C08 – SMT Solvers, Symbolic execution

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

How SMT solvers works?

Symbolic execution

How SMT solvers works?

The SMT problem

The SMT problem:

Given a first-order logic formula, with symbols from (possibly several) theories, does it have a model?

Is the formula satisfiable? If so, how?

The SAT problem is a special case, in which

- the formula is quantifier-free, without function symbols or equality
- no theories are used

SAT solving algorithms are an important ingredient in SMT solvers

First-order theories

- Whereas the language of SAT solvers is Boolean logic, the language of SMT solvers is first-order logic.
- First-order theories allow us to capture structures which are used by programs (e.g., arrays, integers) and enable reasoning about them.
- Validity in first order logic (FOL) is undecidable!
  - Lambda calculus – Alonzo Church (1936)
  - Turing machines – Alan Turing (1937)
  - Recursive functions – Kurt Gödel (1934) and Stephen Kleene (1936)
- Validity in particular first order theories is (sometimes) decidable.

SMT solvers

Combine propositional satisfiability search techniques with solvers for specific first-order theories:

- Linear arithmetic
- Bit vectors
- Arrays
- ...

Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
  - Tries to find ways of encoding an entire SMT problem into SAT.
  - There are a variety of techniques
  - For some theories, this works quite well.

Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
  - Tries to find ways of encoding an entire SMT problem into SAT.
  - There are a variety of techniques
  - For some theories, this works quite well.
- The lazy approach
  - Tries to combine SAT and theory reasoning.
  - The basis for most modern SMT solvers.

SMT: The Big Questions

1. How to solve conjunctions of literals in a theory?
   - Use a Theory solver
2. How to combine a theory solver and a SAT solver to reason about arbitrary formulas?
   - The DPLL(T) framework
3. How to combine theory solvers for several theories?
   - The Nelson-Oppen method and its variants

Theory solver

- 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 if there is some model $\mathcal{M}$ of T such that φ holds in $\mathcal{M}$.
- In order to integrate a Theory solver into a modern SMT solver, it is helpful if the Theory solver can do more than just check satisfiability.

Characteristics of Theory solvers

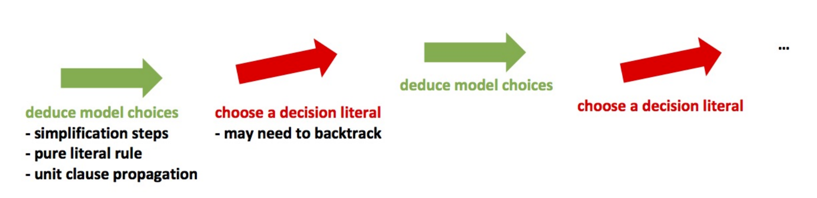
Some desirable characteristics of Theory solvers include:

- 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
- Model Generating – when reporting T-satisfiable, the Theory solver also provides a concrete value for each variable or function symbol
- Proof Generating – when reporting T-unsatisfiable, the Theory solver also provides a checkable proof

Propositional Abstraction

- An atom is a formula without propositional connectives or quantifiers
  - depending on the signature $f(a) = b, m * n \leq 42$ could be atoms; 42 is not
  - a propositional atom is an uninterpreted constant symbol of sort Bool
- A (first-order) literal is an atom or its negation
- For a given signature Σ, we define a signature Σ^p containing only:
  - the propositional Σ-atoms
  - a fresh propositional atom for each non-propositional Σ-atom
- We then fix an injective mapping from the non-propositional Σ-atoms to the Σ^p-atoms.
- For a Σ-formula φ, the formula φ^p is the propositional abstraction of φ, given by replacing all non-propositional Σ-atoms in φ with their image under this mapping.
- An Σ-formula φ is propositional unsatisfiable if $\varphi^p \models \bot$.
- An Σ-formula φ propositionally entails an Σ-formula ψ if $\varphi^p \models \psi^p$.
  - Note that $\varphi^p \models \psi^p$ implies $\varphi \models \psi$, but not necessarily vice-versa.

