A Framework for Cooperating Decision Procedures

Clark W. Barrett, David L. Dill, and Aaron Stump

Stanford University, Stanford, CA 94305, USA,
http://verify.stanford.edu

Abstract. We present a flexible framework for cooperating decision procedures. We describe the properties needed to ensure correctness and show how it can be applied to implement an efficient version of Nelson and Oppen's algorithm for combining decision procedures. We also show how a Shostak style decision procedure can be implemented in the framework in such a way that it can be integrated with the Nelson-Oppen method.

1 Introduction

Decision procedures for fragments of first-order or higher-order logic are potentially of great interest because of their versatility. Many practical problems can be reduced to problems in some decidable theory. The availability of robust decision procedures that can solve these problem within reasonable time and memory could save a great deal of effort that would otherwise go into implementing special cases of these procedures.

Indeed, there are several publicly distributed prototype implementations of decision procedures, such as Presburger arithmetic [15], and decidable combinations of quantifier-free first-order theories [2]. These and similar procedures have been used as components in applications, including interactive theorem provers [13, 9], infinite-state model checkers [7, 10, 4], symbolic simulators [18], software specification checkers [14], and static program analyzers [8].

Nelson and Oppen [12] showed that satisfiability procedures for several theories that satisfy certain conditions can be combined into a single satisfiability procedure by propagating equalities. Many others have built upon this work, offering new proofs and applications [19, 1].

Shostak [17, 6, 16] gave an alternative method for combining decision procedures. His method is applicable to a more restricted set of theories, but is reported to be more efficient and is the basis for combination methods found in SVC [2], PVS [13], and STeP [9, 3]. An understanding of his algorithm has proven to be elusive.

Both STeP and PVS have at least some ability to combine the methods of Nelson and Oppen and Shostak [5, 3], but not much detail has been given, and the methods used in PVS have never been published. As a result, there is still significant confusion about the relationship between these two methods and how to implement them efficiently and correctly.

Our experience with SVC, a decision procedure for quantifier-free first-order logic based loosely on Shostak's method for combining cooperating decision procedures, has been both positive and negative. On the one hand, it has been implemented and is efficient and reliable enough to enable new capabilities in our research group and at a surprisingly large number of other sites. However, efforts to extend and modify SVC have revealed unnecessary constraints in the underlying theory, as well as gaps in our understanding of it.

This paper is an outcome of ongoing attempts to re-architect SVC to resolve these difficulties. We present an architecture for cooperating decision procedures that is simple yet flexible and show how the soundness, completeness, and termination of the combined decision procedure can be proved from a small list of clearly stated assumptions about the constituent theories. As an example of the application of this framework, we show how it can be used to implement and integrate the methods of Nelson and Oppen and Shostak. In so doing, we also describe an optimization applicable to the original Nelson and Oppen procedure and show how our framework simplifies the proof of correctness of Shostak's method. Due to the scope of this paper and space restrictions, many of the proofs have been abbreviated or omitted.

2 Definitions and Notation

Expressions in the framework are represented using the logical symbols **true**, **false**, and '**=**', an arbitrary number of variables, and non-logical symbols consisting of constants, and function and predicate symbols. We call **true** and **false** *constant formulas*. An *atomic formula* is either a constant formula, an equality between terms, or a predicate applied to terms. A *literal* is either an atomic formula or an equality between a non-constant atomic formula and **false**. Equality with **false** is used to represent negation. Formulas include atomic formulas, and are closed under the application of equality, conjunction and quantifiers. An expression is either a term or a formula. An expression is a *leaf* if it is a variable or constant. Otherwise, it is a *compound* expression, containing an operator applied to one or more children.

A *theory* is a set of first-order sentences. For the purposes of this paper, we assume that all theories include the axioms of equality. The *signature* of a theory is the set of function, predicate, and constant symbols appearing in those sentences. The *language* of a signature Σ is the set of all expressions whose function, predicate, and constant symbols come from Σ. Given a theory T with signature Σ, if ϕ is a sentence in the language of Σ, then we write $T \models \phi$ to mean that every model of T is also a model of ϕ. For a given model, M, an *interpretation* is a function which assigns an element of the domain of M to each variable. If Γ is a set of formulas and ϕ is a formula, then we write $\Gamma \models \phi$ to mean that for every model and interpretation satisfying each formula in Γ, the same model and interpretation satisfy ϕ. Finally, if Φ is a set of formulas, then $\Gamma \models \Phi$ indicates that $\Gamma \models \phi$ for each ϕ in Φ.

Expressions are represented using a directed acyclic graph (DAG) data structure such that any two expressions which are syntactically identical are uniquely represented by a single DAG. The following operations on expressions are supported.

`Op(e)`	the operator of e (just e itself if e is a leaf).
`e[i]`	the i^{th} child of e, where e[1] is the first child.

If $\mathtt{e}_1$ and $\mathtt{e}_2$ are expressions, then we write $\mathtt{e}_1 \equiv \mathtt{e}_2$ to indicate that $\mathtt{e}_1$ and $\mathtt{e}_2$ are the same expression (syntactically identical). In contrast, $\mathtt{e}_1 = \mathtt{e}_2$ is simply intended to represent the expression formed by applying the equality operator to $\mathtt{e}_1$ and $\mathtt{e}_2$. Expressions can be annotated with various attributes. If `a` is an attribute, `e.a` is the value of that attribute for expression `e`. Initially, $\mathtt{e.a} = \perp$ for each `e` and `a`, where $\perp$ is a special undefined value.

The following simple operations make use of an expression attribute called `find` to maintain equivalence classes of expressions. We assume that these are the only functions that reference the attribute. Note that when presenting pseudocode here and below, some required preconditions may be given next to the name and parameters of the function.

```
HasFind(a)                         SetFind(a) {a.find = ⊥ }
  RETURN a.find ≠ ⊥;                 a.find := a;

Find(a) {HasFind(a)}               Union(a,b) {a.find ≡ a ∧ b.find ≡ b }
  IF (a.find ≡ a) THEN RETURN a;     a.find := b.find;
  ELSE RETURN Find(a.find);
```

In some similar algorithms, `e.find` is initially set to `e`, rather than $\perp$. The reason we don't do this is that it turns out to be convenient to use an initialized `find` attribute as a marker that the expression has been seen before. This not only simplifies the algorithm, but it also makes it easier to describe certain invariants about expressions.

The `find` attribute induces a relation $\sim$ on expressions: $\mathtt{a} \sim \mathtt{b}$ if and only if $\mathtt{HasFind(a)} \wedge \mathtt{HasFind(b)} \wedge [\mathtt{Find(a)} \equiv \mathtt{Find(b)}]$. For the set of all expressions whose `find` attributes have been set, this relation is an equivalence relation. The *find database*, denoted by $\mathcal{F}$, is defined as follows: $\mathtt{a} = \mathtt{b} \in \mathcal{F}$ iff $\mathtt{a} \sim \mathtt{b}$. The following facts will be used below.

