A Generic Framework for Implicate Generation Modulo Theories

Mnacho Echenim, Nicolas Peltier, Yanis Sellami

A Generic Framework for Implicate Generation Modulo Theories

Mnacho Echenim, Nicolas Peltier, and Yanis Sellami

Univ. Grenoble Alpes, CNRS, LIG, F-38000 Grenoble France
[Mnacho.Echenim|Nicolas.Peltier|Yanis.Sellami]@univ-grenoble-alpes.fr

Abstract. The clausal logical consequences of a formula are called its implicates. The generation of these implicates has several applications, such as the identification of missing hypotheses in a logical specification. We present a procedure that generates the implicates of a quantifier-free formula modulo a theory. No assumption is made on the considered theory, other than the existence of a decision procedure. The algorithm has been implemented (using the solvers MiniSAT, CVC4 and Z3) and experimental results show evidence of the practical relevance of the proposed approach.

1 Introduction

We present a novel approach based on the usage of a generic SMT solver as a black box to generate ground implicates of a formula modulo a theory. Formally, the implicates of a formula ϕ modulo a theory $\mathcal{T}$ are the ground clauses C such that every model of $\mathcal{T}$ that satisfies ϕ also satisfies C; in other words, these are the *clausal $\mathcal{T}$-consequences of ϕ*. The problem of generating such implicates (up to logical entailment) is of great practical relevance, since for any implicate $\bigvee_{i=1}^{n} l_i$, the formula $\bigwedge_{i=1}^{n} \neg l_i \wedge \phi$ is $\mathcal{T}$-unsatisfiable. The set $\{\neg l_i \mid i \in [1, n]\}$ can thus be viewed as a set of hypotheses under which ϕ is $\mathcal{T}$-unsatisfiable or, dually, $\neg\phi$ is provable. This means that generating implicates can permit to identify missing hypothesis in a theorem, such as omitted lemmata or side conditions. Such hypotheses are useful to correct mistakes in specifications, but also to quickly spot why a given statement is not provable. They can be far more informative than counter-examples in this respect, since the latter are hard to analyze and can be clouded with superfluous information.

Consider for example the simple program over an array defined in Algorithm 1. It turns out that the postcondition of the program is not verified. This can be evidenced by translating the preconditions, the algorithm and the negation of the post-condition into a conjunction of logical formulas, and using an SMT solver to construct a model for this conjunction; this model can then be analyzed to determine what precondition is missing. The obtained model, however, will generally contain a hard to read array definition, and the missing precondition will not be explicitly returned. For instance, the model returned by the Z3 SMT solver [6] is (using our notations):

Algorithm 1: `Example`(Array[Int] T, Int a, Int b)

1 **requires** $\forall x, y \in [a, b], x \leq y \implies T[x] \leq T[y]$;
2 **requires** $T[a] \geq 0$;
3 **let** $T[b+1] = T[b-1] + T[b]$;
4 **ensures** $\forall x, y \in [a, b+1], x \leq y \implies T[x] \leq T[y]$;

$a : 533,\ b : 533,$
$f : x \mapsto x \geq 533\ ?\ (x \geq 534\ ?\ 534 : 533) : 532,$
$g : x \mapsto x = 533\ ?\ 535 : (x = 534\ ?\ 19 : -516),$
$T : x \mapsto g(f(x)).$

Implicate generation on the other hand permits to identify the missing precondition in a more efficient manner. The first step consists in selecting the literals that can be used to generate potential explanations; these are called *abducible literals*. In this example, the natural literals to consider are all the (negations of) equalities and inequalities constructed using constants a and b, along with additional predefined constants such as 0 and 1. The second step simply consists in invoking our system, GPiD, to generate the potential missing preconditions. For this example, GPiD plugged with Z3 generates the missing precondition $a \neq b$ in less than 0.2 seconds. If abducible literals can be constructed using also the function symbol T, then our tool generates the other potential precondition $T[b-1] \geq 0$ in the same amount of time.

In previous work [8,9,10,12], we devised refinements of the superposition calculus specially tuned to derive such implicates for quantifier-free formula modulo equality with uninterpreted function symbols. We proved the soundness and deductive-completeness of the obtained procedures, i.e., we showed that the procedure derives all implicates up to redundancy. In the present work, we investigate a different approach. We provide a generic algorithm for generating such implicates, relying only on the existence of a decision procedure for the underlying theory, possibly augmented with counter-example generation capabilities to further restrict the search space. The main advantage of this approach is that it is possible to use efficient SMT solvers as black boxes, instead of having to develop specific systems for the purpose of implicate generation. Our method is based on decomposition, in the spirit of the DPLL approach. The generated implicates are constructed on a given set of candidate literals, called *abducible* literals, which is assumed to be fixed before the beginning of the search, e.g., by a human user. As far as flexibility is concerned, the algorithm also permits to only generate implicates satisfying so-called $\subseteq$-*closed* predicates without any post-processing step. We show that the algorithm is sound and complete, and we provide experimental results showing that the obtained system is much more efficient than the previous one based on superposition. We also devise generic approaches to store sets of implicates efficiently, while removing implicates that are redundant modulo the considered theory. Again, the proposed procedure relies only on the possibility of deciding validity in the underlying theory.

Related work. The implicate generation problem has been thoroughfully investigated in the context of propositional logic (see for instance [19]). Earlier approaches are based mainly on refinements of the Resolution rule [15,16,26,30], and they focus on the definition of efficient strategies to generate saturated clause sets and of compact data structures for storing the generated sets of implicates [5,14,23,29]. Other approaches use decomposition-based methods, in the style of the DPLL procedure, for generating trie-based representations of sets of prime implicates [20,21]. Recently [25], a new approach that outperforms previous algorithms has been proposed, based on max-satisfiability solving and problem reformulation. Our algorithm can be used for propositional implicate generation but it is not competitive with this new approach. Our aim with this work was rather to extend the scope of implicate generation to more expressive logics. Indeed, there have been only very few approaches dealing with logics other than propositional. Some extensions have been considered in modal logics [3,4], and algorithms have been proposed for first-order formulas, based on first-order resolution [17,18] or tableaux [22,24]. However, none of these approaches is capable of handling equality efficiently. More recently, algorithms were devised to generate sets of implicants of formulas interpreted in decidable theories [7], by combining quantifier-elimination for discarding useless variables, with model building to construct sufficient conditions for satisfiability.

