SAT solving, SMT solving and Program Verification

Hans Zantema

Department of Computer Science, TU Eindhoven, and Computing Science Department, Radboud University Nijmegen The Netherlands
email: h.zantema@tue.nl

MDSEminar, February 17, 2011
A propositional formula is a formula composed from the propositional operators $\neg$, $\lor$, $\land$, $\rightarrow$, $\leftrightarrow$ and a finite set $V$ of boolean valued variables. An assignment or valuation is a map $v : V \rightarrow \{\text{false}, \text{true}\}$. An assignment $v$ is lifted to propositional formulas by defining $v(\neg \phi) = \neg v(\phi)$, $v(\phi \lor \psi) = v(\phi) \lor v(\psi)$, $v(\phi \land \psi) = v(\phi) \land v(\psi)$, and so on.

A propositional formula $\phi$ is called satisfiable (SAT) if there exists $v$ such that $v(\phi) = \text{true}$; such a $v$ is called a satisfying assignment.
A *propositional formula* is a formula composed from the propositional operators \( \neg, \lor, \land, \rightarrow, \leftrightarrow \) and a finite set \( V \) of boolean valued variables.
A *propositional formula* is a formula composed from the propositional operators $\neg$, $\lor$, $\land$, $\rightarrow$, $\leftrightarrow$ and a finite set $V$ of boolean valued variables.

An *assignment* or valuation is a map $v : V \rightarrow \{false, true\}$.
A propositional formula is a formula composed from the propositional operators $\neg, \lor, \land, \rightarrow, \leftrightarrow$ and a finite set $V$ of boolean valued variables.

An assignment or valuation is a map $v : V \rightarrow \{false, true\}$.

An assignment $v$ is lifted to propositional formulas by defining:

$$v(\neg \phi) = \neg v(\phi), \quad v(\phi \lor \psi) = v(\phi) \lor v(\psi), \quad v(\phi \land \psi) = v(\phi) \land v(\psi),$$

and so on.
A propositional formula is a formula composed from the propositional operators $\neg$, $\lor$, $\land$, $\rightarrow$, $\leftrightarrow$ and a finite set $V$ of boolean valued variables.

An assignment or valuation is a map $v : V \rightarrow \{\text{false, true}\}$.

An assignment $v$ is lifted to propositional formulas by defining:

$$v(\neg \phi) = \neg v(\phi), \quad v(\phi \lor \psi) = v(\phi) \lor v(\psi), \quad v(\phi \land \psi) = v(\phi) \land v(\psi),$$

and so on.

A propositional formula $\phi$ is called satisfiable (SAT) if there exists $v$ such that $v(\phi) = \text{true}$; such a $v$ is called a satisfying assignment.
Example:

$p \land (q \lor \neg p) \land (\neg q \lor \neg r)$ is satisfiable: choose assignment $v$:

$v(p) = true, \quad v(q) = true, \quad v(r) = false$
Example:

\[ p \land (q \lor \neg p) \land (\neg q \lor \neg r) \] is satisfiable: choose assignment \( v \):

\[ v(p) = true, \ v(q) = true, \ v(r) = false \]

Example:

\[ p \land (q \lor \neg p) \land (\neg q \lor \neg p) \] is unsatisfiable
Example:

\( p \land (q \lor \neg p) \land (\neg q \lor \neg r) \) is satisfiable: choose assignment \( v \):

\[ v(p) = true, \ v(q) = true, \ v(r) = false \]

Example:

\( p \land (q \lor \neg p) \land (\neg q \lor \neg p) \) is unsatisfiable

Simple method for checking satisfiability: \textit{truth table} = check all \( 2^n \) possibilities for \( v \) where \( n = \) number of variables
Example:
\(p \land (q \lor \neg p) \land (\neg q \lor \neg r)\) is satisfiable: choose assignment \(v\):

\[v(p) = true, \quad v(q) = true, \quad v(r) = false\]