Recall DPLL/CDCL Algorithms

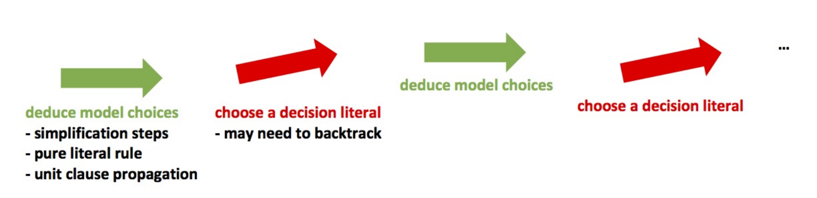


...until...

- conflict reached
  - backtrack - try flipping a decision literal
  - (if CDCL) learn new clause, back-jump
- model found
  - return the model

Adapting DPLL to DPLL(T)

Run DPLL on the propositional abstraction φ^p of the T-input formula φ



...until...

- conflict reached: backtrack/jump, learn clauses as usual
- model found (represented by a set Γ of literals)
  - It is not necessarily a T-model!
  - Ask theory solver: is Γ T-satisfiable?
    - If yes, we are done.
    - If no, backtrack in the original search.
    - (CDCL) get a T-unsatisfiable subset for clause learning/back-jumping

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \land (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \lor \underbrace{g(a) = d}_{3}) \land \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \lor 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$
- SAT solver detects unsat

Theory combination

- 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.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Theory combination

- 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.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Example
The following formula uses both T_{E} and T_{Z}

$$\varphi := 1 \leq x \land x \leq 2 \land f(x) \neq f(1) \land f(x) \neq f(2)$$

The Nelson-Oppen Method

A very general method for combining theory solvers is the Nelson-Oppen method.

This method is applicable when:

1. The signatures Σ_i are disjoint.
2. The theories T_i are stably-infinite.
   - A Σ-theory T is stably-infinite if every T-satisfiable quantifier-free Σ-formula in satisfiable in an infinite model.
3. The formulas to be tested for satisfiability are conjunctions of quantifier-free literals.

Extensions exist that can relax each of these restrictions in some cases.

The Nelson-Oppen Method

Some definitions:

- A member of Σ_i is an *i*-symbol.
- A term t is an *i*-term if it starts with an i-symbol.
- An atomic *i*-formula is
  - an application of an i-predicate,
  - an equation whose lhs in an i-term, or
  - an equation whose lhs is a variable and whose rhs in an i-term
- An *i*-literal is an atomic i-formula or the negation of one.
- An occurrence of a term t in either an i-term or an i-literal is *i*-alien if it is a j-term with $i \neq j$ and all of its super-terms (if any) are i-terms.
- An expression is pure if it contains only variables and i-symbols for some i.

Conversion to separate form

Given a conjunction of literals φ, we want to convert it into a separate form: a T-equisatisfiable conjunction of literals $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ where each φ_i is a Σ_i-formula.

We have the following algorithm:

1. Let ψ be some literal in φ.
2. If ψ is a pure i-literal, for some i, remove ψ from φ and add ψ to φ_i.
   If φ is empty then stop; otherwise goto step 1.
3. Otherwise, ψ is an i-literal for some i.
   Let t be a term occurring i-alien in ψ.
   Replace t in φ with a new variable z and add $z = t$ to φ.
   Goto step 1.

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} :=?$
- $\varphi_{\mathrm{Z}} :=?$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} :=?$
- $\varphi_{\mathrm{Z}} := 1 \leq x$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \land} \; x \leq 2 \; \land \; f(x) \neq f(1) \; \land \; f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} :=?$
- $\varphi_{\mathrm{Z}} := 1 \leq x \land x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2 \wedge}\ f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} :=?$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \ \wedge \ x \leq 2 \wedge} \ f(x) \neq f(y) \ \wedge \ f(x) \neq f(2) \ \wedge \ y = 1$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} :=?$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \land x \leq 2 \land} \; f(x) \neq f(y) \land f(x) \neq f(2) \land y = 1$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \land x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(y) \wedge} \ f(x) \neq f(2) \ \wedge \ y = 1$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(y) \wedge} \ f(x) \neq f(z) \ \wedge \ y = 1 \ \wedge \ z = 2$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge} f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(y) \ \wedge \ f(x) \neq f(z) \wedge} \ y = 1 \ \wedge \ z = 2$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2 \wedge y = 1$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(y) \ \wedge \ f(x) \neq f(z) \ \wedge \ y = 1 \wedge} \ z = 2$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