The rest of the paper is structured as follows. In Section 2, basic definitions and notations are introduced. Section 3 contains the definition of the algorithm for generating implicates, starting with a straightforward, naive algorithm and refining it to make it more efficient. In Section 4 data-structures and algorithms are presented to store implicates efficiently modulo redundancy. Section 5 contains the description of the implementation and experimental results, and Section 6 concludes the paper. Due to space restrictions, some of the proofs are omitted. The full version is available on arXiv.

2 Preliminary notions

Ground terms and non-quantified formulas are built inductively as usual on a sorted signature Σ. The notions of validity, models, satisfiability, etc. are defined as usual. The set of literals built on Σ is denoted by $\mathcal{L}$. Let $\mathcal{T}$ be a theory. A set of formulas S is *$\mathcal{T}$-satisfiable* if there exists an interpretation I such that $I \models S$ and $I \models \mathcal{T}$. We assume that the $\mathcal{T}$-satisfiability problem is decidable, i.e., that there exists an SMT solver that, given a formula ϕ with no quantifier, can decide whether ϕ is $\mathcal{T}$-satisfiable.

We consider clauses as unordered disjunctions of literals with no repetition. Thus, when we write $C \vee D$, we implicitly assume that C and D share no literal. We also identify unit clauses with the literal they contain. For every literal l, $\overline{l}$ denotes the literal complementary of l. The empty clause is denoted by `false`. If $Q = \{l_1, \ldots, l_n\}$ is a set of literals, then we denote by $\overline{Q}$ the clause $\overline{l_1} \vee \cdots \vee \overline{l_n}$. Conversely, given a clause $C = l_1 \vee \cdots \vee l_n$, we denote by $\overline{C}$ the set of literals (or unit clauses) $\{\overline{l_1}, \ldots, \overline{l_n}\}$.

We consider a finite set of *abducible literals* $\mathcal{A}$. We assume that each of these literals is $\mathcal{T}$-satisfiable. Given a set of clauses S, we call a clause C a $(\mathcal{T},\mathcal{A})$-*implicate of* S if $\overline{C} \subseteq \mathcal{A}$ and $S \models_{\mathcal{T}} C$. We say that C is a *prime* $(\mathcal{T},\mathcal{A})$-*implicate of* S if C is a $(\mathcal{T},\mathcal{A})$-implicate of S and for every $(\mathcal{T},\mathcal{A})$-implicate D of S, if $D \models_{\mathcal{T}} C$ then $C \models_{\mathcal{T}} D$. The set of $(\mathcal{T},\mathcal{A})$-implicates of S is denoted by $\mathfrak{I}_{\mathcal{A}}(S)$, and the set of prime $(\mathcal{T},\mathcal{A})$-implicates of S is denoted by $\mathfrak{P}_{\mathcal{A}}(S)$.

Given a set of clauses S and a clause C, we write $S \trianglelefteq_{\mathcal{T}} C$ if there is a clause $D \in S$ such that $D \models_{\mathcal{T}} C$. If S' is a set of clauses, then we write $S \trianglelefteq_{\mathcal{T}} S'$ if for all $C \in S'$, we have $S \trianglelefteq_{\mathcal{T}} C$. We write $S \sim S'$ if $S \trianglelefteq_{\mathcal{T}} S'$ and $S' \trianglelefteq_{\mathcal{T}} S$ (i.e., S and S' are identical modulo $\mathcal{T}$-equivalence).

Proposition 1. *Let l be a literal and let C, D be clauses. The following statements hold:*

1. $l \vee C \models_{\mathcal{T}} D$ *iff* $l \models_{\mathcal{T}} D$ *and* $C \models_{\mathcal{T}} D$.
2. $C \models_{\mathcal{T}} l \vee D$ *iff* $C \wedge \bar{l} \models_{\mathcal{T}} D$.

We assume an order $\prec$ is given on clauses built on $\mathcal{A}$ that agrees with inclusion, i.e., such that $C \subsetneq D \Rightarrow C \prec D$.

Definition 2. *A* $\mathcal{T}$-tautology *is a clause that is satisfied by every model of $\mathcal{T}$. Given a set of clauses S, we denote by* $\mathrm{SubMin}(S)$ *the set obtained by deleting from S all clauses D such that either D is a $\mathcal{T}$-tautology, or there exists $C \in S$ such that $C \models_{\mathcal{T}} D$ and ($D \not\models_{\mathcal{T}} C$ or $C \prec D$).*

Note that in particular, we have $\mathfrak{P}_{\mathcal{A}}(S) \sim \mathrm{SubMin}(\mathfrak{I}_{\mathcal{A}}(S))$.

3 On the generation of prime $(\mathcal{T},\mathcal{A})$-implicates

3.1 A basic algorithm

We present a simple and intuitive algorithm that permits to generate the $(\mathcal{T},\mathcal{A})$-implicates of a set of formulas S. This algorithm is based on the fact that a clause C is a $(\mathcal{T},\mathcal{A})$-implicate of S if and only if $\overline{C} \subseteq \mathcal{A}$ and $S \cup \overline{C} \models_{\mathcal{T}} \texttt{false}$. It will thus basically consist in enumerating the subsets of $\mathcal{A}$ and searching for those whose union with S is $\mathcal{T}$-unsatisfiable. This may be done by starting with an empty set of hypotheses M and repeatedly and nondeterministically adding new abductible literals to M until $S \cup M$ is $\mathcal{T}$-unsatisfiable. This algorithm is naive, as the same clauses will be produced multiple times, but it forms the basis of the more efficient algorithm in Section 3.2.

Definition 3. *Let S be a set of formulas. Let M, A be sets of literals such that $M \cup A \subseteq \mathcal{A}$. We define*

$$\mathfrak{I}_{M,A}(S) = \left\{ C \in \mathfrak{I}_{\mathcal{A}}(S) \;\middle|\; \exists Q \subseteq A,\, C = \overline{M} \vee \overline{Q} \right\},$$
$$\mathfrak{P}_{M,A}(S) = \mathrm{SubMin}(\mathfrak{I}_{M,A}(S)).$$

Intuitively, a clause $\overline{M} \vee \overline{Q}$ thus belongs to $\mathfrak{I}_{M,A}(S)$ if and only if $\overline{Q}$ is a $(\mathcal{T}, \mathcal{A})$-implicate of $S \cup M$.