Example:
\(p \land (q \lor \neg p) \land (\neg q \lor \neg p)\) is unsatisfiable

Simple method for checking satisfiability: *truth table* = check all \(2^n\) possibilities for \(v\) where \(n = \text{number of variables}\)

Good news: it always works
Example:
\( p \land (q \lor \neg p) \land (\neg q \lor \neg r) \) is satisfiable: choose assignment \( v: \)

\[ v(p) = true, \quad v(q) = true, \quad v(r) = false \]

Example:
\( p \land (q \lor \neg p) \land (\neg q \lor \neg p) \) is unsatisfiable

Simple method for checking satisfiability: *truth table* = check all \( 2^n \) possibilities for \( v \) where \( n = \) number of variables

Good news: it always works

Bad news: it is exponential, so only feasible for very small \( n \)
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.

For NP-complete problems the existence of a polynomial algorithm is very unlikely: this would violate the conjecture $P \neq NP$. 
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.

For NP-complete problems the existence of a polynomial algorithm is very unlikely: this would violate the conjecture $P \neq NP$.

Main message of complexity course:
- polynomial $\approx$ feasible
- NP-hard $\Rightarrow$ unfeasible
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.

For NP-complete problems the existence of a polynomial algorithm is very unlikely: this would violate the conjecture \( P \neq NP \).

Main message of complexity course:
- polynomial \( \approx \) feasible
- NP-hard \( \Rightarrow \) unfeasible

Main message of this presentation:
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.

For NP-complete problems the existence of a polynomial algorithm is very unlikely: this would violate the conjecture $P \neq NP$.

Main message of complexity course:
- polynomial $\approx$ feasible
- NP-hard $\Rightarrow$ unfeasible

Main message of this presentation:

*Never believe that NP-hard is hard*
It has been proven that SAT is NP-complete (1970), in fact this was the starting point of NP-completeness results.

For NP-complete problems the existence of a polynomial algorithm is very unlikely: this would violate the conjecture $P \neq NP$.

Main message of complexity course:
- polynomial $\approx$ feasible
- NP-hard $\Rightarrow$ unfeasible

Main message of this presentation:

*Never believe that NP-hard is hard*

Modern SAT solvers are successful in a wide range of areas, dealing with formulas over thousands of variables.
Example: eight queens problem

Eight queens have to be put on a chess board such that no two share a column, row or diagonal.
Example: eight queens problem

Eight queens have to be put on a chess board such that no two share a column, row or diagonal.

64 boolean variables $p_{ij}$ for $i, j = 1, \ldots, 8$
Example: eight queens problem

Eight queens have to be put on a chess board such that no two share a column, row or diagonal

64 boolean variables $p_{ij}$ for $i, j = 1, \ldots, 8$

At least one in every row: for every $i$:

$$\bigvee_{j=1}^{8} p_{ij}$$
Example: eight queens problem

Eight queens have to be put on a chess board such that no two share a column, row or diagonal

64 boolean variables \( p_{ij} \) for \( i, j = 1, \ldots, 8 \)

At least one in every row: for every \( i \): 
\[
\bigvee_{j=1}^{8} p_{ij}
\]

At least one in every column: for every \( j \): 
\[
\bigvee_{i=1}^{8} p_{ij}
\]
Example: eight queens problem

Eight queens have to be put on a chess board such that no two share a column, row or diagonal

64 boolean variables $p_{ij}$ for $i, j = 1, \ldots, 8$

At least one in every row: for every $i$:

$$\bigvee_{j=1}^{8} p_{ij}$$

At least one in every column: for every $j$:

$$\bigvee_{i=1}^{8} p_{ij}$$

At most one in every row: for every $i$:

$$\bigwedge_{1 \leq j < k \leq 8} \neg p_{ij} \lor \neg p_{ik}$$
Example: eight queens problem

At most one in every column: for every $j$:

$$\bigwedge_{1 \leq i < k \leq 8} \neg p_{ij} \lor \neg p_{kj}$$
Example: eight queens problem

