Theory Solvers in SMT

Clark Barrett

barrett@cs.nyu.edu

New York University

Roadmap

Theory Solvers in Satisfiability Modulo Theories

- *Theories*
- Theory Solvers
- Examples of Theory Solvers
- Modeling Software using SMT

Theories

Recall that a theory is a set of sentences (formulas with no free variables) that can be used to restrict the models we wish to consider.

In principle, SMT can be applied to any theory T.

In practice, when people talk about SMT, they are usually referring to a small set of specific theories.

We will consider a few examples of theories which are of particular interest in program analysis and verification applications [MZ03].

All of these assume first order logic *with equality*.

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

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

The theory does not restrict the possible values of symbols 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 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 $T_{\mathcal{Z}}^{\times}$ (equivalent to Diophantine Equations) is undecidable.

Note that this was Hilbert's 10th problem - and the negative answer was the result of work by Davis, Putnam, Robinson, and (most famously) Matiyasevich.

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 the Simplex method which is worst-case exponential performs well 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 $(\mathit{select}, \mathit{store})$.

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

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

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

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

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

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

The *first order theory* of an 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

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

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

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

For IDTs with a single constructor, a conjunction of literals is decidable in polynomial time [Opp80].

For more general IDTs, the problem is NP complete, but reasonbly efficient algorithms exist in practice [ZSM04a, ZSM04b, BST07].

The Theory T_{BV} of Bit-vectors

The theory of bit-vectors is a very active area of research (see e.g. [GD07, BB09, HBJ+14]).

In this theory, each variable represents a fixed-length vector of bits, and the signature includes operations commonly found in software and hardware.

The Theory T_{BV} of Bit-vectors

constants	$\mathbf{0} :: [1], \mathbf{1} :: [1]$
concat	$\_ \circ \_ :: [m], [n] \to [m+n]$ for all $m, n \geq 0$
extract	$\_[i:j] :: [m] \to [i-j+1]$ for all $m > i, j \geq 0$ with $i - j \geq -1$
and	$\_ \,\&\, \_ :: [n], [n] \to [n]$ for all $n \geq 0$
or	$\_ \mid \_ :: [n], [n] \to [n]$ for all $n \geq 0$
exclusive or	$\_ \oplus \_ :: [n], [n] \to [n]$ for all $n \geq 0$
not	$\sim\_ :: [n] \to [n]$ for all $n \geq 0$
plus	$\_ + \_ :: [n], [n] \to [n]$ for all $n \geq 0$
times	$\_ \cdot \_ :: [n], [n] \to [n]$ for all $n \geq 0$
shift left	$\_ \ll \_ :: [n], [n] \to [n]$ for all $n \geq 0$
shift right	$\_ \gg \_ :: [n], [n] \to [n]$ for all $n \geq 0$
equal	$\_ \approx \_ :: [n], [n]$ for all $n \geq 0$
less	$\_ < \_ :: [n], [n]$ for all $n \geq 0$

The Theory T_{BV} of Bit-vectors

The theory is decidable because all of its models are finite.

Efficiency is another matter of course.

Practical procedures for the theory are usually based on reductions to SAT.

Other Interesting Theories

Some other interesting theories include:

- Fragments of set theory [CZ00]
- Theories of pointers and reachability: [RBH07, YRS+06, LQ08]
- Theories of strings: [LRT+14, LTR+15]

Roadmap

Theory Solvers in Satisfiability Modulo Theories

- Theories
- *Theory Solvers*
- Examples of Theory Solvers
- Modeling Software using SMT

Theory Solvers

Given a theory T, a *Theory Solver* for T takes as input a set (interpreted as an implicit conjunction) Φ of literals and determines whether Φ is T-satisfiable.

Φ is T-satisfiable iff there is some model M of T such that each formula in Φ holds in M.

In order to integrate a theory solver into a modern SMT solver, it is helpful if the theory solvers can do more than just check satisfiability.

