Using SMT Solvers (to reason about programs)

Martin Kellogg

Reading quiz: SMT solvers

Reading quiz: SMT solvers

Q1: **TRUE** or **FALSE**: Z3 supports finding an "optimal" satisfying assignment that maximizes or minimizes some objective function

Q2: Which of these theories was **NOT** mentioned as one of the theories supported by Z3 in the reading?

A. polygons
B. equality of uninterpreted functions
C. linear real arithmetic
D. arrays

Reading quiz: SMT solvers

Q1: **TRUE** or **FALSE**: Z3 supports finding an "optimal" satisfying assignment that maximizes or minimizes some objective function

Q2: Which of these theories was **NOT** mentioned as one of the theories supported by Z3 in the reading?
A. polygons
B. equality of uninterpreted functions
C. linear real arithmetic
D. arrays

Reading quiz: SMT solvers

Q1: **TRUE** or **FALSE**: Z3 supports finding an "optimal" satisfying assignment that maximizes or minimizes some objective function

Q2: Which of these theories was **NOT** mentioned as one of the theories supported by Z3 in the reading?

A. polygons
B. equality of uninterpreted functions
C. linear real arithmetic
D. arrays

Agenda: SMT solvers

- **Motivation: reasoning about formulas**
- SAT solving: DPLL
- SMT solving: Nelson-Oppen and DPLL(T)
- SMT in practice: brief intro to Z3 and SMT-LIB

Motivation: reasoning about formulas

- Recall our discussion of **symbolic execution** from earlier in this class

Motivation: reasoning about formulas

- Recall our discussion of **symbolic execution** from earlier in this class
  - effectively, it **uses math** to figure out which values of each variable will cause the program to take particular paths

Motivation: reasoning about formulas

- Recall our discussion of symbolic execution from earlier in this class
    - effectively, it uses math to figure out which values of each variable will cause the program to take particular paths
        - goal: create test cases that definitely increase coverage

Motivation: reasoning about formulas

- Recall our discussion of symbolic execution from earlier in this class
  - effectively, it uses math to figure out which values of each variable will cause the program to take particular paths
    - goal: create test cases that definitely increase coverage
- At the time, we deferred the question of how we would solve path predicates **automatically**

Motivation: reasoning about formulas

- Recall our discussion of **symbolic execution** from earlier in this class
  - effectively, it **uses math** to figure out which values of each variable will cause the program to take particular paths
    - goal: create test cases that definitely increase coverage
- At the time, we deferred the question of how we would solve path predicates **automatically**
  - recall that a ***path predicate*** is a formula over program variables that is true when a particular path is executed

Motivation: reasoning about formulas

For example, consider this program:

```
int simpleMath(int a, int b) {
  assert(b > 0);
  if(a + b == a * b) {
    return 1;
  }
  return 0;
}
```

Motivation: reasoning about formulas

For example, consider this program:

```
int simpleMath(int a, int b) {
  assert(b > 0);
  if(a + b == a * b) {
    return 1;
  }
  return 0;
}
```

suppose we want to **cover** this line (`return 1`)

Motivation: reasoning about formulas

For example, consider this program:

```
int simpleMath(int a, int b) {
  assert(b > 0);
  if(a + b == a * b) {
    return 1;
  }
  return 0;
}
```

suppose we want to **cover** this line (`return 1`)

what's its **path predicate**?

Motivation: reasoning about formulas

For example, consider this program:

```
int simpleMath(int a, int b) {
  assert(b > 0);
  if(a + b == a * b) {
    return 1;
  }
  return 0;
}
```

suppose we want to **cover** this line (`return 1`)

what's its **path predicate**?

$b > 0 \,\&\&\, a + b == a * b$

Motivation: reasoning about formulas

For example, consider this program:

```
int simpleMath(int a, int b) {
  assert(b > 0);
  if(a + b == a * b) {
    return 1;
  }
  return 0;
}
```

suppose we want to **cover** this line (`return 1`)

what's its **path predicate**?

$b > 0 \; \&\& \; a + b == a * b$

Key question: are there a, b **such that this is true**?

Motivation: reasoning about formulas

- As a human, it is relatively easy to solve the example on the previous slide

Motivation: reasoning about formulas

- As a human, it is relatively easy to solve the example on the previous slide
  - but real examples are **many orders of magnitude larger**!

Motivation: reasoning about formulas

- As a human, it is relatively easy to solve the example on the previous slide
  - but real examples are **many orders of magnitude larger**!
  - we'd like to **automate** the task of checking if there's a solution

Motivation: reasoning about formulas

- As a human, it is relatively easy to solve the example on the previous slide
    - but real examples are **many orders of magnitude larger**!
    - we'd like to **automate** the task of checking if there's a solution
- In our lecture on symbolic execution, I briefly mentioned that ***SMT solvers*** are the modern tool that we'd use to do so

Motivation: reasoning about formulas

- As a human, it is relatively easy to solve the example on the previous slide
  - but real examples are **many orders of magnitude larger**!
  - we'd like to **automate** the task of checking if there's a solution
- In our lecture on symbolic execution, I briefly mentioned that ***SMT solvers*** are the modern tool that we'd use to do so
  - let's do it now: https://www.philipzucker.com/z3-rise4fun/

Motivation: reasoning about formulas

- Reasoning about formulas is useful for **more than symbolic execution**

Motivation: reasoning about formulas

- Reasoning about formulas is useful for **more than symbolic execution**
- Other applications **include**:

Motivation: reasoning about formulas

- Reasoning about formulas is useful for **more than symbolic execution**
- Other applications **include**:
  - reasoning about program correctness (automating pen-and-paper proofs!)
  - reasoning about program equivalence (cf. equivalent mutant problem)
  - program synthesis
  - program repair
  - etc.