Find Database Monotonicity. *If the preconditions for* `SetFind` *and* `Union` *are met, then if $\mathcal{F}$ is the find database at some previous time and $\mathcal{F}'$ is the find database now, then $\mathcal{F} \subseteq \mathcal{F}'$.*

Find Lemma. *If the preconditions for* `Find`, `SetFind`, *and* `Union` *hold, then* `Find` *always terminates.*

3 The Basic Framework

As mentioned above, the purpose of the framework presented in this paper is to combine satisfiability procedures for several first-order theories into a satisfiability procedure for their union. Suppose that $T_1, \ldots, T_n$ are n first-order theories, with signatures $\Sigma_1, \ldots \Sigma_n$. Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$. The goal is to provide a framework for a satisfiability procedure which determines the satisfiability in T of a set of formulas in the language of Σ. Our approach follows that of Nelson and Oppen [12]. We assume that the intersection of any two signatures is empty and that each theory is *stably-infinite*. A theory T with signature Σ is called stably-infinite if any quantifier-free formula in the language of Σ is satisfiable in T only if it is satisfiable in an infinite model of T. We also assume that the theories are *convex*. A theory is convex if there is no conjunction of literals in the language of the theory which implies a disjunction of equalities without implying one of the equalities itself.

The interface to the framework from a client program consists of three methods: `AddFormula`, `Satisfiable`, and `Simplify`. Conceptually, `AddFormula` adds its argument (which must be a literal) to a set $\mathcal{A}$, called the *assumption history*. `Simplify` transforms an expression into a new expression which is equivalent modulo $T \cup \mathcal{A}$, and `Satisfiable` returns false if and only if $T \cup \mathcal{A} \models$ **false**. Since any quantifier-free formula can be converted to disjunctive normal form, after which each conjunction of literals can be checked separately for satisfiability, the restriction that the arguments to `AddFormula` be literals does not restrict the power of framework.

The framework includes sets of functions which are parameterized by theory. For example, if `f` is such a function, we denote by $\mathtt{f}_i$ the instance of `f` associated with theory T_i. If for some `f` and T_i, we do not explicitly define the instance $\mathtt{f}_i$, it is assumed that a call to $\mathtt{f}_i$ does nothing. It is convenient to be able to call these functions based on the theory associated with some expression `e`. Expressions are associated with theories as follows. First, variables are partitioned among the theories arbitrarily. In some cases, one choice may be better than another, as discussed in Sec. 5.1 below. An expression in the language of Σ is associated with theory T_i if and only if it is a variable associated with T_i, its operator is a symbol in Σ_i, or it is an equality and its left side is associated with theory T_i. If an expression is associated with theory T_i, we call it an i-expression. We denote by $T(\mathtt{e})$ the index i, where `e` is an i-expression.

Figure 1 shows pseudocode for the basic framework. An input formula is first simplified it because it might already be known or reduce to something easier to handle. Simplification involves the recursive application of `Find` as well as certain rewrite rules. `Assert` calls `Merge` which merges two $\sim$-equivalence classes. `Merge` first calls `Setup` which ensures that the expressions are in an equivalence class.

There are four places in the framework in which *theory-specific* functionality can be introduced. `TheorySetup`, `TheoryRewrite` and `PropagateEqualities` are theory-parameterized functions. Also, each expression has a `notify` attribute containing a set of pairs $\langle \mathtt{f,d} \rangle$, where f is a function and d is some data. Whenever `Merge` is called on an expression `a = b`, the `find` attribute of `a` changes to

```
AddFormula(e) { e is a literal }
  Assert(e);
  REPEAT
    done := true;
    FOREACH theory T_i DO IF PropagateEqualities_i() THEN done := false;
  UNTIL done;

Assert(e) { e is a literal; T ∪ A ⊨ e }
  IF ¬Satisfiable() THEN RETURN;
  e' := Simplify(e);
  IF e' ≡ true THEN RETURN;
  IF Op(e') ≠ '=' THEN e' := (e' = true);
  Merge(e');

Merge(e) { Op(e) = '='; T ∪ A ⊨ e; see text for others }
  Setup(e[1]); Setup(e[2]);
  IF e[1] and e[2] are terms THEN TheorySetup_T(e)(e);
  Union(e[1],e[2]);
  FOREACH ⟨f,d⟩ ∈ e[1].notify DO f(e,d);

Setup(e)
  IF HasFind(e) THEN RETURN;
  FOREACH child c of e DO Setup(c);
  TheorySetup_T(e)(e);
  SetFind(e);

Simplify(e)
  IF HasFind(e) THEN RETURN Find(e);
  Replace each child c of e with Simplify(c);
  RETURN Rewrite(e);

Rewrite(e)
  IF HasFind(e) THEN RETURN Find(e);
  IF Op(e) = '=' THEN e' := RewriteEquality(e);
  ELSE e' := TheoryRewrite_T(e)(e);
  IF e ≢ e' THEN e' := Rewrite(e');
  RETURN e';

RewriteEquality(e)
  IF e[1] ≡ e[2] THEN RETURN true;
  IF one child of e is true THEN RETURN the other child;
  IF e[1] ≡ false THEN RETURN (e[2] = e[1]);
  RETURN e;

Satisfiable()
  RETURN true ≁ false;
```

Fig. 1. Basic Framework

b, and f(a = b,d) is called for each $\langle$f,d$\rangle \in$ a.notify. Typically, TheorySetup adds callback functions to the notify attribute of various expressions to guarantee that the theory's satisfiability procedure will be notified if one of those expressions is merged with another expression. Finally, before returning from AddFormula, each theory may notify the framework of additional equalities it has deduced until each theory reports that there are no more equalities to propagate.

Theory-specific code is distinguished from the framework code shown in Fig. 1 and from *user code* which is the rest of the program. It may call functions in the framework, provided any required preconditions are met. Examples of theory-specific code for both Nelson-Oppen and Shostak style theories are given below, following a discussion of the abstract requirements which must be fulfilled by theory-specific code to ensure correctness.

4 Correctness of the Basic Framework

In order to prove correctness, we give a specification in terms of preconditions and postconditions and show that the framework meets the specification. Sometimes it is necessary to talk about the state of the program. Each run of a program is considered to be a sequence of *states*, where a state includes a value for each variable in the program and a location in the code.

4.1 Preconditions and Postconditions

The preconditions for each function in the framework except for Merge are shown in the pseudocode. In order to give the precondition for Merge, a few definitions are required.