Proposition 4. *Let S be a set of formulas, M and A be sets of literals such that $M \cup A \subseteq \mathcal{A}$. If M is $\mathcal{T}$-satisfiable, then $S \cup M$ is $\mathcal{T}$-unsatisfiable iff $\mathfrak{P}_{M,A}(S) = \{\overline{M}\}$. If M is $\mathcal{T}$-unsatisfiable, then $\mathfrak{P}_{M,A}(S) = \emptyset$.*

It is clear that it is useless to add a new hypothesis l into M both if $M \cup \{l\}$ is $\mathcal{T}$-unsatisfiable (because the obtained $(\mathcal{T}, \mathcal{A})$-implicate would be a $\mathcal{T}$-tautology), or if this set is equivalent to M (because the $(\mathcal{T}, \mathcal{A})$-implicate would not be minimal). This motivates the following definition:

Definition 5. *Let S be a set of formulas and let M, A be two sets of literals. We denote by $\texttt{fix}(S, M, A)$ a set obtained by deleting from A some literals l such that either $M \cup S \models_{\mathcal{T}} l$ or $M \models_{\mathcal{T}} \bar{l}$.*

The use of this definition aims to reduce the number of abducible hypotheses to try, and thus the search space of the algorithm. Still, we do not assume that all the literals l satisfying the condition above are deleted because, in practice, such literals may be hard to detect. However, we assume that no element from M is in $\texttt{fix}(S, M, A)$.

Proposition 6. *Consider a set of formulas S and two sets of literals M, A such that $\mathfrak{P}_{M,A}(S) \neq \{\overline{M}\}$. The following equalities hold:*

1. $\mathfrak{P}_{M,A}(S) = \text{SubMin}(\bigcup_{l \in A} \mathfrak{P}_{M \cup \{l\},A}(S))$.
2. $\mathfrak{P}_{M,A}(S) = \mathfrak{P}_{M,\texttt{fix}(S,M,A)}(S)$.

The results above lead to a basic algorithm for generating $(\mathcal{T}, \mathcal{A})$-implicates which is described in Algorithm 2. As explained above, the algorithm works by adding literals from A as hypotheses until a contradiction can be derived. The **return** statement at Line 5 avoids enumerating the subsets that contain M, once it is known that $S \cup M$ is $\mathcal{T}$-unsatisfiable.

Algorithm 2: $\text{BP}(S, M, A)$

```
1  if M is T-unsatisfiable then
2      return ∅;
3  else
4      if S ∪ M is T-unsatisfiable then
5          return {M̄};
6      else
7          B = fix(S, M, A);
8          foreach l ∈ B do
9              let P_l = BP(S, M ∪ {l}, B);
10         return SubMin(⋃_{l∈B} P_l);
```

Lemma 7. $\mathfrak{P}_{M,A}(S) = \text{BP}(S, M, A)$.

Proof. The result is proved by a straightforward induction on $\text{card}(\mathcal{A} \setminus M)$. By Proposition 4, $\mathfrak{P}_{M,A}(S) = \emptyset$ if M is $\mathcal{T}$-unsatisfiable, and $\mathfrak{P}_{M,A}(S) = \{\overline{M}\}$ if M is $\mathcal{T}$-satisfiable and $S \cup M$ is $\mathcal{T}$-unsatisfiable. Otherwise, by Proposition 6(2), we have $\mathfrak{P}_{M,A}(S) = \mathfrak{P}_{M,\texttt{fix}(S,M,A)}(S) = \mathfrak{P}_{M,B}(S)$. By the induction hypothesis, for each $l \in A$, $P_l = \mathfrak{P}_{M\cup\{l\},B}(S)$, and by Proposition 6(1), $P_l = \mathfrak{P}_{M\cup\{l\},A}(S)$; we deduce that $\mathfrak{P}_{M,A}(S) = \text{SubMin}(\bigcup_{l\in A} P_l)$. Note that at each recursive call, a new element is added to M, since $\texttt{fix}(S, M, A)$ is assumed not to contain any element from M.

Theorem 8. *If S is a set of formulas then $\mathfrak{P}_{\mathcal{A}}(S) = \text{BP}(S, \emptyset, \mathcal{A})$.*

Although the algorithm described above computes all the prime $(\mathcal{T}, \mathcal{A})$-implicates of any clause set as required, it is very inefficient, in particular because of the large number of useless and redundant recursive calls that are made. In what follows we present several improvements to the algorithm in order to generate implicates as efficiently as possible.

3.2 Restricting the set of candidate hypotheses

It is obvious that the algorithm BP makes a lot of redundant calls: for example, if $l_1 \vee l_2$ is a prime $(\mathcal{T}, \mathcal{A})$-implicate of a clause set S, then this $(\mathcal{T}, \mathcal{A})$-implicate will be generated twice, first as $l_1 \vee l_2$, and then as $l_2 \vee l_1$. Such redundant calls are quite straightforward to avoid by ensuring that every invocation of the algorithm contains a distinct set of literals M. This can be done by fixing an ordering $<$ among literals in $\mathcal{A}$, and by assuming that hypotheses are always added in this order. Another way of restricting the set of candidate hypotheses is to exploit information extracted from the previous satisfiability test. For example, if $S \cup \{l_1\}$ is satisfiable for some literal l_1, and that a model of this set satisfies another literal l_2, then $S \cup \{l_2\}$ is also satisfiable and it is not necessary to consider l_2 as a hypothesis. In particular, if a model of $S \cup \{l_1\}$ validates all the literals in A, then $\mathfrak{P}_{M,A}(S)$ is necessarily empty and no literal should be selected. We can thus take advantage of the existence of a model of $S \cup M$ in order to guide the choice of the next literals in A. However, observe that this refinement interferes with the previous one based on the order $<$. Indeed, non-minimal hypotheses will have to be considered if all the smaller hypotheses are dismissed because they are true in the model. We formalize these principles below.

