$$\forall s \forall t \forall u(s = t \wedge t = u \supset s = u) \tag{2.3}$$

Further, for every n-ary function symbol $f \in \mathcal{F}$ it holds, that

$$\forall s_1...s_n \forall t_1...t_n(s_1 = t_1 \wedge ... \wedge s_n = t_n \supset f(s_1, ...s_n) = f(t_1, ...t_n)) \tag{2.4}$$

The first three axioms define the equality predicate as an *equivalence relation*, i.e., it is reflexive (2.1), symmetric (2.2), and transitive (2.3). The last axiom (2.4) guarantees that this equivalence relation is preserved by all the functions, that is to say, all the functions are consistent and therefore for equal arguments they assign the same result. All in all, the equality predicate symbol represents a *congruence relation* on $\mathcal{G}$ in the context of $\mathcal{T}_\varepsilon$. Some relevant properties of a congruence relation are as follows.

A congruence relation R over a set $\mathcal{H}$ is a binary equivalence relation that is compatible with all the functions defined over $\mathcal{H}$. A binary relation R over a set $\mathcal{H}$ is a set of pairs of $\mathcal{H}$ elements, namely a subset of $\mathcal{H} \times \mathcal{H}$. An equivalence relation R over a set $\mathcal{H}$ divides the elements of $\mathcal{H}$ into so-called *equivalence classes*. Two elements belong to the same equivalence class, if and only if they are in relation by R and in that case the elements are called *equivalent* in R. The *equivalence closure* is the smallest set that contains R. Similarly, a congruence relation R over a set $\mathcal{H}$ divides the elements of $\mathcal{H}$ into so-called *congruence classes*. Two elements belong to the same congruence class, if and only if they are in relation by R, in which case the elements are called *congruent* in R.

In the context of the thesis, the focus is on the *congruence closure* of a set of equalities. Since, the equality predicate is symmetric, equalities will be treated as unordered pairs of terms on the level of notation, i.e., an equality $s = t$ will denote indifferently $s = t$ or $t = s$. A set of equalities built over $\mathcal{G}$ is denoted as E, while $E_{\neq}$ denotes the set of disequalities over $\mathcal{G}$. The congruence closure of a set E, denoted as E^*, is the smallest congruence relation over $\mathcal{G}$ that contains E. The expression $E \models s = t$ denotes that two terms s and t belong to the congruence closure of E, namely $(s, t) \in E^*$. When for all equations $s = t$ in a set E' holds that $E \models s = t$, then $E \models E'$. The expression $E \equiv E'$ denotes that $E \models E'$ and $E' \models E$ holds.

2.3 EUF Solver of SMT4J

The primary purpose of the thesis is to extend the EUF theory solver of SMT4J with unsatisfiable core and detailed proof generation abilities. SMT4J is an educational lazy SMT solver that supports satisfiability decisions in multiple quantifier free theories with (limited) proof-production capabilities. It is under development by the Institute for Formal Models and Verification (FMV) of the Johannes Kepler University Linz. The essential strategy employed by the SMT4J Solver is the lazy so-called *lemmas on demand* approach. Algorithm 2.1 from [25] contains the basic pseudo-code that implements, in a simplified way, a decision procedure called *LAZY-BASIC*, that exploits lemmas on demand.

Algorithm 2.1 LAZY-BASIC (from [25])

```
1:  function LAZY-BASIC($\varphi$)
2:      $\mathcal{B} \leftarrow e(\varphi)$
3:      while (TRUE) do
4:          $\langle \alpha, res \rangle \leftarrow$SAT-SOLVER($\mathcal{B}$);
5:          if $res =$ "Unsatisfiable" then return "Unsatisfiable";
6:          else
7:              $\langle t, res \rangle \leftarrow$DEDUCTION($\hat{Th}(\alpha)$);
8:              if $res =$ "Satisfiable" then return "Satisfiable";
9:              else
10:                 $\mathcal{B} \leftarrow \mathcal{B} \wedge e(t)$;
```

The key idea of the approach is to combine a theory specific decision procedure of a theory $\mathcal{T}$ with a propositional SAT solver in order to decide satisfiability of a formula φ in a given theory $\mathcal{T}$. As a preparatory step (line 2 of Alg. 2.1) φ is encoded into a Boolean formula $\mathcal{B}$ by substituting each $\mathcal{T}$-related literal in φ with a unique Boolean variable. The resulting formula of the encoding is called the *propositional abstraction* or *skeleton* of φ. Next, an unconditional loop commences. In each iteration, $\mathcal{B}$ is passed to a SAT-Solver which returns with an assignment α and a result *res*. In the case $\mathcal{B}$ is unsatisfiable, there is no assignment that could satisfy it, therefor α is empty and the method returns with *"Unsatisfiable"* as a result (line 5). Otherwise *res* is *"Satisfiable"* and α defines a mapping of the literals of $\mathcal{B}$ to truth values (*propositional assignment*) in a way that $\mathcal{B}$ is satisfied. Then, in line 7, $\hat{Th}(\alpha)$, a conjunction of the involved literals of φ assigned by α, is passed to the method DEDUCTION, in order to test whether $\hat{Th}(\alpha)$ satisfies φ in $\mathcal{T}$. DEDUCTION returns with an answer *res* for this question and with a clause t. If the result of DEDUCTION is *"Satisfiable"*, *LAZY-BASIC* returns *"Satisfiable"* in line 8. Otherwise DEDUCTION generates a $\mathcal{T}$-valid lemma t, that contradicts on the Boolean level with α (and optionally with further extensions of this assignment). In line 10, the method conjoins the propositional abstraction of this lemma ($e(t)$) with the current state of $\mathcal{B}$ and starts again the iteration with this strengthened formula [25]. For some elaborated applications of this approach see [12, 16].

The design of SMT4J follows this approach closely. The SMT4J framework coordinates the collaboration between a SAT Solver and the theory solvers in order to iteratively refine the propositional abstraction of the input formula based on the lemmas generated on demand by the theory solvers. The set of theories supported by SMT4J depends on the actual configuration of the framework. The supported theories and their solvers are loaded dynamically based on a configuration file, where the properties and options of the solvers can be fine tuned. Although SMT4J also supports, with limitations, combinations of theories, the way of combining multiple theories is beyond the scope of the thesis. Therefore, henceforth it is assumed, that the input of the framework contains only propositional and

$\mathcal{T}_\varepsilon$ atoms connected by boolean operators and that initially the EUF Solver is loaded by SMT4J. The configuration file of this scenario can be found on Figure A.1 of Appendix A.

The instantiation of the SMT4J framework considered in the thesis consists of the following parts:

Parser: Converts the input text file written in the SMT-LIBv2 [7] language into an input formula represented by Java objects. It is made up of a lexical analyser and a grammar parser, generated with jflex[1] and byaccj[2], respectively.

Encoder: Performs the semantic analysis of the input formula and applies several simplification steps (e.g. expansion and purification of the terms) on it. The Encoder constructs the CNF propositional abstraction of the input formula by using polarity based Plaisted-Greenbaum encoding [32] and provides the necessary encoding-decoding steps during the execution.

SAT Solver: Decides incrementally the satisfiability of a CNF propositional formula and provides satisfying (partial or full) assignments in the case of positive decisions and proofs in the case of negative decisions. The framework uses the SAT4J-Core[3] open source SAT Solver as a 'black box' for this purpose without any further tuning.

Extractor: Extracts a set of theory related literals from the input formula, based on the model generated by the SAT Solver.

EUF Solver: Decides the satisfiability of a set of $\mathcal{T}_\varepsilon$-literals. When the set is found to be inconsistent, the component extracts the literals which are involved in a conflict, and computes a deductive proof to explain that conflict. The EUF Solver is the solver of the equalities over uninterpreted functions theory.

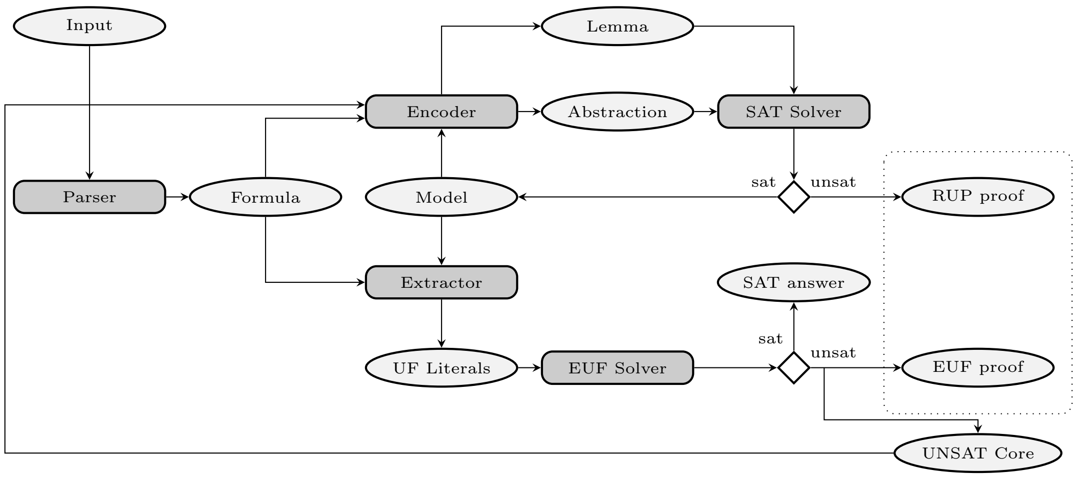

Figure 2.1: Simplified workflow of SMT4J and EUF theory solver

Figure 2.1 illustrates – without claim of being exhaustive – a simplified workflow of SMT4J when the extended EUF Solver is loaded. The input of the framework is a SMT-LIBv2 file. The parser of the framework constructs from this input file a formula φ, which represents a satisfiability problem with $\mathcal{T}_\varepsilon$ as background theory. In the second step, the Encoder processes φ, by checking the type correctness of the formula and applying several simplification steps on it. Then it converts this simplified formula into CNF and constructs

[1]http://jflex.de/
[2]http://byaccj.sourceforge.net/
[3]http://www.sat4j.org/maven234/org.ow2.sat4j.core/index.html

the propositional abstraction ($e(\varphi)$) of it. After these steps, the SAT Solver can decide the satisfiability (on the boolean level) of $e(\varphi)$. If $e(\varphi)$ is decided to be propositionally unsatisfiable, then it is unsatisfiable in the EUF theory as well. Therefore the framework returns with **unsat** as result. Otherwise, the SAT Solver provides a propositional model, which has to be checked for $\mathcal{T}_\varepsilon$-consistency by the EUF Solver. For this, the propositional model generated by the SAT Solver and the original input formula φ is passed to the Extractor. It extracts the corresponding literals (i.e., the literals whose abstraction appear in the model) of the input formula into a set $\mu = \{l_1, l_2, ...l_n\}$ based on their assignment in the abstract model. Then, this result set μ is incrementally added to the EUF Solver. If μ is found to be consistent in $\mathcal{T}_\varepsilon$, then the framework accepts the model as a satisfying interpretation and returns with **sat** as result. In case μ is $\mathcal{T}_\varepsilon$-unsatisfiable, the extension of the EUF Solver begins its work. Namely, it produces an unsatisfiable core η, which is a subset of μ and contains some literals whose conjunction is still unsatisfiable. More precisely η can be defined as an arbitrary set of literals where $\eta \subseteq \mu$ and $\eta \models_\mathcal{T} \bot$ holds. As soon as the EUF Solver returns with η, the Encoder converts all the literals back to the propositional level by using the abstract model and builds up a reduced conflict clause $e(\eta)$ from it. Then $\neg e(\eta)$ becomes a so called *theory lemma*, which can be conjuncted to $e(\varphi)$. With this step, the SAT solver is prevented from finding the same model again and can search for further solutions in a narrowed space. The refinement of the abstraction by these theory lemmas is repeated until the SAT solver either finds a $\mathcal{T}_\varepsilon$-consistent model or returns **unsat**.

Since the theory lemmas produced by the EUF Solver do not introduce new atoms into the formula, the number of possible propositional assignments is finite. In each iteration at least one of these satisfying assignments is ruled out, therefore the procedure will terminate after a finite number of iterations. The following example illustrates the decision procedure on a formula.

Example 2.13 (Methodology)**.** Assume that the following φ formula is given as an input to the framework:

$$\underbrace{x = y}_{a} \wedge \underbrace{y = z}_{b} \wedge \underbrace{f(z) \neq t}_{c} \wedge (\underbrace{x \neq z}_{d} \vee \underbrace{f(x) = t}_{e}).$$

The propositional abstraction ($e(\varphi)$) of this formula is $a \wedge b \wedge c \wedge (d \vee e)$. Assume that the SAT Solver finds the $\{a \mapsto true, b \mapsto true, c \mapsto true, d \mapsto true, e \mapsto false\}$ satisfying assignment. The extraction of the $\mathcal{T}_\varepsilon$-literals based on this assignment produces the $\{x = y, y = z, f(z) \neq t, x \neq z\}$ set of literals as input for the EUF Solver. The theory solver decides that this set of literals is inconsistent in $\mathcal{T}_\varepsilon$, and returns with the conflict set $\{x = y, y = z, x \neq z\}$. The Encoder component constructs from this set the theory-lemma of the form $\neg a \vee \neg b \vee \neg d$ and sends it to the SAT solver as a new clause of φ. Then the SAT Solver has to find a model for this extended formula: $a \wedge b \wedge c \wedge (d \vee e) \wedge (\neg a \vee \neg b \vee \neg d)$, and returns with $\{a \mapsto true, b \mapsto true, c \mapsto true, d \mapsto false, e \mapsto true\}$. Now the EUF Solver has to decide the $\mathcal{T}_\varepsilon$-satisfiability of the $\{x = y, y = z, f(z) \neq t, f(x) = t\}$ set of literals. This set is also inconsistent on the level of the EUF theory, therefore the theory solver constructs the $\{x = y, y = z, f(z) \neq t, f(x) = t\}$ unsatisfiable core. The encoding of these literals results in the $a \wedge b \wedge c \wedge e$ formula, so the new theory-lemma is $\neg a \vee \neg b \vee \neg c \vee \neg e$. The framework extends φ with this new clause and now the SAT Solver has to find a model for the formula: $a \wedge b \wedge c \wedge (d \vee e) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg b \vee \neg c \vee \neg e)$. This propositional formula is unsatisfiable, therefore SMT4J returns with **unsat**.

When a formula is found to be unsatisfiable, SMT4J is expected to produce a proof about the inconsistency of the input formula as a compound of propositional and theory specific proofs. The propositional part of this produced proof is a reverse unit propagation

(RUP) refutation that explains why the conjunction of the input formula together with the theory lemmas are unsatisfiable. For the details of this clausal proof form see [41]. The other part of the proof explains the inconsistency of each unsatisfiable core, which were produced by the theory solvers, in order to validate the theory lemmas used in the propositional proof. So far, this feature of the solver was not implemented and this is one of the main contributions of this thesis. The SMT-LIBv2 input and the proof output of Example 2.13 can be found in Appendix A in Example A.2 and Figure A.3.

2.4 Union-Find Data Structure

The heart of the EUF Solver is a congruence closure algorithm [29] which is based on the union-find data structure. This structure is applied in many graph related algorithms and provides an efficient way to handle multiple non-overlapping sets, such as equivalence classes. A detailed description of the data structure can be found in [15]. The structure maintains a collection of non-empty disjoint sets: $C = \{C_1, C_2, .., C_n\}$. Each set is identified by its so-called *representative*, which is a certain element of the set. As long as a set C_i is not modified, the representative of it, denoted C'_i, has to stay the same element. The surjective map of an element t to its corresponding set is denoted by $s(t)$. The operations supported by this data structure are the following:

makeSet(t): Initializes the data structure by creating the $C_t = \{t\}$ singleton set, where the representative C'_t is t.
Precondition: $\nexists i \in \{1..n\} : t \in C_i$

union(t_1,t_2): Merges the two disjoint sets which contain t_1 and t_2 into a new set. The representative of the new set depends on the details of the implementation.
Precondition: $s(t_1) \cap s(t_2) = \emptyset$

find(t): Returns t', the current representative of the set which contains t.
Precondition: $\exists i \in \{1..n\} : t \in C_i$

Different realizations of this data type can be found in the text book by Cormen et al. [15]. In this thesis, a proof producing congruence closure algorithm is realized that extends the so-called disjoint set forests implementation. In this realization each set is represented by a directed rooted tree, where a node is an element of the set and the root node is the representative of the set. Each node points only to its parent (parent pointer representation), and the root node points only to itself. Figure 2.2a shows an example of a disjoint set forest. With this representation, the operations work as follows: The *makeSet(t)* method creates a new tree with a single node, whose parent is itself. The *find(t)* method traverses through the parent pointers starting from t, until it finds the root of the tree. The *union*(t_1, t_2) method directs the parent pointer of either t'_1 to t'_2 or t'_2 to t'_1. The illustration of this operation can be seen on Figure 2.2b.

The worst case time complexity of the operations on this data structure can be expressed based on two numbers: n, the number of *makeSet* operations, and m, the sum of the number of *makeSet*, *find* and *union* methods. Since the sets are disjoint, each *union* method decreases the number of sets by one, therefore the *union* method can be called maximum *n-1* times. Furthermore, $m \geq n$ since m also contains the number of *makeSet* calls. The effectiveness of the data structure strongly depends on the height of the trees in the forest, therefore perceptible improvement can be achieved by reducing them. Two simple heuristics for that purpose are as follows.

Weighted union: If the *union* operation connects the trees in an arbitrarily way, then $n-1$ *union* operations might, in the worst case, result in an n-long chain of nodes. To

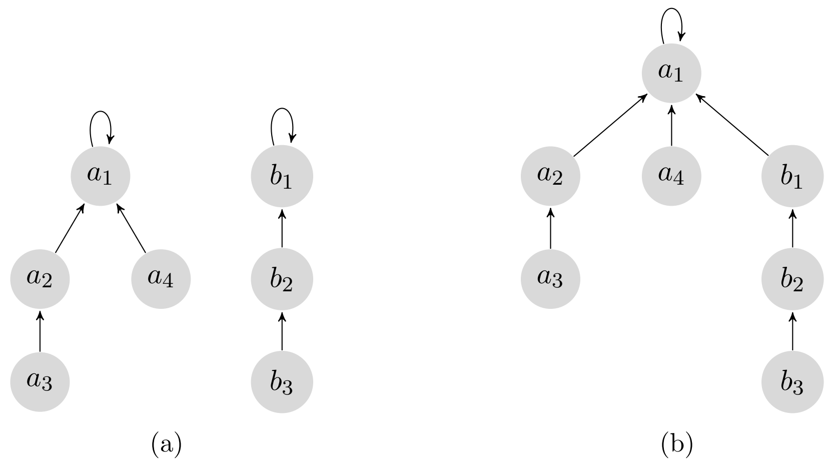

Figure 2.2: An example of a disjoint set forest adapted from [15]. **(a)** Represents two sets: $\{a_1, a_2, a_3, a_4\}$ with a_1 as representative and $\{b_1, b_2, b_3\}$ with b_1 as representative. **(b)** The forest after the union of e.g. a_2 and b_2.

avoid this situation, the *union* operation decides the direction of the new connection based on the size of the involved trees, i.e., it makes the smaller tree's root point to the root of the bigger tree. This heuristic reduces the upper-bound of the tree heights in a disjoint set forest from n to $log\ n$. The requirement of this optimization is to maintain the size information of each tree, which can be achieved in many different ways with minimal costs.

Path compression: This optimization extends the *find* operation with a further step. After the traverse from t to the root of the tree (t'), the parent pointer of t and all its ancestors (except t') are changed to point directly to t'.

In a disjoint set forest of n elements, the worst case running time of an m long sequence of *makeSet*, *find*, and *union* operations, using weighted union and path compression simultaneously, is $O(m\ \alpha(n))$, where $\alpha(n)$ denotes the inverse of the Ackermann function. The Ackermann function is a simple μ-recursive function with an extremely fast rate of growth, therefore α is a function with extremely slow rate of growth (for all practical $n\ \alpha(n) \leq 4$). For the precise derivation of the function α and the computation of the algorithms' complexity see [15].