A *path* from an expression e to a sub-expression s of e is a sequence of expressions $e_0, e_1, ..., e_n$ such that $e_0 \equiv e$, e_{i+1} is a child of e_i, and s is a child of e_n. A sub-expression s of an expression e is called a *highest find-initialized* sub-expression of e if HasFind(s) and there is a path from e to s such that for each expression e' on the path, ¬HasFind(e'). An expression e is called *find-reduced* if Find(s) $\equiv$ s for each highest find-initialized sub-expression s of e.

An expression e is called *merge-acceptable* if e is an equation and one of the following holds: e is a literal; e[1] is **false** or an atomic predicate and e[2] $\equiv$ **true**; or e[1] $\equiv$ **true** and e[2] $\equiv$ **false**.

Merge Precondition.
Whenever Merge(e) is called, the following must hold.

1. e is merge-acceptable,
2. e[1] and e[2] are find-reduced,
3. e[1] $\not\equiv$ e[2], and
4. $T \cup \mathcal{A} \models$ e.

In addition to the preconditions, the following postconditions must be satisfied by the parameterized functions.

TheoryRewrite Postcondition.
After `e' := TheoryRewrite(e)` or `e' := RewriteEquality(e)` is executed, the following must hold:

1. $\mathcal{F}$ is unchanged by the call,
2. if `e` is a literal, then `e'` is a literal,
3. if `e` is find-reduced, then `HasFind(e')` or `e'` is find-reduced, and
4. $T \cup \mathcal{F} \models \mathtt{e} = \mathtt{e'}$.

TheorySetup Postcondition.
After `TheorySetup` is executed, the find database is unchanged.

If all preconditions and postconditions hold for all functions called so far, we say that the program is in an *uncorrupted* state. Also, if **true** $\not\sim$ **false**, we say the program is in a *consistent* state. A few lemmas are required before proving that the preconditions and postconditions hold for the framework code.

Lemma 1. *If the program is in an uncorrupted state and* `Union(a,b)` *has been called, then since that call there have been no calls to* `Union` *where either argument was* `a`.

Proof. Once `Union(a,b)` is called, $\mathtt{a.find} \not\equiv \mathtt{a}$ and this remains true since it can never again be an argument to `SetFind` or `Union`.

Lemma 2 (Equality Find Lemma). *If* $\mathtt{e} \equiv \mathtt{a} = \mathtt{b}$ *and the program is in an uncorrupted and consistent state whose location is not between the call to* `SetFind(e)` *and the next call to* `Union` *and* `HasFind(e)`*, then* `a` *and* `b` *are terms and* `Find(e)` $\equiv$ **false**.

Proof. Suppose `HasFind(e)`. Then `Setup(e)` was called. But by the definition of merge-acceptable, this can only happen if `e`[1] and `e`[2] are terms and `Merge(e =` **false**`)` was called, in which case `Union(e,`**false**`)` is called immediately afterwards. It is clear from the definition of merge-acceptable, that `Union` is never called with first argument **false** unless the second argument is **true**. Thus, if **true** $\not\sim$ **false**, it follows from Lemma 1 that `Find(e)` $\equiv$ **false**. □

Lemma 3 (Literal Find Lemma). *If the program is in an uncorrupted state and* `e` *is a literal, then* `Find(e)` *is either* `e`*,* **true***, or* **false**.

Proof. From the previous lemma, it follows that if `e` is an equality, then `Find(e)` is either `e`, **true**, or **false**. A similar argument shows that the same is true for a predicate. □

Lemma 4 (Simplify Lemma).
If the program is in an uncorrupted state after `e' := Simplify(e)` *is executed, then following are true:*

1. *$\mathcal{F}$ is unchanged by the call,*
2. *if* `e` *is a literal then* `e'` *is a literal,*
3. *if* `e` *is a literal or term, then* `e'` *is find-reduced, and*
4. $T \cup \mathcal{F} \models$ `e` $=$ `e'`.

We must prove the following theorem. A similar theorem is required every time we introduce theory-specific code.

Theorem 1. *If the program is in an uncorrupted state located in the framework code, then the next state is also uncorrupted.*

Proof.
Find Precondition: `Find` is called in two places by the framework. In each case, we check the precondition before calling it.
SetFind Precondition: `SetFind(e)` is only called from `Setup(e)` which returns if `HasFind(e)`. Otherwise, `Setup` performs a depth-first traversal of the expression and calls `SetFind`. It follows from the TheorySetup Postcondition and the fact that expressions are acyclic that the precondition is satisfied.
Union Precondition: `Union(a,b)` is only called if `Merge(a = b)` is called first. By the `Merge` precondition, `a` and `b` are find-reduced. It is easy to see that after `Setup(a)` and `Setup(b)` are called, `Find(a)` $\equiv$ `a` and `Find(b)` $\equiv$ `b`.
AddFormula Precondition: We assume that `AddFormula` is only called with literals.
Assert Precondition: `Assert(e)` is only called from `AddFormula`. In this case, `e` $\in \mathcal{A}$, so it follows that $T \cup \mathcal{A} \models$ `e`.
Merge Precondition: `Merge(e')` is called from `Assert(e)`. We know that `e` is a literal, so by the Simplify Lemma, `Simplify(e)` is a literal and is find-reduced. It follows that `e'` is merge-acceptable and `e'`[1] and `e'`[2] are find-reduced and unequal. From the Simplify Lemma, we can conclude that $T \cup \mathcal{F} \models$ `e` $=$ `e'`. It follows from the soundness property (described next) that $T \cup \mathcal{A} \models$ `e` $=$ `e'`. We know that $T \cup \mathcal{A} \models$ `e`, so it follows that $T \cup \mathcal{A} \models$ `e'`.
TheoryRewrite Postcondition: It is straight-forward to check that each of the requirements hold for `RewriteEquality`.

□

4.2 Soundness

The satisfiability procedure is *sound* if whenever the program state is inconsistent, $T \cup \mathcal{A} \models$ **false**. Soundness depends on the invariance of the following property.

Soundness Property. $T \cup \mathcal{A} \models \mathcal{F}$.

Lemma 5. *If the program is in an uncorrupted state, then the soundness property holds.*

Proof. Initially, the find database is empty. New formulas are added in two places. The first is in `Setup`, when `SetFind` is called. This preserves the soundness property since it only adds a reflexive formula to $\mathcal{F}$. The other is in `Merge(e)`, when `Union(e[1],e[2])` is called. This adds the formula `e` to $\mathcal{F}$, but we know that $T \cup \mathcal{A} \models \mathtt{e}$ by the Merge Precondition. It also results in the addition of any formulas which can be deduced using transitivity and symmetry, but these are also entailed because T includes equality. □

Theorem 2. *If the program is in an uncorrupted state, then the satisfiability procedure is sound.*

Proof. Suppose `Satisfiable` returns false. This means that **true** $\sim$ **false**. It follows from the previous lemma that $T \cup \mathcal{A} \models \mathbf{true} = \mathbf{false}$, so $T \cup \mathcal{A} \models \mathbf{false}$. □

4.3 Completeness

The satisfiability procedure is *complete* if $T \cup \mathcal{A}$ is satisfiable whenever the program is in a consistent state in the user code.