What is an SMT solver, exactly?

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note “tries to”: boolean satisfiability is **NP-complete**

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note “tries to”: boolean satisfiability is **NP-complete**
- “theories” refers to non-boolean parts of the formula
  - for example, a solver might support a **theory of real numbers**

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note "tries to": boolean satisfiability is NP-complete
- "theories" refers to non-boolean parts of the formula
    - for example, a solver might support a theory of real numbers
- different solvers might support different theories

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note "tries to": boolean satisfiability is **NP-complete**
- "theories" refers to non-boolean parts of the formula
  - for example, a solver might support a **theory of real numbers**
- different solvers might support different theories
  - much of today's reading was about various theories that Z3 supports, such as ***Equality of Uninterpreted Functions*** (EUF) and the theory of Arrays

How SMT solvers are used

How SMT solvers are used

- the **key idea** behind using an SMT solver in program analysis is to **reduce** a problem to the satisfiability of some formula
  - “reduce” here means “do a reduction”, like in your theory class

How SMT solvers are used

- the **key idea** behind using an SMT solver in program analysis is to **reduce** a problem to the satisfiability of some formula
  - “reduce” here means “do a reduction”, like in your theory class
- for example, symbolic execution reduces covering a particular line of code to the problem of whether a path predicate is satisfiable
  - then uses the SMT solver as an **oracle**

How SMT solvers are used

- the **key idea** behind using an SMT solver in program analysis is to **reduce** a problem to the satisfiability of some formula
  - "reduce" here means "do a reduction", like in your theory class
- for example, symbolic execution reduces covering a particular line of code to the problem of whether a path predicate is satisfiable
  - then uses the SMT solver as an **oracle**
- note that in the symbolic execution case, we're interested in the satisfying assignment (it's the test case)

How SMT solvers are used

- the **key idea** behind using an SMT solver in program analysis is to **reduce** a problem to the satisfiability of some formula
  - "reduce" here means "do a reduction", like in your theory class
- for example, symbolic execution reduces covering a particular line of code to the problem of whether a path predicate is satisfiable
  - then uses the SMT solver as an **oracle**
- note that in the symbolic execution case, we're interested in the satisfying assignment (it's the test case)
  - in many other interesting cases, we want to check a formula's **validity**: that is, whether it is true for all values of its inputs

Validity vs satisfiability

- Suppose we have some formula F

Validity vs satisfiability

- Suppose we have some formula F
    - we **want to prove** $\forall x. F(x)$ is true

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)
  - we have an oracle for **finding satisfying assignments** (the SMT solver)

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)
  - we have an oracle for **finding satisfying assignments** (the SMT solver)
- Two-step transformation:
  - ∀x. F(x) is true ->

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)
  - we have an oracle for **finding satisfying assignments** (the SMT solver)
- Two-step transformation:
  - ∀x. F(x) is true -> ¬∃x. F(x) is false

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)
  - we have an oracle for **finding satisfying assignments** (the SMT solver)
- Two-step transformation:
  - ∀x. F(x) is true -> ¬∃x. F(x) is false
  - ¬∃x. F(x) is false -> ¬∃x. ¬F(x) is true

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (*x* here stands for the ***free variables*** of F, which you can think of as the inputs)
  - we have an oracle for **finding satisfying assignments** (the SMT solver)
- Two-step transformation:
  - ∀x. F(x) is true -> ¬∃x. F(x) is false
  - ¬∃x. F(x) is false -> ¬∃x. ¬F(x) is true
    - This is exactly equivalent to asking if ¬F(x) is **satisfiable**

Validity vs satisfiability

- Suppose we have some formula F
  - we **want to prove** ∀x. F(x) is true
    - (x here stands for the ***free varia***
      think of as the inputs)
  - we have an oracle for **finding satis**
    solver)
- Two-step transformation:
  - ∀x. F(x) is true -> ¬∃x. F(x) is false
  - ¬∃x. F(x) is false -> ¬∃x. ¬F(x) is true
    - This is exactly equivalent to asking if ¬F(x) is **satisfiable**

This means that we can use an SMT solver to check either **validity** or **satisfiability**!

- useful for e.g. proving program equivalence

Goals for today

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**
  - I don't expect you to be able to go out and build one right away

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**
  - I don't expect you to be able to go out and build one right away
  - but to use a tool effectively, it's important to understand the basic ideas that make it work

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**
    - I don't expect you to be able to go out and build one right away
    - but to use a tool effectively, it's important to understand the basic ideas that make it work
- Goal #2: understand how to **use and apply an SMT solver** to real-world program analysis problems

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**
  - I don't expect you to be able to go out and build one right away
  - but to use a tool effectively, it's important to understand the basic ideas that make it work
- Goal #2: understand how to **use and apply an SMT solver** to real-world program analysis problems
  - this is what the homework will ask you to do
    - and was also the main subject of today's reading

Goals for today

- Goal #1: understand the basics of **how an SMT solver works**
  - I don't expect you to be able to go out and build one right away
  - but to use a tool effectively, it's important to understand the basic ideas that make it work
- Goal #2: understand how to **use and apply an SMT solver** to real-world program analysis problems
  - this is what the homework will ask you to do
    - and was also the main subject of today's reading
  - hopefully you will also get a sense for **when and when not** to apply an SMT-based tool

