BDD-Driven First-Order Satisfiability Procedures (Extended Version)

David Déharbe, Silvio Ranise

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BDD-Driven First-Order Satisfiability Procedures (Extended Version)

David Déharbe — Silvio Ranise

N° 4630

Novembre 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4630--FR+ENG

BDD-Driven First-Order Satisfiability Procedures (Extended Version)

David Déharbe* † , Silvio Ranise ‡†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Cassis

Rapport de recherche n° 4630 — Novembre 2002 — 24 pages

Abstract: Providing a high degree of automation to discharge proof obligations in (fragments of) first-order logic is a crucial activity in many verification efforts. Unfortunately, this is quite a difficult task. On the one hand, reasoning modulo ubiquitous theories (such as lists, arrays, and Presburger arithmetic) is essential. On the other hand, to effectively incorporate this theory specific reasoning in boolean manipulations requires a substantial work. In this paper, we propose a simple technique to cope with such difficulties whose aim is to check the validity of universally quantified formulae with arbitrary boolean structure modulo an equational theory. Our approach combines BDDs with refutation theorem proving. The former allows us to compactly represent the boolean structure of formulae, the latter to effectively mechanize the reasoning in equational theories. We report some experimental results on formulae extracted from software verification efforts which confirm both the flexibility and the viability of our approach.

Key-words: Automated deduction, saturation theorem proving, satisfiability procedures, first-order equational theories, theory of arrays, theory of lists, boolean reasoning, BDDs

* david@dimap.ufrn.br (DIMAp — UFRN)
† This work was realized while the first author was on a post-doctoral stay at INRIA-Lorraine, thanks in part to a financial support by CAPES grant BEX0006/02-5.
‡ Silvio.Ranise@loria.fr (INRIA — Lorraine)

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Procédures de satisfaisabilité du premier ordre dirigées par des BDDs

Résumé : Pour de nombreuses activités de vérification, il est crucial de disposer d'un niveau important d'automatisation dans des fragments de la logique du premier ordre, afin de pouvoir traiter de forme mécanique des obligations de preuve. Si, le raisonnement dans des théories courantes (comme la théorie des tableaux, des listes, ou l'arithmétique de Presburger) est essentiel, il est cependant difficile d'incorporer de forme efficace de raisonnement spécifique dans des manipulations booléennes. Dans ce rapport, nous proposons une technique simple qui s'attache à traiter ce problème, et dont le but est de vérifier la validité de formules quantifiées universellement, ayant une structure booléenne arbitraire, et cela, modulo une théorie équationnelle. Cette approche combine les BDDs avec la preuve de théorèmes par réfutation. Les BDDs permettent une représentation compacte de la structure booléenne des formules, tandis que la preuve par réfutation est une forme efficace de mécaniser le raisonnement dans des théories équationnelles. Nous relatons également des résultats expérimentaux sur des formules provenant de la vérification de programmes, qui confirment à la fois la flexibilité et la viabilité de cette approche.

Mots-clés : Déduction automatique, preuve de théorèmes par saturation, procédures de satisfaisabilité, théories équationnelles du premier ordre, théorie des tableaux, théorie des listes, raisonnement booléen, BDDs

1 Introduction

It is well known that being able to reason in first-order logic with equality is a crucial task in many verification efforts. For example, it can be shown [MP95] that the problem of checking the satisfiability of the temporal formula $\Box\Pi$ (where Π is an expression of first-order logic) by a system Σ is equivalent to find a formula Π' s.t. Π' is preserved by any computational step of Σ (i.e. it is an inductive invariant of Σ) and Π' implies Π. In both cases, we need to check the validity of formulae of first-order logic. To be practical, all the verification efforts which generate proof-obligations whose validity entails a certain property require a high degree of automation. Unfortunately, providing the adequate level of automation is a rather difficult task. This is so for mainly two reasons. First, domain specific reasoning is often necessary. Many decidable theories (and possibly their combination) for both modeling the system and specifying its behavior are commonly used (e.g. the theories of lists, arrays, and Presburger arithmetic). Second, integrating domain specific reasoning with more generic deductive tasks (such as handling the propositional structure of the formulae) is crucial. For example, on the one hand we have formulae with quite a complex boolean structure encoding the control flow of a system. On the other hand, some algebraic property of the data flowing in the system can be used to modify its control flow. As a consequence, the proof obligations ensuring the correctness of such a system will require a combination of boolean and algebraic reasoning.

In this paper, we consider the problem of combining domain specific reasoning in (equational) theories with boolean manipulations. More precisely, we describe how to effectively check the unsatisfiability of (ground) formulae with arbitrary boolean structure in the presence of a background theory, presented by a finite set of (equational) clauses.

Our Approach

In [ARR02], a uniform methodology to build satisfiability procedures for equational theories based on the rewriting approach is proposed. A satisfiability procedure is an algorithm which is capable of checking whether a conjunction (also written as a set) S of ground literals is satisfiable w.r.t. a certain (equational) theory $\mathcal{T}$ (we also say that S is $\mathcal{T}$-(un)satisfiable). The approach of [ARR02] assumes that the theory $\mathcal{T}$ can be axiomatized by a finite set $Ax(\mathcal{T})$ of (equational) clauses. Many ubiquitous theories satisfy this requirement such as the theory of lists, arrays, and their combination. In the rewriting approach, a satisfiability procedure for $\mathcal{T}$ amounts to the exhaustive application of the inference rules of a superposition calculus (see e.g. [NR01]) to $S \cup Ax(\mathcal{T})$. As a consequence, to implement a satisfiability procedure, it is sufficient to use a state-of-the-art prover mechanizing the superposition calculus. This also offers an efficient alternative to specialized decision procedures (see [ABR+02] for experimental evidence of this fact).

The satisfiability of *any* boolean combination β of ground literals can be reduced to the satisfiability of a set of sets of literals by converting β to disjunctive normal form (DNF) and by splitting on disjunctions. For example, checking whether $A \vee B$ (where A and B are conjunctions of ground literals) is $\mathcal{T}$-satisfiable reduces to checking the $\mathcal{T}$-satisfiability

of A or B. Although satisfactory in theory, the technique is not viable in practice. It is well-known that transforming a formula into its DNF can cause an exponential blow-up in size. The blow-up occurs in many verification effort where complex control part of a system is compactly specified by using nested conditional expressions (also called ites, for short). On the other hand, BDDs are a compact data structure to encode formulae with complex boolean structure which commonly arise in verification efforts. In summary, our approach consists in using BDDs to represent and split formulae into simpler sub-formulae, so that they can be successfully processed by state-of-the-art saturation provers.
Plan of the paper. In Section 2, we overview the rewriting approach to build procedures to decide the satisfiability problem of conjunctions of ground literals (cf. [ARR02]). In Section 3, we explain our approach to case-split on the boolean structure of a formula ϕ by using BDDs in order to generate proof obligations whose unsatisfiability implies the validity of ϕ. In Section 4, we describe some experiments with a prototype implementation of our technique. In Section 5, we discuss the alternative approaches to the problem considered in this paper. We propose some conclusions and the future work in Section 6. In the Appendix, we describe the input syntax of our prototype and we give a proof obligation encoded in such a syntax.

