Satisfiability Modulo Theories in Practice

Clark Barrett

barrett@cs.nyu.edu

New York University

Motivation

Automatic analysis of computer hardware and software requires engines capable of reasoning efficiently about large and complex systems.

Boolean engines such as Binary Decision Diagrams and SAT solvers are typical engines of choice for today's industrial verification applications.

However, systems are usually designed and modeled at a higher level than the Boolean level and the translation to Boolean logic can be expensive.

A primary goal of research in Satisfiability Modulo Theories (SMT) is to create verification engines that can reason natively at a higher level of abstraction, while still retaining the speed and automation of today's Boolean engines.

Modeling Reactive Systems Using CVC3

Consider the following lines of code:

$$
\begin{array}{lcl}
l_0 & : & a[i] := a[i] + 1; \\
l_1 & : & a[i+1] := a[i-1] - 1; \\
l_2 & : &
\end{array}
$$

This can be modeled in CVC3 [BT07] as follows:

```
i0, i1, i2 : INT;
a0, a1, a2 : ARRAY INT OF INT;

ASSERT (a1 = a0 WITH [i0] := a0[i0]+1) AND (i1 = i0);
ASSERT (a2 = a1 WITH [i1+1] := a1[i1-1]-1) AND (i2 = i1);
```

Modeling Reactive Systems Using CVC3

To check whether the result is equivalent when the two statements are swapped, we can use the following CVC3 QUERY.

```
i0, i1, i2, i3, i4 : INT;
a0, a1, a2, a3, a4 : ARRAY INT OF INT;

ASSERT (a1 = a0 WITH [i0] := a0[i0]+1) AND (i1 = i0);
ASSERT (a2 = a1 WITH [i1+1] := a1[i1-1]-1) AND (i2 = i1);

ASSERT (a3 = a0 WITH [i0+1] := a0[i0-1]-1) AND (i3 = i0);
ASSERT (a4 = a3 WITH [i3] := a3[i3]+1) AND (i4 = i3);

QUERY (i2 = i4 AND a2 = a4);
```

Modeling Reactive Systems Using CVC3

A more efficient encoding ignores variables that do not change and uses the LET construct to introduce temporary expressions.

```
i : INT;
a : ARRAY INT OF INT;

QUERY
 (LET a1 = a WITH [i] := a[i]+1 IN
      a1 WITH [i+1] := a1[i-1]-1) =
 (LET a1 = a WITH [i+1] := a[i-1]-1 IN
      a1 WITH [i] := a1[i]+1);
```

Modeling Reactive Systems Using CVC3

Because CVC3 includes arrays, arithmetic, and propositional reasoning, most of the operations found in real programs can be modeled precisely.

Operations not supported by CVC3 can often be modeled as uninterpreted functions.

By modeling memory as an array, pointers can also be dealt with.

Practical Issues for Combinations of Theories

Cesare Tinelli focused on a clean theoretical presentation of theory combination.

In this talk, we will try to answer some of the questions that arise when these procedures are implemented.

As usual, the devil is in the details.

We will focus on the quantifier-free T-satisfiability problem, where T is made up of the union of theories whose signatures are disjoint.

The reason for this focus is that it is still the most common case and the lessons learned are often helpful in more general cases.

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
  - Equality (with “uninterpreted” functions)
  - Arithmetic
  - Arrays
- Shostak’s Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
  - What is Shostak's method?
  - Shostak vs Nelson-Oppen
  - Practical lessons from Shostak
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
  - Eliminating the purification step
  - Integrating rewriting
  - Efficiently searching arrangements
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
  - The importance of being minimal
  - The importance of being eager
  - Decision heuristics
  - SAT heuristics and completeness
  - Non-convexity issues
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Roadmap

- Some Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Theories

Recall that a theory is a set of first-order sentences.

For our purposes, we will define a theory as a set of first-order sentences closed under logical implication.

Define the set $Cn\ \Gamma$ of consequences of Γ to be $\{\sigma \mid \Gamma \models \sigma\}$.

Then T is a theory iff T is a set of sentences and $T = Cn\ T$.

The T-satisfiability problem consists of deciding whether there exists a structure $\mathcal{A}$ and variable assignment α such that $(\mathcal{A}, \alpha) \models T \cup \varphi$ for an arbitrary first-order formula φ.

The quantifier-free T-satisfiability problem restricts φ to be quantifier-free.

Theories

We will start by looking at a few specific interesting theories.

A good reference on many of these theories is [MZ03].

The Theory $T_{\mathcal{E}}$ of Equality

The theory $T_{\mathcal{E}}$ of equality is the theory $Cn\ \emptyset$.

Note that the exact set of sentences in $T_{\mathcal{E}}$ depends on the signature in question.

Note also that the theory does not restrict the possible values of the symbols in its signature in any way. For this reason, it is sometimes called the theory of equality with uninterpreted functions (EUF).

The satisfiability problem for $T_{\mathcal{E}}$ is just the satisfiability problem for first-order logic, which is undecidable.

The satisfiability problem for conjunctions of literals in $T_{\mathcal{E}}$ is decidable in polynomial time using congruence closure.

The Theory $T_{\mathcal{Z}}$ of Integers

Let $\Sigma_{\mathcal{Z}}$ be the signature $(0, 1, +, -, \leq)$.

