SMT Solving

Mădălina Erașcu

West University of Timișoara and Institute e-Austria Timișoara
bvd. V. Parvan 4, Timișsoara, Romania

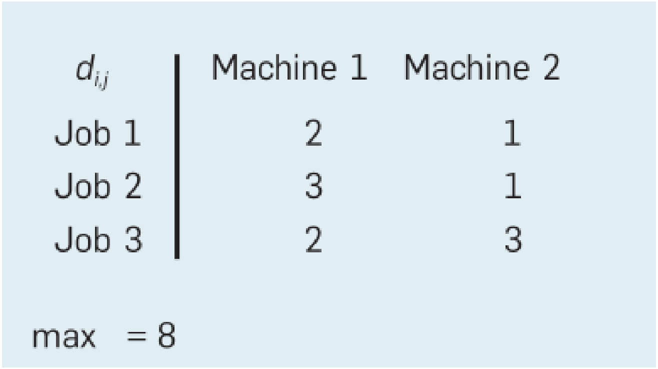
madalina.erascu@e-uvt.ro

Based on the slides of students: Lorena Krocos, Valentina Miron, Anca Vulpe, Mocan-Drăgoescu Andreea Iacob Theodor-Felix, book D. Kroening, O. Strichman. Decision Procedures An Algorithmic Point of View, article by L. De Moura, N. Bjoerner. Satisfiability Modulo Theories: Introduction and Applications.

Outline

Outline

Job-shop-scheduling problem



$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

max = 8

Job-shop-scheduling problem (cont'd)

- n jobs each composed of m tasks
- The problem involves essentially two types of constraints:
  - *Precedence.* Between two tasks in the same job
  - *Resource.* Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Job-shop-scheduling problem (cont'd)

- n jobs each composed of m tasks
- The problem involves essentially two types of constraints:
  - *Precedence*. Between two tasks in the same job
  - *Resource*. Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Job-shop-scheduling problem (cont'd)

- n jobs each composed of m tasks
- The problem involves essentially two types of constraints:
  - *Precedence*. Between two tasks in the same job
  - *Resource*. Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Job-shop-scheduling problem (cont'd)

- n jobs each composed of m tasks
- The problem involves essentially two types of constraints:
  - *Precedence*. Between two tasks in the same job
  - *Resource*. Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Outline

What is SAT? (excursion)

- SAT is the Boolean Satisfiability Problem, it is NP-COMPLETE
- Example: $(a \lor b) \land (\neg a \lor \neg b)$ is SAT for:
- $a = \mathbb{T}$ and $b = \mathbb{F}$

What if we have more than 100000 atoms? Can we solve it as easy as the previous example?

What is SAT? (excursion)

- SAT is the Boolean Satisfiability Problem, it is NP-COMPLETE
- Example: $(a \vee b) \wedge (\neg a \vee \neg b)$ is SAT for:
- $a = \mathbb{T}$ and $b = \mathbb{F}$

What if we have more than 100000 atoms? Can we solve it as easy as the previous example?

What is SAT? (excursion)

- SAT is the Boolean Satisfiability Problem, it is NP-COMPLETE
- Example: $(a \vee b) \wedge (\neg a \vee \neg b)$ is SAT for:
- $a = \mathbb{T}$ and $b = \mathbb{F}$

What if we have more than 100000 atoms? Can we solve it as easy as the previous example?

What is SAT? (excursion)

- SAT is the Boolean Satisfiability Problem, it is NP-COMPLETE
- Example: $(a \lor b) \land (\neg a \lor \neg b)$ is SAT for:
- $a = \mathbb{T}$ and $b = \mathbb{F}$

What if we have more than 100000 atoms? Can we solve it as easy as the previous example?