2 Satisfiability Checking in First-Order Theories

The superposition calculus [NR01] offers an efficient treatment to check the satisfiability of clauses in first-order equational theories. Although equality dramatically enlarges the search space of calculi based on unification, the effectiveness of superposition lies in the powerful criteria (such as term ordering) to prune the search space while maintaining completeness. Any fair application of the rules of the calculus to an unsatisfiable set of clauses will derive the empty clause.[1] In general, the process of applying the rules of the superposition calculus to a set of clauses may not terminate since first-order logic is undecidable. If for a class $\mathcal{C}$ of clauses, we are able to prove that this process terminates, then we are entitled to conclude that the calculus is a satisfiability procedure for $\mathcal{C}$ given its refutation completeness. The methodology to build satisfiability procedures described in [ARR02] is based on this simple observation and it is organized in two phases.

Let $Ax(\mathcal{T})$ be a finite set of equational clauses and S be a set of ground literals. The first phase amounts to flattening all ground literals in S. A flat literal is either an equality of the form $f(c_1, ..., c_n) = c_{n+1}$ or the negation of an equality of the form $c_1 \neq c_2$ (where $c_1, ..., c_{n+1}$ are constants and f is an n-ary function symbol). Flattening is done by extending the signature Σ with new constants for all the distinct non-constant sub-terms in S. For example, let $\Sigma := \{f, g, h, a, b\}$ (where f, g, h are unary function symbols and a, b are constants) and $S := \{f(g(a)) = h(b)\}$. The set $S' := \{g(a) = c_1, f(c_1) = c_2, h(b) = c_3, c_2 = c_3\}$ of ground flat literals can be derived from S over the extended signature $\Sigma' := \Sigma \cup \{c_1, c_2, c_3\}$ (where c_1, c_2, c_3 are constants). It is easy to check that flattening preserves the satisfiability of the

[1] Fairness means that if some inference is possible, it will be performed at some step unless one of the parent clauses gets simplified or deleted (see, e.g. [Rus91] for a formal definition).

set S of ground literals and that it returns sets of literals which are $O(n)$ (where n is the length of the string obtained by concatenating the literals in S written in prefix notation). Some subtlety is required to perform flattening in $O(n)$ time (see [DST80] for the details).

Let S' be the result of flattening S. The second phase of the methodology consists of proving that only finitely many clauses can be generated by the exhaustive application of the rules of the superposition calculus to the clauses in $Ax(\mathcal{T}) \cup S'$. This is proved by a standard induction over the derivation which (roughly) amounts to perform an analysis of all possible inferences between clauses, for any set S' of ground flat literals. As shown in [ARR02], satisfiability procedures for the theory of lists, arrays, sets, and their combination can be derived with this approach. The proofs of termination in [ARR02] rely on particular assumptions on the ordering over terms. It is easy to satisfy these constraints by using a standard ordering (e.g. Lexicographic Path Ordering [DJ90]) whose definition can be made automatic in our context.

3 BDD Based Case Splitting

A satisfiability procedure for an equational theory can be readily implemented by using a superposition prover, once a module for flattening is available. As shown in [ABR+02], satisfiability procedures implemented in this way offer an efficient alternative to *ad hoc* procedures. This fact is no more true when we want to check the (un-)satisfiability of formulae with arbitrary boolean formulae. A preliminary translation of the formula into DNF is required. This can cause an exponential blow-up in the size of the resulting formula. Since the procedures in [ARR02] can be lifted to check the satisfiability of sets of clauses, an alternative approach is to convert the input formula into conjunctive normal form (CNF, for short) and then feed the resulting sets of clauses to the prover. Unfortunately, the size of the resulting CNF can blow-up as in the case of the DNF (see e.g. [WN01]). To overcome this problem, we propose a combination of superposition theorem proving and BDDs to effectively handle both equational and boolean reasoning.

3.1 A Formal Statement of the Problem

We assume the usual syntactic and semantic notions of first-order logic with equality [End72], where the binary symbol $=$ is interpreted as the equality relation. Let Σ be a finite signature s.t. $\Sigma := \Sigma_U \cup \Sigma_I$ where Σ_U and Σ_I are disjoint sets. By Σ_P we denote a finite set of predicate symbols disjoint from Σ containing two distinct constant symbols $\top$ and $\bot$ denoting the truth values "true" and "false", respectively. The symbols in Σ_P of arity 0 are also called propositional letters. We assume the usual inductive definition of first-order ground terms over Σ. A (first-order) atom is either an expression of the form $p(t_1, ..., t_n)$ or an expression of the form $l = r$, where p is an n-ary predicate symbol in Σ_P and $t_1, ..., t_n$ (for $n \geq 0$) are first-order ground terms over Σ. A (first-order) literal is either an atom or its negation. A clause is a finite disjunction of literals (also written as a set). We assume that $Ax(\mathcal{T})$ is a finite set of clauses over $\Sigma_I \cup \mathcal{X}$, where $\mathcal{X}$ is a finite set of variable symbols. We say that

$Ax(\mathcal{T})$ presents the theory $\mathcal{T}$, that the function symbols in Σ_I are interpreted, and that the function symbols in Σ_U are uninterpreted.

The set of *conditional terms* is the smallest set containing the first-order ground terms over Σ and expressions of the form $\mathsf{ite}(\phi, \tau_1, \tau_2)$, where τ_i (for $i = 1, 2$) is a conditional term and ϕ is a conditional formula. A *conditional formula* is either a first-order literal, a boolean combination of conditional formulae, or an expression of the form $\mathsf{ite}(\phi_0, \phi_1, \phi_2)$, where ϕ_i (for $i = 0, 1, 2$) is a conditional formula. We write *conditional expression* to denote either a conditional term or a conditional formula. The first argument of a conditional expression (whose top-most symbol is ite) is called the *guard* of the expression. We assume conditional expressions to satisfy the following three properties.

(Prop.1) ite distributes over operators, i.e. $\diamond(..., \mathsf{ite}(\phi, \eta_1, \eta_2), ...)$ rewrites to $\mathsf{ite}(\phi, \diamond(..., \eta_1, ...), \diamond(..., \eta_2, ...))$, where $\diamond$ is either a boolean connective, the equality symbol $=$, or a symbol in Σ, η_1, η_2 are conditional expressions, and ϕ is a conditional formula.

(Prop.2) $\mathsf{ite}(\phi_0, \phi_1, \phi_2)$ rewrites to $(\phi_0 \Rightarrow \phi_1[\phi_0/\top]) \wedge (\neg\phi_0 \Rightarrow \phi_2[\phi_0/\bot])$ where ϕ_0, ϕ_1, and ϕ_2 are conditional formulae. (We denote by $e[s/s']$ the expression obtained by replacing all occurrences of s in e with s'.) This property is also known as Shannon's decomposition principle (see, e.g. [Gou94]).

(Prop.3) $\mathsf{ite}(\phi, \eta, \eta)$ simplifies to η, where ϕ is a conditional formula and η is a conditional expression.