Let $\mathcal{A}_{\mathcal{Z}}$ be the standard model of the integers with domain $\mathcal{Z}$.

Then $T_{\mathcal{Z}}$ is defined to be the set of all $\Sigma_{\mathcal{Z}}$-sentences true in the model $A_{\mathcal{Z}}$.

As showed by Presburger in 1929, the general satisfiability problem for $T_{\mathcal{Z}}$ is decidable, but its complexity is triply-exponential.

The quantifier-free satisfiability problem for $T_{\mathcal{Z}}$ is "only" NP-complete.

The Theory $T_{\mathcal{Z}}$ of Integers

Let $\Sigma_{\mathcal{Z}}^{\times}$ be the same as $\Sigma_{\mathcal{Z}}$ with the addition of the symbol $\times$ for multiplication, and define $\mathcal{A}_{\mathcal{Z}}^{\times}$ and $T_{\mathcal{Z}}^{\times}$ in the obvious way.

The satisfiability problem for $T_{\mathcal{Z}}^{\times}$ is undecidable (a consequence of Gödel's incompleteness theorem).

In fact, even the quantifier-free satisfiability problem for $T_{\mathcal{Z}}^{\times}$ is undecidable.

The Theory $T_{\mathcal{R}}$ of Reals

Let $\Sigma_{\mathcal{R}}$ be the signature $(0, 1, +, -, \leq)$.

Let $\mathcal{A}_{\mathcal{R}}$ be the standard model of the reals with domain $\mathcal{R}$.

Then $T_{\mathcal{R}}$ is defined to be the set of all $\Sigma_{\mathcal{R}}$-sentences true in the model $A_{\mathcal{R}}$.

The satisfiability problem for $T_{\mathcal{R}}$ is decidable, but the complexity is doubly-exponential.

The quantifier-free satisfiability problem for conjunctions of literals (atomic formulas or their negations) in $T_{\mathcal{R}}$ is solvable in polynomial time, though exponential methods (like Simplex or Fourier-Motzkin) tend to perform best in practice.

The Theory $T_{\mathcal{R}}$ of Reals

Let $\Sigma_{\mathcal{R}}^{\times}$ be the same as $\Sigma_{\mathcal{R}}$ with the addition of the symbol $\times$ for multiplication, and define $\mathcal{A}_{\mathcal{R}}^{\times}$ and $T_{\mathcal{R}}^{\times}$ in the obvious way.

In contrast to the theory of integers, the satisfiability problem for $T_{\mathcal{R}}^{\times}$ is decidable though the complexity is inherently doubly-exponential.

The Theory $T_{\mathcal{A}}$ of Arrays

Let $\Sigma_{\mathcal{A}}$ be the signature $(\mathit{read}, \mathit{write})$.

Let $\Lambda_{\mathcal{A}}$ be the following axioms:

$$\forall\, a\, \forall\, i\, \forall\, v\, (\mathit{read}\,(\mathit{write}\,(a, i, v), i) = v)$$
$$\forall\, a\, \forall\, i\, \forall\, j\, \forall\, v\, (i \neq j \rightarrow \mathit{read}\,(\mathit{write}\,(a, i, v), j) = \mathit{read}\,(a, j))$$
$$\forall\, a\, \forall\, b\, ((\forall\, i\, (\mathit{read}\,(a, i) = \mathit{read}\,(b, i))) \rightarrow a = b)$$

Then $T_{\mathcal{A}} = \mathit{Cn}\ \Lambda_{\mathcal{A}}$.

The satisfiability problem for $T_{\mathcal{A}}$ is undecidable, but the quantifier-free satisfiability problem for $T_{\mathcal{A}}$ is decidable (the problem is NP-complete).

Theories of Inductive Data Types

A inductive data type (IDT) defines one or more constructors, and possibly also selectors and testers.

Example: *list* **of** *int*

- Constructors: *cons*: $(int, list) \longrightarrow list$, *null*: *list*
- Selectors: *car*: $list \longrightarrow int$, *cdr*: $list \longrightarrow list$
- Testers: *is_cons*, *is_null*

The first order theory of a inductive data type associates a function symbol with each constructor and selector and a predicate symbol with each tester.

Example: $\forall\, x : list.\ (x = null \vee \exists\, y : int, z : list.\ x = cons(y, z))$

Theories of Inductive Data Types

A inductive data type (IDT) defines one or more constructors, and possibly also selectors and testers.

Example: *list* **of** *int*

- Constructors: $cons\colon (int, list) \longrightarrow list$, $null\colon list$
- Selectors: $car\colon list \longrightarrow int$, $cdr\colon list \longrightarrow list$
- Testers: is_cons, is_null

For IDTs with a single constructor, a conjunction of literals is decidable in polynomial time using an algorithm by Oppen [Opp80].

For more general IDTs, the problem is NP complete, but reasonbly efficient algorithms exist in practice [BST06].

Other Interesting Theories

Some other interesting theories include:

- Theories of bit-vectors [CMR97, Möl97, BDL98, BP98, EKM98, GBD05]
- Set theory [CZ00]

Roadmap

- Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Shostak's Method

Despite much progress in our understanding in the last few years, there is still an aura of mystique and misunderstanding around Shostak's method.