Chapter 3

Proof Generation of SMT Solvers

This chapter is dedicated to demonstrate the necessity of proof production capabilities in the context of SMT Solvers and to present briefly a few recently employed proof systems. Further, the proof format of the EUF theory solver is introduced in details during this chapter.

SMT Solvers are fairly complicated systems with a large, frequently growing code base. Hence, the objective to ensure the proper functioning of them is highly challenging and the idea to build the system by verified code is not a practically feasible solution. Therefore, some kind of evidence is expected from a solver, that can be checked with an external trusted tool in order to achieve a higher level of confidence about the correctness of the solver. This method discharges the question of trust from a complex system to a simpler verifier tool. Another relevant application domain of the produced proofs is to integrate the results of an SMT Solver into proof assistants or similar systems, without expecting that the tool trusts in the decision procedure. Nevertheless, this aspect of the proof systems is beyond the scope of this thesis. For SAT Solvers the produced evidence is usually either a satisfying assignment for a satisfiable problem instance or a resolution proof for an unsatisfiable problem instance. However, SMT Solvers combine multiple decision procedures during a decision process, therefore their generated certificates are supposed to mirror this complexity. This is one of the main difficulties in finding a sufficient proof system for SMT Solvers.

Another challenge in proof production is in finding the sufficient balance between proof granularity and performance of the solver and the difficulty of the verification process, namely to decide the suitable set of rules that already supports the verification but still not ruins the performance of the solver significantly. If more details are included in the produced proofs, the implementation of the solver becomes more complicated, but the verification process is easier. Nevertheless, if the aim is to keep the code base of the solver simple and the performance untouched, the proof-checker tool has to be more sophisticated and expensive in general. From this point of view, an unsatisfiable core is the simplest proof that an SMT Solver or a theory solver can produce. It narrows down the unsatisfiable problem to a smaller one, but says nothing about how the contradiction is derivable from it. This amount of information requires a more elaborated method to verify. On the other hand, a detailed derivation of a contradiction guides the verification process by referring to the inference rules and axioms of the corresponding theory. For producing these details, the solver has to maintain several types of information during the solving process, which are not necessarily required for the decision. This fact indicates an unavoidable overhead of the proof production. The degree of this overhead mainly depends on the chosen rule steps. A shallow proof describes the biggest steps that are necessary to derive a contradiction, while on the other end, a proof can even contain every necessary steps with clarified semantics. In the former case, during verification the holes of the proof has to be filled, searches that were conducted by the solver already has to be repeated, while in the latter case, the verification process is reduced only to verify each step based on their definition. All in all, the desired supreme properties of a certificate always depends mainly on what the proof is destined for.

Although, for the moment there is no standard or even agreement about the produced certificates of SMT Solvers, there are some recommendations that may be considered during the development of this functionality [10]. First of all, SMT Solvers should share their proof format in order to re-use the already implemented checkers and other proof-based tools and to facilitate proof exchange between systems. Due to the lack of standard and high diversity of solvers, currently this suggestion is hard to follow. The produced proofs are mostly parsed and used with further automatic tools, still it is advisable to choose a format that is readable by the users. The produced certificates of an SMT Solver should be preciously designed, namely the granularity of the produced proofs should not be defined purely by the internal data structures and implementation details of the solver. Furthermore, it is worth to consider to include not only the applied inference rules in the proof format, but additionally the explicit derived formulas also. In that way, the uncertainty of the semantics may reduced. In any case, the most important feature for a successful adoption of a format is to provide an exhaustive documentation of syntax and semantics of the proof system in order to facilitate the application of the proof for verification. For more detailed description of these suggestions see [10].

3.1 Overview

Since the importance of proof certificates is recognized already in SMT community, more and more SMT Solvers provide proof certificates. Some recently popular SMT Solvers with proof production capabilities are for example Z3, veriT and CVC4. Most of the SMT Solvers have their own unique proof system and the aim of this section is to present some of them briefly. The focus in this thesis is restricted to reasoning in the EUF theory, therefore several aspects of the proof systems are not discussed here. For a more detailed survey of current proof formats see e.g. [6]. In Figure 3.1 a very simple formula is presented in SMT-LIBv2 format. This formula will be used as input henceforth to illustrate the proof formats of the different solvers.

```
(set-logic QF_UF)
(declare-sort S 0)
(declare-fun f (S S) S)
(declare-fun a () S)
(declare-fun b () S)
(declare-fun c () S)
(assert (= a b))
(assert (= a c))
(assert (distinct (f b a) (f c a)))
(check-sat)
(exit)
```

Figure 3.1: A simple example problem instance in SMT-LIBv2 format.

3.1.1 Proofs of Z3

Z3[1] is developed at Microsoft Research and is one of the most popular SMT Solvers nowadays due to the high performance provided by it. It has model generation and proof production abilities as well. The architecture of the certificates produced by Z3 is rather

[1]https://github.com/Z3Prover/z3

unique. Figure 3.2 shows an example certificate produced by Z3. First of all, the proof-objects at the propositional level are not the customary resolution proofs but follow more a natural deduction style. The certificates are represented as proof terms, where function symbols representing the inference rules. The last argument of a proof term is always the consequent of the proof.

```
((set-logic QF_UF)
(proof
(let ((?x35 (f c a)))
(let ((?x34 (f b a)))
(let (($x38 (= ?x34 ?x35)))
(let (($x29 (= a b)))
(let ((@x30 (asserted $x29)))
(let ((@x48 (monotonicity (trans (symm (asserted (= a c)) (= c a)) @x30 (= c b)) (=
     ?x35 ?x34))))
(let (($x39 (not $x38)))
(let ((@x44 (mp (asserted (and (distinct ?x34 ?x35) true)) (rewrite (= (and (
    distinct ?x34 ?x35) true) $x39)) $x39)))
(unit-resolution @x44 (symm @x48 $x38) false))))))))))
```

Figure 3.2: Proof of Z3 (version 4.4.0)

Reasoning in the EUF-theory in the Z3 Solver is based on all axioms of the theory, namely even symmetry is handled explicitly in the proofs. The theory axioms are encoded as inference rules (*refl*, *symm*, *trans* and *monotonicity*). In Z3, the equality relation is treated as a logical function that is also contained by the congruence classes. This feature is a relevant difference compared to the congruence closure algorithm implemented in this thesis. For more details about the proof generation capability of Z3 see for instance [17].

3.1.2 Proofs of CVC4

CVC4 is the new version of the Cooperating Validity Checker series, that is currently one of the state-of-the art SMT Solvers. It is developed under the lead of Clark Barrett and Cesare Tinelli and supports several background theories [5]. The proof production capability of the solver is still in progress, nevertheless it is designed to have no influence on the performance of the solver, when it is compiled without this function. The proof format of CVC4 is based on a (dependently) typed lambda calculus, more preciously formulated in the Edinburgh Logical Framework with Side Conditions (LFSC) meta-language. LFSC as a standard for SMT proof format was proposed by Stump et al. [37, 38, 39]. This flexible language facilitates to describe proof systems with the support for the independent verification of the certificates. The language of the certificates follow a functional view of proofs, where the inference rules are represented as functions from proofs to proof. More preciously, in LFSC a proof systems is encoded as a collection of typing declarations.

Figure 3.3 illustrates the proof of unsatisfiability for the example input formula by CVC4. The declared signatures in CVC4 cover already the preprocessing steps of the solver (e.g. CNF clausification), natural deduction steps, resolution steps and theory specific reasoning for example in the theory of equalities over uninterpreted functions, bit-vectors and arrays. Each theory solver logs its deductive steps in a separated component during the solving process and in the final produced proof these components are glued together. The proof begins with the *check* keyword that indicates that the produced proof is already an object of the verification process. The first part of the proof trace defines the involved terms of the input formula. The input formula contains three assertions and the trace refers to them as holding theories. The derivation of the empty clause starts by the *(: (holds cln)* expression. The next line gives to the atomic formula *false* a propositional ($v1$) and a logical atom ($a1$) name that can be used in the later steps. The wild-card symbol in the proof trace means that the value of the argument of the function application is computed

```
(check
(% S sort
(% f (term (arrow S (arrow S S)))
(% a (term S)
(% b (term S)
(% c (term S)
(% A0 (th_holds true)
(% A1 (th_holds (= S a b))
(% A2 (th_holds (= S a c))
(% A3 (th_holds (not (= S (apply _ _ (apply _ _ f b)a) (apply _ _ (apply _ _ f c)a)
    )))
(: (holds cln)
(decl_atom false (\ v1 (\ a1
 ;; Input Clauses
(satlem _ _ (ast _ _ _ a1 (\ l3  (clausify_false trust)))( \ pb1
(satlem _ _ (asf _ _ _ a1 (\ l2  (clausify_false trust)))( \ pb2
 ;; Theory Lemmas
(satlem_simplify _ _ _ (R _ _ pb2 pb1 v1)(\empty empty))))))))))))))))))))
```

Figure 3.3: Proof of CVC4 (version 1.4)

by the types of the further arguments. Although CVC provides proof production for the EUF theory, the recent version of the solver instead of declaring the precise proof of a theory lemma, employs the decision of the theory solver as a trusted step. Therefore, in Figure 3.3 no theory lemmas are depicted explicitly. The last step of the proof shows the derivation of the empty clause by a propositional resolution tree.

3.1.3 Proofs of veriT

Besson et al. [8] propose a flexible, generic proof format for SMT Solvers that is supposed to be easy to generate and that at the same time could be verified with a trustworthy external tool in an efficient way. The proposed proof format is a direct extension of the SMT-LIB syntax, therefore provides familiar environment for solver implementers. The main intent of the proposal is to facilitate a generic format that can be fine-tuned by each SMT Solver based on the actual requirements, that is to say, it can be parametrized by theory solver-specific proof rules. It is recently employed in the veriT SMT Solver, that is developed in the cooperation of LORIA (Nancy, France) and UFRN (Natal, Brazil). In this solver a unique module is responsible for the management of the proof-related information and currently the QF_UF, QF_IDL, QF_RDL, QF_IDL theories are covered by proof production capability. Although the quantifier handling in the proof format of veriT is still under development (see [18]), the skolemization process is already fully proof producing. The generated proof[2] of veriT for the simple example input problem instance is given in Figure 3.4.

As it is obvious from the figure, the proof in this format contains a sequence of *set* commands, where each command introduces a new clause. A clause in this context is a set of arbitrary formulas instead of literals, for instance the clause (not (= c b)) (not (= a a)) (= (f c a) (f b a)) (clause c5) means the disjunction of the formulas $\neg(c = b)$, $\neg(a = a)$ and $f(c, a) = f(b, a)$. The empty clause is denoted as '()'. The proof trace begins with the input formulas (c1-c3) and ends with the final empty clause. In each step elementary inference rules are employed. The inference rules in this format refer to arbitrary many clauses as premises and conclude a single clause. The fine-grainedness of the rules in veriT are planned to be high, namely each rule is as small and simple as possible. The resolution rule represent chain resolution in veriT. The result of each resolution step is explicitly defined in the proof (not like in the LFSC format). Rules of c5, c6 and c8

[2]The longer lines of the proof are slightly rearranged compared to the original output result.

```
(set .c1 (input :conclusion ((= a b))))
(set .c2 (input :conclusion ((= a c))))
(set .c3 (input :conclusion ((distinct (f b a) (f c a)))))
(set .c4 (tmp_distinct_elim :clauses (.c3) :conclusion (
          (not (= (f c a) (f b a))))))
(set .c5 (eq_congruent :conclusion (
          (not (= c b)) (not (= a a)) (= (f c a) (f b a)))))
(set .c6 (eq_transitive :conclusion ((not (= a c)) (not (= a b)) (= c b))))
(set .c7 (resolution :clauses (.c5 .c6) :conclusion (
          (not (= a a)) (= (f c a) (f b a)) (not (= a c)) (not (= a b)))))
(set .c8 (eq_reflexive :conclusion ((= a a))))
(set .c9 (resolution :clauses (.c7 .c8) :conclusion (
          (= (f c a) (f b a)) (not (= a c)) (not (= a b)))))
(set .c10 (resolution :clauses (.c9 .c1 .c2 .c4) :conclusion ()))
```

Figure 3.4: Proof of veriT (version 201410) without term sharing option for improved readability.

introduce valid clauses of the EUF theory, as it will be described more detailed in the further section.

The proposal of this format suggests an extension of the SMT-LIB language with a new *(get-proof-header)* command, that could print the definitions of the implemented rules of the solver. This is the only location for the proof system where the definition of the employed inference rules can take place and this description can be informal. Therefore, the proper formal documentation of the proof system may be incomplete. This can raise difficulties for the users of the produced proofs. veriT provides this information through an external option of the solver.

3.2 Proof Format of EUF Solver

A complete proof that is produced by SMT4J contains a propositional refutation (in RUP form) filled with theory-related sub deductions of the theory lemmas. This section describes in details the format of the proofs generated by the EUF theory solver of SMT4J.

These proofs are justifications of the $\mathcal{T}_\varepsilon$-lemmas and are based on $\mathcal{T}_\varepsilon$-specific reasoning. The theory solver is expected to collect all the necessary information to generate these justifications without ruining the efficiency of the framework significantly. These certificates serve first of all as evidences in order to verify the correct functioning of the theory solver, therefore they are supposed to be well detailed with simple and atomic proof steps. Furthermore, the produced proofs provide a deeper insight into the reasoning mechanism of the theory solver. Thus, readability is an additional important requirement. The chosen proof format of the theory solver follows strongly the proposed certificate format of the veriT SMT Solver. This format is based on natural deduction and easy to read, that is to say, user friendly. Furthermore, it follows the main principles of the SMT-LIBv2 language, thus requires only short time to grow familiar with the format. At the moment, the produced proofs of SMT4J cover only the theory lemmas produced by the EUF theory solver, therefore only a subset of the original proof format is employed.

Each proof of a theory lemma consists of a *context* and a sequence of *proof steps*, where the context means the SMT-LIBv2 definition of the involved input assertions. By default the context section is not printed in the detailed proofs, but on request can be part of the output. Each proof step constructs a new Horn clause by using the *gen_clause* generation rule, where the new clause is identified with a *clause_id*. The gen_clause rule represents clause derivation either by referring directly to an already constructed clause, or by using a named *rule*. The syntax of the produced proofs based on [8] is as follows:

$$\begin{aligned}
\langle proof\rangle &::= \langle context\rangle\ \langle proof\_step\rangle^{*}\\
\langle proof\_step\rangle &::= (set\ \langle clause\_id\rangle\langle gen\_clause\rangle)\\
\langle gen\_clause\rangle &::= \langle clause\_id\rangle\\
&\quad\mid\ \ (\langle rule\_id\rangle\\
&\qquad (:\ clauses\ (\langle gen\_clause\rangle^{*}))?\\
&\qquad (:\ conclusion\ \langle clause\rangle)?)
\end{aligned}$$

A proof produced by the EUF theory solver contains always a single conflict causing disequality (which was asserted as true in the input) and the derivation of the negation of this disequality from other input assertions or further $\mathcal{T}_\varepsilon$-valid formulas. Therefore, the constructed clauses can be grouped into two categories. One set of the clauses in the proof contains only unit clauses, where one of them is the conflict causing disequality and the further clauses are the asserted equalities which are necessary to derive the contradiction. This set of atomic formulas is the unsatisfiable core. Example 3.1 illustrates this part of the proof.

Example 3.1. Assume that the unsatisfiable core of a contradictory set of literals is the set of (dis-)equalities $\{a = c, f(a,d) \neq f(c,d)\}$. Then the produced proof contains the following clauses related with the core:

```
(set .c1 (input :conclusion ((= a c))))
(set .c2 (input :conclusion ((distinct (f a d) (f c d)))))
```

The other set of clauses of a produced proof contains derivations from the asserted input equalities or further $\mathcal{T}_\varepsilon$-valid formulas. One inference rule is to represent the elimination of the distinct predicate from the conflict causing disequality. In the SMT-LIBv2 language, the predicate symbol *distinct* is part of the core theory and denotes a logical function that returns true if and only if the two arguments of it are not identical. Nevertheless, in the theory of equalities over uninterpreted functions only equalities can be derived, therefore this function has to be transformed to an equality predicate in order to reach contradiction. Therefore, in each produced proof one step stands for representing this transformation:

```
(set .c3 (tmp_distinct_elim :clauses (.c2) :conclusion (not (= (f a d) (f c d)))))
```

A derivation step in the proof trace can represent valid formulas constructed based on the axioms of the EUF theory (see Section 2.2), namely it introduces a tautology of $\mathcal{T}_\varepsilon$ in clause form. Since the theory solver handles symmetry only implicitly, the formulas constructed by Axiom 2.2 are not included in the proofs. An example instantiation of the considered axioms are as follows.

Example 3.2. Assume the following instantiation of the $\mathcal{T}_\varepsilon$-axioms:

$$\begin{gathered}
t = t\\
s = t \wedge t = u \supset s = u\\
a = c \wedge b = d \supset f(a,b) = f(c,d)
\end{gathered}$$

After the elimination of the implications the formulas become the following Horn clauses:

$$\begin{gathered}
t = t\\
\neg(s = t) \vee \neg(t = u) \vee s = u\\
\neg(a = c) \vee \neg(b = d) \vee f(a,b) = f(c,d)
\end{gathered}$$

Then the representation of these clauses in the proof language of veriT and the EUF theory solver of SMT4J are as follows:

```
(set .c1 (eq_reflexive :conclusion ((= t t))))
(set .c2 (eq_transitive :conclusion (not (= s t)) (not (= t u)) ((= s u))))
(set .c3 (eq_congruent :conclusion (not(= a c)) (not(= b d)) ((= (f a b)(f c d)))))
```

Although, the transitivity axiom defines an implication based on exactly two equalities, in the generated proofs this rule refers to the combination of multiple transitivity steps. By the default settings, the EUF Theory solver produces all the deduction steps, but the connection between them, namely the derivation of the empty clause from them is generated just on request. The precise definition of deduction based on resolution is beyond the scope of this thesis, for further details see e.g. [21]. The resolution rule in the context of the generated proofs are binary (without refactoring), except the very last step of each proof, where the resolution step is expressed as a chain resolution to resolve the last resolvent with the conflict causing literal and with all the input assertions.

The theory solver provides a minimal informal definition of the here explained inference rules and clauses in a form of header definition of the inference rules. The produced proof of the EUF Theory solver for the example input formula of Figure 3.1 will be explained in detail during the next chapter (see Example 4.11).

Chapter 4

EUF Solver Implementation

The first and foremost responsibility of the EUF theory solver in the SMT4J Solver is to decide whether a conjunction of ground $\mathcal{T}_\varepsilon$-literals is satisfiable. This problem can be approached from various directions and there are various algorithms dedicated to solve this decision problem. Some, but far from all, of these approaches and algorithms can be found in [3, 19, 28, 34]. In general, the common characteristic of these algorithms is that they solve the decision problem through congruence closure. Namely, the methodology of the algorithms is as follows. As a first step, the congruence closure of the given equalities is calculated and the constructed classes are stored. Then the disequalities are examined one-by-one, whether they contradict with the obtained congruence classes or not. A contradiction is found, when a disequality of the input is defined between two terms that belong to the same congruence class. The implementation of the EUF Solver of SMT4J realizes this method by following closely the congruence closure algorithm described by Robert Nieuwenhuis and Albert Oliveras in [29].

Besides the decision of the satisfiability of a given collection of literals, the theory solver is expected to provide several further functionalities in order to simplify and optimize the work of the SMT4J Solver. These features are as follows.

Incremental processing: The theory solver is expected to store the state of its computation between two calls. In practice it means, that whenever a set of input literals is processed by the theory solver and found to be satisfiable, the extension of the problem with further literals is supported. In these cases the solver continues the computation, i.e., modifies the already existing congruence classes based on the additional equalities, without restarting the process. In that way significant acceleration of the decision process can be accomplished. This feature is desirable mainly in the situations, when the theory solver has to interact with other theory solvers.