Our goal is to design an efficient technique to check whether a conditional formula ϕ is a logical consequence of $Ax(\mathcal{T})$ or equivalently, since ϕ is ground, whether $\neg\phi$ is $\mathcal{T}$-unsatisfiable. Variants of this problem have been extensively studied, especially in the context of programming languages (see e.g. [McC63]). If $\mathcal{T}$ is the pure theory of equality, then [Set78] shows that testing the equivalence of conditional terms[2] can be done in polynomial time under the following conditions: (i) conditional expressions are represented as trees and (ii) their guards are of the form $c_1 = c_2$, where c_1, c_2 are constants. Weakening either condition (i) or (ii) yields NP-hard equivalence problem. In almost all real verification efforts, sharing subexpressions is mandatory to handle formulae of reasonable size. Thus, condition (i) cannot be fulfilled.

3.2 Conditional Normal Form

A conditional expression $\mathsf{ite}(\gamma, \phi, \psi)$ is in *(conditional) normal form* when (1) γ contains no ites and it is neither $\top$ nor $\bot$, (2) γ does not occur in ϕ or ψ, (3) ϕ and ψ are not identical, and (4) ϕ and ψ are in normal form. Any conditional formula can be put into conditional normal form, which can be viewed as a decision tree. The decision tree of a conditional formula

[2]The problem of checking the equivalence of two conditional formulae ϕ_1 and ϕ_2 can be reduced to checking the equivalence of conditional terms simply by transforming each occurrence in ϕ_1, ϕ_2 of a non-equational atom $p(t_1, ..., t_n)$ for $p \in \Sigma_P$ of arity n into an equational term $f_p(t_1, ..., t_n) = \mathsf{tt}$, where $f_p \notin \Sigma_F$ is an n-ary function symbol and $\mathsf{tt} \notin \Sigma$ is a constant symbol.

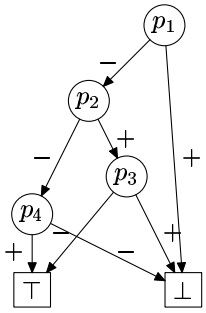


Figure 1: BDD of the formula $p_1 \vee ((p_2 \Rightarrow \neg p_3) \wedge (\neg p_2 \Rightarrow p_4))$, where $p_1 < p_2 < p_3 < p_4$.

ϕ can be constructed by repeatedly applying the Shannon's decomposition principle. For example, the literal $\neg p$ is equivalent to $\mathsf{ite}(p, \bot, \top)$, the formula $p \vee q$ to $\mathsf{ite}(p, \top, q)$, and the term $f(\mathsf{ite}(p, v_0, v_1))$ to $\mathsf{ite}(p, f(v_0), f(v_1))$.

When all atoms are propositional letters, decision trees can be compactly represented by binary decision diagrams (BDDs, for short) [Bry84]. If an ordering $<$ is imposed on the propositional letters and we apply **(Prop.3)** eagerly, then the conditional normal form is also canonical (i.e. unique). In this case, we speak of reduced ordered BDDs and the equivalence of two BDDs can be checked in constant time. An example of BDD is given in Figure 1. The key of the efficiency of BDDs lies in the effective implementation of the Shannon's decomposition principle. To this end, *sharing* and *memoizing* are two crucial ingredients. The former consists of representing expressions by DAGs so that isomorphic subexpressions are stored at the same address in memory and the latter amounts to caching intermediate results to avoid recomputations. All the usual logical operations can be defined by building BDDs bottom-up. For example, negation can be computed in constant time and conjunction in linear time, by using pointers and memoizing. Though the size of a BDD can be exponential in the number of atoms of a formula, experience shows this rarely happens for many interesting verification efforts.

In the more general case of conditional expressions considered here (where terms can contain ites), BDDs cannot be used directly. Furthermore, they no more construct a canonical form. In fact, the activity of building BDDs should be augmented to take into account the background theory $\mathcal{T}$ (see [FG02]). These difficulties can be avoided by considering atoms as propositional letters and by generating a suitable set of proof obligations to take into account $\mathcal{T}$. Our approach is based on these simple observations and the hypothesis that in the richer context considered here (w.r.t. the purely propositional case), BDDs allow for a compact representation of the formulae arising in real verification efforts.

We recall that we take a (ground) conditional formula ϕ as input and we return whether it is a logical consequence of $\mathcal{T}$. Our technique consists of the following three steps.

(Lifting) All the occurrences of conditional terms in ϕ are eliminated. **(Prop.1)** is eagerly applied in order to lift each occurrence of ites at the formula level. This can be done efficiently by using the so-called value numbering algorithm [CS70] to share identical sub-terms and by caching the results of intermediate computations. Let ϕ' be the resulting formula. The atoms in ϕ' are of the form $p(t_1, ..., t_n)$ or $t_1 = t_2$ where $t_1, ..., t_n$ are ground terms not containing ites and p is an n-ary predicate symbol (for $n \geq 0$).

(Boolean Reasoning) The atoms are abstracted to propositional letters so that the standard BDD operations can be used to take into account the boolean structure of the formula. Let $bdd_{\phi'}$ be the BDD of the "abstracted" version of ϕ'.

(Domain Specific Reasoning) A suitable set of proof obligations (namely conjunctions of literals/clauses) are extracted in order to take into account the background theory $\mathcal{T}$. Let $\{\pi_i \mid i = 1, ..., n\}$ (for $n \geq 0$) be the set of such proofs obligations. If each π_i (for $i = 1, ..., n$) is $\mathcal{T}$-unsatisfiable, then ϕ is a logical consequence of $\mathcal{T}$. Notice that if the π_i's are conjunctions of literals and $\mathcal{T}$ is an equational theory presented by the set of clauses $Ax(\mathcal{T})$, then we can check the $\mathcal{T}$-unsatisfiability of π_i by saturating the set $\pi_i \cup Ax(\mathcal{T})$ (for $i = 1, ..., n$), following the approach described in Section 2.

3.3 Case Splitting and $\mathcal{T}$-Satisfiability Checking

We explain how the π_i's are generated by exploiting the structural information encoded in a conditional normal form.

A *branch* π in a conditional normal form (regarded as a decision tree) ϕ is a path from the root of ϕ to either $\bot$ or $\top$. The branch π is identified with the set of literals $\{\lambda_1, \ldots \lambda_n\}$ $(n \geq 1)$ labelling the vertices in π, where each λ_i is either the atom α (if π goes through the conditional normal form $\psi^\top$) or $\neg\alpha$ (if π goes through the conditional normal form $\psi^\bot$) in $\phi[\mathsf{ite}(\alpha, \psi^\top, \psi^\bot)]$.[3] We call a *false* (*true*) *branch*, any branch leading to $\bot$ ($\top$, respectively). Let $\Pi_\bot(\phi)$ ($\Pi_\top(\phi)$) denote the set of false (true, respectively) branches of ϕ (we will also write $\Pi_\bot$ and $\Pi_\top$ when ϕ is clear from the context). For example, the conditional normal form in Figure 1 is such that $\Pi_\bot = \{\{p_1\}, \{\neg p_1, p_2, p_3\}, \{\neg p_1, \neg p_2, \neg p_4\}\}$ and $\Pi_\top = \{\{\neg p_1, \neg p_2, p_4\}, \{\neg p_1, p_2, p_3\}\}$ (where sets of literals are to be read conjunctively whereas sets of sets of literals are to be read disjunctively). The DNF and the CNF of ϕ can be read off from its conditional normal form in linear time [WN01]:

$$\bigvee_{\pi \in \Pi_\top(\phi)} \bigwedge_{\lambda_i \in \pi} \lambda_i \qquad \text{(DNF of } \phi\text{)},$$

$$\bigwedge_{\pi \in \Pi_\bot(\phi)} \bigvee_{\lambda_i \in \pi} \neg\lambda_i \qquad \text{(CNF of } \phi\text{)}.$$

