Efficient resolution of logical models

ENSTA-IA303

Alexandre Chapoutot and Sergio Mover

ENSTA Paris

2020-2021

About the "second part" of IA303

Contact information:

- via email: sergio.mover <at> lix.polytechnique.fr
- "on-demand" office hours: Friday from 2pm to 3pm
- Get in touch also if you are interested in research in:
  - Formal methods (formal verification, decision procedures, . . .)
  - Artificial intelligence (mainly related to the use logics, symbolic AI, planning, . . .)
  - Cyber-Physical systems (e.g., hybrid systems, . . .)

About the “second part” of IA303

Contact information:

- via email: sergio.mover <at> lix.polytechnique.fr
- “on-demand” office hours: Friday from 2pm to 3pm
- Get in touch also if you are interested in research in:
  - Formal methods (formal verification, decision procedures, . . .)
  - Artificial intelligence (mainly related to the use logics, symbolic AI, planning, . . .)
  - Cyber-Physical systems (e.g., hybrid systems, . . .)

In the next 4 classes:

- Introduction to Satisfiability Modulo Theories (SMT) (week 4, today)
- How some theory solvers works: EUF (week 5) and LRA/LIA (week 6)
- Some applications of SMT in formal verification (week 7)

About the "second part" of IA303

Contact information:

- via email: sergio.mover <at> lix.polytechnique.fr
- "on-demand" office hours: Friday from 2pm to 3pm
- Get in touch also if you are interested in research in:
  - Formal methods (formal verification, decision procedures, ...)
  - Artificial intelligence (mainly related to the use logics, symbolic AI, planning, ...)
  - Cyber-Physical systems (e.g., hybrid systems, ...)

In the next 4 classes:

- Introduction to Satisfiability Modulo Theories (SMT) (week 4, today)
- How some theory solvers works: EUF (week 5) and LRA/LIA (week 6)
- Some applications of SMT in formal verification (week 7)

Main take-away points:

- SMT is a powerful and mature tool, with wide applications
- How to formalize a problem in SMT and use an SMT solver
- How an SMT solver works and why is it efficient (despite the problem complexity)

Lecture 4: Satisfiability Modulo Theories and DPLL($\mathcal{T}$)

Main goals for today

In class[1]:

- Why and where do we use Satisfiability Modulo Theories (SMT)?
- What is SMT precisely?
- How can we efficiently decide the SMT problem?

---

[1] Main references:

- The Calculus of Computation [Bradley and Manna, 2007], Chapter 2 (First-Order Logic) and Chapter 3 (First-Order Theories)
- Satisfiability Modulo Theories [Barrett et al., 2009]
- Lazy Satisfiability Modulo Theories [Sebastiani, 2007]
- Satisfiability modulo theories: introduction and applications [de Moura and Bjørner, 2011] - CACM article, good first reading!

Main goals for today

In class[1]:

- Why and where do we use Satisfiability Modulo Theories (SMT)?
- What is SMT precisely?
- How can we efficiently decide the SMT problem?

In the tutorial:

- How do you use an SMT solver?
- How do you formalize a problem (i.e., encode) as an SMT problem?
- How can you write a program that uses an SMT solver?
- How can you automate the encoding generation for a class of problems?

---

[1] Main references:

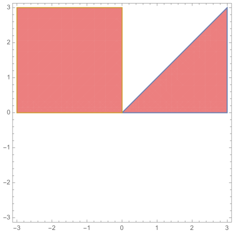
- The Calculus of Computation [Bradley and Manna, 2007], Chapter 2 (First-Order Logic) and Chapter 3 (First-Order Theories)
- Satisfiability Modulo Theories [Barrett et al., 2009]
- Lazy Satisfiability Modulo Theories [Sebastiani, 2007]
- Satisfiability modulo theories: introduction and applications [de Moura and Bjørner, 2011] - CACM article, good first reading!

1. Satisfiability Modulo Theories and DPLL($\mathcal{T}$)
   - Why SMT?
   - The SMT problem
     - First-Order logic - Language and Semantic
     - SMT - Language and Semantic
   - Decision procedures for the SMT Problem
     - Lazy approach - the offline schema
     - Lazy approach - the online approach (DPPL($\mathcal{T}$))

SMT in a nutshell - informal intuition

- Extend of the propositional satisfiability problem to represent specific domains. For example, real numbers:

$$(x - y \geq 0 \vee x < 0) \wedge y > 0$$



- There are more theories: bitvectors, rational and integers linear arithmetic , uninterpreted functions, arrays, strings, separation logics, floating point arithmetic, ...
- SMT is an expressive language.

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])
- Static program analysis (e.g., [Filliâtre and Paskevich, 2013, Barnett et al., 2005, Leino, 2010])

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])
- Static program analysis (e.g., [Filliâtre and Paskevich, 2013, Barnett et al., 2005, Leino, 2010])
- Test-case generation (e.g., [Godefroid et al., 2012, Godefroid et al., 2005])

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])
- Static program analysis (e.g., [Filliâtre and Paskevich, 2013, Barnett et al., 2005, Leino, 2010])
- Test-case generation (e.g., [Godefroid et al., 2012, Godefroid et al., 2005])
- Automatic program repair (e.g., [Mechtaev et al., 2016])

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])
- Static program analysis (e.g., [Filliâtre and Paskevich, 2013, Barnett et al., 2005, Leino, 2010])
- Test-case generation (e.g., [Godefroid et al., 2012, Godefroid et al., 2005])
- Automatic program repair (e.g., [Mechtaev et al., 2016])
- Automatic program synthesis (e.g., [Jha et al., 2010])

What problems can we solve with SMT?

We use SMT to express different have constraints satisfaction problems for several applications:

- Formal Verification for software, hardware, cyber-physical systems, protocols, neural networks (e.g., [Henzinger et al., 2004, McMillan and Padon, 2020, Dutertre et al., 2018, Cimatti et al., 2016, Cimatti et al., 2015])
- Static program analysis (e.g., [Filliâtre and Paskevich, 2013, Barnett et al., 2005, Leino, 2010])
- Test-case generation (e.g., [Godefroid et al., 2012, Godefroid et al., 2005])
- Automatic program repair (e.g., [Mechtaev et al., 2016])
- Automatic program synthesis (e.g., [Jha et al., 2010])
- Planning (e.g., [Wolfman and Weld, 1999, Cashmore et al., 2020, Cimatti et al., 2018])
- ...

Active area of research!
With applications in formal methods, programming languages, software engineering, AI, ...

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?
  1.0 instead of 1.5

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?
  1.0 instead of 1.5
- What if $x = y$?

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?
  1.0 instead of 1.5
- What if $x = y$?
  Infinite loop!