At most one in every column: for every $j$:

$$\bigwedge_{1 \leq i < k \leq 8} \neg p_{ij} \lor \neg p_{kj}$$

At most one in every diagonal: for every $i, j, k, m$ with $(i, j) \neq (k, m)$ and either $i + k = j + m$ or $i - k = j - m$:

$$\neg p_{ij} \lor \neg p_{km}$$
Example: eight queens problem

At most one in every column: for every $j$:

$$\bigwedge_{1 \leq i < k \leq 8} \neg p_{ij} \lor \neg p_{kj}$$

At most one in every diagonal: for every $i, j, k, m$ with $(i, j) \neq (k, m)$ and either $i + k = j + m$ or $i - k = j - m$:

$$\neg p_{ij} \lor \neg p_{km}$$

Formula: conjunction of all these
Example: eight queens problem

At most one in every column: for every $j$:

$$\bigwedge_{1 \leq i < k \leq 8} \neg p_{ij} \lor \neg p_{kj}$$

At most one in every diagonal: for every $i, j, k, m$ with $(i, j) \neq (k, m)$ and either $i + k = j + m$ or $i - k = j - m$:

$$\neg p_{ij} \lor \neg p_{km}$$

Formula: conjunction of all these

Satisfying assignment = solution of eight queens problem
Example: eight queens problem

At most one in every column: for every $j$:

$$\bigwedge_{1 \leq i < k \leq 8} \neg p_{ij} \lor \neg p_{kj}$$

At most one in every diagonal: for every $i, j, k, m$ with $(i, j) \neq (k, m)$ and either $i + k = j + m$ or $i - k = j - m$:

$$\neg p_{ij} \lor \neg p_{km}$$

Formula: conjunction of all these

Satisfying assignment = solution of eight queens problem

SAT solver solves the problem only based on the specification, without providing any algorithmic heuristic like a backtracking strategy.
Modern SAT solvers are based on *resolution*, and only apply on *conjunctive normal form (CNF)*.
Modern SAT solvers are based on resolution, and only apply on conjunctive normal form (CNF).

A conjunctive normal form (CNF) is a conjunction of clauses.
Modern SAT solvers are based on *resolution*, and only apply on *conjunctive normal form* (CNF)

A *conjunctive normal form* (CNF) is a conjunction of clauses

A *clause* is a disjunction of literals
Modern SAT solvers are based on *resolution*, and only apply on *conjunctive normal form* (CNF)

A *conjunctive normal form* (CNF) is a conjunction of clauses

A *clause* is a disjunction of literals

A *literal* is a variable or the negation of a variable
Modern SAT solvers are based on *resolution*, and only apply on *conjunctive normal form (CNF)*

A *conjunctive normal form (CNF)* is a conjunction of clauses

A *clause* is a disjunction of literals

A *literal* is a variable or the negation of a variable

Hence a CNF is of the shape

\[ \bigwedge_{i} \left( \bigvee_{j} \ell_{ij} \right) \]

where \( \ell_{ij} \) are literals
Modern SAT solvers are based on resolution, and only apply on conjunctive normal form (CNF). A conjunctive normal form (CNF) is a conjunction of clauses. A clause is a disjunction of literals. A literal is a variable or the negation of a variable. Hence a CNF is of the shape

\[ \bigwedge_i \left( \bigvee_j \ell_{ij} \right) \]

where \( \ell_{ij} \) are literals. For example, our formula for the eight queens problem is a CNF.
Intuition:
Think of clauses as a set of constraints that all have to hold
*Intuition:*
Think of clauses as a set of *constraints* that all have to hold

SAT = try to find values for the variables such that in every clause at least one literal is true
**Intuition:**
Think of clauses as a set of *constraints* that all have to hold
SAT = try to find values for the variables such that in every clause at least one literal is true
Now we will describe how this search may be implemented
**Intuition:**
Think of clauses as a set of *constraints* that all have to hold

SAT = try to find values for the variables such that in every clause at least one literal is true

Now we will describe how this search may be implemented

In processing such a set of constraints, a stack $M$ of literals is maintained
Intuition:
Think of clauses as a set of constraints that all have to hold