Conversion to separate form

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2}$$

We convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

- As each φ_i is a Σ_i-formula, we can run a Theory solver Sat_i for each φ_i.
- If any Sat_i reports that φ_i is unsatisfiable, then φ is unsatisfiable.
- The converse is not true in general!
- We need a way for the decision procedures to communicate with each other about shared variables.
- If S is a set of terms and $\sim$ is an equivalence relation on S, then the arrangement of S induced by $\sim$ is

$$Ar_{\sim} = \{x = y \mid x \sim y\} \cup \{x \neq y \mid x \not\sim y\}$$

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2.

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ-formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2, we wish to determine if φ is T-satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its separate form $\varphi_1 \wedge \varphi_2$.
2. Let S be the set of variables shared between φ_1 and φ_2.
   Guess an equivalence relation $\sim$ on S.
3. Run Sat_1 on $\varphi_1 \cup Ar_\sim$.
4. Run Sat_2 on $\varphi_2 \cup Ar_\sim$.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2.

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ-formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2, we wish to determine if φ is T-satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its separate form $\varphi_1 \wedge \varphi_2$.
2. Let S be the set of variables shared between φ_1 and φ_2.
   Guess an equivalence relation $\sim$ on S.
3. Run Sat_1 on $\varphi_1 \cup Ar_\sim$.
4. Run Sat_2 on $\varphi_2 \cup Ar_\sim$.

If there exists an equivalence relation $\sim$ such that both Sat_1 and Sat_2 succeed, then φ is T-satisfiable.

If no such equivalence relation exists, then φ is T-unsatisfiable.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2.

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ-formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2, we wish to determine if φ is T-satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its separate form $\varphi_1 \wedge \varphi_2$.
2. Let S be the set of variables shared between φ_1 and φ_2.
   Guess an equivalence relation $\sim$ on S.
3. Run Sat_1 on $\varphi_1 \cup Ar_\sim$.
4. Run Sat_2 on $\varphi_2 \cup Ar_\sim$.

If there exists an equivalence relation $\sim$ such that both Sat_1 and Sat_2 succeed, then φ is T-satisfiable.

If no such equivalence relation exists, then φ is T-unsatisfiable.

The generalization to more than two theories is straightforward.

The Nelson-Oppen Method

Example
Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

Example

Consider the following $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{Z}}$:

$$\varphi := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$

The shared variables are $\{x, y, z\}$.

There are 5 possible arrangements based on equivalence classes of x, y, and z *(see Bell number)*.

The Nelson-Oppen Method

Example

- $\varphi := 1 \le x \land x \le 2 \land f(x) \ne f(1) \land f(x) \ne f(2)$
- $\varphi_{\mathrm{E}} := f(x) \ne f(y) \land f(x) \ne f(z)$
- $\varphi_{\mathrm{Z}} := 1 \le x \land x \le 2 \land y = 1 \land z = 2$

1. $\{x = y, x = z, y = z\}$
2. $\{x = y, x \ne z, y \ne z\}$
3. $\{x \ne y, x = z, y \ne z\}$
4. $\{x \ne y, x \ne z, y = z\}$
5. $\{x \ne y, x \ne z, y \ne z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \,\wedge\, x \leq 2 \,\wedge\, f(x) \neq f(1) \,\wedge\, f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \,\wedge\, f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \,\wedge\, x \leq 2 \,\wedge\, y = 1 \,\wedge\, z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_{E}
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_{E}
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_{E}
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_{E}
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_{E}
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_{Z}
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \land x \leq 2 \land f(x) \neq f(1) \land f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \land f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \land x \leq 2 \land y = 1 \land z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_{E}
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_{E}
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_{Z}
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_{Z}

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2)$
- $\varphi_{\mathrm{E}} := f(x) \neq f(y) \ \wedge \ f(x) \neq f(z)$
- $\varphi_{\mathrm{Z}} := 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ y = 1 \ \wedge \ z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_{E}
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_{E}
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_{E}
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_{Z}
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_{Z}

Conclusion: φ is $T_{\mathrm{E}} \cup T_{\mathrm{Z}}$-unsatisfiable!