Software verification

```
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|} \left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?
  1.0 instead of 1.5
- What if $x = y$?
  Infinite loop!

Need automatic tool to verify that $(x - y) - 1 >= 0$

Software verification

```c
float weighted_sum(unsigned int x,
                   unsigned int y) {
  unsigned int i;
  float sum;

  if (y > x) {    // swap x and y
    x = x^y; y = y^x; x = x^y;
  }

  sum = 0;
  for (i = 0; i <= (x-y)-1; ++i) {
    float tmp;
    tmp = ((i + 1)) / (x - y);
    sum = sum + tmp;
  }

  return sum;
}
```

Compute $\sum_{i=1}^{|x-y|}\left(\frac{i}{|x-y|}\right)$

Is the implementation correct?

- What if $x = 5, y = 3$?
  1.0 instead of 1.5
- What if $x = y$?
  Infinite loop!

Need automatic tool to verify that $(x - y) - 1 >= 0$

Software verification uses SMT to faithfully model the program semantic

1. Satisfiability Modulo Theories and DPLL($\mathcal{T}$)
   - Why SMT?
   - The SMT problem
     - First-Order logic - Language and Semantic
     - SMT - Language and Semantic
   - Decision procedures for the SMT Problem
     - Lazy approach - the offline schema
     - Lazy approach - the online approach (DPPL($\mathcal{T}$))

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

Syntax

A *term* t is either:

- A *constant* $a, b, c, \ldots, 0, 1, \ldots$ (or a 0-ary function)
- A *variable* $x, y, z, \ldots$
- An n-ary *function* $f(t_1, \ldots, f_n)$

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

Syntax

A *term* t is either:

- A *constant* $a, b, c, \ldots, 0, 1, \ldots$ (or a 0-ary function)
- A *variable* $x, y, z, \ldots$
- An n-ary *function* $f(t_1, \ldots, f_n)$

An n-ary *predicate* p is $p(t_1, \ldots, p_n)$

- 0-ary predicates are Propositional variables (denoted with P,Q, ...)

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

Syntax

A *term* t is either:

- A *constant* $a, b, c, \ldots, 0, 1, \ldots$ (or a 0-ary function)
- A *variable* $x, y, z, \ldots$
- An n-ary *function* $f(t_1, \ldots, f_n)$

An n-ary *predicate* p is $p(t_1, \ldots, p_n)$

- 0-ary predicates are Propositional variables (denoted with P,Q, ...)

An *atom* a is either true $\top$, false $\bot$, or a predicate p.

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

Syntax

A *term* t is either:

- A *constant* $a, b, c, \ldots, 0, 1, \ldots$ (or a 0-ary function)
- A *variable* $x, y, z, \ldots$
- An n-ary *function* $f(t_1, \ldots, f_n)$

An n-ary *predicate* p is $p(t_1, \ldots, p_n)$

- 0-ary predicates are Propositional variables (denoted with P,Q, ...)

An *atom* a is either true $\top$, false $\bot$, or a predicate p.

A FOL formula ϕ is either:

- an atom a
- $\neg\psi$, with ψ a FOL formula
- $\psi_1 \wedge \psi_2$, with ψ_1 and ψ_2 FOL formulas.
  Other operators: $\psi_1 \vee \psi_2 := \neg(\neg\psi_1 \wedge \neg\psi_2)$, $\psi_1 \rightarrow \psi_2 := \neg\psi_1 \vee \psi_2$
- $\exists x.\psi$, with x a variable and ψ a FOL formula
- $\forall x.\phi$, with x a variable and ψ a FOL formula

First-Order Logic

Extends Propositional Logic with *predicates*,*functions*, and *quantifiers* to reason about infinite domains.

Syntax

A *term* t is either:

- A *constant* $a, b, c, \dots, 0, 1, \dots$ (or a 0-ary function)
- A *variable* $x, y, z, \dots$
- An n-ary *function* $f(t_1, \dots, f_n)$

An n-ary *predicate* p is $p(t_1, \dots, p_n)$

- 0-ary predicates are Propositional variables (denoted with $P, Q, \dots$)

An *atom* a is either true $\top$, false $\bot$, or a predicate p.

A FOL formula ϕ is either:

- an atom a
- $\neg\psi$, with ψ a FOL formula
- $\psi_1 \wedge \psi_2$, with ψ_1 and ψ_2 FOL formulas.
  Other operators: $\psi_1 \vee \psi_2 := \neg(\neg\psi_1 \wedge \neg\psi_2)$, $\psi_1 \rightarrow \psi_2 := \neg\psi_1 \vee \psi_2$
- $\exists x.\psi$, with x a variable and ψ a FOL formula
- $\forall x.\phi$, with x a variable and ψ a FOL formula
  We will "almost always" avoid quantifiers

First-Order Logic - Some examples of syntax

- Terms:
  - $1, a, b$ are constants
  - x, y, z are variables
  - $f(x)$, $g(x, z)$, and $g(f(x), y)$ are functions

First-Order Logic - Some examples of syntax

- Terms:
  - $1, a, b$ are constants
  - x, y, z are variables
  - $f(x)$, $g(x, z)$, and $g(f(x), y)$ are functions
- Predicates
  - $p(a, b)$, $p(a, f(x))$, $q(x, y)$

First-Order Logic - Some examples of syntax

- Terms:
  - $1, a, b$ are constants
  - x, y, z are variables
  - $f(x)$, $g(x,z)$, and $g(f(x), y)$ are functions
- Predicates
  - $p(a,b)$, $p(a, f(x))$, $q(x,y)$
- FOL formulas
  - $p(a,b)$
  - $\neg p(a,b)$
  - $(p(a,b) \wedge q(x,y))$
  - $\forall x. x = y \wedge f(x,y)$
  - $\forall x. \exists y. (x = y \rightarrow f(x) = f(y))$

First-Order Logic - Semantic

While Propositional Logic evaluates over true and false, in FOL we have domains and assignments .

A *FOL Interpretation* $\mathcal{I}$ is the pair $\mathcal{I} = (\mathcal{D}_{\mathcal{I}}, \alpha_{\mathcal{I}})$:

First-Order Logic - Semantic

While Propositional Logic evaluates over true and false, in FOL we have domains and assignments .

A *FOL Interpretation* $\mathcal{I}$ is the pair $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$:

- $\mathcal{D}_\mathcal{I}$ is the *domain* of $\mathcal{I}$: it's a non-empty set of elements (e.g., values, objects, ...)

First-Order Logic - Semantic

While Propositional Logic evaluates over true and false, in FOL we have domains and assignments .

A *FOL Interpretation* $\mathcal{I}$ is the pair $\mathcal{I} = (\mathcal{D}_{\mathcal{I}}, \alpha_{\mathcal{I}})$:

- $\mathcal{D}_{\mathcal{I}}$ is the *domain* of $\mathcal{I}$: it's a non-empty set of elements (e.g., values, objects, ...)
- $\alpha_{\mathcal{I}}$ is an *assignment* that maps constants, functions, and predicates to elements, functions, and predicates of $\mathcal{D}_{\mathcal{I}}$:

$$\alpha_{\mathcal{I}}(x) := x_{\mathcal{I}} \qquad x_{\mathcal{I}} \in \mathcal{D}_{\mathcal{I}}$$

$$\alpha_{\mathcal{I}}(f) := f_{\mathcal{I}} \qquad f_{\mathcal{I}} : \mathcal{D}_{\mathcal{I}}^{n} \to \mathcal{D}_{\mathcal{I}}$$

Note that constants are 0-ary functions!$\alpha_{\mathcal{I}}(p) \quad := p_{\mathcal{I}} \qquad p_{\mathcal{I}} : \mathcal{D}_{\mathcal{I}}^{n} \to \{tru$

First-Order Logic - an example of interpretation

- Consider the FOL formula [2]: $x + y > z \to y > z - x$
- A possible intepretation over the interger numbers $\mathbb{Z}$
  - $\mathcal{D}_{\mathcal{I}} = \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - the function $+$, $-$ are assigned to the the plus and minus function in $\mathbb{Z}$ (i.e., $+_{\mathbb{Z}}, -_{\mathbb{Z}}$)