SAT = try to find values for the variables such that in every clause at least one literal is true

Now we will describe how this search may be implemented

In processing such a set of constraints, a stack $M$ of literals is maintained

For every literal in $M$ it is important to know whether it was chosen or derived; this information is also stored in $M$
Intuition:
Think of clauses as a set of constraints that all have to hold
SAT = try to find values for the variables such that in every clause at least one literal is true
Now we will describe how this search may be implemented
In processing such a set of constraints, a stack $M$ of literals is maintained
For every literal in $M$ it is important to know whether it was chosen or derived; this information is also stored in $M$
A basic step is called unit propagate, that is, we can derive $\ell$ if there is a clause $C \lor \ell$ such that every literal in $C$ is conflicting with $M$, notation: $M \models \neg C$
In case no unit propagate is possible, we may choose a literal to be added to $M$, starting a case analysis on this literal: decide
In case no unit propagate is possible, we may choose a literal to be added to $M$, starting a case analysis on this literal: decide

A contradiction is obtained if there is a clause $C$ such that $M \models \neg C$
In case no unit propagate is possible, we may choose a literal to be added to $M$, starting a case analysis on this literal: \textit{decide}

A contradiction is obtained if there is a clause $C$ such that $M \models \neg C$

In case $M$ contains a choice $= \text{ literal introduced by}\ \textit{decide}$, this causes a \textit{backtrack} to the last choice, and the negation of this literal is added
In case no unit propagate is possible, we may choose a literal to be added to $M$, starting a case analysis on this literal: *decide*

A contradiction is obtained if there is a clause $C$ such that

$$M \models \neg C$$

In case $M$ contains a choice $\equiv$ literal introduced by *decide*, this causes a *backtrack* to the last choice, and the negation of this literal is added

If $M$ contains no choice, then a contradiction gives rise to unsatisfiability: *fail*
In case no unit propagate is possible, we may choose a literal to be added to $M$, starting a case analysis on this literal: *decide*

A contradiction is obtained if there is a clause $C$ such that $M \models \neg C$

In case $M$ contains a choice $\equiv$ literal introduced by *decide*, this causes a *backtrack* to the last choice, and the negation of this literal is added.

If $M$ contains no choice, then a contradiction gives rise to unsatisfiability: *fail*.

We say that $\ell$ is *undefined* in $M$ if neither $\ell$ nor $\neg \ell$ occurs in $M$. 
The four rules

- **Unit Propagate:** $M \leq M^\ell$ if $\ell$ is undefined in $M$ and the CNF contains a clause $C \lor \ell$ satisfying $M \models \neg C$.

- **Decide:** $M \leq M^\ell d$ if $\ell$ is undefined in $M$.

- **Backtrack:** $M^\ell d N \leq M \neg \ell$ if $M^\ell d N \models \neg C$ for a clause $C$ in the CNF and $N$ contains no decision literals.

- **Fail:** $M \leq \text{fail}$ if $M \models \neg C$ for a clause $C$ in the CNF and $M$ contains no decision literals.
The four rules

**UnitPropagate:** \[ M \implies M\ell \]
if \( \ell \) is undefined in \( M \) and the CNF contains a clause \( C \lor \ell \) satisfying \( M \models \neg C \)
UnitPropagate: \( M \implies M\ell \)
if \( \ell \) is undefined in \( M \) and the CNF contains a clause \( C \lor \ell \)
satisfying \( M \models \neg C \)

Decide: \( M \implies M\ell^d \)
if \( \ell \) is undefined in \( M \)
The four rules

*UnitPropagate:* \[ M \rightarrow M \ell \]
if \( \ell \) is undefined in \( M \) and the CNF contains a clause \( C \lor \ell \)
satisfying \( M \models \neg C \)

*Decide:* \[ M \rightarrow M \ell^d \]
if \( \ell \) is undefined in \( M \)