We define the *merge database*, denoted $\mathcal{M}$, as the set of all expressions e such that there has been a call to `Merge(e)`. In order to describe the property which must hold for completeness, we first introduce a few definitions, adapted from [19].

Recall that an expression in the language of Σ is an i-expression if it is a variable associated with T_i, its operator is a symbol in Σ_i, or it is an equality and its left side is an i-expression. A sub-expression of `e` is called an *i-leaf* if it is a variable or a j-expression, with $j \neq i$, and every expression along some path from `e` is an i-expression. An i-leaf is an *i-alien* if it is not an i-expression. An i-expression in which every i-leaf is a variable is called *pure* (or *i-pure*).

With each term `t` which is not a variable, we associate a fresh variable $v(\mathtt{t})$. We define $v(\mathtt{t})$ to be `t` when `t` is a variable. For some expression or set of expressions S, we define $\gamma_i(S)$ by replacing all of the i-alien terms `t` in S by $v(\mathtt{t})$[1] so that every expression in $\gamma_i(S)$ is i-pure. We denote by $\gamma_0(S)$ the set obtained from S by replacing all maximal terms (i.e. terms without any super-terms) `t` by $v(\mathtt{t})$. Let Θ be the set of all equations $\mathtt{t} = v(\mathtt{t})$, where `t` is a sub-term of some formula in $\mathcal{M}$. It is easy to see that $T \cup \mathcal{M}$ is satisfiable iff $T \cup \mathcal{M} \cup \Theta$ is satisfiable.

Let $\mathcal{M}_i = \{\mathtt{e} \mid \mathtt{e} \in \mathcal{M} \wedge \mathtt{e}$ is an i-expression $\}$. Define Θ_i similarly. Notice that $(\mathcal{M} \cup \Theta)$ is logically equivalent to $\bigcup \gamma_i(\mathcal{M}_i \cup \Theta_i)$, since each can be transformed into the other by repeated substitutions.

[1] Since expressions are DAG's, we must be careful about what is meant by *replacing* a sub-expression. The intended meaning here and throughout is that the expression is considered as a tree, and only occurrences of the term which qualify for replacement in the tree are replaced. This means that some occurrences may not be replaced at all, and the resulting DAG may look significantly different as a result.

We define V, the set of *shared terms* as the set of all terms $\mathtt{t}$ such that $v(\mathtt{t})$ appears in at least two distinct sets $\gamma_i(\mathcal{M}_i \cup \Theta_i), 1 \leq i \leq n$. Let $E(V) = \{\mathtt{a} = \mathtt{b} \mid \mathtt{a}, \mathtt{b} \in V \wedge \mathtt{a} \sim \mathtt{b}\}$, and let $D(V) = \{\mathtt{a} \neq \mathtt{b} \mid \mathtt{a}, \mathtt{b} \in V \wedge \mathtt{a} \not\sim \mathtt{b}\}$. For a set of expressions S, an arrangement $\pi(S)$ is a set such that for every two expressions $\mathtt{a}$ and $\mathtt{b}$ in S, exactly one of $\mathtt{a} = \mathtt{b}$ or $\mathtt{a} \neq \mathtt{b}$ is in $\pi(S)$. We denote by $\pi(V)$ the arrangement $E(V) \cup D(V)$ of V determined by $\sim$. Now we can state the property required for completeness.

Completeness Property. If the program is in a consistent state in the user code, then $T_i \cup \gamma_i(\mathcal{M}_i \cup \pi(V))$ is satisfiable.

The following lemmas are needed before proving completeness.

Lemma 6. *If the program is in an uncorrupted state, then $T \cup \mathcal{M} \models \mathcal{F}$*

Proof. Every formula in $\mathcal{F}$ is either in $\mathcal{M}$ or can be derived from formulas in $\mathcal{M}$ using reflexivity, symmetry, and transitivity of equality. □

Lemma 7. *If the program is in an uncorrupted and consistent state in the user code, then $T \cup \mathcal{M} \models \mathcal{A}$.*

Proof. Suppose $e \in \mathcal{A}$. Then we know that `Assert(e)` was called at some time previously. We can conclude by monotonicity of the find database that **true** $\not\sim$ **false** at the time of that call. Thus, `e' := Simplify(e)` was executed. By the Simplify Lemma, if $\mathcal{F}_1$ was the find database at the time of the call, $T \cup \mathcal{F}_1 \models \mathtt{e} = \mathtt{e'}$. Now, if $e' \equiv$ **true**, then $T \cup \mathcal{F}_1 \models \mathtt{e}$ and so by monotonicity and Lemma 6, $T \cup \mathcal{M} \models \mathtt{e}$. Otherwise, `Merge` is called. Let `x` be the argument to `Merge`. It is easy to see that $T \cup \mathcal{F}_1 \models \mathtt{e} = \mathtt{x}$. But $\mathtt{x} \in \mathcal{M}$, so $T \cup \mathcal{M} \models \mathtt{x}$. It then follows easily by monotonicity and Lemma 6 that $T \cup \mathcal{M} \models \mathtt{e}$. □

The following theorem is from [19].

Theorem 3. *Let T_1 and T_2 be two stably-infinite, signature-disjoint theories and let ϕ_1 be a set of formulas in the language of T_1 and ϕ_2 a set of formulas in the language of T_2. Let v be the set of their shared variables and let $\pi(v)$ be an arrangement of v. If $\phi_i \wedge \pi(v)$ is satisfiable in T_i for i = 1, 2, then $\phi_1 \wedge \phi_2$ is satisfiable in $T_1 \cup T_2$.*

Theorem 4. *If the procedure always maintains an uncorrupted state and the completeness property holds for each theory, then the procedure is complete.*

Proof. Suppose that for a consistent state in the user code, $T_i \cup \gamma_i(\mathcal{M}_i \cup \pi(V))$ is satisfiable for each i. This implies that $T_i \cup \gamma_i(\mathcal{M}_i \cup \Theta_i \cup \pi(V))$ is satisfiable (since each equation in Θ_i simply defines a new variable), which is logically equivalent (by applying substitutions from Θ_i) to $T_i \cup \gamma_i(\mathcal{M}_i \cup \Theta_i) \cup \gamma_0(\pi(V))$. Now, each set $\gamma_i(\mathcal{M}_i \cup \Theta_i)$ is a set of formulas in the language of T_i, and $\gamma_0(\pi(V))$ is an arrangement of the variables shared among these sets, so we can conclude by repeated application of Theorem 3 that $\bigcup \gamma_i(\mathcal{M}_i \cup \Theta_i)$ is satisfiable in T. But $\bigcup \gamma_i(\mathcal{M}_i \cup \Theta_i)$ is equivalent to $\mathcal{M} \cup \Theta$ which is satisfiable in T iff $T \cup \mathcal{M}$ is satisfiable. Finally, by Lemma 7, $T \cup \mathcal{M} \models \mathcal{A}$. Thus we can conclude that $T \cup \mathcal{A}$ is satisfiable. □

4.4 Termination.

We must show that each function in the framework terminates. The following requirements guarantee this.