In this part of the talk, I hope to clear things up, focusing on what we can learn about practical implementation of decision procedures.

Shostak's Method

In 1984 [Sho84], Robert Shostak published a paper which detailed a particular strategy for deciding satisfiability of quantifier-free formulas in certain kinds of theories.

The original version promises three main things:

1. For theories T which meet the criteria (we will call these Shostak theories), the method gives a decision procedure for quantifier-free T-satisfiability.
2. The method has the theory $T_{\mathcal{E}}$ "built-in", so for any Shostak theory T, the method gives a decision procedure for quantifier-free $T \cup T_{\mathcal{E}}$-satisfiability.
3. Any two Shostak theories T_1 and T_2 can be combined to form a new Shostak theory $T_1 \cup T_2$.

Shostak's Method

Unfortunately, the original paper contains many errors and a number of papers have since been dedicated to correcting them [CLS96, RS01, SR02, KC03, BDS02b, Gan02].

When the dust settled, it finally became clear that the first two claims can be justified, but the last one is false.

Most proponents of the method now agree that any attempt to combine theories is best understood in the context of the Nelson-Oppen method.

However, in this context, there is much that can be learned from Shostak's method.

Shostak's Method

The first helpful insight is how to build a decision procedure for a single Shostak theory.

Recall that the Nelson-Oppen method gives a decision procedure for a combination of theories given decision procedures for the component theories.

However, the Nelson-Oppen method provides no help on how to build the decision procedures for the component theories.

Shostak provides one solution for the special case of Shostak theories.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
   Recall that a theory T is convex if for any conjunction of literals φ and variables $x_1, \ldots x_n, y_1, \ldots, y_n$,
   $T \cup \phi \models x_1 = y_1 \vee \cdots \vee x_n = y_n$ implies
   $T \cup \phi \models x_i = y_i$ for some $1 \leq i \leq n$.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.
   The solver *solve* must be a computable function from Σ-equations to sets of Σ-formulas defined as follows:
   (a) If $T \models a \neq b$, then *solve*$(a = b) \equiv \{\mathit{false}\}$.
   (b) Otherwise, *solve*$(a = b)$ returns a set $\mathcal{E}$ of equations in solved form such that $T \models [(a = b) \leftrightarrow \exists\, \overline{w}.\, \mathcal{E}]$, where $\overline{w}$ are fresh variables not appearing in a or b.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.
   The solver *solve* must be a computable function from Σ-equations to sets of Σ-formulas defined as follows:
   (a) If $T \models a \neq b$, then *solve*$(a = b) \equiv \{\mathit{false}\}$.
   (b) Otherwise, *solve*$(a = b)$ returns a set $\mathcal{E}$ of equations in solved form such that $T \models [(a = b) \leftrightarrow \exists\, \overline{w}.\, \mathcal{E}]$, where $\overline{w}$ are fresh variables not appearing in a or b.

   $\mathcal{E}$ is in solved form iff the left-hand side of each equation in $\mathcal{E}$ is a variable which appears only once in $\mathcal{E}$.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.
   The solver *solve* must be a computable function from Σ-equations to sets of Σ-formulas defined as follows:
   (a) If $T \models a \neq b$, then *solve*$(a = b) \equiv \{\mathit{false}\}$.
   (b) Otherwise, *solve*$(a = b)$ returns a set $\mathcal{E}$ of equations in solved form such that $T \models [(a = b) \leftrightarrow \exists\, \overline{w}.\, \mathcal{E}]$, where $\overline{w}$ are fresh variables not appearing in a or b.

   We denote by $\mathcal{E}(X)$ the result of applying $\mathcal{E}$ as a substitution to X.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.
4. T has a canonizer *canon*.

Shostak Theories

A consistent theory T with signature Σ is a Shostak theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. T is convex.
3. T has a solver *solve*.
4. T has a canonizer *canon*.
   The canonizer *canon* must be a computable function from Σ-terms to Σ-terms with the property that
   $$T \models a = b \text{ iff } \mathit{canon}(a) \equiv \mathit{canon}(b).$$

Algorithm Sh

Algorithm Sh checks the satisfiability in T of a set of equalities, Γ, and an set of disequalities, Δ.

Sh$(\Gamma, \Delta, \mathit{canon}, \mathit{solve})$

1. $\mathcal{E}$:= $\emptyset$;
2. **while** $\Gamma \neq \emptyset$ **do begin**
3. Remove some equality $a = b$ from Γ;
4. a^* := $\mathcal{E}(a)$; b^* := $\mathcal{E}(b)$;
5. $\mathcal{E}^*$:= $\mathit{solve}(a^* = b^*)$;
6. **if** $\mathcal{E}^* = \{\mathit{false}\}$ **then return** *false*;
7. $\mathcal{E}$:= $\mathcal{E}^*(\mathcal{E}) \cup \mathcal{E}^*$;
8. **end**
9. **if** $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$ for some $a \neq b \in \Delta$
   then return *false*;
10. **return** *true*;

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$-x - 3y + 2z = 1$	$-x - 3y + 2z = 1$
$x - y - 6z = 1$	
$2x + y - 10z = 3$	