Solution: DPLL-Davis-Putnam-Logemann-Loveland algorithm

- Model based on three operations: *decide*, *propagate*, *backtrack*
- DPLL assumes the input as CNF (conjunction normal form)
- Example: $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$
- Answer: UNSAT

Solution: DPLL-Davis-Putnam-Logemann-Loveland algorithm

- Model based on three operations: *decide*, *propagate*, *backtrack*
- DPLL assumes the input as CNF (conjunction normal form)
- Example: $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$
- Answer: UNSAT

Solution: DPLL-Davis-Putnam-Logemann-Loveland algorithm

- Model based on three operations: *decide*, *propagate*, *backtrack*
- DPLL assumes the input as CNF (conjunction normal form)
- Example: $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$
- Answer: UNSAT

Solution: DPLL-Davis-Putnam-Logemann-Loveland algorithm

- Model based on three operations: *decide*, *propagate*, *backtrack*
- DPLL assumes the input as CNF (conjunction normal form)
- Example: $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$
- Answer: UNSAT

Outline

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \le 2 \wedge \underset{0 \le i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
    - arithmetic
    - bit-vectors
    - arrays
    - uninterpreted functions
    - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

What is SMT?

- Satisfiability Modulo Theories (SMT) problem is a decision problem for first order logical formulas with respect to combinations of background theories such as:
  - arithmetic
  - bit-vectors
  - arrays
  - uninterpreted functions
  - ...

Example:

- For $x + 1 \leq 2 \wedge \underset{0 \leq i < n}{\forall} a[i] \leqslant a[i+1]$
- the background theory $\mathbb{Z} \cup T_{A \cup \mathbb{Z}}$

Software and hardware manipulate structures like numbers, arrays, lists, etc.

First-order theories formalize/encode these structures to enable reasoning about them.

Why study fist-order theories?

- first-order theories are useful in verification and related tasks, e.g. theory of equality, of integers, of rationals and reals, of recursive data structures, of arrays.
- validity in FOL is undecidable while validity in particular theories or fragments of theories is sometimes decidable. Having efficient algorithms for proving decidability is important for verification.

Outline

Decidability

Definition
A formal language L is decidable if there exists a procedure that, given a word w, (1) eventually halts (terminates) and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$.

A procedure for a decidable language is called an algorithm.

Example. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

Definition
A formal language is undecidable if it is not decidable.

Definition
A formal language L is semi-decidable if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$.

The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$.

Decidability

Definition
A formal language L is decidable if there exists a procedure that, given a word w, (1) eventually halts (terminates) and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$.

A procedure for a decidable language is called an algorithm.

Example. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

Definition
A formal language is undecidable if it is not decidable.

Definition
A formal language L is semi-decidable if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$.

The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$.

Decidability

Definition
A formal language L is decidable if there exists a procedure that, given a word w, (1) eventually halts (terminates) and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$.

A procedure for a decidable language is called an algorithm.

Example. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

Definition
A formal language is undecidable if it is not decidable.

Definition
A formal language L is semi-decidable if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$.

The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$.

Decidability

Definition
A formal language L is decidable if there exists a procedure that, given a word w, (1) eventually halts (terminates) and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$.

A procedure for a decidable language is called an algorithm.

Example. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

Definition
A formal language is undecidable if it is not decidable.

Definition
A formal language L is semi-decidable if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$.

The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$.

Decidability

Definition
A formal language L is decidable if there exists a procedure that, given a word w, (1) eventually halts (terminates) and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$.

A procedure for a decidable language is called an algorithm.

Example. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

Definition
A formal language is undecidable if it is not decidable.

Definition
A formal language L is semi-decidable if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$.

The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$.

Complexity

If a language is decidable, then one considers the complexity of the decision problem. The main complexity classes are presented below.

Definition

A language L is polynomial-time decidable (or in PTIME, or in P), if there exists a procedure that, given w, answers *yes* when $w \in L$, answers *no* when $w \notin L$, and halts with the answer in a number of steps that is at most proportionate to some polynomial of the size of w.

Example. Determining if the word w is a well-formed FOL formula is polynomial-time decidable and can be implemented using standard parsing methods.

Definition

A language L is nondeterministic-polynomial-time decidable (or in NPTIME, or in NP), if there exists a nondeterministic procedure that, given w

- guesses a witness W to the fact that $w \in L$ that is at most proportionate in size to some polynomial of the size of w;
- checks in time at most proportionate to some polynomial of the size of w that W really is a witness to $w \in L$;
- and answers *yes* if the check succeeds and no otherwise.