Agenda: SMT solvers

- Motivation: reasoning about formulas
- **SAT solving: DPLL**
- SMT solving: Nelson-Oppen and DPLL(T)
- SMT in practice: brief intro to Z3 and SMT-LIB

Review: basics of SAT

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false)

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false) connected by the ***boolean operators*** ($\wedge$ for logical and, $\vee$ for logical or, and $\neg$ for logical negation)

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false) connected by the ***boolean operators*** ($\wedge$ for logical and, $\vee$ for logical or, and $\neg$ for logical negation)

- a boolean formula is ***satisfiable*** iff there exists an ***assignment*** of the variables to true and false that makes the formula as a whole evaluate to true

Review: basics of SAT

- You sh SAT *problem* course
  - bu

Example boolean formulas:

- $a \lor b \land \neg c$
- $(P \land Q) \lor (Q \land \neg R)$
- etc.

Definition ., symbols that can be *operators* ($\land$ for logical and, $\lor$ for logical or, and $\neg$ for logical negation)

- a boolean formula is ***satisfiable*** iff there exists an ***assignment*** of the variables to true and false that makes the formula as a whole evaluate to true

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
    - is $X \lor Y$ satisfiable?

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \lor Y$ satisfiable?
    - yes: X -> true, Y -> false is a satisfying assignment

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \lor Y$ satisfiable?
    - yes: X -> true, Y -> false is a satisfying assignment
  - is $X \land \neg X$ satisfiable?

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \lor Y$ satisfiable?
    - yes: X -> true, Y -> false is a satisfying assignment
  - is $X \land \neg X$ satisfiable?
    - no: there is no choice of X that makes both X and $\neg X$ true at the same time

SAT solving: how hard is it?

- If I’m asking, it’s probably difficult. But how hard?

SAT solving: how hard is it?

- If I’m asking, it’s probably difficult. But how hard?
- Answer: **NP-Complete**

SAT solving: how hard is it?

- If I'm asking, it's probably difficult. But how hard?
- Answer: **NP-Complete**
  - This is the classic **Cook-Levin theorem** (proved in the 1970s)
    - boolean SAT is the "original" NP-complete problem!

SAT solving: how hard is it?

- If I'm asking, it's probably difficult. But how hard?
- Answer: **NP-Complete**
  - This is the classic **Cook-Levin theorem** (proved in the 1970s)
    - boolean SAT is the "original" NP-complete problem!
    - in NP because you can verify that an assignment makes the formula true by just evaluating the formula

SAT solving: how hard is it?

- If I’m asking, it’s probably difficult. But how hard?
- Answer: **NP-Complete**
  - This is the classic **Cook-Levin theorem** (proved in the 1970s)
    - boolean SAT is the “original” NP-complete problem!
    - in NP because you can verify that an assignment makes the formula true by just evaluating the formula
    - NP-hard by reduction to polynomial-time acceptance by a nondeterministic Turing machine

SAT solving: how hard is it?

- If I'm asking, it's probably difficult. But how hard?
- Answer: **NP-Complete**
  - This is the classic **Cook-Levin theorem** (proved in the 1970s)
    - boolean SAT is the "original" NP-complete problem!
    - in NP because you can verify that an assignment makes the formula true by just evaluating the formula
    - NP-hard by reduction to polynomial-time acceptance by a nondeterministic Turing machine
- Naïve solution: try all possible assignments

SAT solving: how hard is it?

- If I’m asking, it’s probably difficult. But how hard?
- Answer: **NP-Complete**
  - This is the classic **Cook-Levin theorem** (proved in the 1970s)
    - boolean SAT is the “original” NP-complete problem!
    - in NP because you can verify that an assignment makes the formula true by just evaluating the formula
    - NP-hard by reduction to polynomial-time acceptance by a nondeterministic Turing machine
- Naïve solution: try all possible assignments
  - Takes $O(2^n)$ time for a formula with *n* variables (**slow**!)

SAT solving in practice

- I’ve mentioned before (during our symbolic execution lecture) that modern SMT solvers are **fast**

SAT solving in practice

- I’ve mentioned before (during our symbolic execution lecture) that modern SMT solvers are **fast**
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)

SAT solving in practice

- I've mentioned before (during our symbolic execution lecture) that modern SMT solvers are **fast**
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)
- So how do they manage to be so fast when the underlying problem is so hard?

SAT solving in practice

- I've mentioned before (during our symbolic execution lecture) that modern SMT solvers are **fast**
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)
- So how do they manage to be so fast when the underlying problem is so hard?
  - We'll discuss two core algorithms:
    - the **DPLL** algorithm for efficiently solving SAT
    - the **Nelson-Oppen** algorithm for efficiently solving SMT

DPLL: overview

- ***DPLL*** is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961

DPLL: overview

- ***DPLL*** is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm

DPLL: overview

- ***DPLL*** is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in ***conjunctive normal form***

DPLL: overview

- ***DPLL*** is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in ***conjunctive normal form***
- Two key innovations/heuristics:

DPLL: overview

- *DPLL* is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in ***conjunctive normal form***
- Two key innovations/heuristics:
    - ***unit propagation***

DPLL: overview

- ***DPLL*** is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in ***conjunctive normal form***
- Two key innovations/heuristics:
  - ***unit propagation***
  - ***pure literal elimination***