In order to check the $\mathcal{T}$-unsatisfiability of ϕ, we enumerate all the branches $\pi \in \Pi_\top(\phi)$ and we submit $\pi \cup Ax(\mathcal{T})$ to a saturation prover. If the prover derives the empty clause

[3] $e[s]$ denotes that the expression s occurs at a given position in expression e.

for each such proof obligation, then none of the true branches of ϕ is $\mathcal{T}$-satisfiable, and the $\mathcal{T}$-unsatisfiability of ϕ is established. If the prover cannot derive the empty clause for some π', then ϕ is $\mathcal{T}$-satisfiable for the valuation of the atoms corresponding to π', thus establishing a counterexample to the $\mathcal{T}$-unsatisfiability of ϕ.

A *generalized branch* π_v in a conditional normal form ϕ is a path from the root of ϕ to a (not necessarily distinct) vertex v in ϕ. The generalized branch π_v is identified with the set of literals labelling the nodes in the path from the root to v (this last not included) and the CNF of the conditional normal form of the subexpression of ϕ whose root is v. If v is the root of ϕ, then π_v is unique and it is the CNF of ϕ. If v is either $\bot$ or $\top$ then generalized branches degenerate to false or true branches, respectively. Consider again the conditional normal form of Figure 1. The generalized branch leading to the vertex labelled with atom p_2 is $\{\{\neg p_1\}, \{\neg p_2, \neg p_3\}, \{p_2, p_4\}\}$.

We define the *depth* of a conditional normal form to be the maximal length of the true and false branches. To check the $\mathcal{T}$-unsatisfiability of the conditional normal form ϕ whose depth is D, we consider the set $\Pi(\phi, \delta) := \Pi^{=\delta}(\phi) \cup \Pi^{\leq\delta}(\phi)$ of generalized branches for $0 \leq \delta \leq D$, where

$$\begin{array}{lcl} \Pi^{=\delta}(\phi) & := & \{\pi_v \mid v \text{ is a node of } \phi \text{ at depth } \delta\} \\ \Pi^{\leq\delta}(\phi) & := & \{\pi_{v'} \mid v' \text{ is a node of } \phi \text{ at depth } \delta' < \delta \text{ and} \\ & & \quad \text{there exists no } \pi_v \in \Pi^{=\delta}(\phi) \text{ s.t. } v' \text{ is in } \pi_v \ \} \end{array}$$

(notice that $\Pi^{\leq\delta}(\phi)$ does not contain generalized branches and it may be empty, e.g. when $\delta = 0$). We call δ the *splitting depth*. Then, we submit $\pi_v \cup Ax(\mathcal{T})$ (for each $\pi_v \in \Pi(\phi, \delta)$) to a saturation prover which tries to derive the empty clause. It is easy to show that ϕ is logically equivalent to $\bigvee_{\pi_v \in \Pi(\phi,\delta)} \bigwedge_{\eta \in \pi_v} \eta$.

Since the number of paths in a conditional normal form can be exponential in the number of atoms which occurs in it, we can generate exponentially many proof obligations for the prover. By considering generalized paths, the number of proof obligations can be made smaller (possibly one when the node is the root, i.e. the splitting depth is 0). On the other hand, each one of these proof obligations is likely to be more complex than a proof obligation associated to a false or true branch. In fact, generalized paths require to check the $\mathcal{T}$-unsatisfiability of clauses with more than one literal whereas paths only require to check the $\mathcal{T}$-unsatisfiability of literals. Finding the right trade-off between these two aspects requires a thorough experimentation and the development of new heuristics. We report some experiments in Section 4. A remark is in order. The CNF of a formula ϕ obtained from the conditional normal form of ϕ contains "lemmata". To see what this means, consider ϕ to be the formula $p \wedge q$ (where p and q are propositional letters). The "obvious" CNF of ϕ is $\{\{p\}, \{q\}\}$. Now, if we translate ϕ to conditional normal form and then we read off the CNF, we obtain $\{\{p\}, \{\neg p, q\}\}$, where $\neg p$ is the lemma. There is an obvious resolution step between the two clauses so generated. As a consequence, it is not clear that the CNF obtained in this way is "optimal" for saturation theorem provers. In general, it can be proved that there is a close relationship between BDDs and tableaux with lemmata (see [PS95] for details).

The situation is complicated by the fact that some theory $\mathcal{T}$ may require additional case-splitting to establish $\mathcal{T}$-unsatisfiability. Such a theory $\mathcal{T}$ is called *non-convex*. Formally, we say that a formula is *non-convex* if it entails a disjunction of equalities between constants without entailing any of the equalities alone; otherwise, it is *convex*. A theory is *convex* if every conjunction of literals (in the signature of the theory) is convex; otherwise, is is *non-convex*. The pure theory of equality is convex; the theory of arrays $\mathcal{A}$, axiomatized by $\{\mathsf{select}(\mathsf{store}(A,I,E),I) = E, I \neq J \Rightarrow \mathsf{select}(\mathsf{store}(A,I,E),J) = \mathsf{select}(A,J)\}$,[4] is non-convex. In our approach, the saturation prover performs the domain specific case-splitting. For the theory $\mathcal{A}$, we can think of relieving the prover of this burden by transforming the input formula so that each occurrence of $\mathsf{select}(\mathsf{store}(A,I,E),J)$ is replaced by $\mathsf{ite}(I = J, E, \mathsf{select}(A,J))$. After this pre-processing step, only the pure theory of equality is required as the background theory. Since it is convex, no more domain-specific case-splitting is required. Unfortunately, the resulting formula can blow up in size.

3.4 Refinements of our approach

Indeed, the proposed approach is amenable to several optimizations. We briefly hint four of such possible refinements.

First, the activity of **(Lifting)** can be made incremental by introducing "fresh" constants to abstract sub-terms containing ites. This would allow to build small BDDs whose (generalized) branches can be checked for unsatisfiability quickly by the theorem prover. If a given (generalized) branch in the abstracted BDD is found satisfiable, then we are no more entitled to conclude that the input formula is invalid. Rather, we should consider more of the conditional expression abstracted by the constant, until (eventually) the whole expression is exposed. The hope is that—in practice—the unsatisfiability of most branches can be proved by considering only small parts of the conditional expressions.

Second, heuristics to reduce the average length of branches in a conditional normal form should be investigated. Intuitively, this will allow the prover to consider proof obligations of similar form, so that the same strategy for processing clauses can be used. In saturation theorem proving, it is well-known that choosing a suitable order to process the input clauses can dramatically improve the performances. Doing this requires to adapt the existing heuristics to reduce the number of nodes in BDDs as well as to instruct the prover to analyze the structure of the first proof obligation and then to reuse it for the remaining ones.