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$-x - 3y + 2z = 1$	$-x - 3y + 2z = 1$
$x - y - 6z = 1$	
$2x + y - 10z = 3$	

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$x = -3y + 2z + 1$	$-x - 3y + 2z = 1$
$x - y - 6z = 1$	
$2x + y - 10z = 3$	

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$x - y - 6z = 1$ $2x + y - 10z = 3$ $2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$x - y - 6z = 1$	$x = -3y + 2z + 1$
$2x + y - 10z = 3$	
$2x + y - 10z = 3$	

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$-4y - 4z + 1 = 1$	$x = -3y + 2z + 1$
$2x + y - 10z = 3$	
$2x + y - 10z = 3$	

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$y = -z$ $2x + y - 10z = 3$ $2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$y = -z$	$x = 3z + 2z + 1$
$2x + y - 10z = 3$	
$2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$2x + y - 10z = 3$	$x = 5z + 1$
	$y = -z$
$2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$2x + y - 10z = 3$	$x = 5z + 1$
	$y = -z$
$2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$z = -1$	$x = 5z + 1$
	$y = -z$
$2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$z = -1$	$x = 5(-1) + 1$
	$y = -(-1)$
$2x + y - 10z = 3$	$x = -3y + 2z + 1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$2x + y - 10z = 3$	$x = -4$ $y = 1$ $z = -1$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.

Γ	$\mathcal{E}$
$2x + y - 10z = 3$	$x = -4$ $y = 1$ $z = -1$

Note that for this theory, the main loop of Shostak's algorithm is equivalent to Gaussian elimination with back-substitution.

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$x \neq 4y$
$y = 1$	$x + w \neq w + z - 3y$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$x \neq 4y$
$y = 1$	$x + w \neq w + z - 3y$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$-4 \neq 4(1)$
$y = 1$	$x + w \neq w + z - 3y$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$-4 \neq 4$
$y = 1$	$x + w \neq w + z - 3y$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$-4 \neq 4$
$y = 1$	$x + w \neq w + z - 3y$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$-4 \neq 4$
$y = 1$	$-4 + w \neq w + (-1) - 3(1)$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Example

The most obvious example of a Shostak theory is $T_{\mathcal{R}}$ (without $\leq$).

- Step 1: Use the solver to convert Γ into an equisatisfiable set $\mathcal{E}$ of equations in solved form.
- Step 2: Use $\mathcal{E}$ and *canon* to check if any disequality is violated:

  For each $a \neq b \in \Delta$, check if $\mathit{canon}(\mathcal{E}(a)) \equiv \mathit{canon}(\mathcal{E}(b))$.

$\mathcal{E}$	Δ
$x = -4$	$-4 \neq 4$
$y = 1$	$w + (-4) \neq w + (-4)$
$z = -1$	$-4 + w \neq w + (-1) - 3(1)$

Other Shostak Theories

A few other theories can be handled using this algorithm:

- $T_{\mathcal{Z}}$ (without $\leq$) is also a Shostak theory.
- A simple theory of lists (without the NULL list).

However, the idea of using solvers and canonizers can be applied to help decide other theories as well:

- One component in decision procedures for $T_{\mathcal{R}}$ and $T_{\mathcal{Z}}$ (with $\leq$).
- Partial canonizing and solving is useful in $T_{\mathcal{A}}$.
- Partial canonizing and solving is useful for the theory of bit-vectors.

Shostak and Theory Combination

As mentioned, Shostak's second claim is that a combination with $T_{\mathcal{E}}$ can be easily achieved.

The details are a bit technical, and the easiest way to understand it is as a special case of a Nelson-Oppen combination.

The special part is that the abstract Nelson-Oppen is refined in several ways that make implementation easier.

We will look at this next.

Shostak's Method: Summary

Shostak's method provides

- a simple decision procedure for Shostak theories
- insight into the usefulness of solvers and canonizers
- insight into practical ways to refine Nelson-Oppen (next)

Shostak's method does not provide

- a general method for combining theories

Roadmap

- Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Implementing Nelson-Oppen

Let's review the basic Nelson-Oppen procedure.

Suppose that $T_1, \ldots, T_n$ are stably-infinite theories with disjoint signatures $\Sigma_1, \ldots, \Sigma_n$ and Sat_i decides T_i-satisfiability of $\Sigma_i(C)$ literals.

We wish to determine the satisfiability of a ground conjunction Γ of $\Sigma(C)$-literals.

1. Purify Γ to obtain an equisatisfiable set $\bigwedge \varphi_i$, where each φ_i is i-pure.
2. Let S be the set of shared constants (i.e. appearing in more than one φ_i).
3. For each arrangement Δ of S,

   Check $\mathit{Sat}_i(\varphi_i \wedge \Delta)$ for each i.