DPLL: overview

- *DPLL* is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in ***conjunctive normal form***
- Two key innovations/heuristics:
  - ***unit propagation***
  - ***pure literal elimination***
- If those don't apply, default to the naïve algorithm

DPLL: overview

- *DPLL* is a SAT-solving algorithm developed by Davis, Putnam, Logemann, and Loveland (hence the name) in 1961
- Algorithm is still exponential in the worst case, but on many problems is **much faster** than the naïve algorithm
- Input must be in *conjunctive normal form*
- Two key innovations/heuristics:
    - ***unit propagation***
    - *pure literal elimination*
- If those don't apply, default to the naïve algorithm

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):
    - it is a set of ***clauses*** that are separated by ***conjunctions*** ($\wedge$)

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):
    - it is a set of ***clauses*** that are separated by ***conjunctions*** ($\wedge$)
    - each clause contains zero or more ***disjunctions*** ($\vee$) of literals (which may or may not be negated)

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):
  - it is a set of ***clauses*** that are separated by ***conjunctions*** (∧)
  - each clause contains zero or more ***disjunctions*** (∨) of literals (which may or may not be negated)

Example CNF formulas:

- $(a \lor b) \land (\neg c)$
- $(a \lor \neg b) \land (\neg a \lor c) \land (b \lor c)$

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):
  - it is a set of ***clauses*** that are separated by ***conjunctions*** (∧)
  - each clause contains zero or more ***disjunctions*** (∨) of literals (which may or may not be negated)
- if the input formula is not in CNF, we can **transform it into CNF automatically** via DeMorgan's laws, the double negative law, and the distributives laws over boolean operators

DPLL: input

- the DPLL algorithm assumes that the input formula is in ***conjunctive normal form*** (***CNF***):
  - it is a set of ***clauses*** that are separated by ***conjunctions*** (∧)
  - each clause contains zero or more ***disjunctions*** (∨) of literals (which may or may not be negated)
- if the input formula is not in CNF, we can **transform it into CNF automatically** via DeMorgan's laws, the double negative law, and the distributives laws over boolean operators
  - I'm not going to cover this, because you should have had a discrete math class before. If you can't confidently do this now, you should practice before the exam.

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
    - ***literal*** here refers to a variable or its negation

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
    - *literal* here refers to a variable or its negation
- intuition: since the formula is in CNF, for the formula to be satisfiable then **each clause must be true**

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
    - *literal* here refers to a variable or its negation
- intuition: since the formula is in CNF, for the formula to be satisfiable then **each clause must be true**
    - for a one-literal clause to be true, that literal must be true!

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
  - *literal* here refers to a variable or its negation
- intuition: since the formula is in CNF, for the formula to be satisfia
  - for e true!

Consider this CNF formula:

$$(a \lor b) \land (\neg c) \land (\neg a \lor c) \land (b \lor c)$$

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
  - *literal* here refers to a variable or its negation
- intuition: since the formula is in CNF, for the formula to be satisfia
  - for e true!

Consider this CNF formula:

$(a \lor b) \land (\neg c) \land (\neg a \lor c) \land (b \lor c)$

- $\neg c$ appears alone, so c must be false

DPLL: unit propagation

- the first DPLL heuristic is ***unit propagation***: if a literal is the only disjunct in a particular clause, it must be true
  - *literal* here refers to a variable or its negation
- intuition: since the formula is in CNF, for the formula to be satisfia
  - for e true!

Consider this CNF formula:

$$(a \lor b) \land (\neg c) \land (\neg a \cancel{\lor c}) \land (b \cancel{\lor c})$$

- $\neg c$ appears alone, so c must be false

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true
  - if a variable is **always negated**, set it to false

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true
  - if a variable is **always negated**, set it to false
- intuition: a variable that only appears positively can **only help** us satisfy the formula by being true, not by being false

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true
  - if a variable is **always negated**, set it to false
- intuition: a variable that only appears positively can **only help** us satisfy t

Continuing the example:

(a ∨ b) ∧ (¬c) ∧ (¬a ~~∨ c~~) ∧ (b ~~∨ c~~)

- b only appears positively, so we can set it to true

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true
  - if a variable is **always negated**, set it to false
- intuition: a variable that only appears positively can **only help** us satisfy t

Continuing the example:

$(a \lor b) \land (\neg c) \land (\neg a$ ~~$\lor c$~~$) \land (b$ ~~$\lor c$~~$)$

- *b* only appears positively, so we can set it to true

DPLL: pure literal elimination

- the second DPLL heuristic is ***pure literal elimination***:
  - if a variable is **never negated**, set it to true
  - if a variable is **always negated**, set it to false
- intuition: a variable that only appears positively can **only help** us satisfy

Continuing the example:

(~~*a* ∨~~ *b*) ∧ (¬*c*) ∧ (¬*a* ~~∨ *c*~~) ∧ (*b* ~~∨ *c*~~)

- *b* only appears positively, so we can set it to true

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm
- that is, we **guess**

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm
- that is, we **guess**
    - modern solvers use sophisticated heuristics to choose what variable to set in such a guess, but we're going to skip over that

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm
- that is, we **guess**
  - modern solvers use sophisticated heuristics to choose what variable to set in such a guess, but we're going to skip over that
  - generally you can pick whatever variable you'd like if I ask you to do DPLL (e.g., on an exam) when you are stuck

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm
- that is, we **guess**
    - modern solvers use sophisticated heuristics to choose what variable to set in such a guess, but we're going to skip over that
    - generally you can pick whatever variable you'd like if I ask you to do DPLL (e.g., on an exam) when you are stuck
- it is important to **remember** what you guessed