Third, BDDs use an ordering over atoms to obtain canonical forms. On the other hand, saturation provers use an ordering over terms (which can be lifted to literals and clauses) in order to prune their search space. An interesting question is whether these two orderings can be made compatible so that conditional normal forms are of reasonable size while some equational simplification can be performed. In fact, we can simplify ϕ by the equality $s = t$ in $\mathsf{ite}(s = t, \phi, \psi)$.

Fourth, some simple form of domain specific reasoning can be profitably lifted at the level of propositional reasoning in order to produce smaller conditional normal forms. This

[4] Capitalized letters are implicitly universally quantified variables.

is particularly important to make the approach scale up smoothly. As an example of this, many verification efforts requires to manipulate a finite set of first-order constants $\{c_1, ..., c_n\}$ $(n > 1)$, which are assumed to be pairwise distinct, i.e. $\bigwedge_{1 \leq i \neq j \leq n} c_i \neq c_j$. To manipulate such constants, we only need a finite number of instances of the axioms of reflexivity, symmetry, and transitivity of equality. By using well-known techniques (such as Ackermann's reduction), we can obtain a purely propositional formula Δ whose satisfiability is equivalent to the satisfiability of $\bigwedge_{1 \leq i \neq j \leq n} c_i \neq c_j$ (modulo the theory of equality). Now, we can build the conditional normal form ξ of the formula to be proved unsatisfiable and use the *restrict* operator [HBBM97] on ξ with Δ. The restriction of ϕ to ψ is equivalent to ϕ when ψ is valid, and its BDD is simpler. We expect the resulting BDD to be significantly smaller than the BDD obtained by considering the formula $(\bigwedge_{1 \leq i \neq j \leq n} c_i \neq c_j) \Rightarrow \xi$. Other theories can be treated this way as shown, e.g., in [SSB02].

4 Experiments

Our prototype implementation is as follows. Conditional expressions are represented by DAGs using the well-known value numbering algorithm [CS70] for efficiency. **(Lifting)** is implemented by traversing the DAGs bottom-up. **(Boolean Reasoning)** is implemented by Multi-Terminal BDDs (MTBDDs, for short) [CFM+97], whose nodes are labelled by first-order atoms. Since conditional expressions are represented by value numbers, flattening atoms is simply done by creating a "fresh" constant for each value number. Enumerating (possibly generalized) paths is done by a depth-first traversal of the MTBDD. **(Domain Specific Reasoning)** is implemented by running the E prover [Sch02] on paths unioned to the set of clauses axiomatizing the background theory. The communication between the MTBDD library and the E prover is done by reading and writing files. Indeed, this is quite inefficient for two reasons. First, input/output operations are quite expensive in terms of time. Second (and main reason), the prover repeatedly interns the same set of atoms since paths in MTBDDs are quite "similar". The process of interning is costly in state-of-the-art saturation provers since they feature complicated data structures for efficiency. An obvious improvement is to parse each atom once and to enumerate only whether they occur positively or negatively in the various paths.

The proof obligations of our experiments arise in the activity of symbolically debugging imperative programs (written in Pascal-like). They require to reason in the combination of the theory of arrays $\mathcal{A}$ and the theory of lists $\mathcal{L}$, axiomatized by $\{\mathsf{car}(\mathsf{cons}(X, Y)) = X; \mathsf{cdr}(\mathsf{cons}(X, Y)) = Y\}$. Following standard operational semantics techniques, the binary relation $\mu \rightarrow \mu'$ between two states μ and μ' models the execution of commands. A state is represented by a first-order term using the interpreted function symbols of $\mathcal{A}$. For example, the effect of the assignment $prev := ptr$ can be described by $\mu_0 \rightarrow \mathsf{write}(\mu_0, prev, \mathsf{read}(\mu_0, ptr))$, where μ_0 is an (arbitrary) state before the execution of the command. We write $\mu \rightarrow^n \mu'$ to denote the relation between the state μ before and μ' after the execution of n iterations of a loop. Let $pre[M]$ be a pre-condition and $post[M]$ be a

```
type list_τ : pointer to record
  car: τ;
  cdr: list_τ;
end;

func Search(head: list_τ; v:τ): bool
ptr := head;
while ptr ≠ null ∧ ptr ↑ .car ≠ v do
  ptr := ptr ↑ .cdr;
od;
if ptr = null then
  result := false;
else;
  result := true;
fi;
```

```
func Delete(head: list_τ; v:τ): list_τ
ptr := head;
prev := null;
while ptr ≠ null do
  if ptr ↑ .car = v then
    if prev = null then
      head := ptr ↑ .cdr;
    else;
      prev ↑ .cdr := ptr ↑ .cdr;
    fi;
  else;
    prev := ptr;
  fi;
  ptr := ptr ↑ .cdr;
od;
result := head;
```

Figure 2: Two algorithms manipulating lists.

post-condition about the state M of a program. Our aim is to show that if $\mu \rightarrow^n \mu'$, then $pre[\mu] \Rightarrow post[\mu']$ is valid in any model of $\mathcal{A} \cup \mathcal{L}$.

We consider two common list-manipulating functions: *Search* and *Delete* (see Figure 2). The former looks for a certain element in a list, the latter deletes all occurrences of an element in a list. The type of the values stored in the lists is τ. We assume that τ admits equality over its elements. In the experiments, the pre-condition of both programs is that *head* points to the first element of a singly-linked list of length n. The post-condition of *Search* is that *result* is true if and only if at least one of the elements of the list holds the searched value. The post-condition of *Delete* is that none of the elements of the (now possibly empty) list pointed to by *result* holds the deleted value. Appendix A describes the input language of our tool and gives as example the verification condition of the *Search* algorithm with a single unrolling of the loop.

In Table 1, we report the data gathered to check $pre[\mu] \Rightarrow post[\mu]$ both for *Search* and *Delete*, for an increasing number n of loop iterations.[5] The Table reports measures for three different criteria of choosing generalized branches.

Maximal Depth. In this case, generalized branches reduce to true branches.

Optimal Depth. The set of generalized branches is built by choosing a given depth δ and building the generalized branches identified by all the nodes at depth δ.

[5] All the experiments were done on a 2 GHz Pentium with 1 Gb of RAM running Linux Mandrake 2.4.8. The MTBDD library is the CMU BDD 1.0 and the version of the E prover is 0.62.

Optimal Height. The set of generalized branches is built by choosing a given height χ and building the generalized branches identified by all the nodes at height χ.

The aim of these experiments is to better understand how to case-split on the boolean structure of (ground) formulae so to design automatic heuristics to identify an "optimal" set of generalized branches. Both programs in Figure 2 satisfy the pre- and post-conditions. All proof obligations sent to the prover are proved unsatisfiable so that the negation of the input formula is unsatisfiable. Introducing bugs in the algorithms does not change the behavior of our system. (We notice that each satisfiable proof obligation offers a trace of an execution leading to a bug.)

Discussion. First of all, we observe that most of the execution time of our system is spent in the prover (the time taken to build the BDD is negligible).

The proof obligations for the *Delete* algorithm are more "complex" than those for *Search*. Intuitively, this is so because the formulae for the second algorithm contain more occurrences of interpreted function symbols than the former and grow in size more quickly with n.