  - the predicates $>$ is assigned to $>_{\mathbb{Z}}$
  - x, y, z are 0-ary functions of integer type
  - $\alpha_{\mathcal{I}} := \{x \mapsto 13, y \mapsto 2, z \mapsto 4, > \mapsto >_{\mathbb{Z}}, + \mapsto +_{\mathbb{Z}}, - \mapsto -_{\mathbb{Z}}, \ldots\}$

[2] Example 2.7 from [Bradley and Manna, 2007]

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_{\mathcal{I}}, \alpha_{\mathcal{I}})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_{\mathcal{I}}, \alpha_{\mathcal{I}})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$
- we evaluate a term t with $\alpha_{\mathcal{I}}(t)$, recursively:
  - for a variable x $\alpha_{\mathcal{I}}(x)$, $\alpha_{\mathcal{I}}(a)$
  - $\alpha_{\mathcal{I}}(f(t_1, \ldots, t_n)) = \alpha_{\mathcal{I}}(f)(\alpha_{\mathcal{I}}(t_1), \ldots, \alpha_{\mathcal{I}}(t_n))$
  - $\alpha_{\mathcal{I}}(p(t_1, \ldots, t_n)) = \alpha_{\mathcal{I}}(p)(\alpha_{\mathcal{I}}(t_1), \ldots, \alpha_{\mathcal{I}}(t_n))$

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$
- we evaluate a term t with $\alpha_\mathcal{I}(t)$, recursively:
  - for a variable x $\alpha_\mathcal{I}(x)$, $\alpha_\mathcal{I}(a)$
  - $\alpha_\mathcal{I}(f(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(f)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
  - $\alpha_\mathcal{I}(p(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
- $\mathcal{I} \models p(t_1, \ldots, t_n)$ iff $\alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$ is *true*.

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$
- we evaluate a term t with $\alpha_\mathcal{I}(t)$, recursively:
  - for a variable x $\alpha_\mathcal{I}(x)$, $\alpha_\mathcal{I}(a)$
  - $\alpha_\mathcal{I}(f(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(f)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
  - $\alpha_\mathcal{I}(p(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
- $\mathcal{I} \models p(t_1, \ldots, t_n)$ iff $\alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$ is *true*.
- $\mathcal{I} \models \neg\phi$ iff $\mathcal{I} \not\models \phi$

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$
- we evaluate a term t with $\alpha_\mathcal{I}(t)$, recursively:
  - for a variable x $\alpha_\mathcal{I}(x)$, $\alpha_\mathcal{I}(a)$
  - $\alpha_\mathcal{I}(f(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(f)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
  - $\alpha_\mathcal{I}(p(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
- $\mathcal{I} \models p(t_1, \ldots, t_n)$ iff $\alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$ is *true*.
- $\mathcal{I} \models \neg\phi$ iff $\mathcal{I} \not\models \phi$
- $\mathcal{I} \models \phi_1 \wedge \phi_2$ iff $\mathcal{I} \models \phi_1$ and $\mathcal{I} \models \phi_2$

First-Order Logic - Semantic

When does an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I})$ satisfies a FOL formula ϕ, $\mathcal{I} \models \phi$?

- $\mathcal{I} \models \top$ and $\mathcal{I} \not\models \bot$
- we evaluate a term t with $\alpha_\mathcal{I}(t)$, recursively:
  - for a variable x $\alpha_\mathcal{I}(x)$, $\alpha_\mathcal{I}(a)$
  - $\alpha_\mathcal{I}(f(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(f)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
  - $\alpha_\mathcal{I}(p(t_1, \ldots, t_n)) = \alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$
- $\mathcal{I} \models p(t_1, \ldots, t_n)$ iff $\alpha_\mathcal{I}(p)(\alpha_\mathcal{I}(t_1), \ldots, \alpha_\mathcal{I}(t_n))$ is *true*.
- $\mathcal{I} \models \neg\phi$ iff $\mathcal{I} \not\models \phi$
- $\mathcal{I} \models \phi_1 \wedge \phi_2$ iff $\mathcal{I} \models \phi_1$ and $\mathcal{I} \models \phi_2$
- $\mathcal{I} \models \exists x.\psi$, if there is some $a \in \mathcal{D}_\mathcal{I}$, $(\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I}[x \mapsto a]) \models \psi$
- $\mathcal{I} \models \forall x.\phi$, if for all $a \in \mathcal{D}_\mathcal{I}$, $(\mathcal{D}_\mathcal{I}, \alpha_\mathcal{I}[x \mapsto a]) \models \psi$.

First-Order Logic - Semantic (Example)

- formula [3] $\phi := x + y > z \to y > z - x$
- interpretation $\mathcal{I} = (\mathbb{Z}, \alpha_{\mathcal{I}})$:

$$\alpha_{\mathcal{I}} := \{x \mapsto 13, y \mapsto 2, z \mapsto 4, > \mapsto >_{\mathbb{Z}}, + \mapsto +_{\mathbb{Z}}, - \mapsto -_{\mathbb{Z}}, \ldots\}$$

- the truth value of ϕ under $\mathcal{I}$ is:
  1. $\mathcal{I} \models x + y > 0$ — since $\mathcal{I}[x + y > 0] = 13_{\mathbb{Z}} +_{\mathbb{Z}} 42_{\mathbb{Z}} >_{\mathbb{Z}} 1_{\mathbb{Z}}$
  2. $\mathcal{I} \models y > z - x$ — since $\mathcal{I}[y > z - x] = 42_{\mathbb{Z}} >_{\mathbb{Z}} 1_{\mathbb{Z}} -_{\mathbb{Z}} 13_{\mathbb{Z}}$
  3. $\mathcal{I} \models \phi$ — by 1,2, and the semantic of $\to$

[3] Example 2.8 from [Bradley and Manna, 2007]

First-Order Logic - Satisfiability and Validity

A FOL formula ϕ is:

- *satisfiable* iff there exists and interpretation $\mathcal{I}$ such that $\mathcal{I} \models \phi$
- *valid* iff for all interpretations $\mathcal{I}$, $\mathcal{I} \models \phi$

Decidability results (see [Bradley and Manna, 2007]):

- validity is semi-decidable: if ϕ is valid, then there exists a procedure that eventually terminates and says yes
- satisfiability is undecidable.

First-Order Logic - Satisfiability and Validity

A FOL formula ϕ is:

- *satisfiable* iff there exists and interpretation $\mathcal{I}$ such that $\mathcal{I} \models \phi$
- *valid* iff for all interpretations $\mathcal{I}$, $\mathcal{I} \models \phi$

Decidability results (see [Bradley and Manna, 2007]):

- validity is semi-decidable: if ϕ is valid, then there exists a procedure that eventually terminates and says yes
- satisfiability is undecidable.

So, what can we do?

Restricting the domains and interpretations of FOL

In a lot of cases we know the domains and operations appearing in the formulas. For example:

- planning with resources: integer or real numbers
- numerical programs manipulating memory: arrays and integer numbers
- microcode (of CPUs): bounded-length bit vectors
- html web sanitizers: strings
- ...

If we restrict FOL to such domains and operations the satisfiability problem becomes decidable.

Restricting the domains and interpretations of FOL

In a lot of cases we know the domains and operations appearing in the formulas. For example:

- planning with resources: integer or real numbers
- numerical programs manipulating memory: arrays and integer numbers
- microcode (of CPUs): bounded-length bit vectors