Example. For example, L_{PL} is in NP, as exhibited by the following nondeterministic procedure for deciding if a PL formula is satisfiable:

1. parse the input w as formula F (return *no* if w is not a well-formed PL formula);
2. guess an interpretation I, which is linear in the size of w;
3. check that $I \models F$.

I is the witness to the satisfiability of F.

Complexity

If a language is decidable, then one considers the complexity of the decision problem. The main complexity classes are presented below.

Definition

A language L is polynomial-time decidable (or in PTIME, or in P), if there exists a procedure that, given w, answers *yes* when $w \in L$, answers *no* when $w \notin L$, and halts with the answer in a number of steps that is at most proportionate to some polynomial of the size of w.

Example. Determining if the word w is a well-formed FOL formula is polynomial-time decidable and can be implemented using standard parsing methods.

Definition

A language L is nondeterministic-polynomial-time decidable (or in NPTIME, or in NP), if there exists a nondeterministic procedure that, given w

- guesses a witness W to the fact that $w \in L$ that is at most proportionate in size to some polynomial of the size of w;
- checks in time at most proportionate to some polynomial of the size of w that W really is a witness to $w \in L$;
- and answers *yes* if the check succeeds and no otherwise.

Example. For example, L_{PL} is in NP, as exhibited by the following nondeterministic procedure for deciding if a PL formula is satisfiable:

1. parse the input w as formula F (return *no* if w is not a well-formed PL formula);
2. guess an interpretation I, which is linear in the size of w;
3. check that $I \models F$.

I is the witness to the satisfiability of F.

Complexity

If a language is decidable, then one considers the complexity of the decision problem. The main complexity classes are presented below.

Definition
A language L is polynomial-time decidable (or in PTIME, or in P), if there exists a procedure that, given w, answers *yes* when $w \in L$, answers *no* when $w \notin L$, and halts with the answer in a number of steps that is at most proportionate to some polynomial of the size of w.

Example. Determining if the word w is a well-formed FOL formula is polynomial-time decidable and can be implemented using standard parsing methods.

Definition
A language L is nondeterministic-polynomial-time decidable (or in NPTIME, or in NP), if there exists a nondeterministic procedure that, given w

- guesses a witness W to the fact that $w \in L$ that is at most proportionate in size to some polynomial of the size of w;
- checks in time at most proportionate to some polynomial of the size of w that W really is a witness to $w \in L$;
- and answers *yes* if the check succeeds and no otherwise.

Example. For example, L_{PL} is in NP, as exhibited by the following nondeterministic procedure for deciding if a PL formula is satisfiable:

1. parse the input w as formula F (return *no* if w is not a well-formed PL formula);
2. guess an interpretation I, which is linear in the size of w;
3. check that $I \models F$.

I is the witness to the satisfiability of F.

Complexity

If a language is decidable, then one considers the complexity of the decision problem. The main complexity classes are presented below.

Definition

A language L is polynomial-time decidable (or in PTIME, or in P), if there exists a procedure that, given w, answers *yes* when $w \in L$, answers *no* when $w \notin L$, and halts with the answer in a number of steps that is at most proportionate to some polynomial of the size of w.

Example. Determining if the word w is a well-formed FOL formula is polynomial-time decidable and can be implemented using standard parsing methods.

Definition

A language L is nondeterministic-polynomial-time decidable (or in NPTIME, or in NP), if there exists a nondeterministic procedure that, given w

- guesses a witness W to the fact that $w \in L$ that is at most proportionate in size to some polynomial of the size of w;
- checks in time at most proportionate to some polynomial of the size of w that W really is a witness to $w \in L$;
- and answers *yes* if the check succeeds and no otherwise.

Example. For example, L_{PL} is in NP, as exhibited by the following nondeterministic procedure for deciding if a PL formula is satisfiable:

1. parse the input w as formula F (return *no* if w is not a well-formed PL formula);
2. guess an interpretation I, which is linear in the size of w;
3. check that $I \models F$.

I is the witness to the satisfiability of F.

Complexity (cont'd)

Definition

A language L is in CO-NP if its complement language $\bar{L}$ is in NP.