Desirable Characteristics of Theory Solvers

Some desirable characterstics of theory solvers include:

- *Incrementality* - easy to add new literals or backtrack to a previous state
- *Layered/Lazy* - able to detect simple inconsistencies quickly, able to detect difficult inconsistencies eventually
- *Equality Propagating* - If theory solvers can detect when two terms are equivalent, this greatly simplifies theory combination (more about this next time)

Desirable Characteristics of Theory Solvers

Some desirable characterstics of theory solvers include:

- *Model Generating* - When reporting satisfiable, the theory solver also provides a concrete value for each variable or function symbol
- *Proof Generating* - When reporting unsatisfiable, the theory solver also provides a checkable proof
- *Interpolant Generating* - If $\phi \wedge \neg\psi$ is unsatisfiable, find a formula α containing only symbols appearing in both ϕ and ψ such that:
  - $\phi \wedge \neg\alpha$ is unsatisfiable
  - $\alpha \wedge \neg\psi$ is unsatisfiable

Roadmap

Theory Solvers in Satisfiability Modulo Theories

- Theories
- Theory Solvers
- *Examples of Theory Solvers*
- Modeling Software using SMT

Examples of Theory Solvers

We will look at a couple of simple examples of theory solvers

Congruence Closure and $T_{\mathcal{E}}$

Recall that $T_{\mathcal{E}}$ is the theory with only *equality* and *uninterpreted function* symbols.

If Γ is a set of *equalities* and Δ is a set of *disequalities*, then the satisfiability of $\Gamma \cup \Delta$ in $T_{\mathcal{E}}$ can be determined as follows [NO80, DST80]:

- Let τ be the set of terms appearing in $\Gamma \cup \Delta$.
- Let $\sim$ be the equiavlence relation on τ induced by Γ (i.e. $t_1 \sim t_2$ iff $t_1 = t_2 \in \Gamma$ or $t_2 = t_1 \in \Gamma$).
- Let $\sim^*$ be the *congruence closure* of $\sim$, obtained by closing $\sim$ with respect to the congruence property:

$$\overline{s} = \overline{t} \rightarrow f(\overline{s}) = f(\overline{t}).$$

- $\Gamma \cup \Delta$ is satisfiable iff for each $s \neq t \in \Delta$, $s \not\sim^* t$.

A Solver for $T_{\mathcal{E}}$

union and *find* are abstract operations for manipulating equivalence classes.

union(x, y) makes y the new equivalence class representative for x.

find(x) returns the unique representative for the equivalence class containing x.

The *signature* of a term is defined as:
sig$(f(x_1, \ldots, x_n)) = f($*find*$(x_1), \ldots,$ *find*$(x_n))$.

A Solver for $T_{\mathcal{E}}$

```
CC(Γ, Δ)
  while Γ ≠ ∅
    Remove some equality a = b from Γ;
    Merge(find(a), find(b));
  if find(a) = find(b) for some a ≠ b ∈ Δ then
    return false;
  return true;
Merge(a, b)
  if a = b then return;
  Let A be the set of terms containing
      a as an argument
  union(a, b);
  foreach x ∈ A
    if sig(x) = sig(y) for some y then
      Merge(find(x), find(y));
```

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	$f(f(a))$	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(f(f(a)))$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	$f(f(a))$	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(f(f(a)))$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(f(f(a)))$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(f(f(a)))$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(f(f(a)))$	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(a)$	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(a)$	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	$f(a)$	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	$f(a)$	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(f(a))$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(f(a),b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(a,b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(f(a),b)$	$g(a,b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(a,b)$	$g(a,b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(a,b)$	$g(a,b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	$\mathit{find}(t)$	$\mathit{sig}(t)$
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(a,b)$	$g(a,b)$

Example

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ g(a,b) \neq g(f(a),b)$$

t	*find*(t)	*sig*(t)
a	a	a
$f(a)$	a	$f(a)$
$f(f(a))$	a	$f(a)$
$f(f(f(a)))$	a	$f(a)$
b	b	b
$g(a,b)$	$g(a,b)$	$g(a,b)$
$g(f(a),b)$	$g(a,b)$	$g(a,b)$

find$(g(a,b)) =$ *find*$(g(f(a),b)) \rightarrow$ *Unsatisfiable*

Difference Logic

In *difference logic* [NO05], we are interested in the satisfiability of a conjunction of arithmetic atoms.