Termination Requirements.

1. The preconditions for `Find`, `SetFind`, and `Union` always hold.
2. For each i-expression `e`, $\texttt{TheoryRewrite}_i\texttt{(e)}$ terminates.
3. If s is a sequence of expressions in which the next member of the sequence `e'` is formed from the previous member `e` by calling $\texttt{TheoryRewrite}_i\texttt{(e)}$, then beyond some element of the sequence, all the expressions are identical.
4. For each i-expression `e`, $\texttt{TheorySetup}_i\texttt{(e)}$ terminates.
5. After `Union(a,b)` is called,
   (a) No new entries are added to `a.notify`.
   (b) Each call to each funtion in `a.notify` terminates.
6. For each theory T_i, $\texttt{PropagateEqualities}_i$ terminates and after calling $\texttt{PropagateEqualities}_i$ some finite number of times, it will always return false.

Theorem 5. *If the termination requirements hold, then each function in the framework terminates.*

Proof. The first condition guarantees that `Find` terminates, from which it follows that `Satisfiable` terminates. The next two ensure that `Rewrite` terminates. It then follows easily that `Simplify` must terminate. The next few conditions are sufficient to ensure that `Setup` and `Merge` terminate, from which it follows that `Assert` terminates. This, together with the last condition allows us to conclude that `AddFormula` terminates. □

It is not hard to see that without any theory-specific code, these requirements hold.

5 Examples Using the Framework

In this section we will give two examples to show how the framework can accommodate different kinds of theory-specific code.

5.1 Nelson-Oppen Theories

A Nelson-Oppen style satisfiability procedure for a theory T_i must be able to determine the satisfiability of a set of formulas in the language of Σ_i as well as which equalities between variables are entailed by that set of formulas [12]. We present a method for integrating such theories which is flexible and efficient.

Suppose we have a Nelson-Oppen style satisfiability procedure which treats alien terms as variables with the following methods:

$\texttt{AddFormula}_i$	Adds a new formula to the set $\mathcal{A}_i$.
$\texttt{Satisfiable}_i$	True iff $T_i \cup \gamma_i(\mathcal{A}_i)$ is satisfiable.
$\texttt{AddTermToPropagate}_i$	Adds a term to the set Δ_i.
$\texttt{GetEqualities}_i$	Returns the largest set of equalities $\mathcal{E}_i$ between terms in Δ_i such that $T_i \cup \gamma_i(\mathcal{A}_i) \models \gamma_i(\mathcal{E}_i)$.

A new expression attribute, `shared` is used to keep track of which terms are relevant to more than one theory. Each theory is given an index, i, and the `shared` attribute is set to i if the term is used by theory i. If more than one theory uses the term, the `shared` attribute is set to 0. This is encapsulated in the `SetShared` and `IsShared` methods shown below.

```
SetShared(e,i)
  IF e.shared = ⊥ THEN e.shared := i;
  ELSE IF e.shared ≠ i THEN e.shared := 0;
  AddTermToPropagate_i(e);
```

```
IsShared(e)
  RETURN e.shared = 0;
```

Figure 2 shows the theory-specific code needed to add a theory T_i with a satisfiability procedure as described above. We will refer to a theory implemented in this way as a *Nelson-Oppen* theory. Each i-expression is passed to $\texttt{TheorySetup}_i$. $\texttt{TheorySetup}_i$ marks these terms and their alien children as used by T_i. It also ensures that $\texttt{Notify}_i$ will be called if any of these expressions are merged with something else. When $\texttt{Notify}_i$ is called, the formula is passed along to the satisfiability procedure for T_i. These steps correspond to the decomposition into pure formulas in other implementations (but without the introduction of additional variables). $\texttt{PropagateEqualities}_i$ asserts any equations between shared terms that have been deduced by the satisfiability procedure for T_i. This corresponds to the equality propagation step in other methods. It is sufficient to propagate equalities between shared variables, a fact also noted in [19].

We also introduce a new optimization. Not all theories need to know about all equalities between shared terms. A theory is only notified of an equality if the left side of that equality is a term that it has seen before. In order to guarantee that this results in fewer propagations, we have to ensure that whenever an equality between two terms is in $\mathcal{M}$, if one of the terms is not shared, then the left term is not shared. We can easily do this by modifying `RewriteEquality` to put non-shared terms on the left. However, this is not necessary for correctness, a fact which allows the integration of Shostak-style satisfiability procedures which require a different implementation of `RewriteEquality` as described in Sec. 5.2 below.

A final optimization is to associate variable with theories in such a way as to to avoid causing terms to be shared unnecessarily. For example, if $x = t$ is a formula in $\mathcal{M}$ and x is a variable and t is an i-term, it is desirable for x to be an i-term as well (otherwise, t immediately becomes a shared term). In our implementation, expressions are type-checked and each type is associated with

```
TheorySetup_i(e)
  FOREACH i-alien child a of e DO BEGIN
    a.notify := a.notify ∪ { ⟨Notify_i, ∅⟩ };
    SetShared(a,i);
  END
  e.notify := e.notify ∪ { ⟨Notify_i, ∅⟩ };
  IF e is a term THEN SetShared(e,i);

TheoryRewrite_i(e)
  RETURN e;

PropagateEqualities_i()
  propagate := false;
  IF Satisfiable() BEGIN
    IF ¬ Satisfiable_i() THEN Merge(true = false));
    ELSE FOREACH x = y ∈ GetEqualities_i DO
      IF IsShared(x) AND IsShared(y) AND x ≁ y THEN BEGIN
        propagate := true;
        Assert(x = y));
      END
  END
  RETURN propagate;

Notify_i(e)
  IF e[1] is an i-alien term THEN BEGIN
    x := Find(e[2]);
    x.notify := x.notify ∪ { ⟨Notify_i, ∅⟩ };
    e := (e[1] = x);
  END
  AddFormula_i(e);
```

Fig. 2. Code for implementing a Nelson-Oppen theory T_i.

a theory. Thus, we can easily guarantee this by associating x with the theory associated with its type.