DPLL: fallback

- if neither DPLL heuristic applies, then we fallback to the naïve algorithm
- that is, we **guess**
  - modern solvers use sophisticated heuristics to choose what variable to set in such a guess, but we're going to skip over that
  - generally you can pick whatever variable you'd like if I ask you to do DPLL (e.g., on an exam) when you are stuck
- it is important to **remember** what you guessed
  - if you reach an **unsatisfiable** result, you need to ***backtrack*** to the point where you made the guess (and try the other option)

DPLL: algorithm

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
     return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l})
```

pseudo-code from https://en.wikipedia.org/wiki/DPLL_algorithm

DPLL: algorithm

Heuristic: try **unit propagation** first because it creates more units and pure literals.

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l})
```

pseudo-code from https://en.wikipedia.org/wiki/DPLL_algorithm

DPLL: algorithm

Pure literal elimination is tried second because it only eliminates entire clauses (it can’t create new units or pure literals).

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
     return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l})
```

pseudo-code from https://en.wikipedia.org/wiki/DPLL_algorithm

DPLL: algorithm

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause the
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
     return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l})
```

Fallback: try both assignments to a random literal. (Note the short-circuiting “or” operator.)

pseudo-code from https://en.wikipedia.org/wiki/DPLL_algorithm

DPLL: putting it all together

Try to do DPLL in pairs on the following formula:

$(a \lor b) \land (a \lor c) \land (\neg a \lor c) \land (a \lor \neg c) \land (\neg a \lor \neg c) \land (\neg d)$

From SAT to SMT

From SAT to SMT

- We'd like to solve formulas that contain more complex subcomponents than just booleans
  - e.g., involving linear arithmetic like $x > 10$

From SAT to SMT

- We'd like to solve formulas that contain more complex subcomponents than just booleans
  - e.g., involving linear arithmetic like x > 10
- For the moment, we will assume the existence of solvers for these **theories** (such as linear arithmetic)

From SAT to SMT

- We'd like to solve formulas that contain more complex subcomponents than just booleans
  - e.g., involving linear arithmetic like $x > 10$
- For the moment, we will assume the existence of solvers for these **theories** (such as linear arithmetic)
  - but note that separate satisfying assignments for two theories might not be compatible!

From SAT to SMT

- We'd like to solve formulas that contain more complex subcomponents than just booleans
  - e.g., involving linear arithmetic like $x > 10$
- For the moment, we will assume the existence of solvers for these **theories** (such as linear arithmetic)
  - but note that separate satisfying assignments for two theories might not be compatible!
- Core idea of SMT: solve theories **separately**, but use DPLL to combine them (called ***DPLL(T)***)

SMT: Nelson-Oppen

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately
- Requires some assumptions about the theories:

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately
- Requires some assumptions about the theories:
  - **quantifier-free** fragments (“conjunctive”)
  - **equality** is the only symbol in their intersection
  - both must be **stably infinite** (don’t worry about this)

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately
- Requires some assumptions about the theories:
    - **quantifier-free** fragments ("conjunctive")
    - **equality** is the only symbol in their intersection
    - both must be **stably infinite** (don't worry about this)
- **Key idea**: replace expressions from each theory with variables

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately
- Requires some assumptions about the theories:
  - **quantifier-free** fragments (“conjunctive”)
  - **equality** is the only symbol in their intersection
  - both must be **stably infinite** (don’t worry about this)
- **Key idea**: replace expressions from each theory with variables
  - variables introduced by Nelson-Oppen can be shared between theories

SMT: Nelson-Oppen

- Provides a procedure for solving fragments of various theories in the same formula separately
- Requires some assumptions about the theories:
  - **quantifier-free** fragments ("conjunctive")
  - **equality** is the only symbol in their intersection
  - both must be **stably infinite** (don't worry about this)
- **Key idea**: replace expressions from each theory with variables
  - variables introduced by Nelson-Oppen can be shared between theories
  - solve the whole formula with a modified variant of DPLL, then ask the theory solvers if the satisfying assignment makes sense

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \land \quad f(0) = a + 2 \qquad \land \quad x = y$

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \land \quad f(0) = a + 2 \qquad \land \quad x = y$

This formula has literals in two theories. Replace them with shared variables for expressions:

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \wedge \quad f(0) = a + 2 \qquad \wedge \quad x = y$

This formula has literals in two theories. Replace them with shared variables for expressions:

- ***equality of uninterpreted functions*** (***EUF***):

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \wedge \quad f(0) = a + 2 \qquad \wedge \quad x = y$

This formula has literals in two theories. Replace them with shared variables for expressions:

- ***equality of uninterpreted functions*** (***EUF***): f(e1) = a, e2 = $f(x)$, e3 = $f(y)$, f(e4) = e5, $x = y$

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \land \quad f(0) = a + 2 \qquad \land \quad x = y$

This formula has literals in two theories. Replace them with shared variables for expressions:

- ***equality of uninterpreted functions*** (***EUF***): *f*(e1) = *a*, e2 = *f*(*x*), e3 = *f*(*y*), *f*(e4) = e5, *x* = *y*
- ***arithmetic***:

SMT: Nelson-Oppen

At this point in class, I tried to solve this example on the board. I got it wrong; it is not satisfiable. See next week's slides.