Each atom is of the form $x - y \bowtie c$, where x and y are variables, c is a numeric constant, and $\bowtie \in \{=, <, \leq, >, \geq\}$.

The variables can range over either the *integers* (QF_IDL) or the *reals* (QF_RDL).

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \quad \Longrightarrow \quad x - y \leq c \ \wedge \ x - y \geq c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \implies x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \implies y - x \leq -c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \implies x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \implies y - x \leq -c$
- $x - y > c \implies y - x < -c$

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \implies x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \implies y - x \leq -c$
- $x - y > c \implies y - x < -c$
- $x - y < c \implies x - y \leq c - 1$ (integers)

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \implies x - y \leq c \wedge x - y \geq c$
- $x - y \geq c \implies y - x \leq -c$
- $x - y > c \implies y - x < -c$
- $x - y < c \implies x - y \leq c - 1$ (integers)
- $x - y < c \implies x - y \leq c - \delta$ (reals)

Difference Logic

Now we have a conjunction of literals, all of the form $x - y \leq c$.

From these literals, we form a weighted directed graph with a vertex for each variable.

For each literal $x - y \leq c$, there is an edge $x \xrightarrow{c} y$.

The set of literals is satisfiable iff there is no cycle for which the sum of the weights on the edges is negative.

There are a number of efficient algorithms for detecting negative cycles in graphs [CG96].

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

$$\begin{array}{l} x - y = 5 \\ z - y \geq 2 \\ z - x > 2 \\ w - x = 2 \\ z - w < 0 \end{array} \quad \Rightarrow \quad w - x \leq 2 \wedge x - w \leq -2$$

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

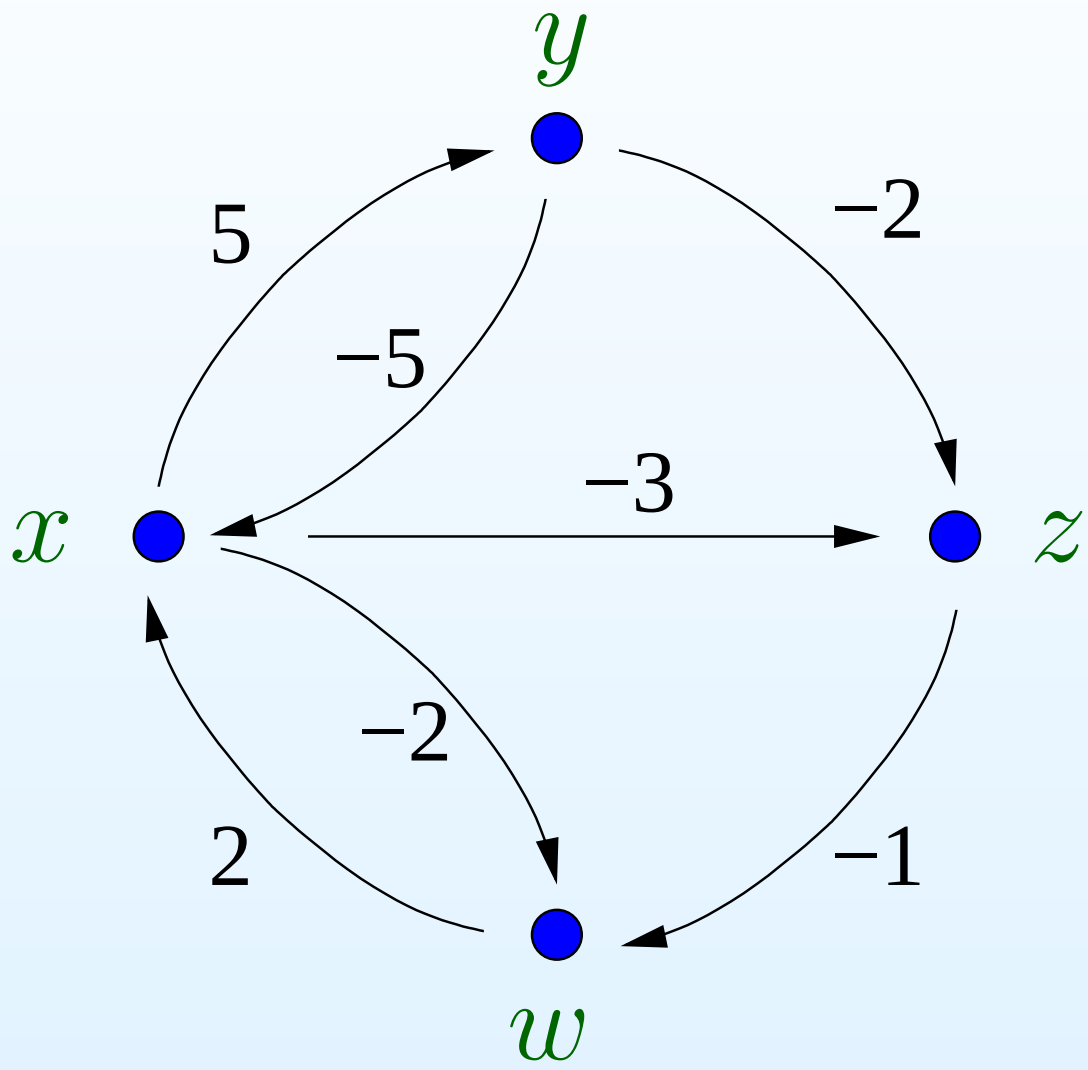
$$\begin{array}{l} x - y = 5 \\ z - y \geq 2 \\ z - x > 2 \quad \Rightarrow \\ w - x = 2 \\ z - w < 0 \end{array} \qquad w - x \leq 2 \wedge x - w \leq -2$$