Example

Consider the following example in a combination of $T_{\mathcal{E}}$, $T_{\mathcal{Z}}$, and $T_{\mathcal{A}}$:

$$\neg p(y) \wedge s = \mathit{write}\,(t, i, 0) \wedge x - y - z = 0 \wedge$$
$$z + \mathit{read}\,(s, i) = f(x - y) \wedge p(x - f(f(z))).$$

After purification, we have the following:

$\varphi_{\mathcal{E}}$	$\varphi_{\mathcal{Z}}$	$\varphi_{\mathcal{A}}$
$\neg p(y)$	$l - z = j$	$s = \mathit{write}\,(t, i, j)$
$m = f(l)$	$j = 0$	$k = \mathit{read}\,(s, i)$
$p(v)$	$l = x - y$	
$n = f(f(z))$	$m = z + k$	
	$v = x - n$	

Example

$\varphi_{\mathcal{E}}$	$\varphi_{\mathcal{Z}}$	$\varphi_{\mathcal{A}}$
$\neg p(y)$	$l - z = j$	$s = \mathit{write}\,(t, i, j)$
$m = f(l)$	$j = 0$	$k = \mathit{read}\,(s, i)$
$p(v)$	$l = x - y$	
$n = f(f(z))$	$m = z + k$	
	$v = x - n$	

There are 12 constants in this example:

- Shared: l, z, j, y, m, k, v, n
- Unshared: x, s, t, i

There are 21147 arrangements of $\{l, z, j, y, m, k, v, n\}$. Clearly, a practical implementation cannot consider all of these separately.

Implementing Nelson-Oppen

In order to obtain a more practical implementation of Nelson-Oppen, we will consider the following refinements:

- Eliminating the purification step
- Incremental processing with theory-specific rewrites
- Strategies for searching through arrangements

Implementing Nelson-Oppen

As most implementers of SMT systems will tell you, the purification step is not really necessary in practice.

In fact, a simple variation of Nelson-Oppen can be obtained that does not require purification [BDS02b].

Given a set of mixed (impure) literals Γ, define a shared term to be any term in Γ which is alien in some literal or sub-term in Γ.

Note that these are exactly the terms that would have been replaced with new constants in the purification step.

Assume also that each Sat_i is modified so that it treats alien terms as constants.

Implementing Nelson-Oppen

The following is a variation of Nelson-Oppen which does not use purification.

1. Partition Γ into sets φ_i, where each literal in φ_i is an i-literal.
2. Let S be the set of shared terms in Γ.
3. For each arrangement Δ of S,

   Check $\mathit{Sat}_i(\varphi_i \wedge \Delta)$ for each i.

Example

Consider again the example from before:

$$\neg p(y) \wedge s = \mathit{write}\,(t, i, 0) \wedge x - y - z = 0 \wedge$$
$$z + \mathit{read}\,(s, i) = f(x - y) \wedge p(x - f(f(z))).$$

After partitioning, we have the following:

$\varphi_{\mathcal{E}}$	$\varphi_{\mathcal{Z}}$	$\varphi_{\mathcal{A}}$
$\neg p(y)$	$x - y - z = 0$	$s = \mathit{write}\,(t, i, 0)$
$p(x - f(f(z)))$	$z + \mathit{read}\,(s, i) = f(x - y)$	

The shared terms are:

$$\mathit{read}\,(s, i), x - y, f(x - y), 0, y, z, f(f(z)), x - f(f(z)).$$

Unfortunately, there are still too many arrangements.

Implementing Nelson-Oppen

The next refinement is to process Γ incrementally, allowing theory-specific rewrites that can potentially reduce the number of shared terms.

Examples of theory-specific rewrites include canonization or partial canonization and simplification based on previously seen literals.

1. For each $\varphi \in \Gamma$
   (a) (Optionally) apply theory-specific rewrites to φ to get φ'
   (b) Identify the shared terms in φ' and add these to S
   (c) Where φ' is an i-literal, add φ' to φ_i
2. For each arrangement Δ of S,
   Check $\mathit{Sat}_i(\varphi_i \wedge \Delta)$ for each i.

Example

Let's see what happens if we process our example incrementally:

$$\neg p(y) \wedge s = \mathit{write}\,(t, i, 0) \wedge x - y - z = 0 \wedge$$
$$z + \mathit{read}\,(s, i) = f(x - y) \wedge p(x - f(f(z))).$$

$\varphi_{\mathcal{E}}$	$\varphi_{\mathcal{Z}}$	$\varphi_{\mathcal{A}}$
$\neg p(y)$ $p(y)$	$x = y + z$ $z = f(z)$	$s = \mathit{write}\,(t, i, 0)$

The shared terms are: $0, y, z, f(z)$. There are only 52 arrangements now.

More importantly, $\varphi_{\mathcal{E}}$ is now inconsistent in the theory $T_{\mathcal{E}}$, making it unnecessary to examine any arrangements.

Implementing Nelson-Oppen

We have seen two ways to avoid searching through too many arrangements:

1. Reduce the number of shared terms
2. Detect an inconsistency early

As a further example of (2), we can build arrangements incrementally, backtracking if any theory detects an inconsistency.

As described in part I, for convex theories, this strategy is very efficient.

For non-convex theories, we may have to explore the entire search space of arrangements.

Implementing Nelson-Oppen

The strategies we have looked at so far do not assume any help from the theory decision procedures (beyond the ability to determine inconsistency).

If the theory decision procedures are able to give additional information, it may significantly help to prune the arrangement search.

The next refinement of our algorithm captures this.