Deduction of interface equalities: The theory solver is expected to be able to derive further equalities from a satisfiable set of equalities. These derived equalities are the so-called *interface equalities* and are mainly used in the case when multiple theory solvers work together in order to solve a decision problem with more than one background theory. For further details about theory combination and the role of interface equalities in it, see [27]. The produced interface equalities are logical consequences (in first-order logic with equality) of the input conjunction of theory literals and they are defined over ground terms. However, these terms not necessarily occur in the input formula.

Conflict set generation: An unsatisfiable core of an unsatisfiable set of literals is essential in order to produce effective $\mathcal{T}_\varepsilon$-lemmas, therefore in each case, when the conjunction of the input literals is found to be unsatisfiable, a conflict set is expected to be produced by the EUF Solver. The generated inconsistent conflict set contains exactly one disequality and an arbitrary number of equalities from the input set of (dis-)equalities, namely it can contain only literals of the input formula.

Detailed proof generation: The theory solver is expected to be able to generate a detailed, human-readable and verifiable proof for each theory lemma which was found by the EUF Solver. Each proof explains the contradiction of one specific conflict set and therefore, the premises of the proofs are just literals of the input formula or tautologies in the equalities over uninterpreted functions theory.

Model generation support: In the case, when the conjunction of the input literals is found to be satisfiable by the theory solver, the model that would justify this consistency, cannot be produced without having a knowledge of the interpretation of the functions and sorts. Therefore, the solver can just support but not accomplish the model generation, by providing the constructed equivalence classes and the required disequalities among them to the framework.

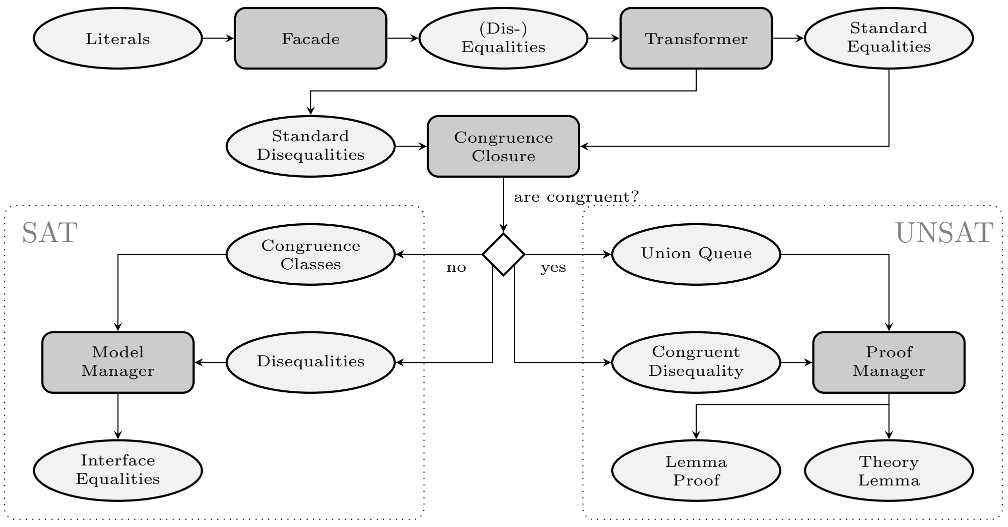

Figure 4.1: Simplified workflow of EUF theory solver

The implementation of the EUF theory solver is based on the following main components.

Facade: Provides a unified and simplified high-level interface to the functionalities of the theory solver. Any interaction with the EUF theory solver is possible only through this component. This interface facilitates the usage of the theory solver by hiding the complexity of the inner architecture and shields the framework from the changes of the theory solver.

Transformer: Responsible for the correct transformation of the input literals into the internal representation (standard form) of the EUF theory solver. Stores the alphabet and signature information of the occurring terms and reconstructs the proper equalities and disequalities from the output of the theory solver. Details are discussed in Section 4.1.

Congruence Closure: Processes incrementally a set of standard equalities (c.f. Def. 4.7) in order to be able to decide, whether an equality belongs to the congruence closure of the processed equalities or not. During the process, stores the newly discovered congruent deductions and the already applied congruence class merge operations. The core module of the EUF theory solver. Details are discussed in Section 4.2.

Proof Manager: Produces the unsatisfiable cores for each contradiction found by the congruence closure algorithm based on the actual state of the congruence classes. Responsible for the outputs of the EUF solver in the cases of unsatisfiable input. The details of this component are described in Section 4.3 to 4.6.

Model Manager: Propagates implied equalities based on the actual state of the congruence classes. Furthermore, this component is expected to support model generation in each call, when the decidable problem found satisfiable by the congruence closure algorithm. The Model Manager is responsible for the outputs of the EUF theory solver in the cases of satisfiable input. The main functionality of the component is presented in Section 4.7.

Figure 4.1 depicts the general working process of the EUF Solver in a simplified way. The input literals are received by the Facade component. It processes the set of the literals and does some minimal preprocessing on the (dis-)equalities, e.g., it applies negation where it is necessary. Then these (dis-)equalities are forwarded to the Transformer component. The Transformer converts each (dis-)equality into the so-called *standard form*, the internal term representation of the theory solver. Then all the standard equalities are sent to the congruence closure algorithm, in order to build up the corresponding congruence classes based on them. When it is done, the standard disequalities are checked one-by-one. For each disequality predicate $s \neq t$ there are two possibilities: Either s and t currently belong to the same congruence class or not.

When s and t do belong to the same congruence class, it means that they are currently congruent, namely the actual disequality is in a contradiction with the already processed set of equalities. In that case, the Proof Manager component of the theory solver starts its work. As a first step, it constructs a so-called *proof forest* based on the sequence of merge operations that occurred in the congruence closure algorithm during the generation of the congruence classes. Then with the help of this proof forest, the Proof Manager produces an unsatisfiable core of the input problem that contains all the input literals, which cause s and t to be congruent. In the case, when the theory solver was invoked with proof generation requirement, the Proof Manager produces an additional detailed explanation of the contradiction. When one contradiction is found already, the theory solver returns with the generated lemma and with the result **unsat**.

When s and t do not belong to the same congruence class, it means that they are not congruent based on the current state of the congruence classes, namely the disequality among s and t does not contradict with the processed set of equalities. In that case, the actual disequality is sent to the Model Manager component. When all disequalities are found non-congruent, the Model Manager based on the congruence classes propagates the interface equalities to the framework and the theory solver returns with **sat**. The collected disequalities and the full set of congruence classes are used only in the scenario, when the theory solver was invoked with model generation support requirement.

Note that each component starts its work as delayed as possible in order to avoid unnecessary overheads. For example, the proof production capability does not influence significantly the running time of the theory solver in the cases when the set of literals found satisfiable, and the interface equality propagation capability does not influence the running time of the unsatisfiable scenarios. The following sections provide a description of the operations of the main components in details.

4.1 Transformations of Input Equalities

Some necessary initial transformations are applied on the input (dis-)equalities in order to simplify the input of the congruence closure algorithm and to enhance the internal processing. These transformations are applied only once on each input term and each step is reversible in order to reconstruct the original proper terms from the outcome of the EUF solver in scenarios when the output of the theory solver is expected to be more than a plain **sat** or **unsat** answer.

4.1.1 Chainable equalities and pairwise distincs

SMT4J handles, as it is proposed in the SMT-LIBv2 standard (see [7]), equalities as chainable and disequalities as pairwise expressions. In the context of the SMT-LIBv2 language, it means only a syntactic sugar to describe formulas, which contain some specific theory-defined function symbols, in a shorter way. However, the (dis-)equality means in the implemented congruence closure algorithm strictly a binary relation, therefore the input (dis-)equalities which are applied on more than two arguments, are supposed to be translated into multiple binary predicates. The infix notation of (dis-)equality in the following examples would be inconvenient, hence it is neglected. The transformation of the multiple-distinct expressions is unambiguous, a new distinct predicate has to be created for all pairs of the argument list.

Example 4.1. Split of disequalities:
$split((\neq\ a\ b\ c\ d)) \rightarrow \{(\neq\ a\ b),\ (\neq\ a\ c),\ (\neq\ a\ d),\ (\neq\ b\ c),\ (\neq\ b\ d),\ (\neq\ c\ d)\}$.

On the other hand, the number of possible conversions of an equality with more than two arguments is not that confined. The following example illustrates the transformation.

Example 4.2. Split of equalities:
$split((=\ a\ b\ c\ d)) \rightarrow \{(=\ a\ b),\ (=\ a\ c),\ (=\ a\ d)\}$.

Note that in the SMT-LIBv2 standard the **:chainable** function symbol annotation implies a different conversion of equalities with more than two arguments, i.e., the equality of Example 4.2 based on the standard supposed to be transformed to the set of equalities $\{(=\ a\ b),\ (=\ b\ c),\ (=\ c\ d)\}$. This deviation is performed to avoid an unnecessary increase of the depth of the proof graph (see Sec. 4.3).

4.1.2 Curryfication

The first more involved transformation of the preparatory process is restricting the maximal number of arguments of the input ground terms by introducing a new function symbol that represents partial function application. This transformation is called *currification* or *function currying* after Haskell Curry and very similar to the well-known technique in the field of functional programming. Formally, during this transformation each function symbol $f \in \mathcal{F}$ of signature $(\pi_1, \pi_2, ..., \pi_k, \pi)$ is replaced with a new unique constant symbol c_f of sort π and at the same time the set of constant symbols $\mathcal{C}$ is extended by these new c_f constant symbols. The extended set of constant symbols is denoted in the following by $\mathcal{C}'$. Furthermore, $\mathcal{F}$ is extended with a new binary function symbol that is called *Curry-function* and denoted as C, of signature (π_c, π_c, π_c), where π_c is the union of all sorts in $\mathcal{S}$ (that is to say, this function can be applied on any terms). The Curry-function is interpreted as a function application, namely its first argument is supposed to be applied on its second argument. The new set of ground terms after this transformation step is denoted by $\mathcal{G}'$. After this transformation, each ground term $t \in \mathcal{G}'$ is either a constant term $c \in \mathcal{C}'$ or has the form of $C(t_1, t_2)$, where $t_1, t_2 \in \mathcal{G}'$, i.e. only constants or the newly introduced C function symbol with exactly two arguments can occur in each term. The *Curry form* of a ground term $t \in \mathcal{G}$ is a term $Curry(t) \in \mathcal{G}'$ and defined based on [29] as follows:

$$Curry(t) = \begin{cases} c & \text{if } t \text{ is } c \in \mathcal{C} \\ C(...C(C(f, Curry(t_1)), Curry(t_2)), ..., Curry(t_n)) & \text{if } t \text{ is } f(t_1, ..., t_n) \in \mathcal{F} \end{cases}$$

Note that the first argument of a Curry-function is either a *function-constant* c_f or another Curry-function, while the second argument of the function is either a constant symbol $c \in \mathcal{C}$ or another Curry-function. A term in Curry form is henceforth called *Curry term.* The following two examples demonstrate the currification transformation first on a simple and then on a slightly more elaborated ground term:

Example 4.3. (Currification of term) The Curry form of the $f(a, b, c)$ term is as follows:

$$Curry(f(a, b, c)) = C(C(C(f, a), b), c).$$

Example 4.4. (Currification of term) The Curry form of the $f(a, h(b, d), g(a))$ term is as follows:

$$Curry(f(a, h(b, d), g(a))) = C(C(C(f, a), C(C(h, b), d)), C(g, a))).$$

The Curry form of a (dis-)equality follows straightforward from the Curry form of the terms, i.e. $Curry(t = s)$ is $Curry(t) = Curry(s)$ and $Curry(t \neq s)$ is $Curry(t) \neq Curry(s)$. The equalities and disequalities are not distinguished during the process, namely the predicates of the input formula are not modified by the transformation at all. Furthermore, the terms of the input literals are modified just on the representation level, their meaning is not influenced by the transformation. However the size of the input formula by this conversion is inevitably augmented. The following more precise observations about the currification transformation are adapted from [29].

Proposition 1. Let t be a term. Then $|Curry(t)| \leq 2|t| - 1$, i.e., the Curry transformation only produces a linear growth of the input.

Proposition 2. Let E be a set of equations over $\mathcal{G}$ and let $s = t$ be an equation over $\mathcal{G}$. Then $Curry(E) \models Curry(s = t)$ if, and only if, $E \models s = t$.

In practice, the theory solver does not have any knowledge about the alphabet and signature of the input terms, therefore these information are built up on the fly during the currification process. Namely, this preprocessing step not just transforms the representation of the terms but produces also a so called *symbol table* of them. The symbol table contains a bidirectional mapping between a pair of term name and signature, and an integer value. As was mentioned before, all the function symbols in the input formula are supposed to be replaced by function-constants. In the implementation of the EUF solver, not just the function symbols but every symbol of the alphabet is replaced by an integer constant. This extended replacement is necessary among other things due to the following reason. The SMT-LIBv2 standard allows the same symbols to represent more than one operation or constant, as long as they have different signature. However, after the preprocessing steps of the EUF solver, these signature information are not accessible any more, hence the different uses of the same symbol could not be told apart easily in the result terms. In the examples above (and further on) the term symbols are represented by their original names, and not by their integer identifier. An elaborated example for the currification with the produced symbol table and integer representation can be found in the Appendix as Example A.1.

4.1.3 Flattening

The second transformation step of the preparatory process restricts the maximal depth of the Curry terms by introducing new constant symbols to represent the non-constant (sub-)expressions. This transformation is defined on the output of the previous transformation step, namely on a set of currified (dis-)equalities, denoted as E. Formally, for each Curry term $t \in \mathcal{G}'$ a new constant symbol c_t is introduced into the set of constants $\mathcal{C}'$. Then all occurrences of t in E are replaced by the new c_t constant symbol and a new equation $t = c_t$ is added to E. The constant symbols which are introduced during this transformation step are called further on *flat-constants*. The following example illustrates this transformation.

Example 4.5. (Flattening equalities) Assume that E contains only one equality:

$$a = C(C(C(f,a),b),d)$$

Then the flattening process on E is as follows. As a first step, all occurrences of the Curry term $C(f,a)$ are replaced by a new constant symbol c_1 and the equality $C(f,a) = c_1$ is added to E. The enlarged E, denoted by E', after this step is $\{C(f,a) = c_1, a = C(C(c_1,b),d)\}$. Then the c_2 constant symbol is introduced in order to replace the $C(c_1,b)$ Curry term and the new generated equality is $C(c_1,b) = c_2$. After this extension E' is $\{C(f,a) = c_1, C(c_1,b) = c_2, a = C(c_2,d)\}$. At last, the $C(c_2,d)$ Curry term is replaced by the new constant symbol c_3 and the $C(c_2,d) = c_3$ equality is included in E'. The final state of E' is as follows.

$$\{C(f,a) = c_1, C(c_1,b) = c_2, C(c_2,d) = c_3, c_3 = a\}$$

As the following example shows, the process is similar when the transformation is applied on a disequality, namely the result set contains multiple equalities and exactly one disequality predicate, which is the original disequality after the replacement of the subterms with the new constant symbols.

Example 4.6. (Flattening disequalities) The $C(C(C(f,a),C(g,a)),d) \neq a$ disequality in flattened form is as follows.

$$\{C(f,a) = c_1, C(g,a) = c_2, C(c_1,c_2) = c_3, C(c_3,d) = c_4, c_4 \neq a\}$$

A closer study of these examples reveals some interesting observations about the result of the transformation. Each atomic formula in E' is either an equality between two constant symbols or an equality between a constant symbol and a Curry term. The preceding (dis-)equalities are called henceforth *constant-equalities* while the latter type of eqalities are denoted as *Curry-equalities*. Among others, this observation about the flattening preprocessing step is described more precisely in [29]:

Proposition 3. Assume that signature $\mathcal{G}'$ is obtained from $\mathcal{C} \cup \mathcal{F}$ by introducing a Curry-function C and converting all other function symbols into function-constants. Let E_0 be a set of equations, let $s = t$ be an equation, (both built over $\mathcal{G}'$), and let E be obtained by applying zero or more constant introduction and replacement steps on E_0. Then the following holds.

1. $E_0 \models s = t$ if, and only if, $E \models s = t$.
2. If a and b are constants not occurring in $E \cup \{s = t\}$, then $E \models s = t$ if, and only if, $E \cup \{s = a, t = b\} \models a = b$.
3. By applying a linear number of constant introduction and replacement steps to E_0 an E can be obtained such that all equations of E have a constant side, E has depth at most 2, and $|E| \leq 2|E_0|$.

After this transformation step, all initial input $\mathcal{T}_\varepsilon$-literals are transformed to constant-equalities and constant-disequalities. The isomorphic subexpressions of the input currified formula are substituted during the process by the same fresh constant symbols and this Curry term sharing is maintained through a global hash table of all Curry terms. This hash table further on is called *flat mapping*. The resulting set of equalities and disequalities E' is called the *standard form* of the initial input equalities and disequalities of the theory solver. More precise definition of standard form of a set of equalities adapted from [29] as follows.

Definition 4.7. [Standard form] A set of equations E is in *standard form* if its equations are of the form $a = b$ or of the form $C(a, b) = c$ whose (respective) left hand side terms a and $C(a, b)$ only occur once in E.

4.2 Decision of Satisfiability

This section is dedicated to the technical description of the congruence closure algorithm introduced by Robert Nieuwenhuis and Albert Oliveras in [29]. The basic Congruence Closure component (without core or proof generation capability) of the EUF theory solver was already implemented in the SMT4J Solver, therefore the concise details of the implementation are not included in this thesis, but the description of the basic algorithms is necessary in order to build on it in the following sections.

Henceforth, it is assumed that all equalities and disequalities are already in standard form, that is to say, all (dis-)equalities have either the form of $C(a, b) = c$ (Curry-equality) or the form of $a = b$ (constant-equality), where a, b and c are constants. This assumption makes the congruence closure algorithm substantially more comprehensible and efficient.

The Congruence Closure component of the EUF theory solver maintains a congruence relation defined by E_0, a set of standard equalities between constant terms. In order to fulfil this requirement, the implementation provides an operation called $Merge(s = t)$, that enlarges E_0 with the equality $s = t$ and updates the congruence closure of this relation. The other expected functionality of the implementation, is to decide whether two terms s and t belong currently to the same congruence class, that is to say, whether $(s, t) \in E_0^*$. In practice, the component accepts multiple standard equality predicates and stores them in order. Later on, when the corresponding congruence closure of the given equalities is required, the component initializes the underlying data structures and starts the calculation by calling the merge operation on each stored equality predicate. When it is done, the disequalities can be examined one-by-one, whether any of them contradicts with the current state of the calculated congruence classes. When a contradiction is found, i.e., two congruent terms are supposed to be disequal, the theory solver returns with **unsat**. It follows that indirectly the Congruence Closure component is responsible to decide the $\mathcal{T}_\varepsilon$-satisfiability of the input literals. The basic data structures used by the procedure are as follows.

Representatives: Maps each c_i constant symbol to its current representative c_i'. Through this data structure, the representative of any symbol can be reached in constant time. Initially, each constant symbol is mapped to itself.

Class Lists: Maps each representative c_i constant to its corresponding congruence class $s(c_i)$. Namely, it contains a list of all the constants who are in the congruence class represented by c_i. At initialization, the class list of each c constant symbol contains only c.

Use Lists: Contains for each representative c_i constant symbol the list of Curry-equalities $C(a, b) = c$, where either a' or b' (or both) is c_i. Namely, it maps each representative constant symbol to the list of those input Curry-equalities, whose arguments (at least one of them) are represented by the given constant. This data structure is initialized as an empty mapping.