Example. Unsatisfiability of PL formulae is in CO-NP because satisfiability is in NP. It is not known if unsatisfiability of PL formulae is in NP. While a satisfiable PL formula has a polynomial size witness of its satisfiability (a satisfying interpretation I), there is no known polynomial size witness of unsatisfiability.

Definition

A language L is NP-HARD if every instance $v \in L'$ of every other NP decidable language L' can be reduced to deciding an instance $w^v_{L'} \in L$. Moreover, the size of $w^v_{L'}$ must be at most proportionate to some polynomial of the size of v. That is, L is NP-HARD if every query $v \in L'$of every NP language L'can be encoded into a query $w \in L$, where w is not much larger than v. L is NP-COMPLETE if it is in NP and is NP-HARD.

Example. L_{PL} is NP-COMPLETE. Indeed, L_{PL}, also called SAT, was the first language shown to be NP-COMPLETE. We proved that L_{PL} is in NP above by describing a nondeterministic polynomial time procedure. The Cook-Levin theorem shows that all NP-LANGUAGES L can be reduced to L_{PL}, so that L_{PL} is NP-HARD. They exhibited a polynomial time algorithm that, given L and input w, constructs an encoding into PL of a simulation of a run of a nondeterministic Turing machine for L on w. The encoding as PL formula F has length that is polynomial in the length of w. F is satisfiable iff the Turing machine decides that $w \in L$.

Complexity (cont'd)

Definition
A language L is in $\textsc{co-NP}$ if its complement language $\bar{L}$ is in NP.

Example. Unsatisfiability of PL formulae is in $\textsc{co-NP}$ because satisfiability is in NP. It is not known if unsatisfiability of PL formulae is in NP. While a satisfiable PL formula has a polynomial size witness of its satisfiability (a satisfying interpretation I), there is no known polynomial size witness of unsatisfiability.

Definition
A language L is $\mathrm{NP}\textsc{-hard}$ if every instance $v \in L'$ of every other NP decidable language L' can be reduced to deciding an instance $w^v_{L'} \in L$. Moreover, the size of $w^v_{L'}$ must be at most proportionate to some polynomial of the size of v. That is, L is $\mathrm{NP}\textsc{-hard}$ if every query $v \in L'$of every NP language L'can be encoded into a query $w \in L$, where w is not much larger than v. L is $\mathrm{NP}\textsc{-complete}$ if it is in NP and is $\mathrm{NP}\textsc{-hard}$.

Example. L_{PL} is $\mathrm{NP}\textsc{-complete}$. Indeed, L_{PL}, also called SAT, was the first language shown to be $\mathrm{NP}\textsc{-complete}$. We proved that L_{PL} is in NP above by describing a nondeterministic polynomial time procedure. The Cook-Levin theorem shows that all $\mathrm{NP}\textsc{-languages}$ L can be reduced to L_{PL}, so that L_{PL} is $\mathrm{NP}\textsc{-hard}$. They exhibited a polynomial time algorithm that, given L and input w, constructs an encoding into PL of a simulation of a run of a nondeterministic Turing machine for L on w. The encoding as PL formula F has length that is polynomial in the length of w. F is satisfiable iff the Turing machine decides that $w \in L$.

Complexity (cont'd)

Definition
A language L is in $\textsc{co-NP}$ if its complement language $\bar{L}$ is in NP.

Example. Unsatisfiability of PL formulae is in $\textsc{co-NP}$ because satisfiability is in NP. It is not known if unsatisfiability of PL formulae is in NP. While a satisfiable PL formula has a polynomial size witness of its satisfiability (a satisfying interpretation I), there is no known polynomial size witness of unsatisfiability.

Definition
A language L is $\textsc{NP-hard}$ if every instance $v \in L'$ of every other NP decidable language L' can be reduced to deciding an instance $w_{L'}^{v} \in L$. Moreover, the size of $w_{L'}^{v}$ must be at most proportionate to some polynomial of the size of v. That is, L is $\textsc{NP-hard}$ if every query $v \in L'$of every NP language L'can be encoded into a query $w \in L$, where w is not much larger than v. L is $\textsc{NP-complete}$ if it is in NP and is $\textsc{NP-hard}$.