Definition 9. *In what follows, we consider a total ordering*[1] $<$ *on the elements of $\mathcal{A}$. For $A \subseteq \mathcal{A}$ and $l \in A$, we define $A[l] \stackrel{def}{=} \{l' \in A \mid l < l'\}$. If I is a set of literals then we denote by $A^I[l]$ the set $\left\{l' \in A \,\middle|\, l' < l \wedge \overline{l'} \notin I\right\} \cup A[l]$.*

Example 10. Assume that $A = \{p_i, \neg p_i \mid i = 1, \ldots, 6\}$ and that for all literals $l \in \{p_i, \neg p_i\}$ and $l' \in \{p_j, \neg p_j\}$, $l < l'$ if and only if either $i < j$ or $(i = j$, $l = p_i$ and $l' = \neg p_i)$. Then $A[p_4] = \{\neg p_4, p_5, \neg p_5, p_6, \neg p_6\}$. If $I = \{p_1, \neg p_2\}$, then $A^I[p_4] = \{p_1, \neg p_2, p_3, \neg p_3, \neg p_4, p_5, \neg p_5, p_6, \neg p_6\}$.

[1] Note that this ordering is not necessarily related to the ordering $\prec$ on clauses.

Definition 11. *Let S be a set of clauses. A set of literals I is* S-compatible with respect to $\mathcal{A}$ *(or simply* S-compatible*) if every prime $(\mathcal{T},\mathcal{A})$-implicate of S contains a literal l such that $l \in I$.*

Intuitively, an S-compatible set I consists of literals l such that $\bar{l}$ will be allowed to be added as a hypothesis to generate $(\mathcal{T},\mathcal{A})$-implicates of S (see Lemma 16 below). The set I can always be defined by taking the negations of all the abducible literals from $\mathcal{A}$. In this case, all literals will remain possible hypotheses. It is possible, however, to restrict the size of I when a model of S is known, as evidenced by the following proposition:

Proposition 12. *If S is a set of clauses and J is a model of S, then the set $I \stackrel{def}{=} \{l \in \mathcal{L} \mid J \models l\}$ is S-compatible.*

Proof. Let Q be a set of literals such that $\overline{Q}$ is a prime $(\mathcal{T},\mathcal{A})$-implicate of S, and assume that for all $l \in Q$, $\bar{l} \notin I$, i.e., that for all $l \in Q$, $J \not\models \bar{l}$. Then $J \models l$ holds for every $l \in Q$, hence $J \models S \cup Q$ and $\overline{Q}$ cannot be a $(\mathcal{T},\mathcal{A})$-implicate of S.

Note that the condition of having a model of S was not added to Definition 11 because in practice, such a model cannot always be constructed efficiently.

Being able to derive unit consequences of the set of axioms (for instance by using unit propagation), can pay off if this additional information can be used to simplify the formula at hand. This motivates the following definition.

Definition 13. *Let S be a set of formulas and $M \subseteq \mathcal{A}$. We denote by $\mathrm{U}_M(S)$ the set of unit clauses logically entailed by $S \cup M$ modulo $\mathcal{T}$, i.e., $\mathrm{U}_M(S) \stackrel{def}{=} \{l \in \mathcal{L} \mid S \cup M \models_{\mathcal{T}} l\}$. Given a set U such that $M \subseteq U \subseteq \mathrm{U}_M(S)$, we denote by $S_{U,M}$ the formula obtained from S by replacing* some *(arbitrarily chosen) literals l' by* `false` *if $U \models_{\mathcal{T}} \overline{l'}$ and by* `true` *if $M \models_{\mathcal{T}} l'$.*

Note that U is not necessarily identical to $\mathrm{U}_M(S)$, because in practice the latter set is hard to generate. Similarly we do not assume that all literals l' are replaced in Definition 13 since testing logical entailment may be costly. Lemma 14 shows that the $(\mathcal{T},\mathcal{A})$-implicates of a set S and those of $S_{U,M}$ are identical.

Lemma 14. *Let S be a set of formulas and $M \subseteq \mathcal{A}$. Consider a set of literals U such that $M \subseteq U \subseteq \mathrm{U}_M(S)$. Then $\mathfrak{I}_{M,A}(S) = \mathfrak{I}_{M,A}(S_{U,M})$*

Definition 15. *Let U, M, A be sets of literals. We define:* $\mathrm{G}_{U,A,M}(S) \stackrel{def}{=} \left\{\overline{M} \vee \bar{l} \;\middle|\; l \in A \wedge \bar{l} \in U\right\}$.

The lemma below can be viewed as a refinement of Proposition 6. It is based on the previous results, according to which, when adding a new hypothesis l, it is possible to remove from the set of abducible literals A every literal that is strictly smaller than l, provided its complementary is in I (because we can always assume that the smallest available hypothesis is considered first). This is why the recursive call is on $A^I[l]$ instead of A. Note also that the use of semantic guidance interferes with the use of the ordering $<$: the smaller the set I, the larger $A^I[l]$.

Lemma 16. *Assume that $S \cup M$ is $\mathcal{T}$-satisfiable and let I be an $(S \cup M)$-compatible set of literals. Let U be a set of literals such that $M \subseteq U \subseteq \mathrm{U}_M(S)$. We have*

$$\mathfrak{P}_{M,A}(S) \;=\; \mathrm{SubMin}\left(\mathrm{G}_{U,A,M}(S) \cup \bigcup_{l \in A, \bar{l} \in I} \mathfrak{P}_{M \cup \{l\}, A^I[l]}(S)\right).$$

Proof. First note that $\mathfrak{P}_{M,A}(S) \neq \{\overline{M}\}$, since $S \cup M$ is $\mathcal{T}$-satisfiable. We first prove that $\mathfrak{P}_{M,A}(S) \subseteq \mathrm{G}_{U,A,M}(S) \cup \bigcup_{l \in A, \bar{l} \in I} \mathfrak{P}_{M \cup \{l\}, A^I[l]}(S)$. Let $C \in \mathfrak{P}_{M,A}(S)$. By hypothesis, C is of the form $\overline{M} \vee \overline{Q}$, where $\emptyset \neq Q \subseteq A$. Since I is $(S \cup M)$-compatible, Q necessarily contains a literal $l \in A$ such that $\bar{l} \in I$. Assume that l is the smallest literal in Q satisfying this property. We distinguish the following cases.