Lookup Table: Maps pairs of representative constant symbols (c_1, c_2) to the list of Curry-equalities $C(a_1, a_2) = a$, where currently a_1' is c_1 and a_2' is c_2. In the case, when such a Curry-equality exists for a given pair of representative constant symbols (c_1, c_2), the Curry-equality is contained by the use lists of c_1 and c_2 too. This data structure is initialized as an empty mapping.

Pending Queue: A queue that represents those equalities, whose merge is currently pending. There are two possible elements of this list: Each element is either a constant-equality or a pair of Curry-equalities. In the case, when the pending element contains two Curry-equalities $C(a_1, a_2) = a$ and $C(b_1, b_2) = b$, the merge of a and b is pending. Initially the Pending Queue is empty.

Union Queue: An ordered sequence of the arguments of each merge operation executed during the congruence closure calculation. Each element of the queue contains two constants that were merged by the algorithm and either a constant-equality or a pair of Curry-equalities as the reason of the union step. Initially the Union Queue is empty.

These data structures effectuate implicitly a union-find data structure (see [15]), where the Representatives mapping stores the disjoint set forest. Before the congruence closure calculation process starts, the initialization of the underlying data structures is necessary. An essential condition of the initialization is to know all constant symbols that can occur in the equalities and disequalities. This information is stored during the preprocessing phase of the theory solver, and passed to the congruence closure procedure as an input for the calculation. In the cases when the congruence procedure is invoked to continue the closure calculation with an enlarged set of equalities, the initialization process is confined only to the newly introduced constant symbols. The responsibility to distinguish these new symbols from the already processed ones, belongs to the Transformer component. When the initialization of the data structures is accomplished, the Congruence Closure component is ready to execute a sequence of merge operations based on the stored equality predicates.

4.2.1 Merge operation

The merge operation on a standard equality predicate is illustrated by Algorithm 4.1 and Algorithm 4.2 from [29]. In the following algorithms of the congruence closure procedure a' denotes always the representative of the a constant symbol.

Algorithm 4.1 Merge from [29]

```
1:  procedure MERGE($s = t$)
2:      if $s$ and $t$ are constants $a$ and $b$ then
3:          add $a = b$ to *Pending*
4:          PROPAGATE()
5:      else                                        ▷ $s = t$ is of the form $f(a_1, a_2) = a$
6:          if *Lookup*($a'_1, a'_2$) is some $f(b_1, b_2) = b$ then
7:              add ( $f(a_1, a_2) = a, f(b_1, b_2) = b$ ) to *Pending*
8:              PROPAGATE()
9:          else
10:             set *Lookup*($a'_1, a'_2$) to $f(a_1, a_2) = a$
11:             add $f(a_1, a_2) = a$ to *UseList*($a'_1$) and to *UseList*($a'_2$)
```

The above procedure divides the input standard equalities into two groups: constant-equalities and Curry-equalities. In line 3 and 4 of Algorithm 4.1 the constant-equalities are forwarded directly to the Pending Queue and the propagate method is called. A Curry-equality $C(a_1, a_2) = a$ is processed as follows: When the Lookup Table contains already a Curry-equality $C(b_1, b_2) = b$ belonging to the (a'_1, a'_2) pair of representatives, a new pending element is constructed from these two Curry-equalities and the propagate method is invoked. If the representative pair (a'_1, a'_2) is not contained in the Lookup Table yet, then the $(a'_1, a'_2) \to C(a_1, a_2) = a$ mapping is stored in the Lookup Table and the Use Lists of a'_1 and a'_2 are amplified with $C(a_1, a_2) = a$, since a_1 and a_2 are arguments in this Curry-equality.

Algorithm 4.2 Propagate adapted from [29]

```
1: procedure Propagate
2:     while $Pending$ is non-empty do
3:         Remove $E$ from $Pending$
                        ▷ $E$ is of the form $a = b$ or $(f(a_1, a_2) = a, f(b_1, b_2) = b)$
4:         if $a' \neq b'$ and wlog., $|ClassList(a')| \leq |ClassList(b')|$ then
5:             $old\_repr\_a := a'$
6:             SaveUnion($a$,$b$,$E$)
7:             for all $c$ in $ClassList(old\_repr\_a)$ do
8:                 set $Representative(c)$ to $b'$
9:                 move $c$ from $ClassList(old\_repr\_a)$ to $ClassList(b')$
10:            for all $f(c_1, c_2) = c$ in $UseList(old\_repr\_a)$ do
11:                if $LookUp(c'_1, c'_2)$ is some $f(d_1, d_2) = d$ then
12:                    add ( $f(c_1, c_2) = c, f(d_1, d_2) = d$ ) to $Pending$
13:                    remove ( $f(c_1, c_2) = c$ from $UseList(old\_repr\_a)$
14:                else
15:                    set $Lookup(c'_1, c'_2)$ to $f(c_1, c2) = c$
16:                    move $f(c_1, c_2) = c$ from $UseList(old\_repr\_a)$ to $UseList(b')$
```

The propagation process is responsible to update the data structures and to find all the new pairs of constants which are supposed to be merged. This procedure is described by Algorithm 4.2, that is adapted from [29]. The method iterates through the elements of the pending list, in order to accomplish the necessary modifications of the data structures. As before was mentioned, each element of the pending list is either a constant equality $a = b$ or a pair of Curry-equalities $C(a_1, a_2) = a, C(b_1, b_2) = b$ and in both cases, the symbols that should be merged are a and b. In line 4 of Algorithm 4.2, the method checks whether a and b belong already to the same congruence class. If they belong to the same class, it means, the current equality is redundant, therefore there is nothing to update. Otherwise the equality $a = b$ is supposed to be added to the union-find data structure. In that case, the corresponding constants of the current equality and pending element is saved in the *Union Queue* in line 6. The union of the corresponding classes is weighted, namely the representative of the already bigger class will be the representative of the new merged class. Lines 7-9 of Algorithm 4.2 accomplish the merge of the classes with path compression. All elements of the originally smaller congruence class are from now on belong to the congruence class of the other representative. Lines 10-16 are updating the Uselist and Lookup Table entries of the old representative constant symbol and detect if whether this merge implied new pairs of constants to be merged.

4.2.2 Congruency checking

After merging all input and propagated equality predicates, the information if two constant symbols are currently congruent is easily obtainable. For this, only the normalization of the terms in question is necessary, namely two constant symbols s and t are congruent, if and only if the Normalize method, as described in Algorithm 4.3 from [29], returns with the same constant for s and t.

The Normalize procedure returns for each t constant symbol its representative t'. For a Curry-function $C(t_1, t_2)$ the procedure recursively normalizes t_1 and t_2 and then checks

Algorithm 4.3 Normalize from [29]

```
1: procedure NORMALIZE($t$)
2:     if $t$ is a constant then
3:         return $t'$
4:     else                                                    ▷ $t$ is $f(t_1, t_2)$
5:         $u_1$ := NORMALIZE($t_1$)
6:         $u_2$ := NORMALIZE($t_2$)
7:         if $u_1$ and $u_2$ are constants and $Lookup(u_1, u_2)$ is $f(a_1, a2) = a$ then
8:             return $a'$
9:         else
10:            return $f(u_1, u_2)$
```

whether there is an input equality in the Lookup Table that belongs to the obtained pair of representatives. If such an equality is found, the method returns with the representative of its constant side. Otherwise, the method returns with a Curry-function with the obtained representatives as arguments.

Based on the result of this method, when exists such a disequality that makes the set of input literals unsatisfiable, the EUF theory solver identifies the conflict causing disequalities without any difficulty.

4.3 Proof Forest Building

When a contradictory disequality is found by the Congruence Closure component, the EUF theory solver is expected to produce an unsatisfiable core of the given input set of literals or a more detailed explanation of the conflict. A disequality $s \neq t$ that causes a conflict is defined between two constants which are congruent, namely the explanation of this conflict is derivable by finding all the equalities that are needed to imply $s = t$. These required equalities are actually the applied union steps on the congruence classes. To keep the congruence closure algorithm fast and efficient, it applies path compression in every step on the internal union-find data structure, therefore at the end of the calculation, the chain of each class unions is not traceable back any more. Hence, during the propagation method (Alg. 4.2, line 6) the storage of the sequence of executed union methods is inevitably necessary. In the cases, when the set of input literals is found satisfiable, this collected information is unused, but when the Congruence Closure component deems the set of literals unsatisfiable, the Proof Manager component accepts as an input the queue of unions and the conflict causing disequality. This sequence of unions provides the basis for the construction of the centrepiece of the Proof Manager, the so-called *proof forest* data structure that is also proposed in [29].

The proof forest of the Proof Manager component is a weighted non-compressed union-find data structure, where the disjoint sets are stored as trees (see Section 2.4). Each tree consists of a root node that contains the current representative of the corresponding set. All edges in the tree are directed towards that root. The nodes of the graph represent the terms, while the edges indicate the equality relation between two terms. Since, there is no redundancy among the given equalities, the forest, as the name suggests, contains no cycles. Each node possesses exactly one outgoing edge that points to the parent of the node. The root node of a tree has also an outgoing edge that points to an artificial node where the actual size of the corresponding tree is stored. These artificial nodes are called henceforth *size nodes*. Each edge between two nodes s and t (where neither s nor t is a size node) is labelled by the reason, that explains why the $s = t$ equality holds. A reason is either an input constant-equality $s = t$ or a pair of Curry-equalities $C(s_1, s_2) = s$ and

$C(t_1, t_2) = t$. Note, that although all the edges are directed, the equality relation that is represented by them is symmetric, therefore the orientation of a given edge is not relevant.

The construction of the proof forest is based on the Union Queue of the Congruence Closure component and uses only the union and the find methods of the proof forest data structure. Each element of the Union Queue is a triple $\langle s, t, E\rangle$, where s and t are constant terms and E is a set of standard equalities. Note, that E already determines the value of s and t, this redundancy serves just as code readability enhancement. The Proof Manager component, as an initialization step, iterates through the Union Queue and calls the union method of the Proof Forest with each current triplet as arguments.

Algorithm 4.4 Union method of Proof Forest

```
1: procedure UNION($s, t, reason$)
2:     $s' := \textsc{Find}(s)$
3:     $t' := \textsc{Find}(t)$
4:     if wlog., $s(s')$.size $\geq s(t')$.size then
5:         $s(s')$.size := $s(s')$.size + $s(t')$.size
6:         REVERSEPATH($t, t'$)
7:         $t$.parentId := $s$
8:         $t$.label := $reason$
```

Algorithm 4.4 describes the union method in details. The main functionality of the method is to place an edge between the nodes of s and t in the forest, i.e., to merge two yet independent trees without breaking the desired structure of the forest. As a first step in line 2 and 3 the corresponding root nodes of the involved trees are determined. These nodes are the actual representatives of s and t. In line 4 the corresponding size nodes are compared in order to determine the orientation of the new edge. The size information of the selected new root node is updated in line 5. When the involved nodes are identified, the corresponding roots are found and the orientation of the required edge is decided, the procedure restructures the tree of the source node, in order to maintain the desired structure of it. Here, "desired structure" means that each node has exactly one parent therefore in the cases when the source node of the new edge is not a root node in its tree, the insertion of the edge would just ruin this structure invariant. However, by reversing the unique path between the source node and its current root, the source of the new edge becomes the root of its tree. After that step, the new edge can be inserted into the forest (line 7) without violating the structure invariant. In line 8, as a last step, the given explanation of the symbolized equality is stored as a label of the new edge. Figure 4.2 depicts a simplified example proof forest and union query when the reorganization of the source tree is necessary. Note that the procedure assumes that s and t do not belong to the same tree yet. This precondition is fulfilled by the Congruence Closure algorithm, since it stored only those equalities in the Union Queue, that gave rise to a merge operation.

The Find method of the proof forest data structure is simple and straightforward. It is described by Algorithm 4.5. In the case, when the searched term is not contained by any of the trees yet (line 2), the method constructs a new tree and returns with the freshly created node. The new tree contains only one node and the size node of it is labelled with -1. By this implicit makeSet step, the initialization of each node is postponed to the very first time, when they would be used. When the searched term is not new (line 5), the method traverses up on the corresponding tree as long as it finds a size node, namely while it finds the current root of the tree and returns with it. As was mentioned already, this method does not execute path compression on the proof forest.

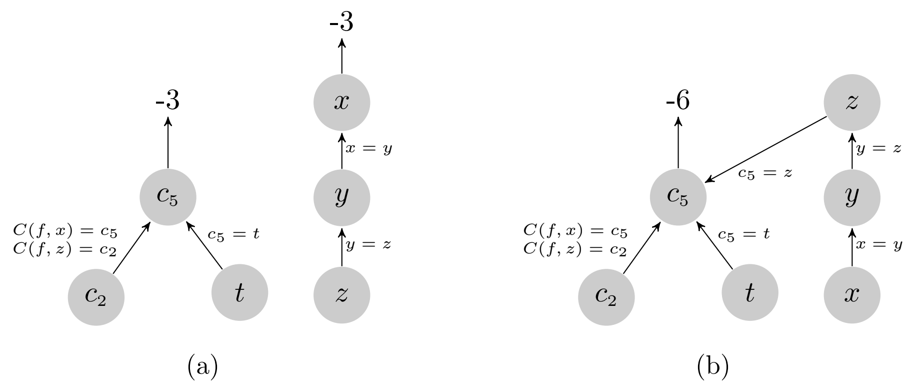

Figure 4.2: An example of a proof forest. **(a)** The state of the proof forest where the edges are constructed based on the following unions: $\langle x, y, \{x = y\}\rangle, \langle y, z, \{y = z\}\rangle, \langle c_5, c_2, \{C(c_f, x) = c_5, C(c_f, z) = c_2\}\rangle, \langle c_5, t, \{c_5 = t\}\rangle$ **(b)** The same forest after reversing the edges between z and x in order to process the $\langle c_5, z, \{c_5 = z\}\rangle$ union.

Algorithm 4.5 Find method of Proof Forest

```
1: procedure FIND($c$)
2:     if $c$ is not in the Proof Forest then
3:         CREATENODE($c$)
4:         return $c$
5:     else
6:         if $c$.parentId $< 0$ then
7:             return $c$
8:         else
9:             return FIND($c$.parentId)
```

4.4 Unsatisfiable Core Production

The main functionality of the Proof Manager component is the same in each case where a conflict causing disequality $s \neq t$ is found. It has to prove that the equality $s = t$ is indicated by the congruence relation generated by the input equalities. However, the component offers two different configurations: either it produces directly such a set of input equalities E_l or it generates a detailed proof of the currently found conflict, where the unsatisfiable core is a byproduct of this proof. This section describes the behaviour how the Proof Manager extracts the unsatisfiable core. When the proof forest is already constructed, it is quite simple to find the explanation of an equality $s = t$. In order to explain why s and t are congruent, one has to only walk through the unique path in the proof forest between s and t and recursively repeat this traversal for the arguments of the upcoming Curry-equalities. Then the edges on the involved paths contain those input equalities that derive the equality $s = t$. Since the proof forest was built based on the Union Queue, it can be assumed that E_l, the collection of these visited equalities will form a subset of the Union Queue. Furthermore, the traversal of the proof forest is exhaustive, i.e., it finds recursively all the necessary equalities, therefore it can be also assumed, that E_l will indicate $s = t$, namely that $(s, t) \in E_l^*$. The details of this traversal method are described by Algorithm 4.6 as it is proposed in [29].

The algorithm strongly exploits the special structure of the built proof forest, namely that if a path exists between c_1 and c_2 then it is unique and acyclic. In the method, the Pending Proofs list always consists of those equalities that still wait to be proved and at the beginning it contains only the equality $c_1 = c_2$, i.e., the negation of the conflict causing

Algorithm 4.6 Explain method from [29]

```
procedure EXPLAIN($c_1$,$c_2$)
    Set *PendingProofs* to $\{c_1 = c_2\}$
    while *PendingProofs* is not empty do
        Remove an equation $a = b$ from *PendingProofs*
        $c$ := NEARESTCOMMONANCESTOR($a, b$)
        EXPLAINALONGPATH($a, c$)
        EXPLAINALONGPATH($b, c$)
```

disequality (Alg. 4.6, line 2). The procedure runs as long as there are still equalities to be proved and in each iteration exactly one equality is investigated. To collect all the necessary equalities that imply $c_1 = c_2$, one has to traverse through the path between c_1 and c_2 in their tree. For this, as a first step (in line 5 of Alg. 4.6) the method identifies the nearest common ancestor of c_1 and c_2. Then it calls the ExplainAlongPath method on both branches of the path between c_1 and c_2, once for the c_1-c section (line 6 of Alg. 4.6), then to the c-c_2 part (line 7 of Alg. 4.6).

Then the ExplainAlongPath method collects all the occurring equalities from the labels along the given path step-by-step. The realization is shown in Algorithm 4.7.

Algorithm 4.7 ExplainAlongPath method adapted from [29]

```
procedure EXPLAINALONGPATH($a$,$c$)
    $a$ := HIGHESTNODE($a$)
    while $a \neq c$ do
        $b$ := PARENT($a$)
        if edge $a \to b$ is labelled with a single input merge $a = b$ then
            Output $a = b$
        else                    ▷ edge is labelled with $\{C(a_1, a_2) = a, C(b_1, b_2) = b\}$
            Add $a_1 = b_1$ and $a_2 = b_2$ to *PendingProofs*
        UNION($a, b$)
        $a$ := HIGHESTNODE($b$)
```

This procedure steps through one specific straight path in the tree that ends at the nearest common ancestor node. From the structure of the tree it follows that just by stepping on the parent in each iteration, once it will reach the ancestor node, namely the method will terminate. In each iteration, the procedure investigates the edge between the actual two nodes. When the edge is labelled with a constant-equality $s = t$, then it can be stored into the output collection without further works on it. Nevertheless, when the current edge contains a pair of Curry-equalities $C(s_1, s_2) = s$ and $C(t_1, t_2) = t$, it means that the proof of the equalities $s_1 = t_1$ and $s_2 = t_2$ are needed in order to assume the $s = t$ equality proved. Hence, the two equalities $s_1 = t_1$ and $s_2 = t_2$ are stored into the Pending Proofs list. Note that in these cases the equality of s and t is derived by the propagation method of the Congruence Closure component or simply introduced by the preprocessing phase, i.e., the equality $s = t$ is not part of the input literals, but implied from them. Therefore, in these steps (line 8 of Alg. 4.7), the $s = t$ equality is not included into the output collection. After processing the actual edge, the procedure steps further up in the tree, in order to examine the next edge of the path. However, the size of this step depends on the already processed edges. Namely, the algorithm narrows down the traversal only for those edges, that were not visited yet.

To avoid already visited edges, the Proof Manager component maintains an additional weighted and compressed union-find data structure, that is called *Explain Forest*. This data structure stores all those equalities (edges), that were already discovered during the traversal of the proof forest. Whenever an edge between a and b was already processed by the ExplainAlongPath method, the union of a and b in the Explain Forest is invoked (line 9 of Alg. 4.7). The Union method merges the equivalence classes of a and b in the Explain Forest based on their current size, while the Find method returns with their actual representatives and applies path compression on the corresponding trees. Each traversal of the Proof Forest starts in an arbitrary node and steps towards an ancestor of the starting node, therefore whenever an already visited sequence of path is found, the direction of the short-cut is always the same, namely it starts deeper or right at the current node and ends in another node that is higher in the tree. Since the unions of the Explain Forest consider the current sizes of the involved trees, and the find methods compress the depth of the trees, the locations of the nodes in the Explain Forest do not mirror the locations in the Proof Forest, i.e., the representatives of the Explain Forest are not necessarily the highest nodes in the Proof Forest. Therefore, beyond the Union and Find methods, the Explain Forest data structure provides an additional HighestNode(t) operation. This method identifies the equivalence class of t in the Explain Forest data structure and returns with the one element of it, which is closest to the root of the corresponding equivalence class tree in the Proof Forest. The maintenance of the highest node information is relative simple, due to the fact, that each union in the Explain Forest is of the form Union(t,parent(t)), i.e, the highest node of the new class is always the highest node of the actual parent node.