Implementing Nelson-Oppen

1. For each $\varphi \in \Gamma$
   (a) (Optional) Apply theory-specific rewrites to φ to get φ'
   (b) Identify the shared terms in φ' and add these to S
   (c) Where φ' is an i-literal, add φ' to φ_i
   (d) (Optional) If $\varphi_i \models s_1 = s_2$ or $\varphi_i \models s_1 \neq s_2$, $s_1, s_2 \in S$, add this fact to Γ.
2. Incrementally search through arrangements Δ of S that are consistent with $\bigwedge \varphi_i$. For each arrangement, check $\mathit{Sat}_i(\varphi_i \wedge \Delta)$ for each i.

Implementing Nelson-Oppen

Finally, for maximum efficiency and flexibility, we can push the entire burden of arrangement finding onto the theory decision procedures.

Suppose $\Phi = \bigwedge \varphi_i$ is the partition of literals from Γ that have been processed so far and that S is the set of shared terms.

The equivalence relation R on S induced by Φ is defined as follows: for $x, y \in S$, xRy iff $x = y \in \varphi_i$ for some i.

The arrangement $\Delta(\Phi)$ of S induced by Φ is the arrangement induced by R.

Implementing Nelson-Oppen

For each $\varphi \in \Gamma$:

1. (Optional) Apply theory-specific rewrites to φ to get φ'
2. Identify the shared terms in φ' and add these to S
3. Where φ' is an i-literal, add φ' to φ_i
4. If $\varphi_i \wedge \Delta(\Phi)$ is not satisfiable, compute some formula ψ such that $\varphi_i \models \psi$ and $\psi \wedge \Delta(\Phi)$ is inconsistent. Add ψ to Γ.

Implementing Nelson-Oppen

For each $\varphi \in \Gamma$:

1. (Optional) Apply theory-specific rewrites to φ to get φ'
2. Identify the shared terms in φ' and add these to S
3. Where φ' is an i-literal, add φ' to φ_i
4. If $\varphi_i \wedge \Delta(\Phi)$ is not satisfiable, compute some formula ψ such that $\varphi_i \models \psi$ and $\psi \wedge \Delta(\Phi)$ is inconsistent. Add ψ to Γ.

Some notes:

- In general, ψ does not have to be a literal. In this case, ψ must be processed by the SAT solver (more on this next).
- Theories can be lazy until Γ is empty.
- Termination becomes the responsibility of the theory decision procedures.

Implementing Nelson-Oppen

For each $\varphi \in \Gamma$:

1. (Optional) Apply theory-specific rewrites to φ to get φ'
2. Identify the shared terms in φ' and add these to S
3. Where φ' is an i-literal, add φ' to φ_i
4. If $\varphi_i \wedge \Delta(\Phi)$ is not satisfiable, compute some formula ψ such that $\varphi_i \models \psi$ and $\psi \wedge \Delta(\Phi)$ is inconsistent. Add ψ to Γ.

More notes:

- It is not hard to fit a Shostak-style decision procedure into this framework.
- This is essentially the algorithm used in SVC, CVC, and CVC3.

Roadmap

- Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Efficient Boolean Reasoning

In part I, we saw how SAT reasoning can be combined with theory reasoning.

Let's take a look at some of the practical issues this raises.

Assume that Sat_{FO} is an algorithm for satisfiability of conjunctions of literals in some theory T of interest.

Assume also that we have a propositional abstraction function Abs which replaces each atomic formula α by a propositional variable p_α.

Let's review the "very lazy" approach.

Very Lazy Approach

Suppose we wish to check the T-satisfiability of a quantifier-free formula φ.

1. Check $Abs(\varphi)$ for satisfiability using a Boolean SAT solver.
2. If $Abs(\varphi)$ is unsatisfiable, φ is unsatisfiable.
3. Otherwise, let ψ be a var. assignment satisfying $Abs(\varphi)$.
4. Let $Abs^{-1}(\psi)$ be the corresponding theory literals.
5. If $Sat_{FO}(Abs^{-1}(\psi))$, then φ is satisfiable.
6. Otherwise, refine $Abs(\varphi)$ by adding $\neg\psi$ and repeat.

Since there are only a finite number of possible variable assignments to $Abs(\phi)$, the algorithm will eventually terminate.

Improving the Very Lazy Approach

From a practical point of view, there are a number of issues to be addressed in order to make this algorithm work [BDS02a].

- Minimizing learned clauses
- Lazy vs Eager notification
- Decision heuristics
- Sat heuristics and completeness
- Non-convexity issues

Minimizing Learned Clauses

The main difficulty with the naive approach is that the clauses added in the refinement step can be highly redundant.

Suppose the Boolean abstraction $Abs(\phi)$ contains $n + 2$ propositional variables.

When a Boolean assignment is returned by the SAT solver, all $n + 2$ variables will have an assignment.

But what if only 2 of these assignments are sufficient to result in an inconsistency in the atomic formulas associated with the variables?

In the worst case, 2^n clauses will be added when a single clause containing the two offending variables would have sufficed.

Minimizing Learned Clauses

To avoid this kind of redundancy, the refinement must be more precise.

In particular, when Sat_{FO} is given a set of literals to check for consistency, an effort must be made to find the smallest possible subset of the given set which is inconsistent.

The refinement should then add a clause derived from only these literals.

One way to implement this is to start removing literals from the set and repeatedly call Sat_{FO} until a minimal inconsistent set is found.