SMT solvers

Recall the ingredients:

- Theory solvers for different theories
- Combine a Theory solver and a SAT solver
- Combine Theory solvers for different theories

Symbolic execution

Symbolic execution

- Symbolic execution is widely used in practice.
- Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:
  - Networks servers
  - File systems
  - Device drivers
  - Unix utilities
  - Computer vision code
  - ...

Symbolic execution: Tools

- Stanford's KLEE
  http://klee.llvm.org/
- Nasa's Java PathFinder
  http://javapathfinder.sourceforge.net/
- Microsoft Research's SAFE
- UC Berkeley's CUTE

Symbolic execution

At any point during program execution, symbolic execution keeps two formulas:

- symbolic store and
- path constraint

Therefore, at any point in time the symbolic state is described as the conjunction of these two formulas.

Symbolic store

The value of variables at any moment in time are given by a function

$$\sigma_s : \mathit{Var} \to \mathit{Sym}$$

- *Var* is the set of variables
- *Sym* is a set of symbolic values
- σ_s is called a symbolic store

Example
σ_s : $\mathtt{x} \mapsto \mathtt{x0}$, $\mathtt{y} \mapsto \mathtt{y0}$

Semantics

Arithmetic expression evaluation simply manipulates the symbolic values.

Example
Suppose the symbolic store is $\sigma_s \;:\; \mathtt{x} \mapsto \mathtt{x0},\ \mathtt{y} \mapsto \mathtt{y0}$.

Then `z = x + y` will produce the new symbolic store

$$\sigma'_s \;:\; \mathtt{x} \mapsto \mathtt{x0},\ \mathtt{y} \mapsto \mathtt{y0},\ \mathtt{z} \mapsto \mathtt{x0 + y0}$$

We literally keep the symbolic expression `x0 + y0`.

Path constraint

- The analysis keeps a path constraint (pct) which records the history of all branches taken so far.
- The path constraint is simply a formula.
- The formula is typically in a decidable logical fragment without quantifiers.
- At the start of the analysis, the path constraint is true.
- Evaluation of conditionals affects the path constraint, but not the symbolic store.

Path constraint

Example
Suppose the symbolic store is σ_s : `x` $\mapsto$ `x0`, `y` $\mapsto$ `y0`.

Suppose the path constraint is `pct = x0 > 10`.

Let us evaluate `if(x > y + 1) {5: ...}`

At label 5, we will get the symbolic store σ_s. It does not change!

But, at label 5, we will get an updated path constraint:

$$\texttt{pct = x0 > 10} \wedge \texttt{x0 > y0 + 1}$$

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Can you find the inputs that make the program reach the ERROR?

Lets execute this example with classic symbolic execution

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input and because we don't know what those read values are, we set the values of `x` and `y` to fresh symbolic values called `x0` and `y0`

pct is true because so far we have not executed any conditionals

σ_s : `x` $\mapsto$ `x0`,
`y` $\mapsto$ `y0`

pct : true

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : true

Here, we simply executed the function twice() and added the new symbolic value for z.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if x = z:

$$\sigma_s : \begin{array}{l} x \mapsto x0, \\ y \mapsto y0 \\ z \mapsto 2*y0 \end{array} \qquad pct : x0 = 2*y0$$

This is the result if x != z:

$$\sigma_s : \begin{array}{l} x \mapsto x0, \\ y \mapsto y0 \\ z \mapsto 2*y0 \end{array} \qquad pct : x0 \neq 2*y0$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We can avoid further exploring a path if we know the constraint pct is unsatisfiable. In this example, both pct's are satisfiable so we need to keep exploring both paths.