During the ExplainAlongPath procedure, these short-cuts are checked in line 2 and line 10. Before each sub-explanation, the start node of the current path is set to the highest yet not explained equality, and during the iterations, whenever a jump over a sequence of edges is applicable, it is found with the aid of the Explain Forest. Note that already during the nearest common ancestor seek (line 5 of Alg. 4.6) are the short-cuts provided by the Explain Forest exploited. Furthermore, the NearestCommonAncestor method returns in each case with the highest node of the equivalence class where the nearest common ancestor belongs.

The Explain and ExplainAlongPath procedure continue as long as there are no more equalities to be justify. When the list of Pending Proofs is empty, all the input equalities that are involved in the current conflict are identified and the collection of them (E_l) can be retrieved by the Proof Manager component as an unsatisfiable core of the input literals. While the theory solver returns with E_l, the SMT-LIBv2 representation of the unsatisfiable core is printed onto the proof stream. The following example demonstrates the process of the Explain and ExplainAlongPath methods informally on a simplified set of equalities.

Example 4.8. Assume that the following set of equalities is given as an input to the theory solver:

$$E = \{b = h, c = b, e = d, d = c, g(e, e) = a, g(e, h) = b, d \neq a\}.$$

After the preprocessing transformations the equalities in standard form are as follows:

$$E' = \\ \{b = h, c = b, e = d, d = c, C(g, e) = c_1, C(c_1, e) = c_2, c_2 = a, C(c_1, h) = c_3, c_3 = b, d \neq a\}$$

The result of the Proof Forest building process is shown in Figure 4.3. The conflict causing disequality of this example is $d \neq a$, therefore the Proof Manager component invokes the Explain(d, a) method. The equality $d = a$ initializes the Pending Proofs list and the main iteration of the method starts. First, the nearest common ancestor of d and a is identified, that is b. Then the traversal of the path $d \to b$ in the Proof Forest begins. Since at this state the Explain Forest is still empty, the highest node of d will

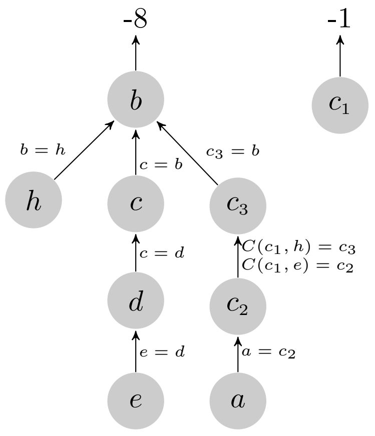

Figure 4.3: Proof Forest of E'

be itself. At the first iteration, the edge between the nodes d and c is examined. This edge is labelled with an input equality, therefore the procedure saves the equality $c = d$ in the output set, E_l. Then the nodes c and d are merged in the Explain Forest in order to avoid the repeated traversal of this edge. Since the highest node of the new equivalence class of c in the Explain Forest is c, the next examined edge is between the nodes c and b. This edge brings the input equality $c = b$ into E_l. Then the equivalence class $[c, d]$ in the Explain Forest is merged with the equivalence class $[b]$, with b as the new highest node. With this, the $d \to b$ path is processed and the ExplainAlongPath procedure starts to work on the $a \to b$ path. The first edge of the new path contains a simple input equality, $a = c_2$, therefore it is passed to E_l, and the union of the two nodes in the Explain Forest is invoked. Since the highest node of the new class is c_2, the next investigated edge is between c_2 and c_3. This edge contains two Curry-equalities, namely to justify the equality $c_2 = c_3$, one has to prove that, $c_1 = c_1$ and $h = e$, therefore these two equalities are appended to the Pending Proofs list. After merging the $[a, c_2]$ tree with the c_3 node in the Explain Forest, the next iteration is about the $c_3 - b$ edge. This edge enlarges E_l with the input equality $c_3 = b$. In the Explain Forest the trees $[d, c, b]$ and $[a, c_2, c_3]$ are merged, where b remains the highest node and the ExplainAlongPath method returns.

Now the next element in the Pending Proofs is the $c_1 = c_1$ equality. The Explain method finds c_1 as nearest common ancestor, therefore there is no path to traverse and the method steps forward to the next pending equality. To prove that $h = e$ holds, the explanation of the $h \to b$ and $e \to b$ paths are required. The $h - b$ path contains just one edge, with an input equality, so E_l is extended with the equality $b = h$. Furthermore, the equivalence classes of h and b are merged in the Explain Forest into the following class: $[c, d, b, a, c_2, c_3, h]$, where b is the highest node. The explanation of the $e \to b$ path demonstrates the benefits of the additional Explain Forest union-find data structure. First the $e - d$ edge is examined, where the input equality $e = d$ is added to E_l and the equivalence classes of e and d in the Explain Forest are merged. The highest node of the new class remains b. If the method would step forward in the tree, it recognizes that all the equalities on the edges in the path $d \to b$ have already been saved in E_l, and therefore the method jumps directly to b as the highest node of the element d and avoids to traverse again this path. With this jump, the ExplainAlongPath(e, b) method stops after only one iteration, instead of three.

Since the Pending Proofs list is empty, the loop of the Explain method terminates. The returned set of the collected equalities is as follows:

$$E_l = \{c = d, c = b, a = c_2, c_3 = b, b = h, e = d\}$$

Although in this example all the input equalities were used in order to prove the given contradiction, it is not difficult to see that in most of the cases E_l, the produced result set, is substantially smaller than E. Furthermore, based on this example, the benefits of the additional Explain Forest may appear minimal, but on realistic problem instances the cumulative enhancement is significant. The input formula of this example in the SMT-LIBv2 format can be found in Figure A.4 in the Appendix. Since all the assertions were required to prove the contradiction, it is not hard to see that the output SMT-LIB proof contains the same assertions (although in different order).

4.5 Using Cores in SMT4J

As described in Section 2.3, the basic lemmas on demand approach expects from the theory solvers to produce an unsatisfiable core in the cases when a formula is found unsatisfiable on the level of the theory. Then the framework transforms this core into a theory lemma and conjoins the propositional abstraction of the lemma to the input problem instance in order to refine the search space of the SAT Solver. The experimental results show that this approach in that form is not sufficiently efficient to solve industrial problem instances (see Section 6.3.2). Therefore, the strategy employed by SMT4J slightly modifies the basic lemmas on demand approach. The implemented version of the approach is described in Algorithm 4.8.

Algorithm 4.8 LAZY-MULTICORES

```
1: function LAZY-MULTICORES($\varphi$)
2:     $\mathcal{B} \leftarrow e(\varphi)$
3:     while (TRUE) do
4:         $\langle \alpha, res \rangle \leftarrow$SAT-SOLVER($\mathcal{B}$);
5:         if $res =$ "Unsatisfiable" then return "Unsatisfiable";
6:         else
7:             $\langle ts, res \rangle \leftarrow$DEDUCTIONMULTICORES($\hat{Th}(\alpha)$);
8:             if $res =$ "Satisfiable" then return "Satisfiable";
9:             else
10:                $\mathcal{B} \leftarrow \mathcal{B} \bigwedge_{t \in ts} e(t)$;
```

The main differences of the algorithm compared to Algorithm 2.1 are in line 7 and 10. While in the original approach, the deduction method returns only with one clause, in the implementation of SMT4J the result of this method is a sequence of clauses. It follows, that in line 10 then not just one theory lemma, but a conjunction of multiple $\mathcal{T}$-valid lemmas is conjoined to the propositional abstraction of the input formula.

The calculation of the congruence classes and the construction of the Proof Forest are unavoidable tasks each time when the theory solver is asked about the satisfiability of a conflicting set of literals. Nevertheless, the same congruence classes and Proof Forest can be employed with minimal further efforts to explain every conflict in the set of literals instead of just one. Thus, the overhead of this modification on the level of the theory solver is not significant, namely just the Explain method is called repeatedly. The number of method calls is exactly the number of contradictions in the input set of literals. In each refinement iteration a theoretical upper bound of the contradictions found by the theory solver is the number of the distinct literals in the model of the corresponding iteration. This is just an upper bound, because most (probably many) of the disequalities do not contradict with the congruence closure generated from the equalities. Still, it can be said, that in most of the cases there is more than one contradiction identifiable, which means that more than one theory lemma can be constructed for the SAT Solver. The influence of this extension to the performance of SMT4J is presented in Section 6.3.2.

4.6 Proof Production

When the Proof Manager component is instrumented to produce a detailed proof of the currently found conflict, the component employs a different method to traverse through the Proof Forest. The main challenges of this procedure are to reveal all the implicit steps of the theory solver in the proof trace. Furthermore, since the solver conducts several transformations on the input literals, the proof production process has to transform back the produced certificates to the signature of the input terms. The base of the concept is exactly the same as at the core production method: traverse through the proof forest in order to find all those equalities which imply the negation of the current conflict causing disequality. Each path in the forest encodes an inference rule, and the task of the procedure is to decide which axiom schema is instantiated on a given path. Then the rules *eq_transitive*, *eq_reflexive*, *eq_congruent* introduce equational tautologies. As described in Section 3.2, the procedure returns with a set of tagged Horn clauses, where the input clauses, also called assertions, represent the original input literals of the theory solver while further clauses are derived from these input clauses by instantiating the axioms of the $\mathcal{T}_\varepsilon$-theory. The generated clauses consist of at most one positive literal, this is called henceforth *conclusion*, and arbitrary many negative literals, denoted as *premises*, where the negation of each premise literal is a conclusion equality of exactly one other clause. The generation of these clauses is the main responsibility of the proof producing Explain method, that consists of the following main components.

Proof Lemma: Contains those clauses that derive the actual contradiction. The clauses are separated into two categories, namely the involved input literals (assertions) are distinguished from the clauses that are instantiations of the theory axioms. The former set forms the unsatisfiable core of the problem instance.

Pending Proofs: Contains those clauses, whose premises are not collected yet, namely clauses that consist of exactly one positive literal. For some of the pending clauses the applied inference rule is already determined when added to the Pending Proofs list.

Proved Clauses: Contains those clauses that are already complemented with all the necessary negative literals, namely those clauses whose premise literals are conclusions of clauses that are either in the Pending Proofs or in the Proved Clauses list. The Proof Lemma is a subset of this set of clauses. This container is in practice a mapping of each proved equality to the clause whose positive literal is the equality in question. Since an equality can be derived just in one way in the Proof Forest, one can search between the proved clauses without the knowledge of the applied inference rule or the involved premises.

Since in the clauses of the proof trace the negative literals are actually references to further clauses, it is an essential requirement that each clause exists only once. Otherwise, either the produced proof would be incorrect (e.g. contains clauses that are not valid in the theory) or the procedure should conduct several repeated traversals to collect the necessary premises of an already proved clause. During the explanation method, when as a premise an equality is identified, there are two possibilities. Either there is already a (proved or pending) clause with this equality as conclusion and in that case the reference of this clause is employed, or a new clause is built up and stored into the Pending Proofs list. This invariant is maintained by the GetClause supportive method in the following algorithms.

Algorithm 4.9 Proof Production

```
procedure EXPLAIN($c_1$,$c_2$)
    $proofLemma$ := new ProofLemma($c_1 \neq c_2$)
    $proofRoot$ := GETCLAUSE($c_1 = c_2$,$undecided$)
    Add $proofRoot$ to $PendingProofs$
    while $PendingProofs$ is not empty do
        $currentClause$ := a clause with conclusion $a = b$ from $PendingProofs$
        $\langle path\_size, nca \rangle$ := NEARESTCOMMONANCESTOR($a, b$)
        SETSTEPRULE($currentClause$,$path\_size$)
        switch $currentClause.StepRule$ do
            case $assertion$
                ADDASSERTION($proofLemma$,$currentClause$)
            case $reflexive$
                ADDSTEP($proofLemma$,$currentClause$)
            case $congruent$
                ADDSTEP($proofLemma$,$currentClause$)
                EXPLAINCONGRUENT($currentClause$)
            case $Curry$
                EXPLAINCURRY($currentClause$)
            case $default$
                ADDSTEP($proofLemma$,$currentClause$)
                EXPLAINALONGPATH($a, nca$)
                EXPLAINALONGPATH($b, nca$)
        Add $currentClause$ to $ProvedClauses$
    MERGECLAUSES($proofLemma$)
    if $withResolution$ is true then
        GENERATERESOLUTIONTREE($proofLemma$)
```

The noteworthy steps of the explain procedure with proof generation are shown in Algorithm 4.9. The structure of the method follows closely the mechanism of the original core producing explain method. The procedure expects two ground terms c_1 and c_2 as input arguments, where $c_1 \neq c_2$ is a conflict causing disequality. Algorithm 4.9 starts with the initialization of the Proof Lemma object, where the starting point of the proof trace is built. The initialization steps of the procedure are depicted in detail through an example later on (Example 4.11). At the beginning, the Pending Proofs list consists only of the clause of the $c_1 = c_2$ equality, namely the clause, whose conclusion is the $c_1 = c_2$ literal. At line 5 of Algorithm 4.9 the method starts to iterate through the Pending Proofs queue. Each iteration fills up a clause with the corresponding premise literals by moving one clause from the Pending Proofs list to the Proved Clauses collection. For this, similar to the original explain method, the nearest common ancestor of a and b is detected in the proof forest. Nevertheless, this time the NearestCommonAncestor method does not employ the Explain Forest but returns the length of the $a \to b$ path in the proof forest beside the common ancestor. Based on this length information, one can identify the applied inference rule of the current clause (line 8 of Alg. 4.9) in the cases when it is still undecided. When the path consists of more than one edge, then the actual equality is proved by using the transitivity axiom of the theory. When the length is zero, it means that the current equality has $a = a$ form i.e., it is an instance of the reflexivity axiom.

A path that consists of exactly one edge represents either an asserted input equality or a congruence step based on two Curry-equalities, depending on the label of the corresponding edge, as it was already in the previous Explain method.

When the inference rule is identified, the method decides how to continue. An input (assertion) or reflexive clause has no premise, i.e., in these cases (line 10-13) there are no further clauses to search, thus the actual unit clause is just simply appended to the Proof Lemma and saved in the Proved Clauses list. A transitive clause refers to a sequence of further equalities as premises, therefore after storing it in the Proof Lemma (line 20 of Alg. 4.9), the ExplainAlongPath method moves along the paths $a \rightarrow nca$ and $b \rightarrow nca$ (line 21-22). This procedure is described in Algorithm 4.10.

Algorithm 4.10 ExplainTransitive method

```
1:  procedure ExplainAlongPath($a$,$nca$)
2:      while $a \neq nca$ do
3:          $b$ := parent($a$)
4:          if clause with conclusion $a = b$ is not in $ProvedClauses$ then
5:              if edge $a \rightarrow b$ is labelled with a single input merge $a = b$ then
6:                  $currentPremise$ := GetClause($a = b$,$assertion$)
7:                  AddAssertion($proofLemma$,$currentPremise$)
8:                  AddPremise($currentClause$,$currentPremise$)
9:                  Add $currentPremise$ to $ProvedClauses$
10:             else        ▷ edge is labelled with $\{C(a_1, a_2) = a, C(b_1, b_2) = b\}$
11:                 $currentPremise$ := GetClause($a = b$,$congruent$)
12:                 AddPremise($currentClause$,$currentPremise$)
13:         else
14:             $currentPremise$ := GetProvedClause($a = b$)
15:             AddPremise($currentClause$,$currentPremise$)
16:         $a := b$
```

The ExplainAlongPath method of the proof production process employs the found equalities on a given sequence of edges to construct or identify the clauses that are involved in the derivation of the equality $a = b$. Note that although the method traverses always an $a \rightarrow nca$ path, the current clause is still the explanation of the $a = b$ equality. Each premise of a transitive clause is either an input or a congruent clause depending on the corresponding edge, as it was already in Algorithm 4.7. The main difference between the two algorithms is that this time the Explain Forest can not be employed. While the responsibility of the ExplainAlongPath method in the unsatisfiable core production was to collect all those input equalities from the proof forest that participate in the conflict and not identified yet, this time the method has to collect all those equalities (both input and congruence) that are necessary to derive the currently examined equality. Thus, all the edges on the path between a and the nearest common ancestor have to be traversed, even if some of them were processed already, in order to add the reference of the corresponding equalities to the premise list of $a = b$. When the clause of any of the edges is already proved or waits to be proved, the GetClause or GetProvedClause method returns with the reference of it (lines 6, 11 and 14 of Alg. 4.10), therefore it will be not proved repeatedly. When the ExplainAlongPath procedure finds a new input equality on an edge (line 5-9 of Alg. 4.10), it builds up a new assertion clause from it and appends it to the Proof Lemma. This new clause is then saved as a premise of the clause under proving. Since input clauses do not require further steps, they are saved in the Proved Clauses list in place without storing them in the Pending Proofs queue. When the actual edge represents a new congruence step (line 11-12), the method constructs the corresponding congruent clause and appends it to the premise list of $a = b$.

When the conclusion equality of the clause that is currently investigated by the Explain method is implied by congruency (line 14 of Alg. 4.9) the two Curry-equalities on the corresponding edge define the two premises of the clause. Algorithm 4.11 describes the details of this scenario. This method assumes, that the conclusion of the current clause is an equality $a = b$ and in the Proof Forest there is only one edge between a and b labelled with two Curry-equalities $C(a_1, a_2) = a, C(b_1, b_2) = b$.

Algorithm 4.11 ExplainCongruent method

```
1: procedure ExplainCongruent(currentClause)
   ▷ edge between $a$ and $b$ is labelled with $\{C(a_1, a_2) = a, C(b_1, b_2) = b\}$
2:     currentPremise1 := GetClause($a_1 = b_1$,curry_undecided)
3:     currentPremise2 := GetClause($a_2 = b_2$,undecided)
4:     AddPremise(currentClause,currentPremise1)
5:     AddPremise(currentClause,currentPremise2)
```

As it was mentioned already in Section 4.1.2, in that case a_2 and b_2 are either constant symbols from $\mathcal{C}$ or flat-constants that were well formed terms before the curryfication and flattening processes. Therefore, the equality $a_2 = b_2$ has to be proven and this proof belongs to the Proof Lemma. Since at this point it is undecided which axiom implies $a_2 = b_2$, the constructed clause is initialized with the temporal *tmp_undecided* inference rule and saved in the Pending Proofs list (line 3 of Alg. 4.11). In the case this clause has already been proved or exists in the Pending Proofs list, the GetClause method returns with the corresponding reference without further steps. Nevertheless, a_1 and b_1 are either function-constants (e.g., $f = f$) or flat-constants that are encoding partial function applications (e.g., $f(c_1) = f(c_2)$, where arity of f is greater than one), that is to say, these terms are not well formed in the language of the input literals. The partial terms were introduced during the currification process of the theory solver, when each function was transformed into a set of binary Curry-functions encased in each other. After this transformation, the only function symbol of the language is the binary Curry-function C. Therefore, in the internal representation of the theory solver there is only one axiom related with congruency, and that one can be used just for proving that the results of two C functions are equal. The following example demonstrates in practice the consequences of this fact in the proof system.