- html web sanitizers: strings
- ...

If we restrict FOL to such ~~domains and operations~~ first-order theories the satisfiability problem becomes decidable.

Restricting the domains and interpretations of FOL

In a lot of cases we know the domains and operations appearing in the formulas. For example:

- planning with resources: integer or real numbers
- numerical programs manipulating memory: arrays and integer numbers
- microcode (of CPUs): bounded-length bit vectors
- html web sanitizers: strings
- ...

If we restrict FOL to such ~~domains and operations~~ first-order theories the satisfiability problem for quantifier-free formulas becomes decidable.

First-Order Theories - Definition

Theory $\mathcal{T}$

A theory $\mathcal{T}$ is defined with:

- a signature Σ: set of constants, functions, predicates
- a set of axioms $\mathcal{A}$: set of closed FOL formulas (i.e., no free variables) containing constants, functions, predicates from Σ

An Σ-formula ϕ is built only using constants, functions, predicates from Σ

First-Order Theories - Definition

Theory $\mathcal{T}$

A theory $\mathcal{T}$ is defined with:

- a signature Σ: set of constants, functions, predicates
- a set of axioms $\mathcal{A}$: set of closed FOL formulas (i.e., no free variables) containing constants, functions, predicates from Σ

An Σ-formula ϕ is built only using constants, functions, predicates from Σ

Validity and Satisfiability

A Σ-formula ϕ is valid in the theory $\mathcal{T}$ ($\mathcal{T}$-*valid*, written as $\models_{\mathcal{T}} \phi$), if:

- **for all** the interpretations $\mathcal{I}$ such that $\mathcal{I}$ satisfies all the axioms of $\mathcal{T}$ (i.e., $\mathcal{I} \models_{\mathcal{T}} A$, for every axiom $A \in \mathcal{A}$ - this is called a $\mathcal{T}$-*interpretation*)
- $\mathcal{I}$ also satisfy ϕ ($\mathcal{I} \models_{\mathcal{T}} \phi$)

A Σ-formula ϕ is satisfiable in the theory $\mathcal{T}$ ($\mathcal{T}$-*satisfiable*) if **there exists** a $\mathcal{T}$-interpretation such that $\mathcal{I} \models_{\mathcal{T}} \phi$

First-Order Theories - Definition

Theory $\mathcal{T}$

A theory $\mathcal{T}$ is defined with:

- a signature Σ: set of constants, functions, predicates
- a set of axioms $\mathcal{A}$: set of closed FOL formulas (i.e., no free variables) containing constants, functions, predicates from Σ

An Σ-formula ϕ is built only using constants, functions, predicates from Σ

Validity and Satisfiability

A Σ-formula ϕ is valid in the theory $\mathcal{T}$ ($\mathcal{T}$-*valid*, written as $\models_{\mathcal{T}} \phi$), if:

- **for all** the interpretations $\mathcal{I}$ such that $\mathcal{I}$ satisfies all the axioms of $\mathcal{T}$ (i.e., $\mathcal{I} \models_{\mathcal{T}} A$, for every axiom $A \in \mathcal{A}$ - this is called a $\mathcal{T}$-*interpretation*)
- $\mathcal{I}$ also satisfy ϕ ($\mathcal{I} \models_{\mathcal{T}} \phi$)

A Σ-formula ϕ is satisfiable in the theory $\mathcal{T}$ ($\mathcal{T}$-*satisfiable*) if **there exists** a $\mathcal{T}$-interpretation such that $\mathcal{I} \models_{\mathcal{T}} \phi$

An example please!

Equalities and Uninterpreted functions

The Theory of Equalities and Uninterpreted functions $\mathcal{T}_E$ is defined as:

- the signature $\Sigma_E := \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$
  - $=$ is a binary predicate and is *interpreted* as the equality
  - all the other function symbols in Σ_E are not interpreted

Equalities and Uninterpreted functions

The Theory of Equalities and Uninterpreted functions $\mathcal{T}_E$ is defined as:

- the signature $\Sigma_E := \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$
  - $=$ is a binary predicate and is *interpreted* as the equality
  - all the other function symbols in Σ_E are not interpreted
- the set of axioms $\mathcal{A}$:
  1. $\forall x. x = x$ [reflexivity]
  2. $\forall x, y. x = y \rightarrow y = x$ [symmetry]
  3. $\forall x, y, z.((x = y \wedge y = z) \rightarrow x = z)$ [transitivity]

Equalities and Uninterpreted functions

The Theory of Equalities and Uninterpreted functions $\mathcal{T}_E$ is defined as:

- the signature $\Sigma_E := \{=, a, b, c, \dots, f, g, h, \dots, p, q, r, \dots\}$
  - $=$ is a binary predicate and is *interpreted* as the equality
  - all the other function symbols in Σ_E are not interpreted
- the set of axioms $\mathcal{A}$:
  1. $\forall x. x = x$ [reflexivity]
  2. $\forall x, y. x = y \rightarrow y = x$ [symmetry]
  3. $\forall x, y, z. ((x = y \wedge y = z) \rightarrow x = z)$ [transitivity]
  4. Function and predicate congruence
     - For each $n \in \mathbb{N}$ and n-ary function symbol f:

$$\forall x_1, \dots, x_n, y_1, \dots, y_n. \left(\bigwedge_{i=1}^{n} x_i = x_j \right) \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$$

Equalities and Uninterpreted functions

The Theory of Equalities and Uninterpreted functions $\mathcal{T}_E$ is defined as:

- the signature $\Sigma_E := \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$
  - $=$ is a binary predicate and is *interpreted* as the equality
  - all the other function symbols in Σ_E are not interpreted
- the set of axioms $\mathcal{A}$:
  1. $\forall x.x = x$ [reflexivity]
  2. $\forall x, y.x = y \rightarrow y = x$ [symmetry]
  3. $\forall x, y, z.((x = y \wedge y = z) \rightarrow x = z)$ [transitivity]
  4. Function and predicate congruence
     - For each $n \in \mathbb{N}$ and n-ary function symbol f:

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \left(\bigwedge_{i=1}^{n} x_i = x_j \right) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

     - For each $n \in \mathbb{N}$ and n-ary predicate symbol p:

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \left(\bigwedge_{i=1}^{n} x_i = x_j \right) \rightarrow p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n)$$

Some concrete examples

$$a \neq b$$

Is sat? Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes*   Is valid? *No*

Some concrete examples

$$a \neq b$$

Is sat? *Yes*   Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat?   Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes* Is valid?

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes* Is valid? *Yes*

Some concrete examples

$$a \neq b$$

Is sat? *Yes*   Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes*   Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes*   Is valid? *Yes*

$$a * (f(b) + f(c)) = d \wedge \neg(b * (f(a) + f(c)) = d) \wedge a = b$$

Hint: treat $*$ and $+$ as an uninterpreted function.

Is sat?   Is valid?