Example: QF_IDL

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

$$\begin{array}{lcl} x - y = 5 & & x - y \leq 5 \wedge y - x \leq -5 \\ z - y \geq 2 & & y - z \leq -2 \\ z - x > 2 & \Rightarrow & x - z \leq -3 \\ w - x = 2 & & w - x \leq 2 \wedge x - w \leq -2 \\ z - w < 0 & & z - w \leq -1 \end{array}$$

Example: QF_IDL



Roadmap

Theory Solvers in Satisfiability Modulo Theories

- Theories
- Theory Solvers
- Examples of Theory Solvers
- *Modeling Software using SMT*

Modeling Software using SMT

Consider the following code:

```
void swap(int& x, int& y) {
  int x0 = x, y0 = y;
  x = x + y;
  y = x - y;
  x = x - y;
  assert(x0 == y && y0 == x);
}
```

A naive way to model using SMT:

- Describe how each statement changes the state of the program
- Do this by writing equations relating the pre-statement state and the post-statement state
- Requires a fresh copy of state (program) variables between every program statement

Modeling Software using SMT

```
x0, x1, x2, x3 : BITVECTOR(32);
y0, y1, y2, y3 : BITVECTOR(32);

ASSERT x1 = BVPLUS(32, x0, y0) AND y1 = y0;
ASSERT x2 = x1 AND y2 = BVSUB(32, x1, y1);
ASSERT x3 = BVSUB(32, x2, y2) AND y3 = y2;

QUERY x0 = y3 AND y0 = x3;
```

Modeling Software using SMT

For comparison, here is the same thing using SMT-LIB syntax:

```
(set-logic QF_BV)
(declare-const x0 (_ BitVec 32))
(declare-const x1 (_ BitVec 32))
(declare-const x2 (_ BitVec 32))
(declare-const x3 (_ BitVec 32))
(declare-const y0 (_ BitVec 32))
(declare-const y1 (_ BitVec 32))
(declare-const y2 (_ BitVec 32))
(declare-const y3 (_ BitVec 32))

(assert (and (= x1 (bvadd x0 y0)) (= y1 y0)))
(assert (and (= x2 x1) (= y2 (bvsub x1 y1))))
(assert (and (= x3 (bvsub x2 y2)) (= y3 y2)))

(assert (not (and (= x0 y3) (= y0 x3))))
(check-sat)
```

Modeling Software using SMT

A more efficient encoding ignores variables that do not change and uses the *let* construct to introduce temporary expressions:

```
x0, y0 : BITVECTOR(32);
QUERY LET x1 = BVPLUS(32, x0, y0) IN
      LET y1 = BVSUB(32, x1, y0) IN
      LET x2 = BVSUB(32, x1, y1) IN
      x0 = y1 AND y0 = x2;
```