Example 4.9. Assume that the following set of equalities is given to the theory solver: $\{a = f(a_1, a_2), b = f(b_1, b_2), a_1 = b_1, a_2 = b_2, a \neq b\}$. To prove the equality $a = b$, one has to instantiate the transitivity (formula 4.1) and then the congruency (formula 4.2) axiom, namely

$$\mathbf{a} = \mathbf{f}(\mathbf{a_1}, \mathbf{a_2}) \wedge f(a_1, a_2) = f(b_1, b_2) \wedge \mathbf{f}(\mathbf{b_1}, \mathbf{b_2}) = \mathbf{b} \supset a = b \quad (4.1)$$

$$\mathbf{a_1} = \mathbf{b_1} \wedge \mathbf{a_2} = \mathbf{b_2} \supset f(a_1, a_2) = f(b_1, b_2) \quad (4.2)$$

where the bold equalities are input literals. Nevertheless, in the internal representation of the theory solver, the proof looks quite different. First of all, the equalities after preprocessing are the following: $\{a_1 = b_1, a_2 = b_2, C(c_f, a_1) = c_1, C(c_1, a_2) = c_2, c_2 = a, C(c_f, b_1) = c_3, C(c_3, b_2) = c_4, c_4 = b, a \neq b\}$. It follows, that during the proof production only these equalities can be employed. Thus, the proof (in unflattened form) looks as follows.

$$\mathbf{a} = \underbrace{\mathbf{C}(\mathbf{C}(\mathbf{c_f}, \mathbf{a_1}), \mathbf{a_2})}_{c_2} \wedge \underbrace{C(C(c_f, a_1), a_2)}_{c_2} = \underbrace{C(C(c_f, b_1), b_2)}_{c_4} \wedge \underbrace{\mathbf{C}(\mathbf{C}(\mathbf{c_f}, \mathbf{b_1}), \mathbf{b_2})}_{c_4} = \mathbf{b} \supset a = b \tag{4.3}$$

$$\underbrace{C(c_f, a_1)}_{c_1} = \underbrace{C(c_f, b_1)}_{c_3} \wedge \mathbf{a_2} = \mathbf{b_2} \supset \underbrace{C(C(c_f, a_1), a_2)}_{c_2} = \underbrace{C(C(c_f, b_1), b_2)}_{c_4} \tag{4.4}$$

$$c_f = c_f \wedge \mathbf{a_1} = \mathbf{b_1} \supset \underbrace{C(c_f, a_1)}_{c_1} = \underbrace{C(c_f, b_1)}_{c_3} \tag{4.5}$$

$$c_f = c_f \tag{4.6}$$

In the internal representation of the produced proof, each congruence step (e.g., Formula 4.4) is proved through a sequence of further steps, due to the currification process.

To be able to transform the latter (inner) proof format of Example 4.9 to the expected format where all terms suit to their initial signatures, the steps that conclude equalities between partial functions have to be distinguished from the other steps of the proof. This purpose is served by an additional clause tag prefix: *tmp_curry*. The clauses that are constructed with this label are henceforth called *Curry-clauses*. Based on this tag, the Explain method easily identifies the case where the current clause requires special management (line 17 of Alg. 4.9) and invokes the ExplainCurry function on it.

The special management of the Curry-clauses requires several modifications in the explanation method of them. First of all, since these clauses contain not well formed terms, they are not allowed to appear in the result Proof Lemma. Nevertheless, the premises of these clauses are indirectly premises of a valid congruent clause of the proof, therefore they have to be proven just like the normal equalities and the valid premises have to be forwarded to the clause where they really belong without appending the Curry-clauses to the Proof Lemma. Algorithm 4.12 presents the main steps of the ExplainCurry method. Each Curry-clause concludes an equality between either two function-constants or two flat-constants. Equality of function-constants (e.g. Formula 4.6 in Example 4.9) is identified as a *tmp_curry_reflexive* clause and it is an instance of the reflexivity axiom in the internal representation of the theory solver. Therefore, in that case there are no necessary premises and the method can simply return (line 3-4 of Alg. 4.12). Nevertheless, equalities between flat-constants can be instances of any of the three theory-axioms. The simplest scenario is when the Curry-clause is an instance of the congruency axiom like Formula 4.5 in Example 4.9. In that case, the flat-constants of the equality are neighbours in the Proof Forest and connected with an edge labeled by two Curry-equalities. Therefore, these clauses can be proved just like the valid congruence clauses with the ExplainCongruent method (line 5-6 of Alg. 4.12). The only difference is that the clause is not appended to the Proof Lemma. Most of the Curry-clauses are proved through a sequence of this step till it reaches a reflexive function-constant equality. Nevertheless, proving the equalities of flat-constants that are implied by transitivity or reflexivity is not that straightforward. An instance of the reflexivity axiom in the context of the Curry-clauses means that the two partial functions are the same, i.e., the same function is applied on the same arguments. If the term in question would not be a partial function application, it would simply be a reflexive instance. However, in that case this reflexivity has to be proven for all of the arguments of the function. In currified and flattened form it can be achieved just through further steps. The following example demonstrates this scenario.

Example 4.10 (Isomorphic Curry-clause)**.** Assume, that the theory solver accepted the following set of literals: $\{b_1 = b_2, f(a, b_1) \neq f(a, b_2)\}$. After currification and flattening,

Algorithm 4.12 ExplainCurry method

```
1: procedure ExplainCurry(currentClause)
2:     switch currentClause.StepRule do
3:         case tmp_curry_reflexive
   ▷ e.g. $c_f = c_f$ where $c_f$ is function-constant
4:             Return
5:         case tmp_curry_congruent
   ▷ e.g. $c_i = c_j$ where $c_i = C(c_f, a_1)$, $c_j = C(c_f, b_1)$ and $arity(f) > 1$
6:             ExplainCongruent(currentClause)
7:         case tmp_curry_isomorph
   ▷ e.g. $c_i = c_i$ where $c_i = C(c_j, a)$ is a partial function
8:             $\langle c_j, a\rangle$ := GetCurryArguments($c_i$)
9:             currentPremise1 := GetClause($c_j = c_j$,curry_undecided)
10:            currentPremise2 := GetClause($a = a$,undecided)
11:            AddPremise(currentClause,currentPremise1)
12:            AddPremise(currentClause,currentPremise2)
13:        case tmp_curry_transitve
   ▷ e.g. $c_i = c_k$ where $c_i = c_j \wedge c_j = c_k$ and $c_i, c_j, c_k$ are flat-constants
14:            $\langle sc_i, a_i\rangle$ := GetCurryArguments($c_i$)
15:            $\langle sc_k, a_k\rangle$ := GetCurryArguments($c_k$)
16:            currentPremise1 := GetClause($sc_i = sc_k$,curry_undecided)
17:            currentPremise2 := GetClause($a_i = a_k$,undecided)
18:            AddPremise(currentClause,currentPremise1)
19:            AddPremise(currentClause,currentPremise2)
```

the set of equalities looks as follows: $\{b_1 = b_2, C(c_f, a) = c_1, C(c_1, b_1) = c_2, C(c_1, b_2) = c_3, c_2 \neq c_3\}$. The negation of the $c_2 \neq c_3$ disequality is simply derivable with congruency, however the currification splits this congruence step into a sequence of several steps:

$$\underbrace{C(c_f, a)}_{c_1} = \underbrace{C(c_f, a)}_{c_1} \wedge \mathbf{b_1} = \mathbf{b_2} \supset \underbrace{C(C(f, a), b_1)}_{c_2} = \underbrace{C(C(f, a), b_2)}_{c_3} \tag{4.7}$$

$$c_f = c_f \wedge a = a \supset \underbrace{C(c_f, a)}_{c_1} = \underbrace{C(c_f, a)}_{c_1} \tag{4.8}$$

$$c_f = c_f \tag{4.9}$$

$$a = a \tag{4.10}$$

Formula 4.8 is an example for a *tmp_curry_isomorph* clause. In that example, the ExplainCurry method is invoked to explain the $c_1 = c_1$ equality. Nevertheless, since c_1 was introduced to encode the $C(c_f, a)$ function, the equality $c_1 = C(c_f, a)$ can not be found in the Proof Forest but in the flat mapping hash table produced by the Transformer component. Therefore, to generate the necessary reflexive premise ($a = a$), the Transformer component of the theory solver is asked to forward the arguments (c_f and a) of the encoded Curry-function (line 8 of 4.12).

When the current Curry-clause defines an equality relation between two flat-constants that are not neighbours in the Proof Forest (line 13 of Alg. 4.12), the algorithm labels it with the special *tmp_curry_transitive* tag. The obvious management of these clauses would be to traverse the path between the constants in the Proof Forest just as with the normal transitive clauses. However, in that case, the involved constants encode partial-functions, that is to say, the current equality was derived by the congruence closure algorithm based on the equalities of the arguments. Therefore, instead of constructing a transitive chain of further Curry-clauses as premises, the ExplainCurry method directly proves the equality of the arguments based on the Curry-functions that are encoded by the current flat-constants. For this, the flat-mapping table of the Transformer component is looked up (line 14-15 of Alg. 4.12). When the represented Curry-functions are given, two new clauses are constructed as premises, where one of them is a Curry-clause while the other is a proper clause (line 16-19 of Alg. 4.12), just as in the ExplainCongruent method.

When all the clauses are already complemented with the necessary premises (the Pending Proofs queue is empty), there are just a few more steps back to complete the proof. First of all, the Curry-clauses have to be eliminated, i.e, their valid premises have to be forwarded to the clauses where they really belong. This step is conducted by the MergeClauses method of the Proof Lemma (line 24 of Alg. 4.9). The procedure iterates through the clauses of the proof (note, that the Curry-clauses are not part of the proof directly) and whenever it finds a Curry-clause as premise it recursively replaces this clause with the non-Curry negative literals of it. Moreover, when it is assumed already that no Curry-clause is referenced in the proof, the clauses have to be serialized. Each clause which was generated during the Explain method gets an internal id number at construction. This helps the debugging process and shows the order of the construction of the clauses. Since the result proof lemma should be started with the input assertions, a reordering of the clauses is necessary, i.e., a new serial number has to be attached to each clause of the lemma.

After merge and serialization, in the case it is expected, the resolution part of the proof is generated (see Section 3.2). The GenerateResolutionTree method recursively resolves each clause with the premise clauses. The starting point of the recursion is the clause that concludes the negation of the conflict causing disequality. In each step, the method declares the result clause which was inferred by the resolution. During the derivation, the input assertion clauses are ignored from the premises, that is to say, they are carried forward by the resolvent clauses. Then the very last step of each Proof Lemma is as follows: Resolve the last resolvent clause with the result clause of the distinct predicate elimination step and with all the input assertions. If the proof was correct, the result clause of this step is empty, otherwise the last clause of the proof contains all those equalities of the trace that are not proven. Hence, the GenerateResolutionTree method provides an internal checking process for the Proof Manager component.

The following example depicts the whole detailed explanation method.

Example 4.11. Assume that the following set of equalities is given to the EUF theory solver (see Example 3.1):

$$E = \{a = b, a = c, f(b, a) \neq f(c, a)\}$$

After preprocessing, the standard form of E is as follows:

$$E' = \{a = b, a = c, C(c_f, b) = c_1, C(c_1, a) = c_2, C(c_f, c) = c_3, C(c_3, a) = c_4, c_2 \neq c_4\}$$

After that the Congruence Closure algorithm finds E' $\mathcal{T}_\varepsilon$-unsatisfiable, the Proof Forest of Figure 4.4 is built by the Proof Manager component. The found contradiction is related with the $c_2 \neq c_4$ disequality, therefore the Explain(c_2,c_4) method is invoked in order to generate a detailed explanation of the conflict. First of all, the object of the trace has to be created (line 2 of Alg. 4.9). Each proof lemma is destined to contain the clauses that are proving the inconsistency related with one disequality. Therefore, the conflict causing disequality is a necessary argument of the initialization. The proof lemma saves the disequality as an input assertion clause. Since the SMT-LIBv2 format has a special predicate symbol for disequalities, as a first step, the distinct predicate is transformed to

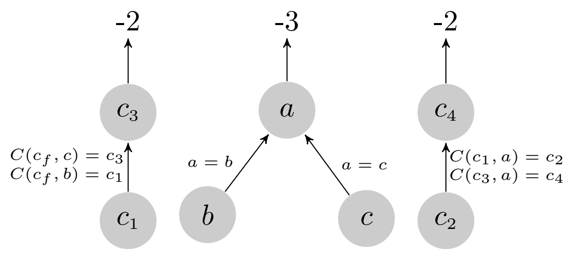

Figure 4.4: Proof Forest of E'

a negation of an equality predicate by the rule *distinct_elimination*. That is to say, the following two clauses are appended to the Proof Lemma:

```
(set .c0 (input :conclusion ((distinct c2 c4))))
(set .c1 (tmp_distinct_elim :clauses ( .c0 ) :conclusion (not (= c2 c4))))
```

The aim of the proof production process is to derive the negation of the $.c1$ clause from the subset of the input equalities and further $\mathcal{T}_\varepsilon$-valid formulas. Therefore, the root of the proof trace has to be the following clause (line 3 of Alg. 4.9):

```
(set .c2 (eq_undecided :conclusion ((= c2 c4))))
```

Since at this point it is not known which inference rule will be applied to draw the $c_2 = c_4$ conclusion, this field of the new clause remains the default value, *eq_undecided*. This clause is appended to the Pending Proofs list (line 4 of Alg. 4.9) and the iteration of the Explain method starts. Since at the first run, the only clause in the Pending Proofs list is $.c2$, the procedure starts with the explanation of this clause. First, the nearest common ancestor and the distance between c_2 and c_4 are identified in the Proof Forest (line 7 of Alg. 4.9). The nearest ancestor in that case is c_4, while the size of the path between c_2 and c_4 is one. Based on the path-size information and on the label of the given edge, the SetStepRule method modifies the inference rule of $.c2$ from *eq_undecided* to *eq_congruent* (line 8 of Alg. 4.9). Then the Explain method appends $.c2$ to the Proof Lemma and invokes the ExplainCongruent method (line 15-16 of Alg. 4.9). The ExplainCongruent method processes the two Curry-equalities $C(c_1, a) = c_2$ and $C(c_3, a) = c_4$ on the edge between c_2 and c_4. First, a clause is searched in the Pending Proofs and Proved Clauses lists which concludes the $c_1 = c_3$ equality. Since, c_1 and c_3 are left arguments of a Curry-function, it can be assumed, that their equality is proved by a Curry-clause. This equality was not examined yet by the explain method, so the GetClause method (line 2 of Alg. 4.11) returns with the following new clause:

```
(set .c3 (tmp_curry_undecided :conclusion ((= c3 c1))))
```

Then the equality of the Curry-function's second arguments is searched, and the following clause is constructed:

```
(set .c4 (eq_undecided :conclusion ((= a a))))
```

Although, in that case it is easy to see that the equality is an instance of the reflexivity axiom, the method constructs it with the *eq_undecided* tag, because this clause belongs to another iteration of the Explain method. The GetClause method saves the constructed clauses in the Pending Proofs queue. The references of the generated new clauses $.c3$ and $.c4$ are appended to the premise literals of the current clause (lines 4-5 of Alg. 4.11) and the ExplainCongruent method returns. After the procedure the current clause looks as follows:

```
(set .c2 (eq_congruent :conclusion (not .c3 )(not .c4 )((= c2 c4))))
```

where the premise clauses at the moment contain only their conclusion equality but no further literals. In that form the clause $.c2$ is assumed to be done and saved into the

Proved Clauses set. The Explain method starts the next iteration, where the current clause is $.c3$ with the equality $c_3 = c_1$ as conclusion. In the Proof Forest c_3 and c_1 are neighbours, therefore the SetStepRule method modifies the inference rule of the clause from *tmp_curry_undecided* to *tmp_curry_congruent* and the clause through the ExplainCurry method ends up in the ExplainCongruent procedure. There, based on the label of the edge between c_1 and c_3, the two necessary premises are constructed:

```
(set .c5 (tmp_curry_undecided :conclusion ((= c_f c_f))))
(set .c6 (eq_undecided :conclusion ((= c b))))
```

Then the references of the clauses $.c5$ and $.c6$ are appended to the premise list of $.c3$:

```
(set .c3 (tmp_curry_congruent :conclusion (not .c5 )(not .c6 )((= c3 c1))))
```

When the ExplainCurry method returns, the Explain method saves $.c3$ into the Proved Clauses list (without saving it into the Proof Lemma) and starts the next iteration. The next clause in the Pending Proofs list is $.c4$ that concludes the equality $a = a$. The SetStepRule method identifies it as a simple reflexive clause. Therefore, the Explain method stores it in the Proof Lemma and Proved Clauses containers (lines 13 and 23 of Alg. 4.9) and moves on to the next clause in the Pending Proofs. The final form of $.c4$ is then:

```
(set .c4 (eq_reflexive :conclusion ((= a a))))
```

The examined clause in the next iteration is $.c5$ with the equality $c_f = c_f$ as conclusion. This equality is also a reflexivity instance. However, this one is a Curry-clause, therefore the ExplainCurry method handles it (line 3-4 of Alg. 4.12). Since it defines an equality between two function-constants, the method has nothing to do and returns. The last element of the Pending Proofs list is then $.c6$, that explains the $c = b$ equality. The nearest common ancestor of c and b is a and the length of the path is two. The SetStepRule method decides that $.c6$ is an instance of the transitivity axiom and, therefore, modifies its inference rule tag from *eq_undecided* to *eq_transitive*. Then the Explain method saves $.c6$ in the Proof Lemma (line 20 of Alg. 4.9) and invokes the ExplainAlongPath method, first on the $c \to a$ path. The ExplainAlongPath method traverses through the path between c and a in the Proof Forest, that in this case contains only one step. The input equality $a = c$ is not contained neither by the Pending Proofs list, nor the Proved Clauses set. Therefore, the ExplainAlongPath method examines the edge between the two constants. The label on this edge is a single input equality, thus, the method constructs only one new clause (line 6 of Alg. 4.10):

```
(set .c7 (input :conclusion ((= a c))))
```

At this point it is sure that $c.7$ is a new input assertion clause that requires no further literals, therefore the ExplainAlongPath method appends $.c7$ to the Proof Lemma and Proved Clauses set. Furthermore, the reference of the new clause is included into the premise list of the current clause $.c6$ (lines 7-9 of Alg. 4.10). Then the course of the ExplainAlongPath method repeats over the path between b and a. This time the procedure constructs the following clause:

```
(set .c8 (input :conclusion ((= a b))))
```

After the two explanation methods, the final form of $.c6$ is as follows:

```
(set .c6 (eq_transitive :conclusion (not .c7 )(not .c8 )((= c b))))
```

At this point the Pending Proofs list is empty, therefore the iteration of the Explain method is over. After merging and serialization, the final form of the proof with the resolution tree is as follows:

```
(set .c1 (input :conclusion ((= a c))))
(set .c2 (input :conclusion ((= a b))))
(set .c3 (input :conclusion ((distinct (f b a) (f c a)))))
```

```
(set .c4 (tmp_distinct_elim :clauses (.c3) :conclusion (not (= (f b a) (f c a)))))
(set .c5 (eq_congruent :conclusion (not (= a a))(not (= c b))((= (f b a)(f c a)))))
(set .c6 (eq_reflexive :conclusion ((= a a))))
(set .c7 (eq_transitive :conclusion (not (= a c))(not (= a b))((= c b))))
(set .c8 (resolution :clauses ( .c5 .c6 ) :conclusion (not (= c b))((= (f b a) (f
    c a)))))
(set .c9 (resolution :clauses ( .c8 .c7 ) :conclusion (not (= a c))(not (= a b))
    ((= (f b a) (f c a)))))
(set .c10 (resolution :clauses ( .c4 .c9 .c1 .c2 ) :conclusion ()))
```

4.7 Interface Equalities

The previous sections described the behaviour of the theory solver in detail, when the set of input (dis-)equalities was found unsatisfiable by the Congruence Closure component. Nevertheless, the theory solver is expected to provide further functionalities for the satisfiable formulas as well. At present, the theory combination strategy of SMT4J is not finally implemented yet, therefore the requirements on the theory solver in order to support the final method of the theory combination might change. However, it is most probable that the final implementation will be an extension or refinement of the Nelson-Oppen approach [27].

For the moment, the theory solver produces as interface deduction the conjunction of those equalities which were propagated by congruency during the congruence closure calculation without considering the involved constants. Moreover, on request the solver generates all equalities that were found so far based on the actual state of the congruence classes. As the resulting set of literals significantly blows up the search space, optimizations are necessary, which is, however, beyond the scope of this work. Still, the former two facilities provide already sufficient information for the combination procedure of stably infinite convex theories without inconvenient effort.