Some examples are from [Bradley and Manna, 2007] and [Barrett et al., 2009]

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes* Is valid? *Yes*

$$a * (f(b) + f(c)) = d \wedge \neg(b * (f(a) + f(c)) = d) \wedge a = b$$

Hint: treat $*$ and $+$ as an uninterpreted function.

Is sat? *No* Is valid?

Some examples are from [Bradley and Manna, 2007] and [Barrett et al., 2009]

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes* Is valid? *Yes*

$$a * (f(b) + f(c)) = d \wedge \neg(b * (f(a) + f(c)) = d) \wedge a = b$$

Hint: treat $*$ and $+$ as an uninterpreted function.

Is sat? *No* Is valid? *No*

Some examples are from [Bradley and Manna, 2007] and [Barrett et al., 2009]

Some concrete examples

$$a \neq b$$

Is sat? *Yes* Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes* Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes* Is valid? *Yes*

$$a * (f(b) + f(c)) = d \wedge \neg(b * (f(a) + f(c)) = d) \wedge a = b$$

Hint: treat $*$ and $+$ as an uninterpreted function.

Is sat? *No* Is valid? *No*

Is the satisfiability of EUF-formulas decidable?

Some examples are from [Bradley and Manna, 2007] and [Barrett et al., 2009]

Some concrete examples

$$a \neq b$$

Is sat? *Yes*   Is valid? *No*

$$a = b \wedge b = c \leftrightarrow f(c) = f(a)$$

Is sat? *Yes*   Is valid? *Yes*

$$a = b \wedge b = c \implies g(f(a), b) = g(f(c), a)$$

Is sat? *Yes*   Is valid? *Yes*

$$a * (f(b) + f(c)) = d \wedge \neg(b * (f(a) + f(c)) = d) \wedge a = b$$

Hint: treat $*$ and $+$ as an uninterpreted function.

Is sat? *No*   Is valid? *No*

Is the satisfiability of EUF-formulas decidable? More next week!

Some examples are from [Bradley and Manna, 2007] and [Barrett et al., 2009]

Some theories of interest I

- Linear rational and Integer Arithmetic (LRA and LIA) - (week 3)

$$(x + y < 3 \wedge y > 2) \rightarrow x < 1$$

Used to model arithmetic (note that constraints are linear!)

- Difference logic

$$(a - b \leq 3 \wedge c - a \leq 2) \vee b - c \leq 10$$

Used to model arithmetic (note the restrictions, only difference of 2 constants, no strict inequalities)

- Reals (i.e., polynomial inequalities over the reals)

$$a^2 + 3ab + c \leq 3 \vee a - b \leq 2$$

Model geometric problems, problems in engineering, . . .

Some theories of interest II

- Arrays

$$\neg((write(a, i, v_1) \wedge j = i + 1) \rightarrow (read(a, i) < read(a, j, v)))$$

Model unbounded memory in programs

- Bit-Vectors - a bit-vector $x_{[n]}$ is a vector of bits of length n

$$x_{32}[15:0] = y_{[16]}[7:0] :: y_{[16]}[15:8]$$

Model hardware operations and low-level software

- Strings

$$y = \text{“}a'' \cdot x \wedge x = z \cdot \text{“}b'') \rightarrow y = \text{“}a'' \cdot w \cdot \text{“}b''$$

Model string constraints, for example for testing or security

Satisfiability Modulo Theory Problem

SMT Problem

The problem of deciding the satisfiability of quantifier-free formulas expressed in some *decidable* first order theory $\mathcal{T}$

Some remarks:

- Usually quantifier-free formulas, but SMT solver can deal with quantifiers (semi-decidable or focus on decidable subsets, like Effectively Propositional Logic)
- Also consider formulas obtained combining multiple theories, e.g., $\mathcal{T}_1 \cup T_2 \cup \ldots \cup T_n$

How can we decide the SMT problem?

Assumption from here onwards:

- ϕ is a quantifier-free formula
- ϕ is in conjunctive normal form (CNF)

How can we decide the SMT problem?

Assumption from here onwards:

- ϕ is a quantifier-free formula
- ϕ is in conjunctive normal form (CNF)

How can we decide the satisfiability of a $\Sigma_{\mathcal{T}}$-formula ϕ?

$$\phi := (x > 3 \vee x + y = 0) \wedge (y < 0 \vee x < 3)$$

How can we decide the SMT problem?

Assumption from here onwards:

- ϕ is a quantifier-free formula
- ϕ is in conjunctive normal form (CNF)

How can we decide the satisfiability of a $\Sigma_{\mathcal{T}}$-formula ϕ?

$$\phi := (x > 3 \vee x + y = 0) \wedge (y < 0 \vee x < 3)$$

- We can see ϕ as a Propositional formula (i.e., interpreting each theory predicate as a Boolean predicate)

$$(P_1 \vee P_2) \wedge (P_3 \vee P_4)$$

How can we decide the SMT problem?

Assumption from here onwards:

- ϕ is a quantifier-free formula
- ϕ is in conjunctive normal form (CNF)

How can we decide the satisfiability of a $\Sigma_{\mathcal{T}}$-formula ϕ?

$$\phi := (x > 3 \vee x + y = 0) \wedge (y < 0 \vee x < 3)$$

- We can see ϕ as a Propositional formula (i.e., interpreting each theory predicate as a Boolean predicate)

$$(P_1 \vee P_2) \wedge (P_3 \vee P_4)$$

- We can enumerate all the μ^b Propositional models of $(P_1 \vee P_2) \wedge (P_3 \vee P_4)$

$$\mu^b := \{P1 \mapsto \mathit{true}, P2 \mapsto \mathit{false}, P3 \mapsto \mathit{true}, P4 \mapsto \mathit{false}\}$$

How can we decide the SMT problem?

Assumption from here onwards:

- ϕ is a quantifier-free formula
- ϕ is in conjunctive normal form (CNF)

How can we decide the satisfiability of a $\Sigma_{\mathcal{T}}$-formula ϕ?

$$\phi := (x > 3 \vee x + y = 0) \wedge (y < 0 \vee x < 3)$$

- We can see ϕ as a Propositional formula (i.e., interpreting each theory predicate as a Boolean predicate)

$$(P_1 \vee P_2) \wedge (P_3 \vee P_4)$$

- We can enumerate all the μ^b Propositional models of $(P_1 \vee P_2) \wedge (P_3 \vee P_4)$

$$\mu^b := \{P1 \mapsto \mathit{true}, P2 \mapsto \mathit{false}, P3 \mapsto \mathit{true}, P4 \mapsto \mathit{false}\}$$

- For each model μ^b, we can check if the conjunction is *consistent* in the theory $\mathcal{T}$:

$$x > 3 \wedge \neg x + y = 0 \wedge y < 0 \wedge \neg x < 3$$

It's satisfiable: $\mu := \{x \mapsto 4, y \mapsto 0\}$

This is the lazy approach to SMT (e.g., see [Sebastiani, 2007])

Offline lazy approach to SMT solving

procedure $\mathcal{T}$-DPLL-offline(ϕ)
  $\phi^b := TO_{\mathbb{B}}(\phi)$
  while true **do**
    $res, \mu^b := DPLL(\phi^b)$
    if $res = true$ **then**
      $\mu := TO_{\mathcal{T}}(\mu^b)$
      $res := \mathcal{T}consistent(\mu)$
      if $res = true$ **then**
        return `SAT`
      else
        $\phi^b := \phi^b \wedge \neg\mu^b$
    else
      return `UNSAT`

Offline lazy approach to SMT solving

procedure $\mathcal{T}$-DPLL-offline(ϕ)
  $\boxed{\phi^b := TO_{\mathbb{B}}(\phi)}$
  while true **do**
    $\boxed{res, \mu^b := DPLL(\phi^b)}$
    if $res = true$ **then**
      $\mu := TO_{\mathcal{T}}(\mu^b)$
      $res := \mathcal{T}consistent(\mu)$
      if $res = true$ **then**
        return `SAT`
      else
        $\phi^b := \phi^b \wedge \neg\mu^b$
    else
      return `UNSAT`

- Boolean reasoning: delegates the enumeration to the DPLL (or CDCL solver)