However, this is typically too slow to be practical.

Minimizing Learned Clauses

A better, but more difficult way to implement this is to instrument Sat_{FO} to keep track of which facts are used to derive an inconsistency.

If Sat_{FO} happens to be proof-producing, the proof can be used to obtain this information.

This is the approach used in the CVC tools.

Other tools such as Verifun [FJOS03] use a more lightweight approach.

Lazy vs Eager Notification

The naive algorithm does not invoke Sat_{FO} until a complete solution is obtained.

In contrast, an eager notification policy would notify Sat_{FO} immediately of every decision made by the SAT solver.

Experimental results show that the eager approach is significantly better.

Eager notification requires that Sat_{FO} be online: able quickly to determine the consistency of incrementally more or fewer literals.

Eager notification also requires that the SAT solver be instrumented to inform Sat_{FO} every time it assigns a variable.

Naive, Lazy, and Eager Implementations

Example	Naive		Lazy		Eager
	Iterations	Time (s)	Iterations	Time (s)	Time (s)
read0	77	0.14	17	0.09	0.07
pp-pc-s2i	?	$>$ 10000	82	1.36	0.10
pp-invariant	?	$>$ 10000	239	5.81	0.22
v-dlx-pc	?	$>$ 10000	6158	792	3.22
v-dlx-dmem	?	$>$ 10000	?	$>$ 10000	4.12

Decision Heuristics

SAT solvers like Chaff have sophisticated heuristics for determining which variable to split on.

However, for some first-order examples, the structure of the original formula is an important consideration when determining which literal to split on.

For example, many industrial benchmarks make heavy use of the $\mathbf{ite}$ (if-then-else) construct.

Suppose an $\mathbf{ite}$ expression of the form $\mathbf{ite}(\alpha, t_1, t_2)$ appears in the formula being checked.

If α is set to *true*, then all of the literals in t_2 can be ignored since they no longer affect the formula.

Decision Heuristics

Unfortunately, the SAT solver doesn’t know the original structure of the formula and as a result, it can waste a lot of time choosing irrelevant variables.

We found that for such examples, it was better to use a depth-first traversal of the original formula to choose splitters than the built-in SAT heuristic.

Again, this requires tighter integration and communication between the two solvers.

This is far from the final word on decision heuristics. This is an interesting but challenging area for research.

Variable Selection Results

Example	SAT		DFS	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	1309	0.69	2522	1.14
bool-dlx2-aa	4974	2.36	792	0.81
bool-dlx2-cc-bug01	10903	11.4	573387	833
v-dlx-pc	4387	3.22	6137	6.10
v-dlx-dmem	5221	4.12	2184	3.48
v-dlx-regfile	6802	5.85	3833	6.64
dlx-pc	39833	19.0	529	1.04
dlx-dmem	34320	18.8	1276	1.90
dlx-regfile	47822	35.5	2739	4.12
pp-bloaddata-a	8695	5.47	1193	1.80

SAT Heuristics and Completeness

A somewhat surprising observation is that some heuristics used by SAT solvers must be disabled or the method will be incomplete.

The main example of this is the pure literal rule.

This rule looks for propositional variables which are either always (or never) negated in the CNF formula being checked.

These variables can instantly be replaced by *true* (or *false*).

However, if such a variable is an abstraction of a first-order atomic formula, this is no longer the case.

This is because propositional literals are independent of each other, but first-order literals may not be.

Non-Convexity Issues

Consider the following set of literals in the array theory $T_{\mathcal{A}}$:

$$\{\mathit{read}\,(\mathit{write}\,(a, i, v), j) = x, x \neq v, x \neq \mathit{read}\,(a, j)\}.$$

One possible implementation of the array theory decision procedure produces the following fact to "refute" the induced arrangement.

$$\mathit{read}\,(\mathit{write}\,(a, i, v), j) = \mathbf{ite}(i = j, v, \mathit{read}\,(a, j)).$$

This requires the SAT solver to accept new formulas in the middle of a search.

It also complicates decision heuristics.

In this case, $i = j$ is the best next thing to split on.

Roadmap

- Specific Theories
- Shostak's Method
- Implementing Nelson-Oppen
- Efficient Boolean Reasoning
- Reasoning about Quantifiers

Quantifiers

The Abstract DPLL Modulo Theories framework can be easily extended to include rules for quantifier instantiation [GBT07].

- First, we extend the notion of literal to that of an abstract literal which may include quantified formulas in place of atomic formulas.
- Add two additional rules:

Inst_$\exists$:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, (\neg\exists\, x.\, P \vee P[x/sk]) \quad \textbf{if} \begin{cases} \exists\, x.\, P \text{ is an abstract literal in } M \\ sk \text{ is a fresh constant.} \end{cases}$$

Inst_$\forall$:

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, (\neg\forall\, x.\, P \vee P[x/t]) \quad \textbf{if} \begin{cases} \forall\, x.\, P \text{ is an abstract literal in } M \\ t \text{ is a ground term.} \end{cases}$$

An Example

Suppose a and b are constant symbols and f is an uninterpreted function symbol. We show how to prove the validity of the following formula:

$$(0 \leq b \wedge (\forall\, x.\, 0 \leq x \rightarrow f(x) = a)) \rightarrow f(b) = a$$