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \wedge \quad f(0) = a + 2 \qquad \wedge \quad x = y$

This formula has literals in two theories. Replace them with shared variables for expressions:

- ***equality of uninterpreted functions*** (***EUF***): f(e1) = a, e2 = $f(x)$, e3 = $f(y)$, f(e4) = e5, $x = y$
- ***arithmetic***: e1 = e2 - e3, e4 = 0, e5 = a + 2, $x = y$

SMT: Nelson-Oppen

Let's use the following formula as an example:

$f(f(x) - f(y)) = a \quad \land \quad f(0) = a + 2 \quad \land \quad x = y$

Note how theories communicate using (only) **equalities**

This formula has literals in two theories. Replace them with shared variables for expressions:

- ***equality of uninterpreted functions*** (***EUF***): *f*(e1) = *a*, e2 = *f*(*x*), e3 = *f*(*y*), *f*(e4) = e5, *x* = *y*
- *arithmetic*: e1 = e2 - e3, e4 = 0, e5 = *a* + 2, *x* = *y*

SMT: DPLL(T) algorithm intuition

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for “theory” in DPLL(T))

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory
  - Replace each theory fragment with a **fresh boolean variable**

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory
  - Replace each theory fragment with a **fresh boolean variable**
  - Run normal DPLL (with one exception, which I'll mention soon)

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory
  - Replace each theory fragment with a **fresh boolean variable**
  - Run normal DPLL (with one exception, which I'll mention soon)
  - Assuming we get a satisfying assignment, ask theories if **all of the assignments can be true at the same time**

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory
  - Replace each theory fragment with a **fresh boolean variable**
  - Run normal DPLL (with one exception, which I'll mention soon)
  - Assuming we get a satisfying assignment, ask theories if **all of the assignments can be true at the same time**
  - If not, add new clauses and re-run DPLL(T)

SMT: DPLL(T) algorithm intuition

- DPLL(T) is a variant of DPLL for use with theories (T stands for "theory" in DPLL(T))
  - Use Nelson-Oppen to ***purify*** the input formula so that each fragment is in only one theory
  - Replace each theory fragment with a **fresh boolean variable**
  - Run normal DPLL (with one exception, which I'll mention soon)
  - Assuming we get a satisfying assignment, ask theories if **all of the assignments can be true at the same time**
  - If not, add new clauses and re-run DPLL(T)
  - Continue until done

SMT: DPLL(T) example

Consider this formula as an example:

$x >= 0 \land y = x + 1 \land (y > 2 \lor y < 1)$

Conveniently all clauses are in **linear arithmetic**, so we can skip purification

SMT: DPLL(T) example

Consider this formula as an example:

$$x >= 0 \wedge y = x + 1 \wedge (y > 2 \vee y < 1)$$

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

SMT: DPLL(T) example

Consider this formula as an example:

$$x >= 0 \land y = x + 1 \land (y > 2 \lor y < 1)$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$

$$p1 \land p2 \land (p3 \lor p4)$$

We now solve this with DPLL. We get a satisfying assignment (e.g., p1, p2, p4 all true). Then, we check this with our theory:

- can p1, p2, and p4 all be true at the same time?

SMT: DPLL(T) example

Consider this formula as an example:

$$x >= 0 \wedge y = x + 1 \wedge (y > 2 \vee y < 1)$$

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

We now solve this with DPLL. We get a satisfying assignment (e.g., p1, p2, p4 all true). Then, we check this with our theory:

- can p1, p2, and p4 all be true at the same time?
  - **no**! theory of linear arithmetic says p1 and p2 imply not p4

SMT: DPLL(T) example

Consider this formula as an example:

$x >= 0 \land y = x + 1 \land (y > 2 \lor y < 1)$

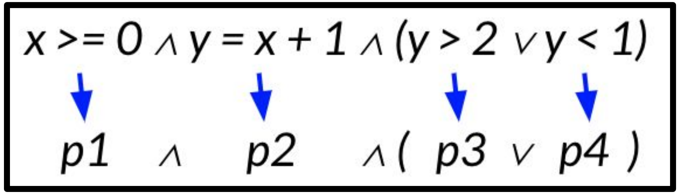
$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

$p1 \land p2 \land (p3 \lor p4)$

We now solve this with DPLL. We get a satisfying assignment (e.g., p1, p2, p4 all true). Then, we check this with our theory:

- can p1, p2, and p4 all be true at the same time?
  - **no**! theory of linear arithmetic says p1 and p2 imply not p4
  - add **new clause** $(\neg p1 \lor \neg p2 \lor \neg p4)$, try again

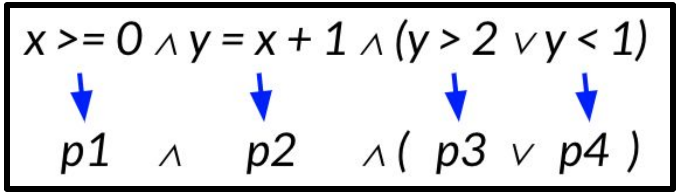
SMT: DPLL(T) example



We now have:

$p1 \land p2 \land (p3 \lor p4) \land (\neg p1 \lor \neg p2 \lor \neg p4)$

SMT: DPLL(T) example

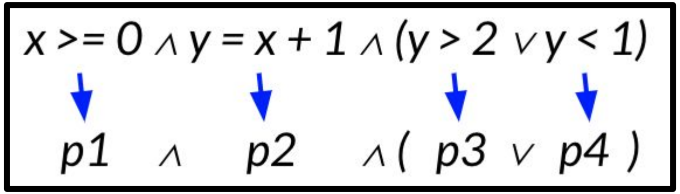


We now have:

$p1 \land p2 \land (p3 \lor p4) \land (\neg p1 \lor \neg p2 \lor \neg p4)$

Run DPLL again; one satisfying assignment is p1, p2, p3, ¬p4

SMT: DPLL(T) example



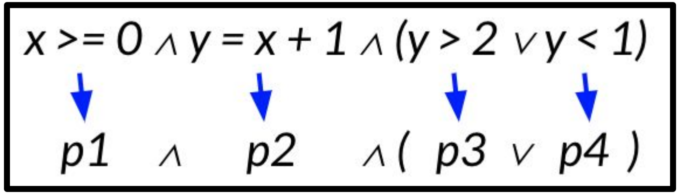
We now have:

$p1 \land p2 \land (p3 \lor p4) \land (\neg p1 \lor \neg p2 \lor \neg p4)$

Run DPLL again; one satisfying assignment is p1, p2, p3, ¬p4

- check these again against our theory. Can these all be true at the same time?

SMT: DPLL(T) example



We now have:

$p1 \land p2 \land (p3 \lor p4) \land (\neg p1 \lor \neg p2 \lor \neg p4)$

Run DPLL again; one satisfying assignment is p1, p2, p3, ¬p4

- check these again against our theory. Can these all be true at the same time?
- **yes!**
    - So, we’re done.

SMT: DPLL(T) vs DPLL

SMT: DPLL(T) vs DPLL

- DPLL(T) cannot use **pure literal elimination**

SMT: DPLL(T) vs DPLL

- DPLL(T) cannot use **pure literal elimination**
  - variables may not be **independent** when they represent a theory formula, so pure literal elimination can only be applied to plain SAT variables

SMT: DPLL(T) vs DPLL

- DPLL(T) cannot use **pure literal elimination**
  - variables may not be **independent** when they represent a theory formula, so pure literal elimination can only be applied to plain SAT variables
  - for example, consider the formula:

$$(\mathbf{x > 10} \lor x < 3) \land (\mathbf{x > 10} \lor x < 9) \land (x < 7)$$

SMT: DPLL(T) vs DPLL

- DPLL(T) cannot use **pure literal elimination**
  - variables may not be **independent** when they represent a theory formula, so pure literal elimination can only be applied to plain SAT variables
  - for example, consider the formula:

    $$(\mathbf{x > 10} \lor x < 3) \land (\mathbf{x > 10} \lor x < 9) \land (x < 7)$$
  - setting the variable for $x > 10$ to true will make $x < 7$ false!

SMT: DPLL(T) vs DPLL

- DPLL(T) cannot use **pure literal elimination**
  - variables may not be **independent** when they represent a theory formula, so pure literal elimination can only be applied to plain SAT variables
  - for example, consider the formula:

    $(\mathbf{x > 10} \lor x < 3) \land (\mathbf{x > 10} \lor x < 9) \land (x < 7)$
  - setting the variable for x > 10 to true will make x < 7 false!
- DPLL(T) must support **adding clauses to the formula**
  - to represent the knowledge gained from theories

Agenda: SMT solvers

- Motivation: reasoning about formulas
- SAT solving: DPLL
- SMT solving: Nelson-Oppen and DPLL(T)
- **SMT in practice: brief intro to Z3 and SMT-LIB**

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**

```
(echo “Running Z3”)
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**
  - **declaring variables/functions**

```
(echo "Running Z3")
(declare-const a Int)
```

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**
  - **declaring variables/functions**
  - **defining constraints**

```
(echo “Running Z3”)
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**
  - **declaring variables/functions**
  - **defining constraints**
  - **checking satisfiability**
  - **obtaining a model if so**

```
(echo "Running Z3")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**
  - **declaring variables/functions**
  - **defining constraints**
  - **checking satisfiability**
  - **obtaining a model if so**

```
(echo "Running Z3")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

What question does this code answer?

SMT in practice: Z3 and SMT-LIB

- Z3 is an SMT solver from Microsoft Research
- Uses a standard input language (SMT-LIB) that is shared with other modern SMT solvers
- SMT-LIB permits:
  - **printing to the screen**
  - **declaring variables/functions**
  - **defining constraints**
  - **checking satisfiability**
  - **obtaining a model if so**

```
(echo "Running Z3")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

What question does this code answer?
"Does an integer greater than 0 exist?"

SMT in practice: a more complex example

Consider this code:

```
int getNumber(int a, int b, int c) {
  if (c == 0) return 0;
  if (c == 4) return 0;
  if (a + b < c) return 1;
  if (a + b > c) return 2;
  if (a * b == c) return 3;
  return 4;
}
```

SMT in practice: a more complex example

Consider this code:

```
int getNumber(int a, int b, int c) {
  if (c == 0) return 0;
  if (c == 4) return 0;
  if (a + b < c) return 1;
  if (a + b > c) return 2;
  if (a * b == c) return 3;
  return 4;
}
```

Suppose we want to know if the **pink** statement is ever executed. What constraints should we pass to the SMT solver to check?

SMT in practice: a more compl

Consider this code:

```
int getNumber(int a, int b, int c) {
  if (c == 0) return 0;
  if (c == 4) return 0;
  if (a + b < c) return 1;
  if (a + b > c) return 2;
  if (a * b == c) return 3;
  return 4;
}
```

Su

the

executed. What constraints should we pass to the SMT solver to check?

All of the following must be true:

- !(c == 0)
- !(c == 4)
- !(a + b < c)
- !(a + b > c)
- a * b == c

SMT in practice: a more compl

Consider this code:

```
int getNumber(int a, int b, int c) {
  if (c == 0) return 0;
  if (c == 4) return 0;
  if (a + b < c) return 1;
  if (a + b > c) return 2;
  if (a * b == c) return 3;
  return 4;
}
```

Su
the
executed. What constraints should we pass to the SMT solver to check?

All of the following must be true:

- !(c == 0)
- !(c == 4)
- !(a + b < c)
- !(a + b > c)
- a * b == c

Let's turn this into code for [the solver](#)!

SMT in practice: a more complex example

- What went wrong?

SMT in practice: a more complex example

- What went wrong?
  - the solver **didn’t terminate quickly** on that example
    - search space is infinite!

SMT in practice: a more complex example

- What went wrong?
    - the solver **didn’t terminate quickly** on that example
        - search space is infinite!
- Z3 also supports reasoning about **bit vectors** of fixed size

SMT in practice: a more complex example

- What went wrong?
  - the solver **didn't terminate quickly** on that example
    - search space is infinite!
- Z3 also supports reasoning about **bit vectors** of fixed size
  - let's model Java ints (32 bits) and ask the same question...

SMT in practice: a more complex example

- What went wrong?
  - the solver **didn’t terminate quickly** on that example
    - search space is infinite!
- Z3 also supports reasoning about **bit vectors** of fixed size
  - let’s model Java ints (32 bits) and ask the same question…
    - it terminates quickly!
    - finite search space

Another example: program equivalence

Consider these two programs:

```
int add1(int a, int b) {
  return a + b;
}

int add2(int a, int b) {
  return a * b;
}
```

Another example: program equivalence

Consider these two programs:

```
int add1(int a, int b) {
  return a + b;
}

int add2(int a, int b) {
  return a * b;
}
```

are these two methods *semantically equivalent*?

("semantically equivalent" methods have the same *meaning*)

Another example: program equivalence

Consider these two programs:

```
int add1(int a, int b) {
  return a + b;
}

int add2(int a, int b) {
  return a * b;
}
```

are these two methods *semantically equivalent*?

("semantically equivalent" methods have the same *meaning*)

answer from Z3: yes, for a = 0 and b = 0

Another example: program equivalence

Consider these two programs:

```
int add1(int a, int b) {
  return a + b;
}

int add2(int a, int b) {
  return a * b;
}
```

are these two methods *semantically equivalent*?

("semantically equivalent" methods have the same *meaning*)

answer from Z3: yes, for a = 0 and b = 0

- does this match our intuition?

Another example: program equivalence

Consider these two programs:

```
int add1(int a, int b) {
  return a + b;
}

int add2(int a, int b) {
  return a * b;
}
```

are these two methods ***semantically equivalent*****?**

("semantically equivalent" methods have the same *meaning*)

answer from Z3: **yes**, for a = 0 and b = 0

- does this match our intuition?
- what have we **actually proven**?

Proving universal claims

- When proving **universal** claims, we need to prove that there are not any counter-examples
  - ***universal*** claims are those that start with “for all…”

Proving universal claims

- When proving **universal** claims, we need to prove that there are not any counter-examples
    - ***universal*** claims are those that start with “for all…”
        - **program equivalence** is universal, because we can phrase it as “for all inputs, these programs have the same output”

Proving universal claims

- When proving **universal** claims, we need to prove that there are not any counter-examples
    - ***universal*** claims are those that start with "for all…"
        - **program equivalence** is universal, because we can phrase it as "for all inputs, these programs have the same output"
    - "proving no counter-examples" via SMT solver means that we're looking for **unsat** as an answer

Proving universal claims

- When proving **universal** claims, we need to prove that there are not any counter-examples
  - ***universal*** claims are those
    - **program equivalence** … it as “for all inputs, these
  - “proving no counter-examples” via SMT solver means that we’re looking for **unsat** as an answer
    - need to phrase the question to the solver as “does there exist an input such that these programs differ”
      - if it says “no” (=unsat), then the programs are the same!

Let’s try with Z3 again, this time changing our question to ask if there are counter-examples.

Summary

- Solver-aided reasoning is used for testing and verification.

Summary

- Solver-aided reasoning is used for testing and verification.
- SMT solvers:
  - Provide one solution, if one exists.
  - Are commonly used to find counter-examples (or prove unsat).
  - Support many theories that can model program semantics.
  - Usually support a standard language (SMT-lib).

Summary

- Solver-aided reasoning is used for testing and verification.
- SMT solvers:
  - Provide one solution, if one exists.
  - Are commonly used to find counter-examples (or prove unsat).
  - Support many theories that can model program semantics.
  - Usually support a standard language (SMT-lib).
- The challenge is to model a problem as a constraint system.

Summary

- Solver-aided reasoning is used for testing and verification.
- SMT solvers:
  - Provide one solution, if one exists.
  - Are commonly used to find counter-examples (or prove unsat).
  - Support many theories that can model program semantics.
  - Usually support a standard language (SMT-lib).
- The challenge is to model a problem as a constraint system.
- Many higher-level DSLs and language bindings exist.
  - but in HW10 you'll mostly use SMT-LIB directly

Course announcements

- Next week's topic will be **DevOps**
    - I have already posted the required readings
- I will soon send out a survey about when you'd like to do a **final exam review**
    - reminder: the final exam is on May 9th at 6pm (here)