Example. L_{PL} is $\textsc{NP-complete}$. Indeed, L_{PL}, also called SAT, was the first language shown to be $\textsc{NP-complete}$. We proved that L_{PL} is in NP above by describing a nondeterministic polynomial time procedure. The Cook-Levin theorem shows that all $\textsc{NP-languages}$ L can be reduced to L_{PL}, so that L_{PL} is $\textsc{NP-hard}$. They exhibited a polynomial time algorithm that, given L and input w, constructs an encoding into PL of a simulation of a run of a nondeterministic Turing machine for L on w. The encoding as PL formula F has length that is polynomial in the length of w. F is satisfiable iff the Turing machine decides that $w \in L$.

Complexity (cont'd)

Definition
A language L is in $\textsc{co-NP}$ if its complement language $\bar{L}$ is in NP.

Example. Unsatisfiability of PL formulae is in $\textsc{co-NP}$ because satisfiability is in NP. It is not known if unsatisfiability of PL formulae is in NP. While a satisfiable PL formula has a polynomial size witness of its satisfiability (a satisfying interpretation I), there is no known polynomial size witness of unsatisfiability.

Definition
A language L is $\mathrm{NP}\text{-}\textsc{hard}$ if every instance $v \in L'$ of every other NP decidable language L' can be reduced to deciding an instance $w_{L'}^{v} \in L$. Moreover, the size of $w_{L'}^{v}$ must be at most proportionate to some polynomial of the size of v. That is, L is $\mathrm{NP}\text{-}\textsc{hard}$ if every query $v \in L'$of every NP language L'can be encoded into a query $w \in L$, where w is not much larger than v. L is $\mathrm{NP}\text{-}\textsc{complete}$ if it is in NP and is $\mathrm{NP}\text{-}\textsc{hard}$.

Example. L_{PL} is $\mathrm{NP}\text{-}\textsc{complete}$. Indeed, L_{PL}, also called SAT, was the first language shown to be $\mathrm{NP}\text{-}\textsc{complete}$. We proved that L_{PL} is in NP above by describing a nondeterministic polynomial time procedure. The Cook-Levin theorem shows that all $\mathrm{NP}\text{-}\textsc{languages}$ L can be reduced to L_{PL}, so that L_{PL} is $\mathrm{NP}\text{-}\textsc{hard}$. They exhibited a polynomial time algorithm that, given L and input w, constructs an encoding into PL of a simulation of a run of a nondeterministic Turing machine for L on w. The encoding as PL formula F has length that is polynomial in the length of w. F is satisfiable iff the Turing machine decides that $w \in L$.

Outline

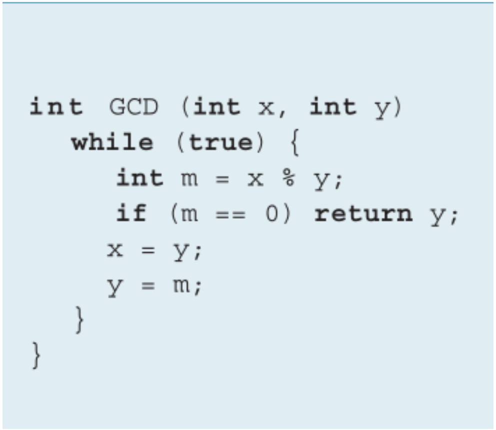
Job-shop-scheduling problem

Encoding

$$
\begin{array}{l}
(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge \\
(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge \\
(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge \\
((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge \\
((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge \\
((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge \\
((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge \\
((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge \\
((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))
\end{array}
$$

DEMO in Z3 (see lab 7)