At maximal splitting depth, each proof obligation is a set of unit clauses which are efficiently treated by the prover. In this case, the proof obligations are "similar" and the time to check the (un-)satisfiability of each one is almost constant. At different splitting depths, the situation is more complex since the sub-BDDs may vary widely in size and form. Most proof obligations are still extremely simple for the saturation prover, while others need considerable time. Our experiments show that choosing an optimal (splitting) depth can have an important impact on the running time, as we observe speedups of a factor of up to (about) 3. We conjectured that better performances are possible if a more appropriate set of (generalized) branches is submitted to the prover. To evaluate this, we choose to build branches by considering an (optimal) height in the BDD. At different heights, the situation is more uniform than for the splitting depth since the sub-BDDs are quite similar in size and form. Our experiments show that choosing an optimal (splitting) height can have a more important impact on the running time, as we observe speedup factors of up to 9. However, more work awaits us to design optimal heuristics to build generalized branches.

We are also interested to evaluate the rôle of domain specific reasoning on the overall performances of the system. To get some insights on this issue, we have collected the execution time of the prover for each path and have plotted it against its size. The size of a path π is the number of unit literals occurring in the proof obligation generated from π, not yet flattened. The plots for *Search* and *Delete* in a logarithmic scale are shown in Figure 3. Some remarks are in order. First, the execution times may vary widely (factors of up to 1000 for the proof obligations in *Delete* are possible). This fact suggests that the proof obligations for *Delete* are difficult from a domain specific reasoning viewpoint since the literals in them are more complex than those in the formulae in *Search*. This may explain why we obtained better speed-ups for *Search* by working at the propositional level, namely by choosing a suitable heuristics to perform case-splitting. Second, while performing our experiments we have used the automatic mode of the E prover which automatically establish the strategy to process the input clauses. Unfortunately, the strategy selected by the prover

	Size	Maximal depth				Optimal depth					Optimal height				
		depth	g.b.	sz.	time	depth	g.b.	sz.	time	speedup	height	g.b.	sz.	time	speedup
n	*Search*														
1	25/135	8	24	7	0.5	5	8	32.4	0.16	3.1	3	6	34.7	0.10	5.0
2	52/408	13	135	11	2.8	7	16	55.4	0.9	3.1	5	15	56.8	0.31	9.0
3	88/900	18	416	63	9.9	11	111	61.8	3.15	3.1	7	26	84.0	1.3	7.6
4	131/1608	23	875	80	25	14	281	75.6	10	2.5	8	70	91.8	2.9	8.6
5	182/2530	28	1584	98	54	18	727	91.7	18	3.0	10	88	117	6.7	8.1
6	241/3700	33	2597	117	119	21	1026	109	63	1.9	11	159	131	17.6	6.8
n	*Delete*														
1	20/101	7	4	32.2	0.08	7	4	32.2	0.08	1.0	1	4	32.2	0.08	1.0
2	80/671	13	48	54	3.2	9	36	53.6	2.3	1.4	4	16	66.2	1.5	2.1
3	330/4729	20	832	86	307	14	430	88.0	170	1.8	4	312	95.8	202	1.5

Table 1: **Experimental data.** The first column (namely, 'Size') identifies the number n of loop iterations. The second column gives the size of the problem k_1/k_2, where k_1 is the number of MTBDD nodes needed to represent (the negation of) the proof obligation and k_2 is the total number of MTBDD nodes created. The following three columns (namely, 'Maximal depth', 'Optimal depth' and 'Optimal height') are divided in sub-columns. The sub-columns 'depth' in 'Maximal depth' and 'Optimal depth' and 'height' in 'Optimal height' record the maximal splitting depth, the optimal splitting depth, and the optimal splitting height w.r.t. time, respectively. The optimal depth and height have been established by trial and error. The remaining sub-columns report the number (g.n.) of generalized branches (which is equal to the number of proof obligations discharged by the saturation prover), the average number (sz.) of literals clauses per proof obligation, and the running time in seconds. The last sub-column gives the speed-up when running at optimal depth or height.

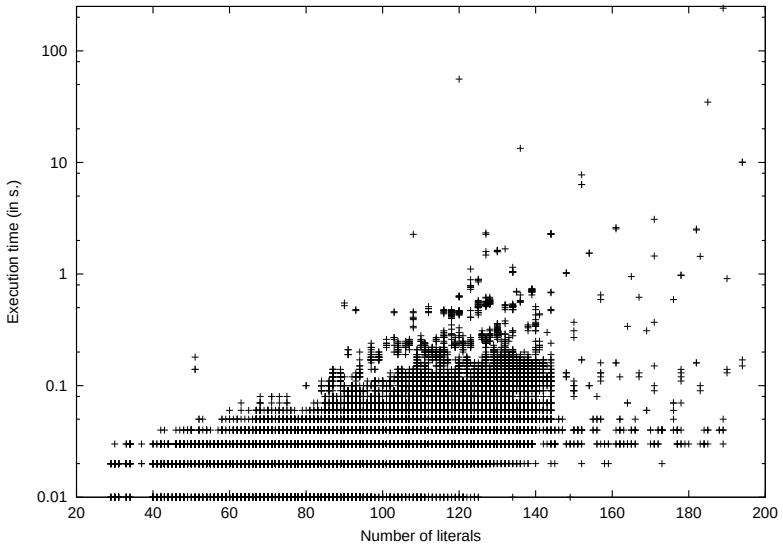
is not always optimal. In fact, by simply changing the order in which the clauses are presented to the prover, we have experienced great variations in the execution times for the same set of input clauses. It is therefore extremely important to understand the causes of these phenomena and try to design more robust strategies to perform the saturations. To sum up, we can say that finding suitable heuristics to effectively generate proof obligations for the domain specific reasoner which are similar in their boolean structure is already quite important. In fact, this can give quite important improvements in performances. However, further improvements can be achieved when considering also the complexity of the literals w.r.t. the background theory. A more thorough investigation of the relationship between the boolean level and the domain specific level is mandatory to design heuristics to speed-up performances. Also, experiments with different theorem provers are required.

5 Related Work

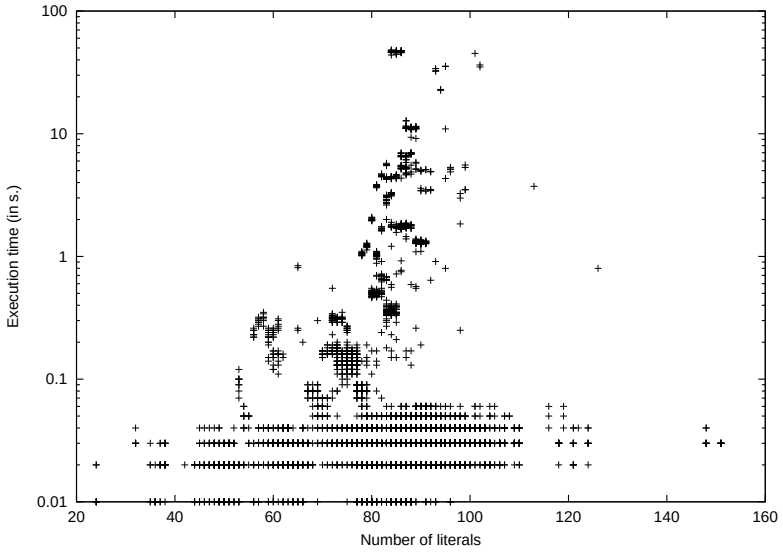
Some attempts have been made to widen the scope of applicability of BDDs to (quantifier-free) first-order logic. In [GSZ+98, GvdP00], the pure theory of equality is combined with BDDs by using Ackermann's finite model property [Ack54]. Our approach is more flexible since we allow many (equational) theories to be combined with BDDs. The work described in [FG02] is the closest in spirit to ours. In fact, satisfiability procedures for (combinations) of theories based on the Nelson and Oppen schema [NO78] are combined with BDDs. The main difference is that our approach handles the ite construct which is not considered in [FG02]. This can be problematic for the kind of applications considered here, since a naive expansion of the ites in terms of the usual boolean connectives can blow up in size.