It is not decided yet, how SMT4J will support the combination of decision procedures for nonconvex or finite background theories, therefore this topic is beyond the scope of this thesis. For further details of these scenarios see e.g. [31, 40]

4.8 Implementation Details

The most important modules of the theory solver and their responsibilities can be found at the beginning of this chapter. This section provides a short overview of their implementation details and briefly describes some decisions made during the development. The implementation of the theory solver consists of 26 classes and approximately 3000 lines of code. Although, it is a medium sized project, the complexity of the system is considerable at some parts, especially when taking into account the required familiarization efforts.

A main objective during development was to decompose the solving process into small steps. Each step is implemented by a small component that is responsible only for a confined set of isolated tasks. This high degree of modularization supports the verification of intermediate results and increases the extensibility of the solver. Furthermore, it provides easy replacement for the implementation of several components in the theory solver.

The module exchange facility of the implementation is required for instance to provide a comfortable way to switch between core and proof production mode. Figure 4.5 depicts the partial class diagram of the related classes. The initialization of the Proof Manager component is based on the actual input options of the solver. The core producing component and the detailed proof producing component implement the same interface and return with a Proof Lemma object. When the theory solver is in core production mode, the textual description of the proof object contains the SMT-LIBv2 definition of the conflict set. When the theory solver is expected to produce detailed certificates of

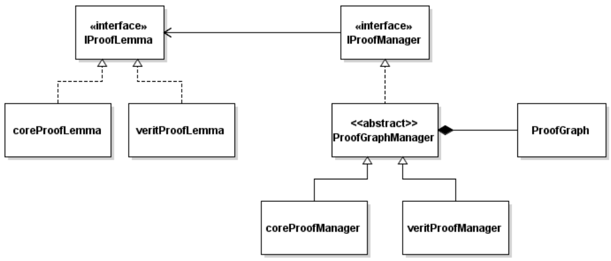

Figure 4.5: UML class diagram of Proof Manager classes

the upcoming conflicts, the proof object contains the textual representation of a sequence of clauses. In both cases, the proof object contains a sequence of (dis-)equalities as the unsatisfiable core. Both Proof Manager implementation works on the same Proof Graph object. An additional abstract base class guarantees that the graph related operations do not modify the graph and provides code sharing between the different algorithms.

Another relevant principle of the development was to extend the theory solver with the core and proof production facilities but without the influence on the performance and memory usage of the solver when these functions are not used. Thus, the core or proof producing component is created just when a contradiction is already found. Furthermore, the information that the congruence closure algorithm has to store during the process is minimized. As a result, the proof production capability of the theory solver has only minimal overhead when the input set of literals is satisfiable.

One decision was made during the implementation of the clause class which may influence the extensibility of the proof generation functionality. At first sight, clauses should be implemented through a common abstract parent clause class and each type of proof step should be a new child of this element. However, in the implementation there is only one general clause class which contains a label to store the current type (e.g, undecided, transitive, etc.) of a proof step. In this way, the different types of proof steps are implemented by the same class where only the label of the instance in question describes the applied rule. Tagged classes usually fall under the category of 'code smell'; however, in this situation it is reasonable. The proof generation algorithm handles the proof steps based on their current type. However, the behaviour of a proof step object does not depend on the current type. Furthermore, many clauses are created without the knowledge of their type, i.e., a proper class hierarchy of the clauses would induce several casting statements in the proof generation algorithm.

Chapter 5

Proof Checking

5.1 Overview

As already described in Section 3, it is difficult to establish a sufficient level of trust in an SMT Solver due to its complex implementation. One possible solution for this challenge is to force the solver to produce some kind of evidence about the correctness of the made decisions. Then, this evidence can be independently verified with a much simpler and more trustworthy tool. In the following, such an evidence of correctness is called *certificate*. Since the format and content requirements of SMT certificates are not standardized yet, most of the proof producing SMT Solvers use proprietary formats and employ different tools for proof verification. The following sections illustrate some verification tools together with experimental results on their performance. The environment of these experiments is described in Section 6.1. Note that these checkers are responsible for the verification of the entire proof of an SMT Solver, while in the context of this thesis the proof generation and checking is restricted only to the deduction of the $\mathcal{T}_\varepsilon$-lemmas. A brief description of the implemented checker for these lemma-proofs is given later on in this chapter.

5.1.1 veriT - SMTCoq

The certificates produced by the veriT SMT Solver [11] can be verified, for example, with SMTCoq. This software contains a modular certified checker for the veriT answers. SMTCoq defines its own general notation of certificates and uses an OCaml preprocessor in order to integrate the produced proofs of veriT into the Coq system. Coq is an interactive proof assistant which facilitates, among other things, the automatic check of proofs by a relatively small verified kernel. The details of Coq are beyond the scope of this thesis, for an exhaustive documentation of Coq see e.g. [4].

To use this verified kernel of Coq, SMTCoq transforms the variables, literals, clauses and theories of the produced proofs into an internal representation of Coq. This means that the checker verifies an encoded version of the produced proofs. SMTCoq handles the CNF transformation steps, resolution on the propositional level and theory reasoning (in EUF and LIA) in a separate manner. That is to say, the checker is built up from the combination of several small independent checkers where each of them is dedicated to the verification in one specific domain. These small checkers are actually computational functions inside Coq and they are proved to be correct. For precise details of SMTCoq see e.g., [2, 24].

Figure 5.1 presents the performance of the veriT SMT Solver (version 201410) on some parts of the QF_UF benchmarks (see Section 6.2.2). The veriT SMT Solver without proof generation could decide the satisfiability of almost all of the formulas; merely three problem instances remained unsolved. Since certificates can be generated only for contradictory

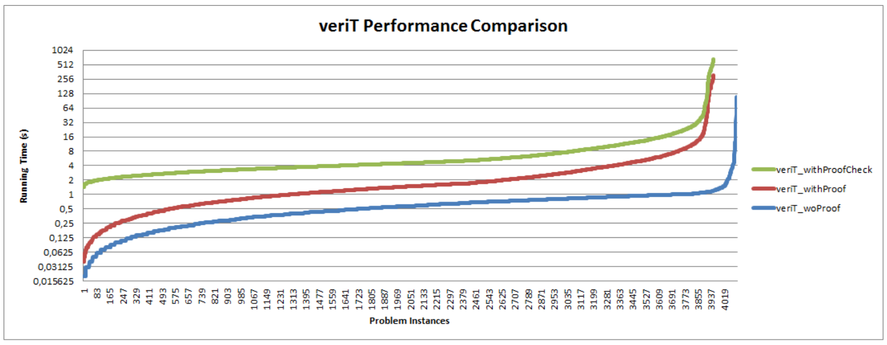

Figure 5.1: veriT overheads of proof production and verification

formulas, the diagram is restricted to those problem instances which the solver decided to be unsatisfiable. The blue line of the chart shows the required running time of the solver on these instances, without proof production. The distance between the blue and red lines of the figure shows the cost (in seconds) of the proof production. The average factor of the slow-down at proof generation is generally 5, but, if the instances which could not been solved by proof production in less than 300 seconds are also considered, this average factor is increased to approximately 30. This may appear as a significant overhead, but still 93% of the problem instances were solved in less than 10 seconds with proof production and only 144 problem instances remained unsolved within 300 seconds. The green line in Figure 5.1 represents the sum of the running times of the veriT solver (version Verit2c2b43b) and the SMTCoq (version 1.2) tool. The proof producing and checking together took approximately five times longer, on average, than just the proof producing experiments. Note that this overhead was measured on the Verit2c2b43b version of the solver (since the checker tool recommended this version), which is slightly slower then version 201410 of the veriT solver. The SMTCoq checker was invoked on 3949 generated certificates. Since the first step of the proof checking process with this tool is to transform the produced proofs into another representation, the overhead of the verification with SMTCoq is not so surprising. When the encoding of a produced certificate is unsuccessful, SMTCoq returns with an error message. Although the proofs were generated by the recommended version of the veriT SMT solver, the current version of SMTCoq failed during the transformation process for most of the proof traces. It could only verify less than 150 proof objects (mainly the proofs of the NEQ, PEQ, SEQ and eq_diamond benchmark families); for all further proofs it returned without a decision. All in all, it appears that SMTCoq as a verification tool for veriT certificates is not complete yet.

5.1.2 CVC4 - LFSC Checker

As described in Section 3.1.2, the CVC4 SMT Solver [5] produces certificates in the format of LFSC. In this format the inference rules are encoded as signatures and a generated certificate is actually a term, that is to say, type checking of this object can serve as a verification process. An efficient LFSC proof checker is provided in C++ by Andy Reynolds and Aaron Stump that comes together with the CVC4 solver and can be invoked automatically for each generated proof object. This tool takes the signatures and side conditions that are defined as the proof system of the solver and based on them checks whether the type of the proof object is as it is expected.

The LFSC checker employs several performance optimizations in order to facilitate efficient proof checking. A characteristic feature of the checker is that it is designed to handle really large proof traces [36]. First of all, it avoids to read and parse the entire proof at once, but keeps always just the parts that are relevant for the current state of the checking process in the memory. With this incremental checking strategy the memory usage of the tool is considerably reduced. Furthermore, during the verification of deeply nested proof objects the checker employs tail recursion at many points when it is possible instead of naive recursion in order to avoid the possibility of a stack overflow.

In LFSC one can define computational side conditions for the inference rules in a simple functional programming language. These side conditions are then compiled into C++ code and accessible by the checker directly. In each step where an inference rule with constraints is employed, the compiled code of the condition is called. If the code fails (either explicitly or implicitly), it means that the side condition of the rule is not granted and therefore the application of the rule is not possible. In the proof system of CVC4 side conditions are defined corresponding to, for example, the resolution steps. These conditions facilitate the so-called *deferred resolution* during proof checking. The main idea of this strategy is to delay the computation of the resolvent clauses and perform the condition checks only in a final simplification step. In that way, checking resolution steps requires only constant time. For more details about deferred resolution and the LFSC checker see e.g. [30, 35, 38, 39].

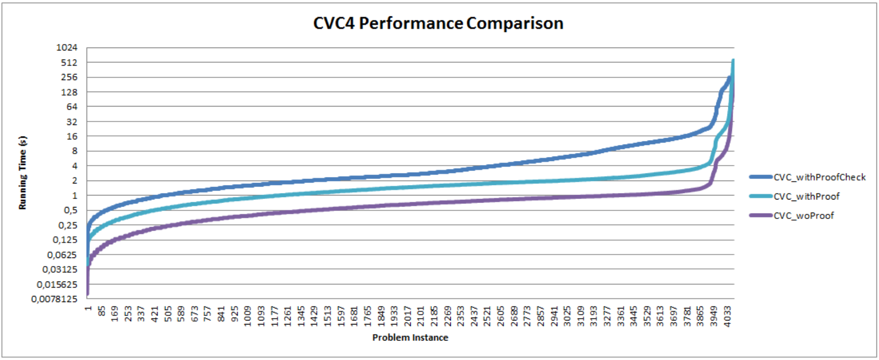

Figure 5.2: CVC4 overheads of proof production and verification

Figure 5.2 shows the performance on a logarithmic scale of the CVC4 SMT Solver on the QF_UF benchmarks (see Section 6.2.2). CVC4 without proof production could solve almost all of the problem instances (except 19). Figure 5.2 contains the running times of those instances which were unsatisfiable. It appears, that the average overhead of proof production in CVC4 is approximately 85%. The solving process with the proof production and checking process together is on average more than 5 times slower than the pure solving process. With proof producing and checking only 49 formulas remained unsolved, where 16 instances could not be solved because of the space limitation (which in that experiment was only 5GB). It indicates, that the memory management of the LFSC checker with the before described optimizations is effective. All in all, although the factor of slow-down with proof production and validation appears great, with this configuration still 89% (3611 from 4052) of the instances were solved and verified in less than 10 seconds.

5.2 EUFChecker

This section briefly presents the prototypical checker that was developed in C++ to efficiently verify the produced $\mathcal{T}_\varepsilon$-proof witnesses of the EUF theory solver of SMT4J. Since the produced certificates are well detailed and contain all necessary information, there is no need for elaborated reasoning techniques during the checking process. The implementation of the EUF Checker tool consists of 13 classes, where 12 of them contain less than 100 lines of code. The biggest class (with 380 LOC) is responsible for the parsing tasks in the verification process. The generated unsatisfiable cores of SMT4J are generally quite small (see Section 6.3.2,'avg. core size'). Therefore, it can be assumed, that the generated certificates can be read into memory and there is no need for special memory management. Note that the main purpose of the checker is to provide information about the correctness of the proofs generated by the EUF theory solver of SMT4J, and nothing more. Thus, extensibility was not a main requirement on the system during development. All in all, although the code of the checker is not verified, due to its simplicity, it can be employed to increase the trust in the generated certificates and therefore also in the solver results.

The full proof generated by SMT4J contains several sections beside the theory related certificates, nevertheless the checker is focusing on the steps of the EUF deductions and does not require the context or the resolution tree sections of the proofs. The identification of these parts of the proofs is simple. The responsibility of the EUF Checker is to decide the correctness of each of these proof traces. Two important components of the tool are the following.

Term Table: Contains all occurring terms of the proof. Each term is described by its name and arity and represented by an integer value. The responsibility of the Term Table is to guarantee that one term is constructed only once and the same instance is shared in all occurrences.

Equality Table: Contains all occurring atoms of the proof, namely it is a collection of all the equalities in the proof. The responsibility of the Equality Table is to guarantee that one equality is constructed only once and the same instance is shared everywhere. The conflict causing disequality is also stored here separated from the other equalities.

When an EUF-theory proof trace is extracted from the output of SMT4J, the checker starts to parse the proof line by line. The checker assumes, that the equalities are defined over same-sorted terms and the function symbols do not violate their signatures, namely type checking is not included into the process. It is not difficult to see that by including the context of the proofs into the output of SMT4J or by referring to the original problem instance that was solved by it, this functionality could be easily added. Nevertheless, in the former case the produced certificates would consume unreasonably more space, while the latter case would involve unnecessary terms into the verification process.

In the EUF Checker the equalities are identified by their arguments, without considering the order of them. In that way, the symmetry property of them is handled implicitly. Each equality object has two arguments. Furthermore, each equality has a state that indicates whether the given equality is proved already or not. An equality is assumed to be proved when it is implied by a correct step of the proof whose all premise equalities are proved. Moreover, each equality contains a sequence of references for those proof steps, where the equality in question is employed as premise.

The parsing process of the EUF Checker conducts syntactical checks on the proof steps and terminates with an error message in the case of problem (e.g., missing parenthesis, unrecognised inference rule etc.). The proof steps are represented as a pair of an equality and a (possibly empty) sequence of further equalities $\{\langle p_1, p_2, ..., p_n\rangle, c\}$. Although the order of the clauses in the produced certificates is fixed, the checker does not rely on it, that is to say, there are no assumptions about the order of the proof steps in the trace. When an input clause is parsed by the checker, it marks already the asserted equality to be proved. The other proof steps are saved in a collection without influencing the state of the involved equalities. When all the steps are parsed, it means that the Term Table and the Equality Table collections of the checker are successfully filled up based on the current proof trace and the asserted equalities are marked to be proved. Moreover, after all steps were processed successfully, it can be assumed that the checker identified the conflict causing disequality and checked, whether there is a proof step that concludes the negation of it or not. In the latter case the checker returns with an error message.

After parsing, the verification of the proof steps starts. Each proof step has to be a valid instantiation of its corresponding inference rule. The reflexive step verifier checks that the premise list is empty and the conclusion equality is defined between identical terms. The congruent step checker examines more details. First of all, the conclusion equality has to be between two function symbols where the signature of the two symbols are the same (in that case it means only the arity, since type information is not handled). The number of the premise equalities has to be equal with the arity of the given function symbol. Then the procedure builds equalities from the pairwise arguments of the function symbols on the two sides of the conclusion equality and checks whether this equality is contained in the premises or not. When an equality was not found, the method returns with an error message. Moreover, in the case when one of the premises was not used during this checking process, the method throws another error message. Note that the checker assumes that each necessary equality is included as premise exactly as many times, as it is used, namely the premise list can be redundant, but the order of the premises is not constrained. The transitive steps have to fulfil the following constraints. The conclusion equality is not allowed to be reflexive and the number of premise equalities has to be at least two. The order of the premises is not fixed in this rule either, therefore the verification of the transitive chain is done step-by-step. Starting from one of the arguments of the conclusion equality, the procedure seeks a premise equality that contains it. Then this premise is marked as used and the step is repeated with the other argument of the premise equality as the starting point. The process picks and seeks as long there are no more unused and fitting premises left or the other argument of the conclusion equality is reached. In the former case the transitive chain is not correct, therefore an error message is sent. In the latter case if there are no unused premise equalities the method returns with true. If at some point there is no premise with the currently searched argument, it means the transitive chain is broken, therefore the return value is false.

When the verification of the proof steps was successful, the checker starts to process them, namely it iterates through them and examines the state of the premise equalities. When a proof step contains only proved premises (or no premises at all), the conclusion equality is marked to be proved. Then this recently proved equality forces all the other proof steps where it was involved as a premise to evaluate their new state. When there are no more proof steps to fire, the method stops. At this point the checker examines the equality that was the negation of the conflict causing disequality, whether it became proved or not. Obviously, when it is not proved, the checker returns with false. As a secondary check, the tool evaluates the state of all the further equalities in the Equality Table.

In that way, the EUF Checker tool recognizes when a produced proof trace does not contain enough information to derive a contradiction. However, the checker does not return with false when the proof contains unnecessary asserted equalities or proof steps as long they are correct instantiation of an inference rule.

Chapter 6

Experimental Results

This chapter is dedicated to summarize and evaluate the experiences. The general performance and properties of the SMT4J framework had minor importance due to its' incompleteness. In order to examine the performance and other attributes of the extended EUF theory solver, a series of experiments has been conducted on the QF_UF benchmarks and on further Fuzzer-generated EUF theory specific problem instances. For comparison, other state-of-the-art SMT Solvers were included in the examination.

6.1 Experimental Setup

The experiments were conducted on a 3.40GHz Intel Core i7-2600 machine with 16 GB memory and Ubuntu 14.04 (64-bit) as operation system. The constraints of the executions were controlled by the tool called RunLim[1]. The default limits of each formula instance were 300 seconds real time and 10GB space. The involved further SMT Solvers of the experiments are CVC4, veriT and Z3. Henceforth the experiments without proof generation nor checking are considered as the base experiments of each SMT solver. For this base scenario, CVC4 (version 1.4, compiled with gcc 4.8.2) was built with its standard configuration, without any optimization support, i.e., without the --best and --enable-gpl options. For the proof generation capabilities, another instance of CVC4 was installed with the --enable-proof option. The veriT SMT Solver was necessary also in two different versions. The default instance has version 201410 and was used for the base case running time experiments. The SMTCoq-1.2 verifier (see Section 5.1.1) recommends a specific development version of veriT (Verit2c2b43b). Furthermore, the verifier builds on native-coq (version 140702). For independent reference results the Z3 SMT Solver was employed (version 4.4.0 - 64 bit).

6.2 Input

6.2.1 EUF Fuzzer

The main intention of the EUF Fuzzer tool is not to tense the performance of the theory solver but to reveal as many bugs and malfunctioning steps of the solver and the proof checker as possible. Therefore, the problem instances produced by the fuzzer tool consist only of $\mathcal{T}_{\mathcal{E}}$-literals, namely each assertion of the generated problem instances is an atomic formula with either an equality or a distinct predicate symbol. The propositional abstraction of a generated formula is satisfiable always by assigning true for each literal, but in no other way. It follows, that the theory solver is invoked only once during the solving process and the result of the decision depends just on the EUF theory solver. All in all,

[1]http://fmv.jku.at/runlim/

the generated formulas are suitable to test directly the theory solver by minimizing the role of the SAT Solver and the entire framework in the decision process.

The EUF Fuzzer tool is designed to be flexible and provides several calibration possibilities. For problem instance generation, seven properties of the formulas can be restricted. First of all, the maximum number of involved sorts, as well the maximum number of constant and function symbols is decidable. For the given theory, multiple sorts are not necessarily usefull, since it splits the problem space of the theory solver into several independent partitions, i.e., it simplifies the instances. The maximum number of symbols and the properties of the function symbols influence the likely size of the congruence classes. For the function symbols the maximal width (arity) and depth is also configurable. Furthermore, the number of generated assertions and the proportion of equalities and disequalities are also parametrized.