- Assume that Q contains a literal l' such that $\overline{l'} \in U$. In this case, since $U \subseteq \mathrm{U}_M(S)$, $S \cup M \models_{\mathcal{T}} \overline{l'}$. Since $Q \subseteq A$, we also have $l' \in A$, and since $\mathfrak{P}_{M,A}(S) \neq \{\overline{M}\}$, we deduce that $\overline{M} \vee \overline{l'} \in \mathfrak{P}_{M,A}(S)$. Since $\overline{M} \vee \overline{l'} \models_{\mathcal{T}} C$ and $C \in \mathfrak{P}_{M,A}(S)$, C must be smaller or equal to $\overline{M} \vee \overline{l'}$, which is possible only if $C = \overline{M} \vee \overline{l'}$. We deduce that $C \in \mathrm{G}_{U,A,M}(S)$.
- Otherwise, we show that $Q \setminus \{l\} \subseteq A^I[l]$. By Definition 9, we have $A[l] = \{l' \in A \mid l < l'\}$ and $A^I[l] = \{l' \in A \mid l' < l \wedge l' \notin I\} \cup A[l]$. Let $l' \in Q$, with $l' \neq l$. If $l' > l$ then $l' \in A[l] \subseteq A^I[l]$. If $l' \not> l$, then since $>$ is total and $l \neq l'$, necessarily $l > l'$. Since l is the smallest literal in Q such that $\bar{l} \in I$, we deduce that $\overline{l'} \notin I$. Thus $l' < l$ and $\overline{l'} \notin I$, which entails that $l' \in A^I[l]$. Consequently, $Q \setminus \{l\} \subseteq A^I[l]$. Since $C = \overline{M \cup \{l\}} \vee \overline{Q \setminus \{l\}}$, this entails that $C \in \mathfrak{P}_{M \cup \{l\}, A^I[l]}(S)$.

We now prove that $\mathrm{G}_{U,A,M}(S) \cup \bigcup_{l \in B, \bar{l} \in I} \mathfrak{P}_{M \cup \{l\}, B^I[l]}(S) \subseteq \mathfrak{I}_{M,A}(S)$.

- Let $C \in \mathrm{G}_{U,A,M}(S)$. By definition, C is of the form $\overline{M} \cup l$ with $l \in \overline{A} \cap U$. Since $U \subseteq \mathrm{U}_M(S)$, we deduce that $S \cup M \models_{\mathcal{T}} l$, i.e., that $S \models_{\mathcal{T}} \overline{M} \vee l$. Since $\bar{l} \in A$, this entails that $\overline{M} \vee l \in \mathfrak{I}_{M,A}(S)$, hence $C \in \mathfrak{I}_{M,A}(S)$.
- Let $C \in \mathfrak{P}_{M \cup \{l\}, A^I[l]}(S)$ with $l \in A$, $\bar{l} \in I$. By definition, $C = \overline{M} \vee \bar{l} \vee \overline{Q}$, with $Q \subseteq A^I[l]$ and $C \in \mathfrak{I}_{\mathcal{A}}(S)$. But $A^I[l] \subseteq A$ by definition, thus $Q \cup \{l\} \subseteq A$ and $C = M \vee (\overline{Q} \vee \bar{l}) \in \mathfrak{I}_{M,A}(S)$.

Similarly to CSP (see [12, Sect. 4.2]), we parameterize our algorithm by a predicate in order to filter the implicates that are generated. The goal of this parametrization is to allow the user to restrict the form of the generated implicates. Typically, one could want to generate implicates only up to a given size limit, or only those satisfying some specific semantic constraints.

Definition 17. *A predicate $\mathcal{P}$ on sets of literals is* $\subseteq$-closed *if for all sets of literals A such that $\mathcal{P}(A)$ holds, if $B \subseteq A$ then $\mathcal{P}(B)$ also holds.*

Examples of $\subseteq$-closed predicates include cardinality constraints: $\mathcal{P}_k \stackrel{\text{def}}{=} \lambda A.\ \mathrm{card}(A) \leq k$, where $k \in \mathbb{N}$, or implicant constraints: $\mathcal{P}_\phi \stackrel{\text{def}}{=} \lambda A.\ \phi \models A$,

where ϕ is a formula. Note that $\subseteq$-closed predicates can safely be combined by the conjunction and disjunction operators.

An important feature of $\subseteq$-closed predicates is that implicates verifying such predicates can be generated on the fly without any post-processing step, thanks to the following result:

Proposition 18. *If $\mathcal{P}$ is $\subseteq$-closed and $\mathcal{P}(M)$ does not hold, then for all sets of literals A, $\mathcal{P}(M \cup A)$ does not hold either.*

The inclusion of these improvements to the original algorithm results in the one described in Algorithm 3.

Algorithm 3: $\text{IMP}(S, M, A, \mathcal{P})$

```
1  if M is T-unsatisfiable or ¬P(M) then
2      return ∅;
3  if S ∪ M is T-unsatisfiable then
4      return {M̄};
5  let U ⊆ U_M(S) such that M ⊆ U ;
6  let S = S_{U,M} ;
7  let A = fix(S, M, A) ;
8  let I be an (S ∪ M)-compatible set of literals ;
9  foreach l ∈ A such that l̄ ∈ I do
10     let P_l = IMP(S, M ∪ {l}, A^I[l], P);
11 return SubMin(G_{U,A,M}(S) ∪ ⋃_{l∈A} P_l);
```

Lemma 19. *If $\mathcal{P}$ is $\subseteq$-closed then* $\text{IMP}(S, M, A, \mathcal{P}) = \mathfrak{P}_{M,A}(S) \cap \left\{\overline{A} \,\middle|\, A \in \mathcal{P}\right\}$.

Proof. If one of M or $S \cup M$ is $\mathcal{T}$-unsatisfiable, or $\mathcal{P}(M)$ does not hold, then the result follows from Propositions 4 and 18. Otherwise the result is proved by induction on $\text{card}(\mathcal{A} \setminus M)$, using Proposition 6 and Lemmata 16 and 14.

Theorem 20. *If $\mathcal{P}$ is $\subseteq$-closed then* $\mathfrak{P}_{\mathcal{A}}(S) \cap \left\{\overline{A} \,\middle|\, A \in \mathcal{P}\right\} = \text{IMP}(S, \emptyset, \mathcal{A}, \mathcal{P})$.