Modeling Software using SMT

Again, here is the SMT-LIB for comparison.

```
(set-logic QF_BV)
(declare-const x0 (_ BitVec 32))
(declare-const y0 (_ BitVec 32))
(assert (let ((x1 (bvadd x0 y0)))
        (let ((y1 (bvsub x1 y0)))
        (let ((x2 (bvsub x1 y1)))
        (not (and (= x0 y1) (= y0 x2)))))))
(check-sat)
```

Modeling Software using SMT

So far, we have treated every program variable independently. This is fine as long as there are no pointers.

Pointers require a more sophisticated model. We will discuss three [WBW16]:

- Flat model
- Burstall model
- Partitioned model

Flat Memory Model

A single array is used to model all of memory. Every variable is modeled as an entry in the array.

- Let `M` be the memory array
- The value of a variable is given by its entry in the array, e.g. `M[x]`
- If `p` is a pointer, then its value is `M[p])` and the value of `*p` is `M[M[p]]`.
- Writes to memory are modeled using the array store operation.

Modeling Software using SMT

Consider again the `swap` code:

```
void swap(int& x, int& y) {
  int x0 = x, y0 = y;
  x = x + y;
  y = x - y;
  x = x - y;
  assert(x0 == y && y0 == x);
}
```

This time, we will model `x` and `y` as locations in memory.

Example

```
M0 : ARRAY BITVECTOR(32) OF BITVECTOR(32);
x, y : BITVECTOR(32);

QUERY LET M1 = M0 WITH [x] := BVPLUS(32, M0[x], M0[y]) IN
      LET M2 = M1 WITH [y] := BVSUB(32, M1[x], M1[y]) IN
      LET M3 = M2 WITH [x] := BVSUB(32, M2[x], M2[y]) IN
      M0[x] = M3[y] AND M0[y] = M3[x];
```

Example

```
(set-logic QF_ABV)
(declare-const M0 (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))

(assert (let ((M1 (store M0 x (bvadd (select M0 x) (select M0 y)))))
        (let ((M2 (store M1 y (bvsub (select M1 x) (select M1 y)))))
        (let ((M3 (store M2 x (bvsub (select M2 x) (select M2 y)))))
        (not (and (= (select M0 x) (select M3 y))
                  (= (select M0 y) (select M3 x))))))))
(check-sat)
```

Challenges with the Flat Model

There are two main challenges that affect the scalability of the flat model:

- *Deeply nested stores*: Every assignment layers another store. Performance of array solvers suffers as number of stores grows.
- *Disjointness constraints*: For variables that are guaranteed to not alias (e.g. different local/global variables), explicit constraints must be added:

$$disjoint(p, q) \equiv p + size(p) \leq q \vee q + size(q) \leq p.$$

Note that these are required *pairwise*, so as the number of variables grows, the number of disjointness constraints grows quadratically.

Burstall's model

Burstall's model uses one array for each type in the program

- Improves scalability significantly
- Assumes that variables of different types can never alias

For languages like C that can include type-unsafe constructs, Burstall's model may produce incorrect results.

Example

Consider the following code.

```
int a ;
void foo() {
  int* b = &a;
  *b = 0xFFFF;
  char* c = (char*) b;
  *c = 0x0;
  assert (a == 0xFF00);
}
```

There are 4 different types: `int`, `int*`, `char`, and `char*`.

In the Burstall model, each type has its own memory array.