Offline lazy approach to SMT solving

procedure $\mathcal{T}$-DPLL-offline(ϕ)
  $\boxed{\phi^b := TO_{\mathbb{B}}(\phi)}$
  while true **do**
    $\boxed{res, \mu^b := DPLL(\phi^b)}$
    if $res = true$ **then**
      $\mu := TO_{\mathcal{T}}(\mu^b)$
      $\boxed{res := \mathcal{T}consistent(\mu)}$
      if $res = true$ **then**
        return `SAT`
      else
        $\phi^b := \phi^b \land \neg\mu^b$
    else
      return `UNSAT`

- Boolean reasoning: delegates the enumeration to the DPLL (or CDCL solver)
- Theory reasoning: check consistency of $\mathcal{T}$-literals (simpler problem) with a dedicated $\mathcal{T}$-**solver**

Offline lazy approach to SMT solving

procedure $\mathcal{T}$-DPLL-offline(ϕ)
  $\boxed{\phi^b := TO_{\mathbb{B}}(\phi)}$
  while true **do**
    $\boxed{res, \mu^b := DPLL(\phi^b)}$
    if $res = true$ **then**
      $\mu := TO_{\mathcal{T}}(\mu^b)$
      $\boxed{res := \mathcal{T}consistent(\mu)}$
      if $res = true$ **then**
        return `SAT`
      else
        $\boxed{\phi^b := \phi^b \wedge \neg\mu^b}$
    else
      return `UNSAT`

- Boolean reasoning: delegates the enumeration to the DPLL (or CDCL solver)
- Theory reasoning: check consistency of $\mathcal{T}$-literals (simpler problem) with a dedicated $\mathcal{T}$-**solver**
- Boolean reasoning: add a blocking clause μ_b to avoid to "visit" the same Boolean model

Drawbacks of the offline approach

The offline approach “loosely” integrates the CDCL solver and the theory solvers:

Drawbacks of the offline approach

The offline approach “loosely” integrates the CDCL solver and the theory solvers:

- Restart the Boolean search from scratch after blocking a model μ^b

  loose learned clauses, arbitrary “restart”

Drawbacks of the offline approach

The offline approach “loosely” integrates the CDCL solver and the theory solvers:

- Restart the Boolean search from scratch after blocking a model μ^b

  loose learned clauses, arbitrary “restart”

- Only blocks a complete model μ^b “weak” pruning of the search space
  - What is the effect on the backjumping of CDCL?
  - What is the effect on learning clauses?

Drawbacks of the offline approach

The offline approach "loosely" integrates the CDCL solver and the theory solvers:

- Restart the Boolean search from scratch after blocking a model μ^b

  loose learned clauses, arbitrary "restart"
- Only blocks a complete model μ^b "weak" pruning of the search space
  - What is the effect on the backjumping of CDCL?
  - What is the effect on learning clauses?
- Check for $\mathcal{T}$-consistency of full models μ could be unsatisfiable "earlier"
  - Can we detect unsatisfiability due to theory "earlier" in the search?
  - Can we generalize Boolean constraint Propagation to the theory $\mathcal{T}$?

Could we have a tighter integration of CDCL and the $\mathcal{T}$-Solver?

The online approach - DPLL($\mathcal{T}$)

Similar architecture to CDCL, but integrates the theory reasoning:

procedure DPLL-$\mathcal{T}(\phi)$
  May pre-process ϕ (e.g., propagation)
  $\mu := \emptyset$
  $\phi^b := TO_{\mathbb{B}}(\phi); \mu^b = TO_{\mathbb{B}}(\mu)$
  while true **do**
    $\mathcal{T} - Decide(\phi^b, \mu^b)$
    while true **do**
      $res := \mathcal{T} - Deduce(\phi^b)$
      if $res = true$ **then**
        $\mu := TO_{\mathcal{T}}(\mu^b)$
        return `SAT`
      else if $res = conflict$ **then**
        $lvl := \mathcal{T} - Analyze(\phi^b, \mu^b)$
        if $lvl = 0$ **then**
          return `UNSAT`
        else
          $\mathcal{T} - Backtrack(lvl, \phi^b, \mu^b)$

The online approach - DPLL($\mathcal{T}$)

procedure DPLL-$\mathcal{T}(\phi)$
  May pre-process ϕ (e.g., propagation)
  $\mu := \emptyset$
  $\phi^b := TO_{\mathbb{B}}(\phi); \mu^b = TO_{\mathbb{B}}(\mu)$
  while true **do**
    $\boxed{\mathcal{T} - Decide(\phi^b, \mu^b)}$
    while true **do**
      $res := \mathcal{T} - Deduce(\phi^b)$
      if $res = true$ **then**
        $\mu := TO_{\mathcal{T}}(\mu^b)$
        return `SAT`
      else if $res = conflict$ **then**
        $lvl := \mathcal{T} - Analyze(\phi^b, \mu^b)$
        if $lvl = 0$ **then**
          return `UNSAT`
        else
          $\mathcal{T} - Backtrack(lvl, \phi^b, \mu^b)$

Similar architecture to CDCL, but integrates the theory reasoning:

- *decision*: choose an unassigned literal l from ϕ^b (similar to DPLL)

The online approach - DPLL($\mathcal{T}$)

procedure DPLL-$\mathcal{T}(\phi)$
  May pre-process ϕ (e.g., propagation)
  $\mu := \emptyset$
  $\phi^b := TO_{\mathbb{B}}(\phi); \mu^b = TO_{\mathbb{B}}(\mu)$
  while true **do**
    $\boxed{\mathcal{T} - Decide(\phi^b, \mu^b)}$
    while true **do**
      $\boxed{res := \mathcal{T} - Deduce(\phi^b)}$
      if $res = true$ **then**
        $\mu := TO_{\mathcal{T}}(\mu^b)$
        return SAT
      else if $res = conflict$ **then**
        $lvl := \mathcal{T} - Analyze(\phi^b, \mu^b)$
        if $lvl = 0$ **then**
          return UNSAT
        else
          $\mathcal{T} - Backtrack(lvl, \phi^b, \mu^b)$

Similar architecture to CDCL, but integrates the theory reasoning:

- *decision*: choose an unassigned literal l from ϕ^b (similar to DPLL)
- *deduce*: iteratively deduces a literal l^b s.t. $\phi^b \wedge \mu^b \models l^b$
  - In case, add l to μ and check the consistency of μ (in the theory)
  - Optimized with $\mathcal{T}$*-propagation* and *early pruning*.

The online approach - DPLL($\mathcal{T}$)

procedure DPLL-$\mathcal{T}(\phi)$
May pre-process ϕ (e.g., propagation)
$\mu := \emptyset$
$\phi^b := TO_{\mathbb{B}}(\phi); \mu^b = TO_{\mathbb{B}}(\mu)$
while true **do**
$\boxed{\mathcal{T} - Decide(\phi^b, \mu^b)}$
while true **do**
$\boxed{res := \mathcal{T} - Deduce(\phi^b)}$
if $res = true$ **then**
$\mu := TO_{\mathcal{T}}(\mu^b)$
return `SAT`
else if $res = conflict$ **then**
$\boxed{lvl := \mathcal{T} - Analyze(\phi^b, \mu^b)}$
if $lvl = 0$ **then**
return `UNSAT`
else
$\mathcal{T} - Backtrack(lvl, \phi^b, \mu^b)$

Similar architecture to CDCL, but integrates the theory reasoning:

- *decision*: choose an unassigned literal l from ϕ^b (similar to DPLL)
- *deduce*: iteratively deduces a literal l^b s.t. $\phi^b \wedge \mu^b \models l^b$
  - In case, add l to μ and check the consistency of μ (in the theory)
  - Optimized with $\mathcal{T}$*-propagation* and *early pruning*.
- *analyze*: detect the conflict clauses and determines the decision level to backtrack to.
  - Produces also a *theory conflicts*

The online approach - DPLL($\mathcal{T}$)

procedure DPLL-$\mathcal{T}(\phi)$
    May pre-process ϕ (e.g., propagation)
    $\mu := \emptyset$