Correctness. The proof of the following theorem is similar to that given for the framework code and is omitted.

Theorem 6. *If the program is in an uncorrupted state located in the theory-specific code for a Nelson-Oppen theory, then the next state is also uncorrupted.*

To show that the completeness property holds, we must show that if the program is in a consistent state in the user code, then $T_i \cup \gamma_i(\mathcal{M}_i \cup \pi(V))$ is satisfiable. This requires the following invariant to hold for each theory T_i.

Shared Term Requirement There has been a call to `SetShared(e,i)` if $v(\mathtt{e})$ appears in $\gamma_i(\mathcal{M}_i \cup \Theta_i)$.

Lemma 8. *If T_i is a Nelson-Oppen theory, then the shared term requirement holds for T_i.*

Corollary 1. *If T_i is a Nelson-Oppen theory, and $v(\mathtt{t})$ appears in $\gamma_i(\mathcal{M}_i \cup \Theta_i)$, then $\mathtt{t} \in \Delta_i$.*

Let $\Delta'_i = \Delta_i \cup \{\mathtt{x} \mid \mathtt{x}$ is a term and $\mathtt{t} = \mathtt{x} \in \mathcal{A}_i$ for some term $\mathtt{t}\}$.

Lemma 9. *If T_i is a Nelson-Oppen theory and the program is in an uncorrupted state in the user code and $\mathtt{x} = \mathtt{y} \in \mathcal{M}$, where $\mathtt{x} \in \Delta'_i$, then $\mathtt{x} = \mathtt{z} \in \mathcal{A}_i$, where $z \equiv$ `Find(y)` at some previous time.*

Proof. Suppose $\mathtt{x} \in \Delta_i$. Then `SetShared` was called. It is easy to see from the code that at the time it was called, $\mathtt{Notify}_i$ was added to `x.notify`. If on the other hand, $\mathtt{x} \notin \Delta_i$, then $\mathtt{t} = \mathtt{x} \in \mathcal{A}_i$ for some `t` which is not an i-term. But then, when $\mathtt{t} = \mathtt{x}$ was added to $\mathcal{A}_i$, $\mathtt{Notify}_i$ was added to `x.notify`. In each case, $\mathtt{Notify}_i\mathtt{(x = y)}$ will be called after `Merge(x = y)` is called, so that $\mathtt{x} =$ `Find(y)` is added to $\mathcal{A}_i$. □

Lemma 10. *If T_i is a Nelson-Oppen theory and the program is in an uncorrupted state in the user code and $\mathtt{x} \sim \mathtt{y}$, where $\mathtt{x}, \mathtt{y} \in \Delta'_i$, then $T_i \cup \gamma_i(\mathcal{A}_i) \models \gamma_i(x = y)$.*

Proof. We can show by the previous lemma that since `Find(x)` $\equiv$ `Find(y)`, there is a chain of equalities in $\mathcal{A}_i$ linking `x` to `y`. □

Let $D_i = \{\mathtt{a} \neq \mathtt{b} \mid \mathtt{a}, \mathtt{b} \in (\Delta_i \cap V)\}$, and let $D'_i = \{\mathtt{a} \neq \mathtt{b} \mid \mathtt{a}, \mathtt{b} \in (\Delta'_i \cap V)\}$.

Lemma 11. *If T_i is a Nelson-Oppen theory and the program is in an uncorrupted and consistent state in the user code, then $T_i \cup \gamma_i(\mathcal{A}_i \cup D_i)$ is satisfiable.*

Proof. No single disequality $x \neq y \in D_i$ can be inconsistent because if it were, that would mean $T_i \cup \gamma_i(\mathcal{A}_i) \models \gamma_i(x = y)$. But if this is the case, since $\mathtt{PropagateEqualities}_i$ terminated, it must be the case that $\mathtt{x} \sim \mathtt{y}$. Since no single equality $\mathtt{x} = \mathtt{y}$ is entailed, it follows from the convexity of T_i, that no disjunction of equalities can be entailed. □

Lemma 12. *If T_i is a Nelson-Oppen theory and the program is in an uncorrupted and consistent state in the user code, then $T_i \cup \gamma_i(\mathcal{A}_i \cup D'_i)$ is satisfiable.*

Proof. If $\mathtt{t}'_1 \neq \mathtt{t}'_2 \in D'_i$, we can find (by the definition of Δ'_i) some $\mathtt{t}_1$ and $\mathtt{t}_2$ such that $\mathtt{t}_1 \neq \mathtt{t}_2 \in D_i$ and $\mathcal{A}_i \models (\mathtt{t}_1 = \mathtt{t}'_1 \wedge \mathtt{t}_2 = \mathtt{t}'_2)$. The result follows by the previous lemma. □

Theorem 7. *If each theory satisfies the shared term requirement and the program is in an uncorrupted and consistent state in the user code, then if T_i is a Nelson-Oppen theory, the completeness property holds for T_i.*

Proof. It is not hard to show that if $v(\mathtt{x}) \in \gamma_i(\mathcal{A}_i \cup \Theta_i)$, then $\mathtt{x} \in \Delta'_i$. It then follows that an interpretation satisfying $T_i \cup \gamma_i(\mathcal{A}_i \cup D'_i)$ can be modified to also satisfy $\gamma_i(\pi(V))$. □

Termination. The only termination condition that is non-trivial is the last one. The following requirement is sufficient to fulfill this condition.

Nelson-Oppen Termination Requirement
Suppose that before a call to `Assert` from $\mathtt{PropagateEqualities}_i$, n is the number of equivalence classes in $\sim$ containing at least one term $\mathtt{t} \in V$. Then, either the state following the call to `Assert` is inconsistent or if m is the number of equivalence classes in $\sim$ containing at least one term $\mathtt{t} \in V$ after returning from `Assert`, $m < n$.

If every theory is a Nelson-Oppen theory, it is not hard to see that this requirement holds. This is because each call to `Assert` merges the equivalence classes of two shared variables without creating any new ones.

5.2 Adding Shostak Theories

Suppose we have a theory T_i with no predicate symbols which provides two functions, σ and ω which we refer to as the canonizer and solver respectively. Note that if we have more than one such theory, we can often combine the canonizers and solvers to form a canonizer and solver for the combined theory, as described in [17][2]. The functions σ and ω have the following properties.