An alternative stream of research to handle combination of domain specific and propositional reasoning consists of integrating satisfiability procedures with generic propositional satisfiability (SAT) solvers [BDS02, dMR02, ACG00, ABC+02] (see [Tin02] for a rational reconstruction of the combination of the Davis-Putnam procedure and satisfiability procedures for first-order theories). In this approach, atoms are abstracted to propositional constants, the resulting propositional formulae are sent to the SAT solver which enumerates the assignments making them satisfiable. Then, each propositional literal is translated back to a literal in the background theory and their conjunction is checked for (un-)satisfiability by an available procedure for that theory. Indeed, various techniques should be used to make the combination effective.

Another approach to the problem considered in this paper consists of reducing the satisfiability problem for some theories to the pure SAT problem [BV01, SSB02, BLS02]. This approach aims at exploiting the recent advances in state-of-the-art SAT solvers. The key point is to find a propositional encoding of the problem which is then submitted to a SAT solver. If suitable tricks are used to control the size of the resulting propositional formulae, the technique scales up significantly. The weakness of the approach lies in the fact that the reduction process must be designed from scratch each time a new theory is considered. On the contrary, our approach sharply separates the propositional and the domain specific part



(a) *Search*



(b) *Delete*

Figure 3: Proof size vs. time

of the problem allowing for a high degree of flexibility w.r.t. the background (equational) theory.

Existing state-of-art systems for verification incorporates dedicated modules to effectively handle (variants of) the fragment of first-order logic with equality considered here. *Simplify* [DNS96] (the theorem prover of the ESC/Java tool [FLL+02]) adopts the Nelson and Oppen schema [NO79] for domain specific reasoning and the CNF to handle the boolean structure of formulae. It does not support the ite construct; a serious limitation for the kind of formulae considered here. However, *Simplify* features a heuristic mechanism to handle quantified formulae. In our approach, quantified formulae can be factored out and sent to the available refutational theorem prover; the unification mechanism will (possibly) find suitable instantiations for the variables. In this way, we use the prover as a semi-decision procedure for the satisfiability problem of rich background theories (we envisage to investigate this issue in the future). SVC [BDL96] tightly integrates the process of building the conditional normal form of a formula and a combination of decision procedures for domain specific reasoning based on Shostak's schema [Sho84]. PVS [ORS92] loosely incorporates a module to lift ites occurring below function symbols, a module to case-split on the resulting formulae, and a combination of decision procedures based on Shostak's schema. STeP [Bjø98] combines a generalization of the Davis-Putnam algorithm to first-order logic with decision procedures for combination of various theories (based on a variant of Shostak's schema [Sho84]).

6 Conclusion and Future Work

A combination of BDDs and equational satisfiability checking to reason in first-order logic extended with the ite construct and various equational theories has been described. It is based on the simple observations that BDD compactly represent the DNF of a formula and that the satisfiability (w.r.t. the background theory) of each disjunct can be effectively decided by superposition. The technique can be implemented with little effort by re-using state-of-the-art BDD libraries and superposition provers. A prototype implementation showed encouraging results on some verification problems arising in the context of debugging imperative programs.

Several refinements (along the lines of Section 3.4) of the proposed approach await to be investigated. First, reuse and/or adapt existing heuristics to reduce the size of and to balance BDDs in order to minimize the number of proof obligations sent to the prover while reducing their average complexity (specified as the number of literals in each proof obligations). Second, evaluate the impact of simplification mechanisms. We identify two kinds of such mechanisms: (*i*) pure equational simplification (i.e. using equalities to simplify branches) and (*ii*) lifting (part of) the domain specific reasoning at the propositional level by well-known means (such as Ackermann's reduction). Finally, the combined effect of the refinements should be studied.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Korniowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Arithmetic Propositions. In *CADE-18, 18th Intl. Conf. on Automated Deduction, Copenhagen, Denmark*, number 2392 in LNCS. Springer-Verlag, 2002.

[ABR+02] A. Armando, M.P. Bonacina, S. Ranise, M. Rusinowitch, and A. K. Sehgal. High-Performance Deduction for Verification: A Case Study in the Theory of Arrays. In *Proc. of VERIFY'02 (FLoC'02 Affiliated Wokshop)*, 2002.

[ACG00] A. Armando, C. Castellini, and E. Giunchiglia. SAT-Based Procedures for Temporal Reasoning. In *5th European Conference on Planning, ECP'99, Durham, UK, 1999*, number 1809 in LNCS, pages 97–108. Springer-Verlag, 2000.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problems.* North-Holland, 1954.

[ARR02] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, to appear, 2002.

[BDL96] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, November 1996.

[BDS02] C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proc. of the Computer Aided Verification (CAV'02)*, LNCS, 2002.

[Bjø98] N. Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Stanford University, 1998.

[BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proc. of CAV'02, LNCS 2404, Copenhagen, Denmark*, 2002.

[Bry84] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 1984.

[BV01] R. E. Bryant and M. N. Velev. Processor Verification Using Efficient Reductions of the logic of Uninterpreted Functions to Propositional Logic. *ACM Trans. on Computational Logic (TOCL)*, 2(1):93–134, 2001.

[CFM+97] Edmund Clarke, Masahiro Fujita, P. McGeer, Kenneth McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[CS70] J. Cocke and J. T. Schwartz. Programming Languages and their Compilers: Preliminary Notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[dMR02] L. de Moura and H. Ruess. Lemmas on Demand for Satisfiability Solvers. In *Proc. of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002.

[DNS96] D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. Technical report, DEC, 1996.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpression Problem. *J. of the ACM*, 27(4):758–771, October 1980.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, Inc., 1972.

[FG02] Pascal Fontaine and E. Pascal Gribomont. Using bdds with combinations of theories. In *9th Intl. Conf. on Logic for Programming and Automated Reasoning (LPAR'2002)*, 2002.

[FLL+02] C. Flanagan, K. R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. of the 2002 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI), Berlin, Germany*, pages 234–245, 2002.

[Gou94] Jean Goubault. Proving with bdds and control of information. In Alan Bundy, editor, *12th International Conference on Automated Deduction (CADE'12)*, volume 814 of *Lecture Notes in Computer Science*, 1994.

[GSZ+98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD Based Procedures for a Theory of Equality with Uninterpreted Functions. In *Proc. 13th Conf. Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 244–255, 1998.

[GvdP00] J. F. Groote and J. van de Pol. Equational binary decision diagrams. In *7th Intl. Conf. on Logic for Programming and Automated Reasoning (LPAR'2000)*, number 1955 in LNCS, pages 161–178. Springer Verlag, 2000.