At initialization the tool generates the possible signatures based on the given parameters. Then the formulas are built from top to bottom where each new assertion combines the generated signatures randomly. For each term of the constructed expressions first the arity of the signature is picked. This selection is random based, where the possibility is decreasing for the bigger numbers. When the arity is decided, a function symbol, whose signature suits to the expected arity and sort requirements, is selected randomly. The sub-expressions are built in a same way as long either a constant symbol is picked or a function symbol reaches the depth constraint (in that case a random constant symbol is used). The procedure does not try to avoid repetition in the generated expressions, since the congruence closure component has to correctly handle redundancy. Actually, the repetition on the level of sub-expressions is essential, in order to generate unsatisfiable instances.

Solving the problem instances generated by the EUF Fuzzer tool invokes the EUF theory solver only once. When the generated problem instance is $\mathcal{T}_{\mathcal{E}}$-satisfiable, the theory solver does not produce theory lemmas or proofs. Therefore, to use the tool for testing these functions of the theory solver, it is important to calibrate it in such a way, that the $\mathcal{T}_{\mathcal{E}}$-unsatisfiable formulas outnumber the $\mathcal{T}_{\mathcal{E}}$-satisfiable instances. Furthermore, the complexity of the generated proofs is also relevant, i.e., the fuzzer should avoid to generate too obvious contradictions (e.g. conflict that involves only two literals). Therefore, as an initialization step of the running experiments, the tool was executed with several parameter combinations in a small interval, in order to find the best calibration for the proper problem instances. As a result, three different calibrations of the tool were selected as settings for the test case generation. Two of them generate more probably unsatisfiable problem instances, while the third one provides a balanced distribution of satisfiable and unsatisfiable formulas. In the running experiments each calibration was employed to generate 300 problem instances.

6.2.2 Benchmarks from the SMT-LIB

The SMT-LIB initiative provides a large library of input problem instances that are formulated in the SMT-LIBv2 language and classified by logic. These formulas facilitate the evaluation and the proper comparison of SMT Solvers. For the (quantifier free) logic of equalities over uninterpreted functions currently 6650 problem instances are provided grouped into 6 families.

6.3 Results

6.3.1 Fuzzer Results

The fuzzer tool arranged with the SMT4J solver and the EUF Checker tool in a pipeline forms a perfect test bed for the system. In that way, during the development process an exhaustive, automatic regression testing environment focusing on the correct functioning for the theory solver was built. As a secondary check for the decisions made by the EUF theory solver of SMT4J, each generated input formula is evaluated by the veriT and the CVC SMT Solvers as well, in order to compare the results. Solving of the problem instances generated by the EUF Fuzzer is very fast. Thus, it does not make sense to include performance comparison for them.

6.3.2 Benchmark Results

The lazy SMT solving approach of SMT4J with the single core producing EUF theory solver unfortunately did not prove to be efficient enough for the standard QF_UF benchmarks, that is to say, the solver could decide the satisfiability only of 42 formulas from the 6650 problem instances. Nevertheless, these experiments provide also some interesting statistics about the properties of the theory solver, since even if the satisfiability of the problem instances at the end mostly were not decided, during the process the theory solver was asked to decide the $\mathcal{T}_\varepsilon$-satisfiability of enormous amount of propositional assignments.

benchmark		avg. iteration	avg. input size	avg. core size	avg. red.
eq_diamond		10473,9	99,1	99,1	0%
loops6		3869,4	374,2	8,3	97,8%
NEQ[2]		1315,5	456,0	4,9	98,9%
PEQ		1576,2	902,0	15,3	98,3%
QG-class	qg5	5982,5	228,7	7,5	96,7%
	qg6	3324,0	349,0	8,4	97,5%
	qg7	1898,0	535,9	9,1	98,3%
SEQ		2507,0	418,2	11,1	97,3%
TypeSafe		1,0	4,0	3,0	25%

Table 6.1: Average results of the EUF theory solver with single core production on the QF_UF benchmark grouped family-wise

Table 6.1 presents the aggregated results of the EUF theory solver with single core production on the QF_UF benchmark grouped by families. As in Section 2.3 was described, the main concept of the solver is to iteratively refine the propositional abstraction of the input formula by the generated theory-lemmas. The first column ('avg. iteration') indicates how many times the theory solver was asked to decide the $\mathcal{T}_\varepsilon$-satisfiability of a conjunction of $\mathcal{T}_\varepsilon$-literals in average, namely how many lemmas were produced by the theory solver during the decision of one problem instance in average. The second column ('avg. input size') shows that for a given benchmark family in average how many $\mathcal{T}_\varepsilon$-literals were assigned by the SAT-Solver during the iterations, namely this column describes the average size of the input set of literals of the theory solver. The third column ('avg. core size') presents the average size of the unsatisfiable cores produced by the theory solver in each iteration. The last column of the table labelled 'avg. red.' presents average reduction of the input problem size in percentage.

[2]The version of SMT4J that was employed for this experiment contained still a minor bug in the Extractor component that was influencing the results of the problem instances in this family.

The large differences between the average input size and average core size, and so the high percentages in the last column, indicate effective problem reduction. Although, the problem instances of the *eq_diamond* family were not reduced at all (as expected), all things considered, one can conclude that the produced unsatisfiable cores prune the search space of the SAT Solver significantly. The average number of iterations shows that the theory solver does not form the bottleneck in the solving process. Furthermore, the large size of input sets indicates that the theory solver serves well under large loads.

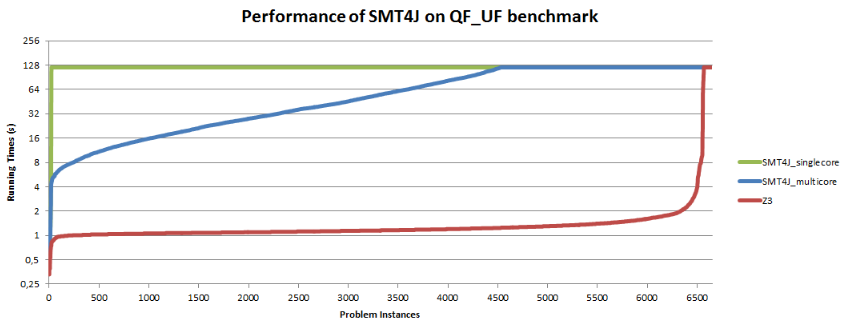

Figure 6.1: Experimental Results for multiple core production compared to single core production and Z3

Figure 6.1 presents the experimental results of the enhanced approach (see Section 4.5). The results show the superiority of the multiple-core production over the single-core generation method in the solver. While with single-core production the solver could decide only less than 50 problem instances from 6650 within 2 minutes, the multi-core production was already able to decide more than 4500 instances. The third line depicts the running results of Z3 for the same benchmarks, and shows that it could solve almost all instances. The results are presented numerically in Table 6.2.

	SMT4J_SCore		**SMT4J_MCore**		**Z3**	
	#	%	#	%	#	%
unsat	22	0,33	2558	56,4	4017	61,18
sat	0	0	1977	43,6	2549	38,82
solved	22	0,33	4535	68,2	6566	98,74
unsolved	6628	99,67	2115	31,8	84	1,26
sum	6650	100	6650	100	6650	100

Table 6.2: Numerical results of SMT4J and Z3 for the QF_UF benchmark

Note that the experiments of Figure 6.1 and Table 6.2 were conducted on a cluster of approximately 30 machines (Intel(R) Core(TM)2 Quad CPU,Q9550,@2.83GHz,8 GB memory) and with a more recent version of SMT4J, therefore the results are not directly comparable to the former experiences.

Chapter 7

Conclusion

7.1 Summary

Based on the experimental results, the main conclusion of this thesis is as follows. Although the exploitation of theory lemmas on demand approach is a simple way to integrate the support of various background theories into an SMT Solver, without further strategies and optimizations the method is not adequately efficient to produce a competitive SMT Solver. One possible strategy to enhance the performance of the approach is to generate as many theory lemmas in each refinement iteration as possible, in order to prune the searching space more effectively. In the implemented algorithm the number of possible contradictions is finite, the exhaustive core production is feasible without significant further efforts.

The EUF theory solver satisfactorily withstood the execution experiences and did not hinder the performance of SMT4J. The module provided correct and fast functionalities even under large loads. Moreover, the improved version of the theory solver with the multiple-core production functionality served already as an effective basis for the solving process of industrial problem instances.

Beyond the lemmas on demand approach, there are further satisfiability decision procedures, where the produced unsatisfiable cores of a theory solver are applicable in order to provide performance enhancement. The cost of production showed minimal in the case of the EUF theory, therefore it is a remunerative investment of effort to provide this functionality. A detailed proof production requires additional expenditures from an SMT Solver without concrete benefits in the decision process and this functionality may seem not really necessary. Nevertheless, a generated proof significantly reduces the required efforts to verify the correctness of the theory solver, consequently it is a supportive capability during the development process in order to ensure the quality of the produced results.

7.2 Further Works

Several options to improve the implemented procedure arose during the development process. This section is dedicated to describe shortly some of them.

There is one particularly notable failing of the extension of the union-find based explain method. It can not efficiently exploit when a concerned path in the proof forest was already traversed. More precisely, future work should examine how the proof generation procedure could take advantage of the Explain Forest in order to collect the necessary, and just the necessary clauses of an already discovered path in the proof forest. The most obvious optimization of the algorithm would be to store the already built and proved clauses in the proof forest right at the corresponding edges. This solution would provide a significant improvement of the proof generation procedure in multiple-core production mode, since the already constructed and proved clauses could be reused in the proving process of the next unsatisfiable core. Namely, the algorithm would be influenced in a positive way by this optimization in the cases, when the same proof forest is reused for proving more than one contradiction. Nevertheless, the current solution is more supportive for the debugging process by the separation of the proof forest from the proof generation process.

Another performance enhancement could maybe achieved by improving the quality of the produced unsatisfiable cores for example by the reduction of their size when it is possible. A smaller unsatisfiable core precludes more propositional assignments from the search space of the SAT Solver, hereby each call of the theory solver would provide substantially more benefits to the decision procedure. Thus, it would be fruitful to consider a simple post-process of the generated unsatisfiable cores, in order to eliminate unnecessary literals.

An obvious side effect of the implemented congruence closure algorithm is that redundant equalities are not exploited. This feature is relevant only in the case, when an input equality is found redundant because it was already deducted by the congruence closure algorithm through transitivity or congruency. When such an equality is involved in a contradiction, the produced unsatisfiable core contains all those equalities, which are necessary to deduce the equality in question. It may be worth to store the unprocessed, redundant equalities during the congruence closure algorithm, in order to find reduction possibilities during core production.

The current implementation of the EUF theory solver provides deduction of interface equalities in order to support the theory exchange capabilities of SMT4J. The set of deducted equalities either contains all the found equalities based on the calculated congruence classes, or only those equalities, that were propagated by the Congruence Closure algorithm directly. The former case generates an impracticable large set, while the latter solution may not provide all the necessary equalities. Since SMT4J is still under development and the principles of the theory combination method are not completely clear, the development of an efficient strategy to select the most relevant deductions remains on the list of further works. Anyway, the actual state of the implementation stores all the necessary information that would be required to identify a specific subset of the deducted equalities, i.e., to identify the equalities over shared variables.

Besson et al. in [8] gave some suggestions about the implementation of a proof verifier for their flexible proof format. Since the produced proofs of the EUF theory solver differ in many points from their proof script proposal, the implemented checker is overly simplified and less generic compared to their recommendation. When other theory solvers of SMT4J also start to produce detailed proofs and the main principles of the supported proof format are final, a programmable and more flexible verifier tool will be required.

The theory of equalities over uninterpreted functions has the so called *ground interpolation property.* Computation of ground interpolants has a valued role in predicate refinement methods [23] and in model checking systems [26]. Fuchs at al. proposed in [22] an efficient and simple algorithm to compute ground interpolants from coloured congruence graphs for the EUF Theory. Although, the proof forest of the theory solver differs from the graphs described in the paper, in future implementations the algorithm could be employed.

References

[1] W. Ackermann. *Solvable cases of the decision problem.* North Holland Publishing Co., 1954.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *Proc. of International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT)*, 2011.

[3] L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Proc. of 17th Conference on Automated Deduction (CADE)*, pages 64–78. Springer, 2000.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, et al. *The Coq Proof Assistant Reference Manual*, 1999.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. of 23rd International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[6] C. Barrett, L. De Moura, and P. Fontaine. Proofs in satisfiability modulo theories. *All about Proofs, Proofs for All. College Publications (to appear in 2015)*, 2014.

[7] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.

[8] F. Besson, P. Fontaine, and L. Théry. A flexible proof format for SMT: a proposal. In *Proc. of First International Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.

[9] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability, chapter 26: Satisfiability Modulo Theories*, volume 185. IOS press, 2009.

[10] S. Böhme and T. Weber. Designing proof formats: A user's perspective. *Proc. of First International Workshop on Proof Exchange for Theorem Proving (PxTP)*, pages 27–32, 2011.

[11] T. Bouton, D. C. B. De Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Proc. of the 22nd International Conference on Automated Deduction (CADE)*, pages 151–156. Springer, 2009.

[12] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. In *Proc. of 6th International Workshop on Satisfiability Modulo Theories and First International Workshop on Bit-Precise Reasoning*, pages 6–11, 2008.

[13] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. of 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 174–177. Springer, 2009.

[14] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of 6th International Conference on Computer Aided Verification (CAV)*, pages 68–80. Springer, 1994.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms, chapter 21: Data Structures for Disjoint Sets.* MIT press, 2009.

[16] L. De Moura, H. Rueß, and M. Sorea. Lemmas on Demand for Satisfiability Solvers. In *Proc. of Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 244–251, 2002.

[17] L. M. de Moura and N. Bjørner. Proofs and refutations, and Z3. In *Proc. of 15th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2008.

[18] D. Deharbe, P. Fontaine, and B. W. Paleo. Quantifier inference rules for SMT proofs. In *Proc. of First International Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.

[19] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[20] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of 18th International Conference on Computer Aided Verification (CAV)*, pages 81–94. Springer, 2006.

[21] M. Fitting. *First-order logic and automated theorem proving.* Springer Science & Business Media, 2. edition, 1996.

[22] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In *Proc. of 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–427. Springer, 2009.

[23] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proc. of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 459–473. Springer, 2006.

[24] C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers.* PhD thesis, Ecole Polytechnique X, 2013.

[25] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer Publishing Company, Incorporated, 1. edition, 2008.

[26] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[27] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

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

[29] R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Information and Computation*, 205(4):557–580, 2007.

[30] D. Oe, A. Reynolds, and A. Stump. Fast and flexible proof checking for SMT. In *Proc. of 7th International Workshop on Satisfiability Modulo Theories*, pages 6–13. ACM, 2009.

[31] D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical computer science*, 12(3):291–302, 1980.

[32] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[33] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 151–166. Springer, 1998.

[34] R. E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, 1978.

[35] A. Stump. *Checking Validities and Proofs with CVC and flea.* PhD thesis, Stanford University, 2002.

[36] A. Stump. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science*, 228:121–133, 2009.

[37] A. Stump and D. Oe. Towards an SMT proof format. In *Proc. of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 27–32. ACM, 2008.

[38] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[39] A. Stump, A. Reynolds, C. Tinelli, A. Laugesen, H. Eades, C. Oliver, and R. Zhang. LFSC for SMT Proofs: Work in Progress. In *Proc. of Second International Workshop on Proof Exchange for Theorem Proving (PxTP)*, page 21, 2012.

[40] C. Tinelli and C. G. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, 2005.

[41] A. Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *Proc. of 10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

[42] K. P. Varga and M. Várterész. *A matematikai logika alkalmazásszemléletű tárgyalása.* Panem, 2003.

Appendices

Appendix A

```
<?nonexml noneversion="1.0" encoding="UTF-8"?>
<smt4j:SMT4JConfig xmlns:smt4j="http://www.smt4j.org/SMT4JConfig" xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
    www.smt4j.org/SMT4JConfig SMT4JConfig.xsd ">
  <smt4j:TSolver classPath="org.smt4j.solver.tsolver.CoreSolver" name="
      CoreSolver"></smt4j:TSolver>
  <smt4j:TSolver classPath="org.smt4j.solver.tsolver.UFSolver" name="
      UFSolver"></smt4j:TSolver>
  <smt4j:Logic name="QF_UF" >
    <smt4j:tsolver>CoreSolver</smt4j:tsolver>
    <smt4j:tsolver>UFSolver</smt4j:tsolver>
    <smt4j:uninterpretedFuns>UFSolver</smt4j:uninterpretedFuns>
    <smt4j:uninterpretedSorts>true</smt4j:uninterpretedSorts>
    <smt4j:quantifiers>false</smt4j:quantifiers></smt4j:Logic>
  <smt4j:SATSolver classPath="org.smt4j.solver.dpll.sat4j.SAT4J" name="
      SAT4J" />
</smt4j:SMT4JConfig>
```

Figure A.1: Configuration file of SMT4J

```
(set-logic QF_UF)
(declare-sort S1 0)
(declare-fun x () S1)
(declare-fun y () S1)
(declare-fun z () S1)
(declare-fun t () S1)
(declare-fun f (S1) S1)
(assert (= x y))
(assert (= y z))
(assert (distinct (f z) t))
(assert (or (distinct x z) (= (f x) t)))
(check-sat)
(get-proof)
(exit)
```

Figure A.2: Input formula of Example 2.13

```
4 7 9 12 13 −14
(set .c1 (input :conclusion ((= y z))))
(set .c2 (input :conclusion ((= x y))))
(set .c3 (input :conclusion ((distinct x z))))
(set .c4 (tmp_distinct_elim :clauses ( .c3 )  :conclusion (not (= x z))))
(set .c5 (eq_transitive :conclusion (not (= y z))(not (= x y))((= x z))))
4 7 9 12 −13 14
(set .c1 (input :conclusion ((= (f x) t))))
(set .c2 (input :conclusion ((= y z))))
(set .c3 (input :conclusion ((= x y))))
(set .c4 (input :conclusion ((distinct (f z) t))))
(set .c5 (tmp_distinct_elim :clauses ( .c4 )  :conclusion (not (= (f z) t))))
(set .c6 (eq_transitive :conclusion (not (= (f x) (f z)))(not (= (f x) t))
        ((= (f z) t))))
(set .c7 (eq_congruent :conclusion (not (= x z))((= (f x) (f z)))))
(set .c8 (eq_transitive :conclusion (not (= y z))(not (= x y))((= x z))))
```

Figure A.3: SMT4J proof

Example A.1. (Symbol table) After the currification of the $f(a,b,c)$ term, the symbol table and result Curry term is as follows:

$$\begin{aligned} Curry(f(a,b,c)) = C(C(C(0,1),2),3),\ &where\\ 0 \longleftrightarrow &\langle f,(\pi_1,\pi_1,\pi_2,\pi)\rangle,\\ 1 \longleftrightarrow &\langle a,(\pi_1)\rangle,\\ 2 \longleftrightarrow &\langle b,(\pi_1)\rangle,\\ 3 \longleftrightarrow &\langle c,(\pi_2)\rangle. \end{aligned}$$

```
(set-logic QF_UF)
(declare-sort S 0)
(declare-fun g (S S) S)
(declare-fun a () S)
(declare-fun b () S)
(declare-fun c () S)
(declare-fun d () S)
(declare-fun e () S)
(declare-fun h () S)
(assert (= b h))
(assert (= c b))
(assert (= e d))
(assert (= d c))
(assert (= (g e e) a))
(assert (= (g e h) b))

(assert (distinct a d))

(check-sat)
(exit)
```

Figure A.4: Input formula of Example 4.8 and 4.11

Sworn Declaration

Linz, 17 August 2015