Greatest Common Divisor



```
int GCD (int x, int y)
  while (true) {
     int m = x % y;
     if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

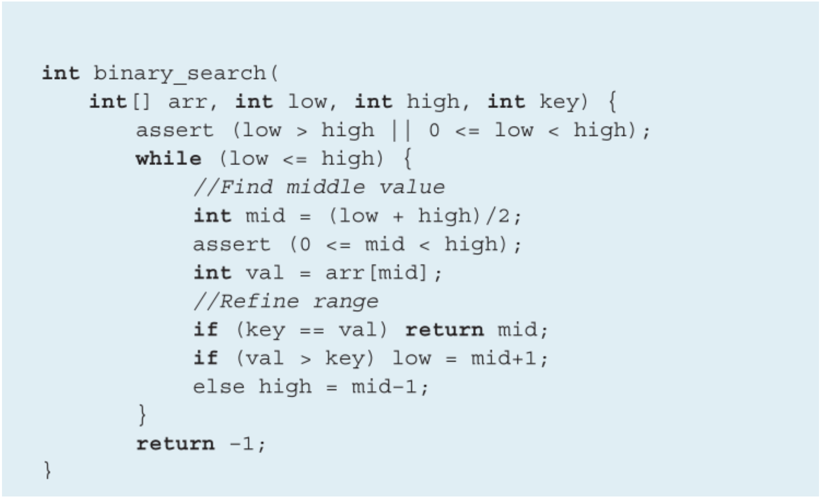
Greatest Common Divisor (con'd)

- In software engineering, in special in testing, SMT has an important place, because it represents the central role of dynamic symbolic execution.
- Using dynamic symbolic computation, inputs can be found, which can guide execution into bugs but it does not guarantee that the programs are free of all the errors being checked for.

```
int GCD (int x_0, int y_0) {
   int m_0 = x_0 % y_0;            (m_0 = x_0 % y_0)    ∧
   assert (m_0 != 0);             ¬(m_0 = 0)           ∧
   int x_1 = y_0;                  (x_1 = y_0)          ∧
   int y_1 = m_0;                  (y_1 = m_0)          ∧
   int m_1 = x_1 % y_1;            (m_1 = x_1 % y_1)    ∧
   assert (m_1 == 0);              (m_1 = 0)
}
```

DEMO in Z3 (see lab 7)

Binary Search



```
int binary_search(
    int[] arr, int low, int high, int key) {
        assert (low > high || 0 <= low < high);
        while (low <= high) {
            //Find middle value
            int mid = (low + high)/2;
            assert (0 <= mid < high);
            int val = arr[mid];
            //Refine range
            if (key == val) return mid;
            if (val > key) low = mid+1;
            else high = mid-1;
        }
        return -1;
}
```

DEMO in Z3 (see lab 7)

Outline

Array logic

- Arrays are a basic data structure that are used in most software programs.
  - modeling memories and caches in hardware design.
- Array logic permits expressions over arrays.
  - Arrays can be formalized as maps from an *index type* to and *element type*.
  - Array logic has two basic operations
    - Reading
    - Writing

Array logic (cont'd)

- Index type denoted by T_I
- Element type denoted by T_E.
- Type of the arrays denoted by T_A,

Let $a \in T_A$ denote an array.

Basic operations on arrays:

1. Reading an element with index $i \in T_I$ from a. The value of the element that has index i is denoted by $a[i]$.
2. Writing an element with index $i \in T_I$. Let $e \in T_E$ denote the value to be written. The array a where element i has been replaced by e is denoted by $a\{i \leftarrow e\}$.

Array logic (cont'd)

Definition (Array logic)

The syntax of a formula in array logic is defined by extending the syntactic rules for the index logic and the element logic. Let $atom_I$ and $atom_E$ denote an atom in the index logic and element logic, respectively, and let $term_I$ and $term_E$ denote a term in the index and element logic, respectively:

$$atom : atom_I \mid atom_E \mid \neg atom \mid atom \ \wedge \ atom \mid \forall arrayIdentifier . atom$$

$$term_A : arrayIdentifier \mid term_A\{term_I \leftarrow term_E\}$$

$$term_E : term_A[term_I]$$

Read-over-write axiom

Definition (Read-over-write axiom)

$$a \in T_A.\ \forall e \in T_E.\ \forall i, j \in T_I. a\{i \leftarrow e\}[j] = \begin{cases} e, & i = j, \\ a[j], & \mathit{otherwise} \end{cases} \tag{1}$$

Array logic in program verification

Pseudocode fragment that initializes an array of size 100 with zeros, annotated with the invariants that are maintained