[HBBM97] Y. Hong, P. Beerel, Jerry Burch, and Kenneth McMillan. Safe bdd minimization using don't cares. In 34^{th} *Design Automation Conference*, 1997.

[McC63] J. McCarthy. *Computing Programming and Formal Systems*, chapter A Basis of a Mathematical Theory of Computation. North-Holland, 1963.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[NO78] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. Report STAN-CS-78-662, Stanford Artificial Intelligence Laboratory, Computer Science Department, 1978.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning.* 2001.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Proc. CADE-11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[PS95] J. Posegga and P. H. Schmidt. Automated Deduction with Shannon Graphs. *J. of Logic and Computation*, 5(6):697–729, 1995.

[Rus91] M. Rusinowitch. Theorem-proving with Resolution and Superposition. *JSC*, 11(1&2):21–50, January/February 1991.

[Sch02] S. Schulz. E–a brainiac theorem prover. *AI Communications*, 2002.

[Set78] Ravi Sethi. Conditional expressions with equality tests. *Journal of the Association of Computing Machinery*, 25(4):667–674, October 1978.

[Sho84] R. E. Shostak. Deciding Combinations of Theories. *J. of the ACM*, 31(1):1–12, 1984.

[SSB02] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding Separation Formulas with SAT. In *Proc. of CAV'02, LNCS 2404, Copenhagen, Denmark*, 2002.

[Tin02] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. of the 8th European Conference on Logics in Artificial Intelligence, Italy)*, volume 2424 of *LNCS*. Springer, 2002.

[WN01] C. Weidenbach and A. Nonnengart. Small clause normal form. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning.* 2001.

A Input Syntax of the Prototype Implementation

The input syntax of the prototype implementation described in the paper is LISP-like.

Constant, function, and predicate symbols are (finite) sequences of alphanumeric characters starting with a small letter or a digit. Variable symbols are also (finite) sequences of alphanumeric characters starting with a capital letter. (The underscore can also occur in symbol identifiers.)

First order terms and atoms are written in prefix notation. So, for example, the term $f(c_1, c_2)$ can be written as `(f c1 c2)`, the atom $f(c_1, c_2) = g(d_1, d_2, d_3)$ is written as `(= (f c1 c2) (g d1 d2 d3))`, and the atom $p(a)$ is written as `(p a)`, where $c_1, c_2, d_1, d_2, d_3, a$ are constant symbols, f, g are function symbols, $=$ is the equality symbol, and p is a predicate symbol.

The following table gives the correspondence between the usual syntax and the the concrete syntax of the boolean connectives used in our tool.

Logic Syntax	Concrete Syntax
$\neg$	`not`
$\wedge$	`and`
$\vee$	`or`
$\Rightarrow$	`->`
$\Leftrightarrow$	`<->`
if-then-else	`ite`

Notice the presence of the ternary 'if-then-else' connective. Its first argument is a formula, whereas both the second and the third arguments can be either formulae or terms.

Quantifier-free first order formulae are written in prefix notation. So, for example, the formula $\neg(f(c_1, c_2) = g(d_1, d_2, d_3)) \Rightarrow p(a)$ is written as `(-> (not (= (f c1 c2) (g d1 d2 d3))) (p a))` and the formula $f(X, Y) = g(U, V, Z)$ is written as `(= (f X Y) (g U V Z))`, where X, Y, U, V, Z are (implicitly) universally quantified variables.

Recall that our tool is capable of checking whether a ground formula ϕ (i.e. a LISP-like expression not containing symbol identifiers whose first character is a capital letter) is a logical consequence of a theory T, presented by a finite set of equational clauses $Ax(T)$. The input to our tool is a list[6] of (LISP-like) expressions encoding the formulae in $Ax(T)$ and a (LISP-like) expression encoding the formula ϕ. As an example, Figure 4 gives the proof obligation checked by our tool for the first line of Table 1 below *Search*. Finally, Figure 5 gives the grammar (in BNF) of the input syntax of our tool. Below, `THEORY` denotes the set $Ax(T)$ of equational clauses presenting the theory, `QUESTION` the boolean combination of ground literals ϕ, and `TERM` denotes the set of conditional expressions. A final remark is in order. Our tool extends the first-order syntax above with the following two constructs: (i) `#`NUM`=`EXP and (ii) `#`NUM`#`, where NUM is a finite sequence of digits and EXP is either a first-order term or a first-order formula. Intuitively, (i) introduces an abbreviation (namely, NUM) for the first-order term or formula EXP and (ii) abbreviates a first-order

[6] in the sense of LISP

```
;; the set Ax(T), where T := A_s ∪ L ∪ D
;; and D is a theory of pairwise distinct constants
(
 ;; theory of arrays A_s
 (= E (read (write A I E) I))     ;; read(write(A, I, E), I) = E
 (-> (not (= I J))
     (= (read A J)
        (read (write A I E) J))) ;; I ≠ J ⇒ read(write(A, I, E), I) = read(A, J)
 ;; (simple) theory of lists L
 (= (car (cons X Y)) X)           ;; car(cons(X, Y)) = X
 (= (cdr (cons X Y)) Y)           ;; cdr(cons(X, Y)) = Y
 ;; theory of pairwise distinct (first-order) constants D
 (not (= tt ff))
 (not (= head found)) (not (= head ptr))  (not (= found ptr))
 (not (= head in))    (not (= found in))  (not (= ptr in))
 (not (= head i1))    (not (= found i1))  (not (= ptr i1))    (not (= in i1))
)
;; the boolean combination of ground conditional expressions φ
(-> (and (= (read initial head) i1)
         (= (read initial i1) (cons v1 in))
         )
    (-> (and (= m1 (write initial ptr (read initial head)))
             (= m2 (ite (and (not (= (read m1 ptr) in))
                             (not (= v (car (read m1 (read m1 ptr))))))
                        (write m1 ptr (cdr (read m1 (read m1 ptr))))
                        m1))
             (= final
                (write m2 found (ite (not (= (read m2 ptr) in)) tt ff))))
        (and (not (= (read final head) in))
             (<-> (= (read final found) tt)
                  (= (car (read final (read final head))) v)))))
```

Figure 4: Input file for symbolic debugging of one loop iteration of *Search*.

```
PROBLEM::= THEORY QUESTION

THEORY::= ( OPT_LIST_OF_TERMS )

QUESTION::= TERM

TERM::= CONSTANT_SYMBOL
    | VARIABLE_SYMBOL
    | true
    | false
    | ( ite term term term )
    | ( FUNCTION_SYMBOL list_of_terms )
    | ( and list_of_terms )
    | ( or list_of_terms )
    | ( not term )
    | ( = term term )
    | ( -> term term )
    | ( <-> term term )

LIST_OF_TERMS::= TERM
             | LIST_OF_TERMS TERM

OPT_LIST_OF_TERMS::= ϵ
                   | LIST_OF_TERMS
```

Figure 5: Input grammar of our tool.

term or formula defined by means of a construct of type (*i*). Notice that this is only a syntactic device (similar to the `LET` construct of many functional languages but whose scope is the whole formula) to produce more compact proof obligations and it does not extend the expressivity of the fragment of first-order logic considered.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399