We first negate the formula and put it into abstract CNF. The result is three unit clauses:

$$(0 \leq b) \wedge (\forall\, x.\, (\neg 0 \leq x \vee f(x) = a)) \wedge (\neg f(b) = a)$$

An Example

Let l_1, l_2, l_3 denote the three abstract literals in the above clauses. Then the following is a derivation in the extended framework:

$$\begin{array}{rcll} \emptyset & \| & (l_1)(l_2)(l_3) & \Longrightarrow (\text{UnitProp}) \\ l_1, l_2, l_3 & \| & (l_1)(l_2)(l_3) & \Longrightarrow (\text{Inst\_}\forall) \\ l_1, l_2, l_3 & \| & (l_1)(l_2)(l_3)(\neg(0 \leq b) \vee f(b) = a) & \Longrightarrow (\text{Fail}) \\ fail & & & \end{array}$$

The last transition is possible because M falsifies the last clause in F and contains no decisions (case-splits). As a result, we may conclude that the original set of clauses is unsatisfiable, which implies that the original formula is valid.

Conclusion: SAT vs SMT

Currently, SAT solvers are much more mature than SMT solvers.

However, SMT solvers are starting to catch up:

- Annual SMT-COMP competition has helped spur innovation
- New breed of solvers much faster than before: Yices, Barcelogic Tools, CVC3, etc.

My opinion: SMT solvers will emerge as the verification engine of choice for many applications.

References

[ARR03] A. Armando, S. Ranise, and M. Rusinowitch. The rewriting approach to satisfi ability procedures. *Information and Computation*, 183:140–164, 2003

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the* 35^{th} *Design Automation Conference (DAC '98)*, pages 522–527. Association for Computing Machinery, June 1998. San Francisco, California. *Best paper award*

[BDS02a] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfi ability of fi rst-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002. Copenhagen, Denmark

[BDS02b] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In Alessandro Armando, editor, *Proceedings of the* 4^{th} *International Workshop on Frontiers of Combining Systems (FroCoS '02)*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 132–146. Springer-Verlag, April 2002. Santa Margherita Ligure, Italy

References

[BP98] Nikolaj Bjørner and Mark C. Pichora. Deciding fi xed and non-fi xed size bit-vectors. In *TACAS '98: Proceedings of the 4^{th} International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 376–392. Springer-Verlag, 1998

[BST06] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfi ability in the theory of recursive data types. In *Proceedings of the 4^{th} Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '06)*, August 2006. Seattle, Washington

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07)*, Lecture Notes in Computer Science. Springer-Verlag, July 2007. Berlin, Germany

[CLS96] 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

[CMR97] David Cyrluk, M. Oliver Möller, and Harald Ruess. An effi cient decision procedure for the theory of fi xed-size bit-vectors. In *Proceedings of the 9^{th} International Conference on Computer Aided Verification (CAV '97)*, pages 60–71. Springer-Verlag, 1997

References

[CZ00] Domenico. Cantone and Calogero G. Zarba. A new fast tableau-based decision procedure for an unquantifi ed fragment of set theory. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *Lecture Notes in Artificial Intelligence*, pages 127–137. Springer, 2000

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Möller. Mona 1.x: New techniques for WS1S and WS2S. In *Proceedings of the* 10^{th} *International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explanation. In *Procs. 15th Int. Conf. on Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, 2003

[Gan02] Harald Ganzinger. Shostak light. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2002

References

[GBD02] Vijay Ganesh, Sergey Berezin, and David L. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In M.D. Aagaard and J.W. O'Leary, editors, *4th International Conference FMCAD'02*, volume 2517 of *Lecture Notes in Computer Science*, pages 171–186, Portland OR USA, November 2002. Springer Verlag

[GBD05] Vijay Ganesh, Sergey Berezin, and David L. Dill. A decision procedure for fi xed-width bit-vectors, January 2005. Unpublished Manuscript

[GBT07] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantifi ed verifi cation condisions using satisfi ability modulo theories. In *Proceedings of the* 21^{st} *International Conference on Automated Deduction (CADE '07)*, Lecture Notes in Artifi cial Intelligence. Springer-Verlag, July 2007. Bremen, Germany

[KC03] Sava Krstic and Sylvain Conchon. Canonization for disjoint union of theories. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Deduction (CADE '03)*, 2003

References

[Lev99] Jeremy Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, 1999

[Möl97] M. Oliver Möller. *Solving Bit-Vector Equations – a Decision Procedure for Hardware Verification*. PhD thesis, University of Ulm, 1997

[MZ03] Zohar Manna and Calogero Zarba. Combining decision procedures. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*, volume 2787 of *Lecture Notes in Computer Science*, pages 381–422. Springer-Verlag, November 2003

[Opp80] Derek C. Oppen. Reasoning about recursively defi ned data structures. *J. ACM*, 27(3):403–411, 1980

References

[RS01] H. Ruess and N. Shankar. Deconstructing shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, June 2001

[SDBL01] Aaron Stump, David L. Dill, Clark W. Barrett, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the* 16^{th} *IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 29–37. IEEE Computer Society, June 2001. Boston, Massachusetts

[Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *Int'l Conf. Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002