[2] Although it has been claimed that solvers can always be combined to form a solver for the combined theory [6, 17], this is not always possible, as pointed out in [11]

σ is a canonizer for T_i if

1. $T_i \models \gamma_i(a = b)$ iff $\sigma(\mathtt{a}) \equiv \sigma(\mathtt{b})$
2. $\sigma(\sigma(\mathtt{t})) \equiv \sigma(\mathtt{t})$ for all terms $\mathtt{t}$.
3. $\gamma_i(\sigma(\mathtt{t}))$ contains only variables occurring in $\gamma_i(\mathtt{t})$.
4. $\sigma(\mathtt{t}) \equiv \mathtt{t}$ if $\mathtt{t}$ is a variable or not an i-term.
5. If $\sigma(\mathtt{t})$ is a compound i-term, then $\sigma(\mathtt{x}) = \mathtt{x}$ for each child $\mathtt{x}$ of $\sigma(\mathtt{t})$.

ω is a solver[3] for T_i if

1. If $T_i \models \gamma_i(x \neq y)$ then $\omega(\mathtt{x = y}) \equiv$ **false**.
2. Otherwise, $\omega(\mathtt{x = y}) \equiv \mathtt{a = b}$ where $\mathtt{a}$ and $\mathtt{b}$ are terms,
3. $T_i \models (x = y) \leftrightarrow (a = b)$,
4. $\gamma_i(a)$ is a variable and does not appear in $\gamma_i(b)$,
5. neither $\gamma_i(\mathtt{a})$ nor $\gamma_i(\mathtt{b})$ contain variables not occurring in $\gamma_i(\mathtt{x} = \mathtt{y})$,
6. $\omega(a = b) \equiv a = b$ and $\sigma(b) \equiv b$.

We call such a theory a Shostak theory. The code in Fig. 3 shows the additional code needed to integrate a Shostak theory.

Correctness. It is not hard to show that this code satisfies the preconditions and requirements of the framework.

Theorem 8. *If the program is in an uncorrupted state located in the theory-specific code for a Shostak theory, then the next state is also uncorrupted.*

Included in the Shostak code are the calls to `SetShared` necessary to allow this theory to be integrated with Nelson-Oppen theories. We have not included the code typically included for handling uninterpreted functions. This is because our approach allows us to consider uninterpreted functions as belonging to a separate Nelson-Oppen theory. Though we do not show how in this paper, any simple congruence closure algorithm can be integrated as a Nelson-Oppen theory. Omitting details related to uninterpreted functions simplifies the presentation and proof. We have also included code for handling disequalities, which Shostak's original procedure does not handle directly. We will give some intuition for how this works after making a few definitions.

Let $\Delta_i = \{\mathtt{t} \mid \mathtt{t}$ is an i-leaf in some expression $\mathtt{e} \in \mathcal{M}\}$. Let $\mathcal{E} = \{\mathtt{a} = \mathtt{b} \mid \mathtt{a} \in \Delta_i \wedge \mathtt{b} \equiv \mathtt{Find(a)}\ \}$. For an expression $\mathtt{e}$, define $\tau(\mathtt{e})$ to be the expression obtained from $\mathtt{e}$ by replacing each i-leaf $\mathtt{x}$ in $\mathtt{e}$ by `Find(x)`. Shostak's method works by ensuring that $\mathtt{Find(t)} \equiv \sigma(\tau(\mathtt{t}))$. This together with the properties of the solver ensure that the set $\mathcal{E}$ is equivalent to a substitution, meaning it is easily satisfiable. These are the key ideas of the completeness argument.

Lemma 13. *If the program is in an uncorrupted and consistent state which is not inside of a call to* `Merge`*, then for each term* $\mathtt{t}$ *such that* `HasFind(t)`*,* $\mathtt{Find(t)} \equiv \sigma(\tau(\mathtt{t}))$*. Also, if* $\mathtt{Find(t)} \equiv \mathtt{t}$*, then* $\tau(\mathtt{t}) \equiv \mathtt{t}$*.*

[3] Shostak allows the solved form to be more general. To simplify the presentation, we assume the solver returns a single, logically equivalent, equation.

```
RewriteEquality(e)
  IF e[1] ≡ e[2] THEN RETURN true;
  IF one child of e is true THEN RETURN the other child;
  IF e[1] ≡ false THEN RETURN (e[2] = e[1]);
  IF e[1] is a term THEN RETURN ω(e);
  RETURN e;

TheorySetup_i(e)
  FOREACH a which is an i-leaf in e DO BEGIN
    IF Op(e) = '=' THEN a.notify := a.notify ∪ {⟨UpdateDisequality,e⟩};
    ELSE a.notify := a.notify ∪ {⟨UpdateShostak,e⟩};
    SetShared(a,i);
  END
  IF e is a term THEN SetShared(e,i);

TheoryRewrite_i(e)
  RETURN σ(e);

PropagateEqualities_i()
  RETURN false;

UpdateDisequality(x,y)
  IF ¬Satisfiable() ∨ ¬HasFind(y) THEN RETURN;
  Replace each i-leaf c in y with Find(c);
  y' := Rewrite(y);
  IF y' ≢ false THEN Merge(y' = false);

UpdateShostak(x,y)
  IF Find(y) ≡ y THEN BEGIN
    Replace each i-leaf c in y with Find(c) to get y';
    Merge(y = σ(y'));
  END
```

Fig. 3. Code for implementing a Shostak theory T_i.

Proof. When $\texttt{SetFind}$ is first called on an expression $\texttt{e}$, the $\texttt{Merge}$ preconditions together with the solver and canonizer guarantee that $\texttt{e} \equiv \sigma(\tau(\texttt{e}))$. Then, whenever an i-leaf is merged, $\texttt{UpdateShostak}$ is called to preserve the invariant. $\square$

Lemma 14. *If the program is in an uncorrupted and consistent state in the user code, and T_i is a Shostak theory, then $T_i \cup \gamma_i(\mathcal{E})$ is satisfiable.*

Proof. Let M be a model of T_i, and let $\texttt{x} \in \Delta_i$. If $\texttt{Find(x)} \equiv \texttt{x}$, then assign $v(\texttt{x})$ an arbitrary value. Otherwise, assign $v(\texttt{x})$ the same value as $\gamma_i(\texttt{Find}(\texttt{x}))$. By the above lemma, this assignment satisfies $\gamma_i(\mathcal{E})$. $\square$

Lemma 15. *If the program is in an uncorrupted and consistent state in the user code and T_i is a Shostak theory, then $T_i \cup \gamma_i(\mathcal{M}_i)$ is satisfiable.*