Example

```
M_int, M_intp, M_charp : ARRAY BITVECTOR(32) OF BITVECTOR(32);
M_char : ARRAY BITVECTOR(32) OF BITVECTOR(8);
a, b, c : BITVECTOR(32);

QUERY LET M_intp' = M_intp WITH [b] := a IN
      LET M_int' = M_int WITH [M_intp'[b]] := 0hex0000ffff IN
      LET M_charp' = M_charp WITH [c] := M_intp'[b] IN
      LET M_char' = M_char WITH [M_charp'[c]] := 0hex00 IN
      M_int'[a] = 0hex0000ff00;
```

Example

```
(set-logic QF_ABV)
(declare-const M_int (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const M_intp (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const M_char (Array (_ BitVec 32) (_ BitVec 8)))
(declare-const M_charp (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))

(assert (let ((M_intp1 (store M_intp b a)))
        (let ((M_int1 (store M_int (select M_intp1 b) #x0000ffff)))
        (let ((M_charp1 (store M_charp c (select M_intp1 b))))
        (let ((M_char1 (store M_char (select M_charp c) #x00)))
        (not (= (select M_int1 a) #x0000ff00)))))))
(check-sat)
```

Points-to Analysis

Before introducing the partitioned model, we need to explain what a points-to analysis is.

A *points-to* analysis builds a directed acyclic graph (DAG):

- Each *vertex* is an equivalence class of program variables.
- An *edge* in the DAG from N_1 to N_2 indicates that at some point in the program execution, the *value* of a variable associated with N_1 might point to the *location* of a variable in N_2.

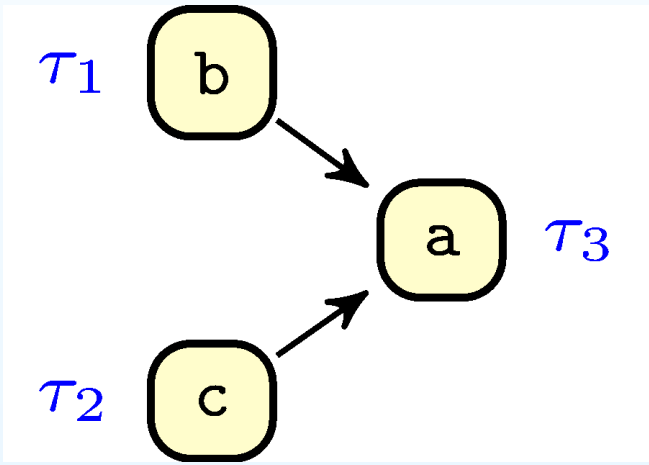
There is a rich literature on points-to analyses that we will not cover here.

Instead, we illustrate with an example.

Points-to Analysis

Consider again the following code:

```
int a ;
void foo() {
  int* b = &a;
  *b = 0xFFFF;
  char* c = (char*) b;
  *c = 0x0;
  assert (a == 0xFF00);
}
```

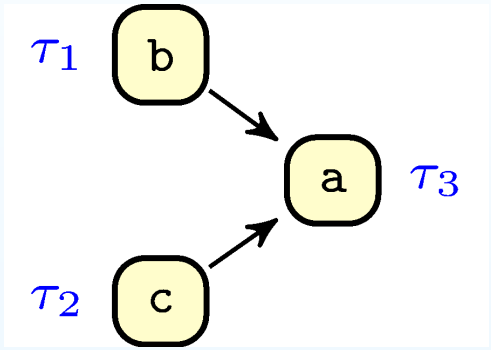


A points-to graph for the code is shown on the right.

- Because `&a` is assigned to `b`, there is an edge from τ_1 to τ_3.
- Because `b` is assigned to `c`, there is also an edge from τ_2 to τ_3.

Partitioned Memory Model

```
int a ;
void foo() {
  int* b = &a;
  *b = 0xFFFF;
  char* c = (char*) b;
  *c = 0x0;
  assert (a == 0xFF00);
}
```



The points-to graph tells us how to build the partitioned memory model:

- Each vertex in the graph has its own memory array.
- The value of a variable is obtained by reading from the memory array associated with its vertex.
- The value of a *dereferenced* variable is obtained by reading from the memory array associated with the points-to *target* of the outgoing edge from the variable's vertex.