    $\phi^b := TO_{\mathbb{B}}(\phi); \mu^b = TO_{\mathbb{B}}(\mu)$
    while true **do**
        $\boxed{\mathcal{T} - Decide(\phi^b, \mu^b)}$
        while true **do**
            $\boxed{res := \mathcal{T} - Deduce(\phi^b)}$
            if $res = true$ **then**
                $\mu := TO_{\mathcal{T}}(\mu^b)$
                return `SAT`
            else if $res = conflict$ **then**
                $\boxed{lvl := \mathcal{T} - Analyze(\phi^b, \mu^b)}$
                if $lvl = 0$ **then**
                    return `UNSAT`
                else
                    $\boxed{\mathcal{T} - Backtrack(lvl, \phi^b, \mu^b)}$

Similar architecture to CDCL, but integrates the theory reasoning:

- *decision*: choose an unassigned literal l from ϕ^b (similar to DPLL)
- *deduce*: iteratively deduces a literal l^b s.t. $\phi^b \wedge \mu^b \models l^b$
    - In case, add l to μ and check the consistency of μ (in the theory)
    - Optimized with $\mathcal{T}$*-propagation* and *early pruning*.
- *analyze*: detect the conflict clauses and determines the decision level to backtrack to.
    - Produces also a *theory conflicts*
- *backtrack*: block the conflict clause and bactracks to the level lvl (similar to DPLL)
    - $\mathcal{T}$-backjumping and $\mathcal{T}$-learning

$\mathcal{T}$-backjumping and $\mathcal{T}$-learning

When we invoke the $\mathcal{T}$-solver on an assignment μ, and μ is not consistent:

- we would like to infer a small subset $\nu \subseteq \mu$ such that ν is not consistent (i.e., ν is a *conflict set*)

a smaller ν can reduce more the search space

$\mathcal{T}$-backjumping and $\mathcal{T}$-learning

When we invoke the $\mathcal{T}$-solver on an assignment μ, and μ is not consistent:

- we would like to infer a small subset $\nu \subseteq \mu$ such that ν is not consistent (i.e., ν is a *conflict set*)

  a smaller ν can reduce more the search space

- we can use $\neg\nu^b$ to guide the conflict analysis of CDCL
  In practice, we can consider $\mathcal{T}$-propagations (see later) as unit-propagation in the implication graph.

$\mathcal{T}$-backjumping and $\mathcal{T}$-learning

When we invoke the $\mathcal{T}$-solver on an assignment μ, and μ is not consistent:

- we would like to infer a small subset $\nu \subseteq \mu$ such that ν is not consistent (i.e., ν is a *conflict set*)

  a smaller ν can reduce more the search space

- we can use $\neg\nu^b$ to guide the conflict analysis of CDCL
  In practice, we can consider $\mathcal{T}$-propagations (see later) as unit-propagation in the implication graph.
- $\neg\nu^b$ can be learned as a conflict clause by the sat solver

Ideally, the $\mathcal{T}$-solver should search for a minimal conflict set $\nu \subseteq \mu$

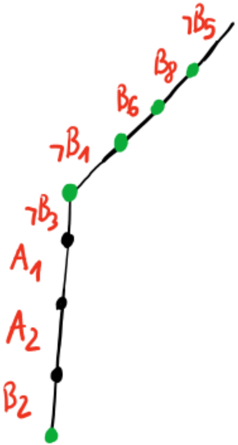
- in practice, finding a minimal set ν is expensive
- $\mathcal{T}$-solvers compromise performance and size of the conflict set ν

$\mathcal{T}$-Backjumping and $\mathcal{T}$-Learning

Examples [4] over the theory of Linear Integer Arithmetic:

$\neg(2x_2 - x3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$\neg A_2 \vee x1 - x5 \leq 1$	$\neg A_2 \vee B_2$
$3x_1 - 2x_2 \leq 3 \vee A_2$	$B_3 \vee A_2$
$\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$A_1 \vee 3x_1 - 2x_2 \leq 3$	$A_1 \vee B_3$
$x_2 - x_4 \leq 6 \vee x_5 = 5 - 3x_4 \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$A_1 \vee x_3 = 3x_5 + 4 \vee A_2$	$A_1 \vee B_8 \vee A_2$

- $\mu^b := \neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2$



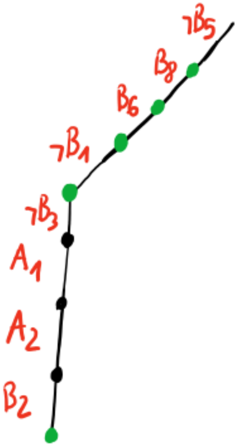
[4] Example 5.2 [Sebastiani, 2007]

$\mathcal{T}$-Backjumping and $\mathcal{T}$-Learning

Examples [4] over the theory of Linear Integer Arithmetic:

$\neg(2x_2 - x3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$\neg A_2 \vee x1 - x5 \leq 1$	$\neg A_2 \vee B_2$
$3x_1 - 2x_2 \leq 3 \vee A_2$	$B_3 \vee A_2$
$\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$A_1 \vee 3x_1 - 2x_2 \leq 3$	$A_1 \vee B_3$
$x_2 - x_4 \leq 6 \vee x_5 = 5 - 3x_4 \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$A_1 \vee x_3 = 3x_5 + 4 \vee A_2$	$A_1 \vee B_8 \vee A_2$

- $\mu^b := \neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2$
- $\neg B_5 \wedge B_8 \wedge B_2$ is inconsistent in the theory.
  We have a conflict clause $B_5 \vee \neg B_8 \vee \neg B_2$

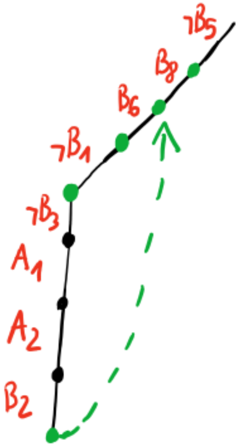


4 Example 5.2 [Sebastiani, 2007]

$\mathcal{T}$-Backjumping and $\mathcal{T}$-Learning

Examples [4] over the theory of Linear Integer Arithmetic:

$\neg(2x_2 - x3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$\neg A_2 \vee x1 - x5 \leq 1$	$\neg A_2 \vee B_2$
$3x_1 - 2x_2 \leq 3 \vee A_2$	$B_3 \vee A_2$
$\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$A_1 \vee 3x_1 - 2x_2 \leq 3$	$A_1 \vee B_3$
$x_2 - x_4 \leq 6 \vee x_5 = 5 - 3x_4 \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$A_1 \vee x_3 = 3x_5 + 4 \vee A_2$	$A_1 \vee B_8 \vee A_2$

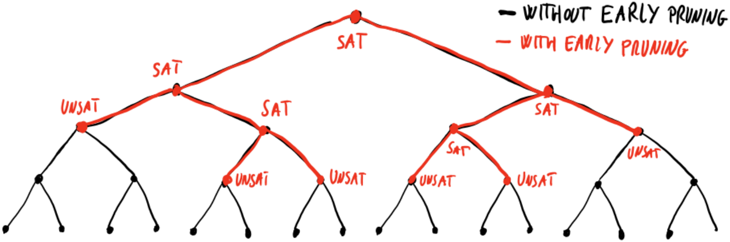


- $\mu^b := \neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2$
- $\neg B_5 \wedge B_8 \wedge B_2$ is inconsistent in the theory. We have a conflict clause $B_5 \vee \neg B_8 \vee \neg B_2$
- The solver backtracks removing all literals up to $\{\neg B_5, B_8\}$.

[4] Example 5.2 [Sebastiani, 2007]

Early Pruning

- Check the $\mathcal{T}$-consistency of each partial assignment found by CDCL
- Backtrack immediately if the assignment is not consitent
- Main advantage: prunes the search space



Technical considerations:

- Requires an *incremental* and *backtrackable* $\mathcal{T}$-solver
- Checking consistency for every decision is not cheap!
  Heuristics, use incomplete but cheap consistency checks (e.g., simplex on integer arithmetic)

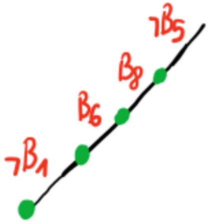
$\mathcal{T}$-Propagation

- Used in $\mathcal{T} - \textit{decide}$ to deduce the value of unassigned literals:
  - When the current (partial) assignment μ is satisfiable
  - The $\mathcal{T}$-solver can return a set ν of unassigned literals such that $\mu \models_{\mathcal{T}} \nu$
  - $\mathcal{T}$-Propagation can unit propagate the implied ν (similarly to Boolean Constraint Propagation)

$\mathcal{T}$-Propagation

- Used in $\mathcal{T} - decide$ to deduce the value of unassigned literals:
  - When the current (partial) assignment μ is satisfiable
  - The $\mathcal{T}$-solver can return a set ν of unassigned literals such that $\mu \models_{\mathcal{T}} \nu$
  - $\mathcal{T}$-Propagation can unit propagate the implied ν (similarly to Boolean Constraint Propagation)