Proof. Suppose $\texttt{e} \in \mathcal{M}_i$. Clearly $\texttt{e}[1] \sim \texttt{e}[2]$. If $\texttt{e}$ is an equality between terms, it follows from Lemma 13 that $\sigma(\tau(\texttt{e}[1])) \equiv \sigma(\tau(\texttt{e}[2]))$. By properties of σ, it follows that $T_i \models \tau(\texttt{e}[1]) = \tau(\texttt{e}[2])$. Then, by the definition of $\mathcal{E}$, it follows that $T_i \cup \mathcal{E} \models \texttt{e}[1] = \texttt{e}[2]$ and hence $T_i \cup \gamma_i(\mathcal{E}) \models \gamma_i(\texttt{e}[1] = \texttt{e}[2])$. Suppose on the other hand that $\texttt{e}$ is the literal $(\texttt{x} = \texttt{y}) = \textbf{false}$, and suppose that $T_i \cup \gamma_i(\mathcal{E}) \models \gamma_i(\texttt{x} = \texttt{y})$. The same argument as above in reverse shows that $\texttt{Find(x)} \equiv \texttt{Find(y)}$. The $\texttt{UpdateDisequality}$ code ensures that in this case **true** will get merged with **false**, contradicting the assumption that the state is consistent. Thus, $T_i \cup \gamma_i(\mathcal{E}) \not\models \gamma_i(\texttt{x} = \texttt{y})$. Since T_i is convex, it follows that $T_i \cup \gamma_i(\mathcal{E} \cup \mathcal{M}_i)$ is satisfiable. $\square$

Theorem 9. *If the program is in an uncorrupted and consistent state in the user code and T_i is a Shostak theory, then the completeness property holds for T_i.*

Proof. The above lemma shows that $T_i \cup \gamma_i(\mathcal{E} \cup \mathcal{M}_i)$ is satisfiable. Suppose $\texttt{a}$ and $\texttt{b}$ are shared terms. If $\texttt{a} \sim \texttt{b}$, a similar argument to that given above shows that $T_i \cup \gamma_i(\mathcal{E}) \models \gamma_i(\texttt{a} = \texttt{b})$. If, on the other hand $\texttt{a} \not\sim \texttt{b}$, it follows easily that $T_i \cup \gamma_i(\mathcal{E}) \not\models \gamma_i(\texttt{a} = \texttt{b})$. Since each equality in $\gamma_i(\mathcal{M}_i \cup \pi(V))$ is entailed by $T_i \cup \gamma_i(\mathcal{E})$ and none of the disequalities are, it follows by convexity that $T_i \cup \gamma_i(\mathcal{M}_i \cup \pi(V))$ is satisfiable. $\square$

Termination. The idempotency of the solver and canonizer are sufficient to guarantee termination of rewrites. For each expression $\texttt{e}$, it is not hard to show that something is added to $\texttt{e.notify}$ only if $\texttt{Find(e)} \equiv \texttt{e}$. Consider the functions called by $\texttt{Merge}$ which are $\texttt{UpdateDisequality}$ and $\texttt{UpdateShostak}$. Both of them call $\texttt{Merge}$ recursively. Each of them reduce the value of some measure of the program state. For $\texttt{UpdateDisequality}$, the measure is the number of equality expressions $\texttt{e}$ such that $\texttt{HasFind(e)}$ and $\omega(\tau(\texttt{e})) \not\equiv \textbf{false}$. For $\texttt{UpdateShostak}$, the measure is the number of expressions $\texttt{e}$ such that $\texttt{Find(e)} \equiv \texttt{e}$ and $\texttt{Find(c)}$ $\not\equiv \texttt{c}$ for some i-leaf $\texttt{c}$ of $\texttt{e}$. With some effort, it can be verified that none of the functions in the theory-specific code presented thus far which can be called after $\texttt{Union}$ increase either of these measures. The other termination conditions are trivial.

Finally, in order to combine Shostak and Nelson-Oppen, the Shostak code must not break the Nelson-Oppen Termination Requirement. Any new call to `Merge` has the potential to "create" new shared terms by causing a new term to show up in $\mathcal{M}_i$ for some i. A careful analysis shows that if `Assert(x = y)` is called from the Nelson-Oppen code, any resulting call to `Merge` does not increase the number of equivalence classes containing shared terms. Lemma 13 ensures that by the time `Assert` has returned, $\mathtt{x} \sim \mathtt{y}$, so the number of equivalence classes containing shared terms decreases as required.

6 Conclusion

We have presented a framework for combining decision procedures for disjoint first-order theories, and shown how it can be used to implement and integrate Nelson-Oppen and Shostak style decision procedures.

This work has shed considerable light on the individual methods as well as on what is required to combine them. We discovered that a more restricted set of equalities can be propagated in the Nelson-Oppen framework without losing completeness. Also, by separating the uninterpreted functions from the Shostak method, the code is simpler and easier to verify.

We are working on an extension of the framework which would handle non-convex theories and more general Shostak solvers. In future work, we hope also to be able to relax the requirements that the theories be disjoint and stably-infinite. We also plan to complete and distribute a new version of SVC based on these results.

Acknowledgments

We would like to thank Natarajan Shankar at SRI for helpful discussions and insight into Shostak's decision procedure. This work was partially supported by the National Science Foundation Grant MIPS-9806889 and NASA contract NASI-98139. The third author is supported by a National Science Foundation Graduate Fellowship.

References

1. F. Baader and C. Tinelli. A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method. In W. McCune, editor, *14th International Conference on Computer Aided Deduction*, Lecture Notes in Computer Science, pages 19–33. Springer-Verlag, 1997.
2. Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 1996.
3. N. Bjorner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.

4. Michael A. Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.
5. D. Cyrluk. Private communication. 1999.
6. D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combinations of Theories. In M. McRobbie and J. Slaney, editors, *13th International Conference on Computer Aided Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477. Springer-Verlag, 1996.
7. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*, pages 160–172. Springer-Verlag, July 1999. Trento, Italy.
8. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, , and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.
9. Z. Manna et al. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.
10. H.Saidi and N.Shankar. Abstract and model check while you prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.
11. J. Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, 1999.
12. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
13. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.
14. David Y.W. Park, Jens U. Skakkebæk, Mats P.E. Heimdahl, Barbara J. Czerny, and David L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98: Second Workshop on Formal Methods in Software Practice*, pages 34–43, March 1998.
15. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, volume 8, pages 102–114, August 1992.
16. H. Ruess and N. Shankar. Deconstructing Shostak. In *17th International Conference on Computer Aided Deduction*, 2000.
17. R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.
18. J. Su, D. Dill, and J. Skakkebæk. Formally verifying data and control with weak reachability invariants. In *Formal Method In Computer-Aided Design*, 1998.
19. C. Tinelli and M. Harandi. A new Correctness Proof of the Nelson-Oppen Combination Procedure. In F. Baader and K. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (FroCoS'96)*, volume 3 of *Applied Logic Series*. Kluwer Academic Publishers, 1996.