4 On the storage of $(\mathcal{T}, \mathcal{A})$-implicates

The number of implicates of a given formula may be huge, hence it is essential in practice to have appropriate data structures to store them in a compact way and efficient algorithms to check that a newly generated implicate C is not redundant (forward subsumption modulo $\mathcal{T}$), and if so, to delete all the already generated implicates that are less general than C (backward subsumption modulo $\mathcal{T}$), before inserting C into the stored implicates. In this section, we devise a trie-like data-structure to perform these tasks. As in the previous section, we only rely on the existence of a decision procedure for testing $\mathcal{T}$-satisfiability.

Definition 21. *Let $<_t$ be an order on the literals in $\mathcal{A}$, possibly, but not necessarily, equal to the order $<$ used for literal ordering in the implicate generation algorithm. An $\mathcal{A}$-*tree *is inductively defined as $\bot$ or a possibly empty set of pairs $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, where $l_1, \ldots, l_n$ are pairwise distinct literals in $\mathcal{A}$ and τ_i (for $i = 1, \ldots, n$) is an $\mathcal{A}$-tree only containing literals that are strictly $<_t$-greater than l_i. An $\mathcal{A}$-tree is associated with a set of $\mathcal{A}$-clauses inductively defined as follows:*

$$\begin{array}{rcl} \mathcal{S}(\bot) & \stackrel{def}{=} & \{\mathtt{false}\}, \\ \mathcal{S}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) & \stackrel{def}{=} & \bigcup_{i=1}^{n} \{l_i \vee C \mid C \in \mathcal{S}(\tau_i)\}. \end{array}$$

In particular, $\mathcal{S}(\emptyset) = \emptyset$. Intuitively an $\mathcal{A}$-tree may be seen as a tree in which the edges are labeled by literals and the leaves are labeled by $\emptyset$ or $\bot$, and represents a set of clauses corresponding to paths from the root to $\bot$. We introduce the following simplification rule (which may be applied at any depth inside a tree, not only at the root level):

$$\mathtt{Simp} : \tau \cup \{l : \emptyset\} \rightarrow \tau$$

Informally, the rule deletes all leaves labeled by $\emptyset$ except for the root. It may be applied recursively, for instance $\{l : \{l_1 : \emptyset, \ldots, l_n : \emptyset\}\} \rightarrow_{\mathtt{Simp}}^{n+1} \emptyset$. Termination is immediate since the size of the tree is strictly decreasing.

Proposition 22. *If $\tau \rightarrow_{\mathtt{Simp}} \tau'$ then τ' is an $\mathcal{A}$-tree and $\mathcal{S}(\tau) = \mathcal{S}(\tau')$.*

The algorithm permitting the insertion of a clause in an $\mathcal{A}$-tree is straightforward and thus omitted. The following lemma provides a simple algorithm to check whether a clause is a logical consequence modulo $\mathcal{T}$ of some clause in $\mathcal{S}(\tau)$ (forward subsumption). The algorithm proceeds by induction on the $\mathcal{A}$-tree.

Lemma 23. *Let C be a clause and let τ be an $\mathcal{A}$-tree. We have $\mathcal{S}(\tau) \trianglelefteq_{\mathcal{T}} C$ iff one of the following conditions hold:*

- $\tau = \bot$.
- $\tau = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ *and there exists $i \in [1, n]$ such that $l_i \models_{\mathcal{T}} C$ and $\mathcal{S}(\tau_i) \trianglelefteq_{\mathcal{T}} C$.*

Proof. If $\tau = \bot$ then $\mathcal{S}(\tau) = \{\mathtt{false}\} \trianglelefteq_{\mathcal{T}} C$ hence the equivalence holds. Otherwise, let $\tau = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. By definition, $\mathcal{S}(\tau) \trianglelefteq_{\mathcal{T}} C$ holds iff there exists a clause $D \in \mathcal{S}(\tau)$ such that $D \models_{\mathcal{T}} C$. Since $\mathcal{S}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) = \bigcup_{i=1}^{n} \{l_i \vee E \mid E \in \mathcal{S}(\tau_i)\}$, the previously property holds iff there exists $i \in [1, n]$ and $E \in \mathcal{S}(\tau_i)$ such that $l_i \vee E \models_{\mathcal{T}} C$, i.e., such that $l_i \models_{\mathcal{T}} C$ and $E \models_{\mathcal{T}} C$ by Proposition 1(1). By definition, $\exists E\, (E \models_{\mathcal{T}} C \wedge E \in \mathcal{S}(\tau_i))$ iff $(\mathcal{S}(\tau_i) \trianglelefteq_{\mathcal{T}} C)$. Furthermore, $l_i \trianglelefteq_{\mathcal{T}} C$ holds iff $\overline{C} \cup \{l_i\}$ is $\mathcal{T}$-unsatisfiable, hence the result.

The following definition provides an algorithm to remove, in a given $\mathcal{A}$-tree, all branches corresponding to clauses that are logical consequences of a given formula modulo $\mathcal{T}$ (backward subsumption).

Definition 24. *Let ϕ be a formula and let τ be an $\mathcal{A}$-tree. $\mathtt{rm}(\tau, \phi)$ denotes the $\mathcal{A}$-tree defined as follows:*

– *If ϕ is $\mathcal{T}$-unsatisfiable, then* $\texttt{rm}(\tau, \phi) \stackrel{def}{=} \emptyset$.
– *If ϕ is $\mathcal{T}$-satisfiable, then:*
  • $\texttt{rm}(\bot, \phi) \stackrel{def}{=} \bot$,
  • $\texttt{rm}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}, \phi) \stackrel{def}{=} \bigcup_{i=1}^{n} \{l_i : \texttt{rm}(\tau_i, \phi \wedge \overline{l_i})\}$.

Intuitively, starting with some clause C, the algorithm incrementally adds literals $\overline{l_1}, \ldots, \overline{l_n}$ occurring in the clauses $D = l_1 \vee \cdots \vee l_n \in \mathcal{S}(\tau)$ and invokes the SMT solver after each addition. If a contradiction is found then this means that $C \models_{\mathcal{T}} D$, hence the branch corresponding to D can be removed. The calls are shared among all common prefixes. Of course, this algorithm is interesting mainly if the SMT solver is able to perform incremental satisfiability testing, with "push" and "pop" commands to add and remove formulas from the set of axioms (which is usually the case).

Lemma 25. *Let ϕ be a formula and let τ be an $\mathcal{A}$-tree. Then* $\texttt{rm}(\tau, \phi)$ *is an $\mathcal{A}$-tree, and* $\mathcal{S}(\texttt{rm}(\tau, \phi)) = \{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\}$.