This is the result if x = z:

σ_s : $x \mapsto x0$, $y \mapsto y0$, $z \mapsto 2*y0$

pct : $x0 = 2*y0$

This is the result if x != z:

σ_s : $x \mapsto x0$, $y \mapsto y0$, $z \mapsto 2*y0$

pct : $x0 \neq 2*y0$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Lets explore the path when x == z is true.
Once again we get 2 more paths.

This is the result if x > y + 10:

σ_s : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0 ∧ x0 > y0+10

This is the result if x ≤ y + 10:

σ_s : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0 ∧ x0 ≤ y0+10

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

So the following path reaches "**ERROR**".

This is the result if x > y + 10:

σ_s : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0 ∧ x0 > y0+10

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance, `x0 = 40, y0 = 20` is a satisfying assignment. That is, running the program with those concrete inputs triggers the error.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Handling Loops - a limitation

```
int F(unsigned int k) {
  int sum = 0;
  int i   = 0;
  for ( ; i < k; i++)
     sum += i;
  return sum;
}
```

- A serious limitation of symbolic execution is handling unbounded loops.
- Symbolic execution runs the program for a finite number of paths.
- But what happens if we do not know the bound on a loop?
- The symbolic execution will keep running forever!

Handling Loops - bound loops

```
int F(unsigned int k) {
  int sum = 0;
  int i   = 0;
  for ( ; i < 2; i++)
     sum += i;
  return sum;
}
```

- A common solution in practice is to provide some loop bound.
  - In the above example, we can bound `k` to say 2.
  - This is an example of an under-approximation
- Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.

Handling Loops - loop invariants

```
int F(unsigned int k) {
  int sum = 0;
  int i   = 0;
  for ( ; i < k; i++)
     sum += i;
  return sum;
}
```

loop invariant

- Another solution is to provide a loop invariant.
- This technique is rarely used for large programs because it is difficult to provide such invariants manually.
- It can also lead to over-approximation.

Constraint solving - challenges

- Constraint solving is fundamental to symbolic execution.
- An SMT solver is continuously invoked during analysis.
- Often, the main roadblock to performance of symbolic execution engines is the time spent in constraint solving.
- Important features:
    - The SMT solver supports as many decidable logical fragments as possible.
        - Some tools use more than one SMT solver.
    - The SMT solver can solve large formulas quickly.
    - The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

Key optimization - caching

- The analyzer will invoke the SMT solver with similar formulas.
- The symbolic execution engine can keep a map (cache) of formulas to a satisfying assignment for the formulas.
- When the engine builds a new formula and would like to find a satisfying assignment for that formula, it can first access the cache, before calling the SMT solver.

Key optimization - caching

Example
Suppose the cache contains the mapping:

$$\begin{array}{ccc} \text{Formula} & & \text{Solution} \\ (x + y < 10) \land (x > 5) & \rightarrow & \{x = 6, y = 3\} \end{array}$$

If we get a weaker formula as a query, say $(x + y < 10)$, then we can immediately reuse the solution already found in the cache, without calling the SMT solver.

If we get a stronger formula as a query, say $(x + y < 10) \land (x > 5) \land (y \geq 0)$, then we can quickly try the solution in the cache and see if it works, without calling the solver (in this example, it works).

When constraint solving fails

Despite best efforts, the program may be using constraints in a fragment which the SMT solver does not handle (well).

For example, the SMT solver does not handle non-linear constraints well.

When constraint solving fails - example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Here, we changed the twice() function to contain a non-linear result.

Let us see what happens when we symbolically execute the program now...

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

When constraint solving fails - example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

This is the result if x = z:

σ_s : `x ↦ x0,`
`y ↦ y0`
`z ↦ y0*y0`

pct : `x0 = y0*y0`

Now, if we are to invoke the SMT solver with the pct formula, it would be unable to compute satisfying assignments, precluding us from knowing whether the path is feasible or not.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emina Torlak.