```
1. $a$ : **array** 0..99 **of integer**;
2. $i$ : **integer**;
3.
4. **for** $i := 0$ **to** 99 **do**
5. $/ * \forall x \in \mathbb{N}. x < i \implies a[x] = 0 * /$
6. $a[i] := 0$;
7. $/ * \forall x \in \mathbb{N}. x \leq i \implies a[x] = 0 * /$
8. **done**;
9. $/ * \forall x \in \mathbb{N}. x \leq 99 \implies a[x] = 0 * /$
```

Verification conditions, e.g., using Hoare's axiom system.

- Element write verification:

$$(\forall x \in \mathbb{N}. x < i \implies a[x] = 0) \land a = a\{i \leftarrow 0\}$$
$$\implies (\forall x \in \mathbb{N}. x \leq i \implies a'[x] = 0)$$

- Program verification:

$$(\forall x \in \mathbb{N}. x < i \implies a[x] = 0) \land i = 99 \land a = a\{i \leftarrow 0\})$$
$$\implies (\forall x \in \mathbb{N}. x \leq 99 \implies a[x] = 0)$$

Aside: Array Bounds Checking in Programs

- Array data structures in programs are of bounded size
- **Array bounds violation**
- Exploitable to gain remote control over a system
- Array bounds checking does not require array logic

Example

Consider the following program fragment, which is meant to move the elements of an array:

1. int a[N];
2. for(int i=0; i < N; i++)
3. a[i]=a[i+1];

Despite of the fact that the program contains an array, the verification condition for the array-bounds property does not require array logic:

$$i < N \implies (i < N \land i + 1 < N)$$

Arrays as Uninterpreted Functions

- Replace the index operator with a function, where the function is the array and the function argument is the index.
- Array updates: $a\{i \leftarrow e\}$ replaced with a of array type
- Rewrite the **read-over-write** axiom
  1. $a'[i] = e$ for the value that is written,
  2. $\forall j \neq i.\ a'[j] = a[j]$ for the values that are unchanged.

Example (Arrays as Uninterpreted Functions)

First part of read-over-write axiom:

$$a\{i \leftarrow e\}[i] \geq e$$

Replace $a\{i \leftarrow e\}$ with a' and assume $a'[i] = e$:

$$a'[i] = e \implies a'[i] \geq e$$

Example (Arrays as Uninterpreted Functions (cont'd))

Second part of read-over-write axiom:

$$a[0] = 10 \implies a\{1 \leftarrow 20\}[0] = 10$$

Replace $a\{i \leftarrow e\}$ with a and add the two constraints described above:

$$(a[0] = 10 \ \wedge \ a'[1] = 20 \ \wedge \ (\forall j \neq 1.\ a'[j] = a[j]))$$

$$\implies a'[0] = 10$$

Replace a and a' with uninterpreted-function symbols F_a and $F_{a'}$:

$$(F_a(0) = 10 \ \wedge \ F_{a'}(1) = 20 \ \wedge \ (\forall j \neq 1.\ F_{a'}(j) = Fa(j)))$$

$$\implies F_{a'}(0) = 10$$

Definition (Array property)

An array logic formula is called an array property if and only if it is of the form :

$$\forall i_1, ..., i_k \in T_I.\ \varphi_I(i_1, ..., i_k) \implies \varphi_V(i_1, ..., i_k),$$

and satisfies the following conditions:

1. The predicate φ_I , called the index guard, must follow the grammar

$$iguard : iguard\ \land\ iguard \mid iguard\ \lor\ iguard \mid iterm \leq iterm \mid iterm = iterm$$

$$iterm : i_1 \mid ... \mid i_k \mid term$$

$$term : integerConstant \mid integerConstant\ *\ indexIdentifier \mid term + term$$

   The "*indexIdentifier*" used in "*term*" must not be one of $i_1, \ldots, i_k$.

2. The index variables $i_1, ..., i_k$ can only be used in array read expressions of the form $a[i_j]$.
   The predicate φ_V is called the value constraint.

Example (Array property)