Remark 26. The $\mathcal{A}$-trees may be represented as dags instead of trees. In this case, it is clear that the complexity, defined as the number of satisfiability tests of forward subsumption (as defined in Lemma 23) is of the same order as the size of the dag, since the recursive calls only depend on the considered subtree. For backward subsumption (see Definition 24) the situation is different since the recursive calls have an additional parameter that is the formula ϕ, which depends on the path in the $\mathcal{A}$-tree. The maximal number of satisfiability tests is therefore equal to the size of underlying tree, and not that of the dag. Note that it would be necessary to make copies of some of the subtrees, if two pruning operations are applied on the same (shared) subtree with different formulas.

5 Experimental evaluation

Algorithm 3 has been implemented in a C++ framework called GPiD. The SMT solver is used as a black box and GPiD can thus be plugged with any tool serving this purpose, provided an interface is written for it. As a consequence, the handled theory is only restricted by the SMT solver. Three interfaces were implemented, respectively for MiniSAT [13], CVC4 [1] and Z3 [6]. The implicate generator used in the reported experiments is the one based on Z3, which turned out to be more efficient on the considered benchmarks. All the tests were run on one core of an Intel(R) Core(TM) i5-4250U machine running at 1,9 GHz with 1 GiB of RAM. The benchmarks are extracted from the SMTLib [2] library, the considered theories are quantifier-free uninterpreted functions (QF_UF) and quantifier-free linear integer arithmetic with uninterpreted functions (QF_UFLIA). For obvious reasons, only satisfiable examples have been kept for analysis. Abducible literals are part of the problem input, they are generated by considering all ground equalities and disequalities with a maximal depth provided by the user; all the experiments were conducted using a maximal depth of 1 and the average number of abducible literals is around 13397 (min.

Table 1: Number of problems for which at least one $(\mathcal{T}, \mathcal{A})$-implicate of a given maximal size can be generated in a given amount of time (in seconds), for the QF_UF SMTLib benchmark (2549 examples).

Size \ Time	$[0, 0.5[$	$[0.5, 1[$	$[1, 1.5[$	$[1.5, 2[$	$[2, 5[$	$[5, 10[$	$[10, 35[$	None
1	2235	75	28	16	33	32	61	69
2	2236	81	27	16	30	23	67	69
3	2236	79	27	16	34	23	65	69
4	2230	84	23	18	33	24	68	69
5	2231	79	27	12	36	22	73	69
6	2234	73	29	15	30	24	75	69
7	2231	81	23	15	33	22	75	69
8	2233	78	23	16	33	21	76	69

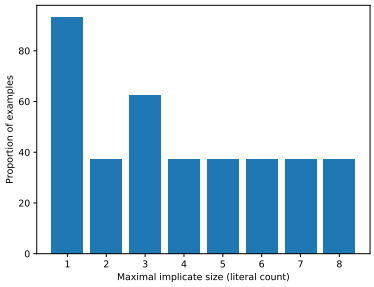
Table 2: Number of problems for which at least one $(\mathcal{T}, \mathcal{A})$-implicate of a given maximal size can be generated in a given amount of time (in seconds), for the QF_UFLIA SMTLib benchmark (400 examples).

Size \ Time	$[0, 0.5[$	$[0.5, 1[$	$[1, 1.5[$	$[1.5, 2[$	$[2, 5[$	$[5, 10[$	$[10, 35[$	None
1	120	23	46	76	100	6	25	4
2	120	23	6	0	0	0	247	4
3	120	23	6	0	96	4	147	4
4	120	23	6	0	0	0	247	4
5	120	23	6	0	0	0	247	4
6	120	22	7	0	0	0	247	4
7	121	22	6	0	0	0	247	4
8	116	24	6	3	0	0	247	4

1741, max. 17.10^6). We chose not to apply unit propagation simplifications to the considered sets of clauses. More precisely, this means that we let $U = M$ at line 5 of Algorithm 3 and delegate the simplifications that could occur in the following line to the satisfiability checker. The reason for this decision is that efficiently performing such simplifications can be difficult and strongly depends on the theory. We also define `fix`$(,,)$ as the complementation on literals and $\mathcal{P}$ as either `true` or a predicate ensuring $\text{card}(M) \leq n$ to generate $(\mathcal{T}, \mathcal{A})$-implicates of size at most n. In all the experiments, the prime implicates filter (SubMin) was not active, so that implicates can be generated on the fly. Finally, if available, we recover models of $S \cup M$ from the SMT solver in order to further prune the set of abducibles (see Line 8 of Algorithm 3). Tables 1 and 2 show the number of examples for which our tool generates at least one $(\mathcal{T}, \mathcal{A})$-implicate for a given timespan, for the QF_UF and QF_UFLIA benchmarks respectively.

The results show that our tool is quite efficient, since it fails to generate any $(\mathcal{T}, \mathcal{A})$-implicate within 35 seconds for only 2% (resp. 1%) of the QF_UF (resp. QF_UFLIA) benchmarks. Figure 1 shows the proportion of the QF_UFLIA set for which GPID generates an implicate in less than 15 seconds, depending on the maximal size constraint. For the QF_UF benchmark, the proportion decreases from 97% for a maximal size constraint of 1 to 95% when there are no size restrictions. We also point out that for 57% of the QF_UF benchmark, we are actually able to generate all the $(\mathcal{T}, \mathcal{A})$-implicates of size 1 in less than 15 seconds.

Fig. 1: Proportion (out of 100) of examples of the QF_UFLIA benchmark where GPID generates at least one implicate under 15 seconds.