Example

```
M_1 : ARRAY BITVECTOR(32) OF BITVECTOR(32);
M_2 : ARRAY BITVECTOR(32) OF BITVECTOR(32);
M_3 : ARRAY BITVECTOR(32) OF BITVECTOR(8);

a, b, c : BITVECTOR(32);

QUERY
  LET M_1' = M_1 WITH [b] := a IN
  LET M_3' = (((M_3 WITH [M_1'[b]] := 0hexFF)
                    WITH [BVPLUS(32, M_1'[b], 0hex00000001)] := 0hexFF)
                    WITH [BVPLUS(32, M_1'[b], 0hex00000002)] := 0hex00)
                    WITH [BVPLUS(32, M_1'[b], 0hex00000003)] := 0hex00 IN
  LET M_2' = M_2 WITH [c] := M_1'[b] IN
  LET M_3'' = M_3' WITH [M_2'[c]] := 0hex00 IN
    M_3''[a] = 0hex00 AND
    M_3''[BVPLUS(32, a, 0hex00000001)] = 0hexFF AND
    M_3''[BVPLUS(32, a, 0hex00000002)] = 0hex00 AND
    M_3''[BVPLUS(32, a, 0hex00000003)] = 0hex00;
```

Example

```
(set-logic QF_ABV)
(declare-const M1 (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const M2 (Array (_ BitVec 32) (_ BitVec 32)))
(declare-const M3 (Array (_ BitVec 32) (_ BitVec 8)))
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))
(assert
  (let ((M1_1 (store M1 b a)))
  (let ((M3_1 (store (store (store (store M3 (select M1_1 b) #xff)
                                   (bvadd (select M1_1 b) #x00000001) #xff)
                            (bvadd (select M1_1 b) #x00000002) #x00)
                     (bvadd (select M1_1 b) #x00000003) #x00)))
  (let ((M2_1 (store M2 c (select M1_1 b))))
  (let ((M3_2 (store M3_1 (select M2_1 c) #x00)))
  (not (and (= (select M3_2 a) #x00)
            (= (select M3_2 (bvadd a #x00000001)) #xff)
            (= (select M3_2 (bvadd a #x00000002)) #x00)
            (= (select M3_2 (bvadd a #x00000003)) #x00)))))))))
(check-sat)
```

References

[BST07] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for Bit-Vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, chapter 16, pages 174–177. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009

[CZ00] Domenico Cantone and Calogero G. Zarba. A new fast tableau-based decision procedure for an unquantified 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

[CG96] B. V. Cherkassy and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, October 1980

References

[GD07] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer Berlin Heidelberg, 2007

[HBJ+14] Liana Hadarean, Clark Barrett, Dejan Jovanović, Cesare Tinelli, and Kshitij Bansal. A tale of two solvers: Eager and lazy approaches to bit-vectors. In Armin Biere and Roderick Bloem, editors, *Proceedings of the* 26^{th} *International Conference on Computer Aided Verification (CAV '14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 680–695. Springer, July 2014. Vienna, Austria

[LRT+14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Proceedings of the* 26^{th} *International Conference on Computer Aided Verification (CAV '14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, July 2014. Vienna, Austria

[LTR+15] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. A decision procedure for regular membership and length constraints over unbounded strings. In Carsten Lutz and Silvio Ranise, editors, *Proceedings of the* 10^{th} *International Symposium on Frontiers of Combining Systems (FroCoS '15)*, volume 9322 of *Lecture Notes in Artificial Intelligence*, pages 135–150. Springer, September 2015. Wroclaw, Poland

References

[LQ08] S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using smt solvers. In *Proceedings of the 35th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, 2008

[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

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980

[NO05] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the* 17^{th} *International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, July 2005

References

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

[RBH07] Zvonimir Rakamarić, Jesse Bingham, and Alan J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *Verification, Model Checking, and Abstract Interpretation: 8th International Conference*, pages 106–121. Springer, 2007. Lecture Notes in Computer Science Vol. 4349

[YRS+06] Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS '06)*, 2006

[WBW16] Wei Wang, Clark Barrett, and Thomas Wies. Partitioned memory models for program analysis. In *Proceedings of the* 22^{nd} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '16)*, 2016. Submitted for review

[ZSM04a] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of the* 2^{nd} *International Joint Conference on Automated Reasoning (IJCAR '04) LNCS 3097*, pages 152–167, 2004

References

[ZSM04b] Ting Zhang, Henny B. Sipma, and Zohar Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of the* 17^{th} *International Conference on Theorem Proving in Higher Order Logics (TPHOLs '04)*, volume 3223 of *Lecture Notes in Computer Science*, pages 321–336, 2004