The **extensionality rule** defines the equality of two arrays a_1 and a_2 as elementwise equality. Extensionality is an array property:

$$\forall i.\ a_1[i] = a_2[i].$$

Note that the index guard is simply true in this case.

Example

Recall the array logic formula:

$$(\forall x \in \mathbb{N}. x < i \implies a[x] = 0)$$

$$\wedge a' = a\{i \leftarrow 0\}$$

$$\implies (\forall x \in \mathbb{N}. x \leq i \implies a'[x] = 0)$$

The first and the third conjunct are obviously array properties. We can apply the write rule on the second conjunct and replace the array update by a fresh variable a' and add two new constraints: $a'[i] = 0$, which is obviously an array property and $\forall j \neq i. a'[j] = a[j]$, which does not comply with the array property syntax. However it can be rewritten as:

$$\forall j.(j \leq i - 1 \vee i + 1 \leq j \implies a'[j] = a[j])$$

Array-Reduction Algorithm

Input: An array property formula ϕ_A in Negation Normal Form (NNF)
Output: A formula ϕ_{UF} in the index and element theories with uninterpreted functions

1. Apply the write rule to remove all array updates from ϕ_A.
2. Replace all existential quantifications of the form $\exists i \in T_I .\ P(i)$ by $P(j)$, where j is a fresh variable.
3. Replace all universal quantifications of the form $\forall i \in T_I .\ P(i)$ by

$$\bigwedge_{i \in I(\phi)} P(i).$$

4. Replace the array read operators by uninterpreted functions and obtain ϕ_{UF}.
5. return ϕ_{UF}.

Example (Array-Reduction Algorithm)

$$\begin{gathered}(\forall x \in \mathbb{N}.\ x \leq i \implies a[x] = 0)\\ \wedge\ a' = a\{i \leftarrow 0\}\\ \implies (\forall x \in \mathbb{N}.\ x \leq i \implies a'[x] = 0)\end{gathered}$$

Transform to array property formula in NNF:

$$\begin{gathered}(\forall x \in \mathbb{N}.\ x \leq i \implies a[x] = 0)\\ \wedge\ a' = a\{i \leftarrow 0\}\\ \wedge(\exists x \in \mathbb{N}.\ x \leq i \wedge a'[x] \neq 0)\end{gathered}$$

Step 1 - apply write rule:

$$\begin{gathered}(\forall x \in \mathbb{N}.\ x \leq i \implies a[x] = 0)\\ \wedge\ a'[i] = 0 \wedge \forall j \neq i. a'[j] = a[j]\\ \wedge(\exists x \in \mathbb{N}.\ x \leq i \wedge a'[x] \neq 0)\end{gathered}$$

Example (Array-Reduction Algorithm (continued))

Step 2 - instantiate the existential quantifier with a new variable $z \in \mathbb{N}$:

$$\begin{gathered}(\forall x \in \mathbb{N}.\ x \leq i \implies a[x] = 0) \\ \wedge\ a' = a\{i \leftarrow 0\} \\ \wedge\ z \leq i \wedge a[z] \neq 0).\end{gathered}$$

Step 3 - replace the two universal quantifications:

$$\begin{gathered}(i < i \implies a[i] = 0) \wedge (z < i \implies a[z] = 0) \\ \wedge a'[i] = 0 \wedge (i \neq i \implies a'[i] = a[i]) \wedge (z \neq i \implies a'[z] = a[z]) \\ \wedge\ z \leq i \wedge a'[z] \neq 0).\end{gathered}$$

Remove the trivially satisfied conjuncts:

$$\begin{gathered}(z < i \implies a[z] = 0) \\ \wedge\ a'[i] = 0 \wedge (z \neq i \implies a'[z] = a[z]) \\ \wedge\ z \leq i \wedge a'[z] \neq 0).\end{gathered}$$

Example (Array-Reduction Algorithm (continued))

Step 4 - replace the arrays by uninterpreted functions:

$$\begin{gathered}(z < i \wedge F_a(z) = 0) \\ \wedge\ F'_a(i) = 0 \wedge (z \neq i \implies F'_a(z) = F_a(z)) \\ \wedge\ z \leq i \wedge F'_a(z) \neq 0).\end{gathered}$$

By distinguishing the three cases $z < i, z = i, and\ z > i$, it is easy to see that this formula is unsatisfiable.