$\neg(2x_2 - x3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$\neg A_2 \vee x1 - x5 \leq 1$	$\neg A_2 \vee B_2$
$3x_1 - 2x_2 \leq 3 \vee A_2$	$B_3 \vee A_2$
$\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$A_1 \vee 3x_1 - 2x_2 \leq 3$	$A_1 \vee B_3$
$x_2 - x_4 \leq 6 \vee x_5 = 5 - 3x_4 \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$A_1 \vee x_3 = 3x_5 + 4 \vee A_2$	$A_1 \vee B_8 \vee A_2$

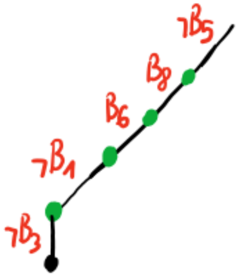


- $\mu^b := \neg B_5, B_8, B_6, \neg B_1$

$\mathcal{T}$-Propagation

- Used in $\mathcal{T}$ − *decide* to deduce the value of unassigned literals:
  - When the current (partial) assignment μ is satisfiable
  - The $\mathcal{T}$-solver can return a set ν of unassigned literals such that $\mu \models_{\mathcal{T}} \nu$
  - $\mathcal{T}$-Propagation can unit propagate the implied ν (similarly to Boolean Constraint Propagation)

$\neg(2x_2 - x3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$\neg A_2 \vee x1 - x5 \leq 1$	$\neg A_2 \vee B_2$
$3x_1 - 2x_2 \leq 3 \vee A_2$	$B_3 \vee A_2$
$\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$A_1 \vee 3x_1 - 2x_2 \leq 3$	$A_1 \vee B_3$
$x_2 - x_4 \leq 6 \vee x_5 = 5 - 3x_4 \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$A_1 \vee x_3 = 3x_5 + 4 \vee A_2$	$A_1 \vee B_8 \vee A_2$



- $\mu^b := \neg B_5, B_8, B_6, \neg B_1$
- $\neg(3x_1 - x_3 \leq 6) \wedge x_3 = 3x_5 + 4 \wedge x_2 - x_4 \leq 6 \wedge (2x_2 - x3 > 2) \models_{\mathcal{T}} \neg(3x_1 - 2x_2 \leq 3)$

Other Approaches to the SMT problem

- The eager approach to SMT: convert the problem to a SAT problem
  Used to decide formulas over the bit-vector theory.

Other Approaches to the SMT problem

- The eager approach to SMT: convert the problem to a SAT problem
  Used to decide formulas over the bit-vector theory.
- Abstract DPLL: abstract formulation of DPLL as a transition system
  Allow to reason about the properties of different variants of the algorith (e.g., correctness, completeness, termination)

Other Approaches to the SMT problem

- The eager approach to SMT: convert the problem to a SAT problem
  Used to decide formulas over the bit-vector theory.
- Abstract DPLL: abstract formulation of DPLL as a transition system
  Allow to reason about the properties of different variants of the algorith (e.g., correctness, completeness, termination)
- Model-Constructing Satisfiability:
  - Assignments (e.g., decisions) to theory variables, not just Propositional
    i.e., no Boolean abstraction anymore, and no enumeration of the models of the Boolean Abstraction.
  - Decisions and explanations can be done for new atoms (obtained from unsatisfiable proofs)
  - Several implementation, efficient for Non-Linear Real Arithmetic

To sum up

What did we see today:

- SMT is a fundamental tool in several area (e.g., verification, program analysis, planning, . . .)
- Satisfiability of (full) First Order logic is undecidable - so what can we do?
- Theories allow us to have decision procedures - motivation to look at the SMT problem
- The lazy approach to SMT: best of both words (CDCL SAT solver) and efficient theory solvers

Next week: how to decide consistency for the theory of Equalities and Uninterpreted Functions

References I

Barnett, M., Chang, B. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2005).
Boogie: A modular reusable verifier for object-oriented programs.
In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer.

Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009).
Satisfiability modulo theories.
In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press.

Bradley, A. R. and Manna, Z. (2007).
The calculus of computation - decision procedures with applications to verification.
Springer.

References II

Cashmore, M., Magazzeni, D., and Zehtabi, P. (2020).
Planning for hybrid systems via satisfiability modulo theories.
J. Artif. Intell. Res., 67:235–283.

Cimatti, A., Do, M., Micheli, A., Roveri, M., and Smith, D. E. (2018).
Strong temporal planning with uncontrollable durations.
Artif. Intell., 256:1–34.

Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. (2015).
Hycomp: An smt-based model checker for hybrid systems.
In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 52–67. Springer.

Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. (2016).
Infinite-state invariant checking with IC3 and predicate abstraction.
Formal Methods Syst. Des., 49(3):190–218.

References III

de Moura, L. M. and Bjørner, N. (2011).
Satisfiability modulo theories: introduction and applications.
Commun. ACM, 54(9):69–77.

Dutertre, B., Jovanovic, D., and Navas, J. A. (2018).
Verification of fault-tolerant protocols with sally.
In Dutle, A., Muñoz, C. A., and Narkawicz, A., editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 113–120. Springer.

Filliâtre, J.-C. and Paskevich, A. (2013).
Why3 — where programs meet provers.
In Felleisen, M. and Gardner, P., editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer.

References IV

Godefroid, P., Klarlund, N., and Sen, K. (2005).
DART: directed automated random testing.
In Sarkar, V. and Hall, M. W., editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM.

Godefroid, P., Levin, M. Y., and Molnar, D. A. (2012).
SAGE: whitebox fuzzing for security testing.
Commun. ACM, 55(3):40–44.

Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. L. (2004).
Abstractions from proofs.
In Jones, N. D. and Leroy, X., editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244. ACM.

References V

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010).
Oracle-guided component-based program synthesis.
In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM.

Leino, K. R. M. (2010).
Dafny: An automatic program verifier for functional correctness.
In Clarke, E. M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer.

References VI

McMillan, K. L. and Padon, O. (2020).
Ivy: A multi-modal verification tool for distributed algorithms.
In Lahiri, S. K. and Wang, C., editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 190–202. Springer.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016).
Angelix: scalable multiline program patch synthesis via symbolic analysis.
In Dillon, L. K., Visser, W., and Williams, L. A., editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 691–701. ACM.

Sebastiani, R. (2007).
Lazy satisability modulo theories.
J. Satisf. Boolean Model. Comput., 3(3-4):141–224.

References VII

Wolfman, S. A. and Weld, D. S. (1999).
The LPSAT engine & its application to resource planning.
In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 310–317. Morgan Kaufmann.