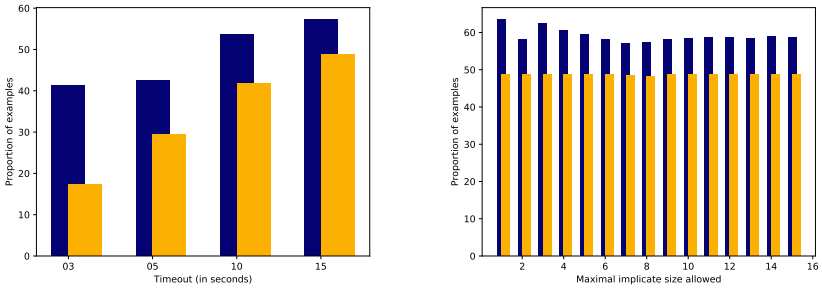


We ran additional experiments to compare this approach with a previous one based on a superposition-based approach [11,12] and implemented in the CSP tool. As far as we are aware, CSP is the only other available tool for implicate generation in the theory of equality with uninterpreted function symbols. Previous experiments (see, e.g., [11,12]) showed that CSP is already more efficient than approaches based on a reduction to propositional logic for generating implicates of ground equational formulas, which is why we did not run comparisons against tools for propositional implicate generation. CSP is based on a constrained calculus defined by the usual inference rules of the superposition calculus together with additional rules to dynamically assert new abducible hypotheses on demand into the search space. The asserted hypotheses are attached to the clauses as constraints and, when an empty clause is generated, the negation of these hypotheses yields a $(\mathcal{T}, \mathcal{A})$-implicate. We chose to compare the tools by focusing on their ability to generate at least one $(\mathcal{T}, \mathcal{A})$-implicate of a given size. Indeed, generating all (prime) $(\mathcal{T}, \mathcal{A})$-implicates is unfeasible within a reasonable amount of time except for very simple formulas, and comparing the raw number of $(\mathcal{T}, \mathcal{A})$-implicates generated is not relevant because some of these may actually be redundant w.r.t. non-generated ones[2]. We believe in practice, being able to efficiently compute a small number of $(\mathcal{T}, \mathcal{A})$-implicates for a complex problem is more useful than computing huge sets of $(\mathcal{T}, \mathcal{A})$-implicates but only for simple formulas. The following experiments are only based on benchmarks that can be solved by both prototypes, as CSP is not capable of handling integer arithmetics. We represented on Figure 2 the number of examples for which both tools can

[2] A refined comparison of the set of generated $(\mathcal{T}, \mathcal{A})$-implicates modulo theory entailment is left for future work.

Fig. 2: Number of examples from the QF_UF benchmark set for which GPID (on the left, darker color) and CSP (on the right, lighter color) generate at least one $(\mathcal{T}, \mathcal{A})$-implicate within a given time (a) and generate at least one implicate of a given maximal size under 15 seconds (b)

(a) (b)



generate at least one $(\mathcal{T}, \mathcal{A})$-implicate with a given maximal size constraint for various timeouts (a) and generate at least one $(\mathcal{T}, \mathcal{A})$-implicate within a given time limit for various maximal size constraints (b).

6 Conclusion

We devised a generic algorithm to generate implicates modulo theories and showed that the corresponding implementation is more efficient than a previous attempt based on superposition. This result was to be expected since the DPLL($\mathcal{T}$) approach is more efficient than engines based on the Superposition Calculus for testing the satisfiability of quantifier-free formulas with a large combinatorial structure. Furthermore, the used superposition engine had to be specifically tuned for implicate generation, and it is far less efficient than state-of-the-art systems such as Vampire [27], E [28] or Spass [31] (this is of course the advantage of having a generic algorithm using decision procedures as black boxes). While our aim was to be completely generic, it is clear than the efficiency of the procedure could be improved in practice by integrating theory-specific algorithms for deriving consequences and normalizing formulas. For instance, in the case of the theory of equality with uninterpreted function symbols, the implicates could be normalized by replacing each term by its minimal representative, as is done in [12]. Efficient, theory-dependent simplification procedures will also be explored in future work. A combination between the superposition-based approach [12], in which the assertion of hypotheses is guided by the proof procedure could also be beneficial. Our approach could also be combined with that of [7], which is based on model building and quantifier-elimination.

References

1. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the* 23^{rd} *International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.
2. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
3. M. Bienvenu. Prime implicates and prime implicants in modal logic. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 379. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.
4. P. Blackburn, J. Van Benthem, and F. Wolter. *Handbook of Modal Logic*. Studies in logic and practical reasoning - ISSN 1570-2464 ; 3. Elsevier, 2007.
5. J. De Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–780. John Wiley & Sons ltd, 1992.
6. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken. Minimum Satisfying Assignments for SMT. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, number 7358 in Lecture Notes in Computer Science, pages 394–409. Springer, 2012.
8. M. Echenim and N. Peltier. A Superposition Calculus for Abductive Reasoning. *Journal of Automated Reasoning*, 57(2):97–134, 2016.
9. M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. In *Proceedings of IJCAI'13 (International Conference on Artificial Intelligence)*, pages 3–9. AAAI, 2013.
10. M. Echenim, N. Peltier, and S. Tourret. A Rewriting Strategy to Generate Prime Implicates in Equational Logic. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'14)*. Springer, 2014.
11. M. Echenim, N. Peltier, and S. Tourret. Quantifier-free equational logic and prime implicate generation. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2015.
12. M. Echenim, N. Peltier, and S. Tourret. Prime Implicate Generation in Equational Logic. *Journal of Artificial Intelligence Research*, 60:827–880, 2017.
13. N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
14. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.
15. P. Jackson. Computing prime implicates incrementally. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 253–267. Springer-Verlag, 1992.
16. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990.

17. E. Knill, P. T. Cox, and T. Pietrzykowski. Equality and abductive residua for Horn clauses. *Theoretical Computer Science*, 120(1):1–44, Nov. 1993.
18. P. Marquis. Extending abduction from propositional to first-order logic. In *Fundamentals of artificial intelligence research*, pages 141–155. Springer, 1991.
19. P. Marquis. Consequence finding algorithms. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, pages 41–145. Springer, 2000.
20. A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009.
21. A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. *Foundations of Intelligent Systems*, pages 203–213, 2011.
22. M. C. Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1):99–117, 1993.
23. A. Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001.
24. H. Nabeshima, K. Iwanuma, K. Inoue, and O. Ray. SOLAR: An automated deduction system for consequence finding. *AI Commun.*, 23(2):183–203, Jan. 2010.
25. A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 1980–1987. AAAI Press, 2015.
26. W. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.
27. A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 376–380. Springer LNCS 2083, 2001.
28. S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
29. L. Simon and A. Del Val. Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 359–370, 2001.
30. P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, 4:446–456, 1967.
31. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In *Proceedings of the 16th Conference on Automated Deduction (CADE-16)*, pages 378–382. Springer LNCS 1632, 2001.