*Backtrack:* \[ M \ell^d N \rightarrow M \neg \ell \]
if \( M \ell^d N \models \neg C \) for a clause \( C \) in the CNF and \( N \) contains no
decision literals

Hans Zantema
SAT solving, SMT solving and Program Verification
The four rules

**UnitPropagate:** \( M \rightarrow M \ell \)
if \( \ell \) is undefined in \( M \) and the CNF contains a clause \( C \lor \ell \) satisfying \( M \models \neg C \)

**Decide:** \( M \rightarrow M \ell^d \)
if \( \ell \) is undefined in \( M \)

**Backtrack:** \( M \ell^d N \rightarrow M \neg \ell \)
if \( M \ell^d N \models \neg C \) for a clause \( C \) in the CNF and \( N \) contains no decision literals

**Fail:** \( M \rightarrow \text{fail} \)
if \( M \models \neg C \) for a clause \( C \) in the CNF and \( M \) contains no decision literals
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or a list $M$ yielding a satisfying assignment.

So this proof system is sound and complete for deciding SAT.

It is natural always to give UnitPropagate priority.

For efficiency it is essential to have good heuristics for which literal to choose in Decide.

This derivational framework is the basis for several optimizations as they are used in modern powerful SAT solvers (SATzilla, Picosat, Rsat, Minisat, March, Yices).

Hans Zantema
SAT solving, SMT solving and Program Verification
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or

Hans Zantema
SAT solving, SMT solving and Program Verification
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list $M$ yielding a satisfying assignment
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list $M$ yielding a satisfying assignment

So this proof system is sound and complete for deciding SAT
Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list $M$ yielding a satisfying assignment

So this proof system is sound and complete for deciding SAT.

It is natural always to give UnitPropagate priority.
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list $M$ yielding a satisfying assignment

So this proof system is sound and complete for deciding SAT

It is natural always to give *UnitPropagate* priority

For efficiency it is essential to have good heuristics for which literal to choose in *Decide*
Observations

Start with $M$ being empty and apply the rules as long as possible (or stopping when all clauses contain a literal from $M$) always ends in either

- fail, proving that the CNF is unsatisfiable since the derivation of Fail can be interpreted as a case analysis yielding a contradiction in all cases, or
- a list $M$ yielding a satisfying assignment

So this proof system is sound and complete for deciding SAT. It is natural always to give UnitPropagate priority. For efficiency it is essential to have good heuristics for which literal to choose in Decide. This derivational framework is the basis for several optimizations as they are used in modern powerful SAT solvers (SATzilla, Picosat, Rsat, Minisat, March, Yices).
Example

Let the CNF consist of the four clauses

1. \( p \lor q \)
2. \( p \lor \neg q \)
3. \( \neg p \lor r \)
4. \( \neg p \lor \neg r \)

We get the following derivation proving unsatisfiability:

\[ \emptyset = \Rightarrow \text{Decide} \]
\[ p \]
\[ \Rightarrow \text{UnitPropagate, clause 3} \]
\[ p \]
\[ \Rightarrow \text{Backtrack, clause 4} \]
\[ \neg p \]
\[ \Rightarrow \text{UnitPropagate, clause 1} \]
\[ \neg p \]
\[ \Rightarrow \text{Fail, clause 2} \]

fail
Example

Let the CNF consist of the four clauses

1. \( p \lor q \)
2. \( p \lor \neg q \)
3. \( \neg p \lor r \)
4. \( \neg p \lor \neg r \)
Let the CNF consist of the four clauses

1. \( p \lor q \)
2. \( p \lor \neg q \)
3. \( \neg p \lor r \)
4. \( \neg p \lor \neg r \)

We get the following derivation proving unsatisfiability:
Example

Let the CNF consist of the four clauses

1. \( p \lor q \)
2. \( p \lor \neg q \)
3. \( \neg p \lor r \)
4. \( \neg p \lor \neg r \)

We get the following derivation proving unsatisfiability:

\[
\begin{align*}
\emptyset & \implies \text{Decide} \\
p^d & \implies \text{UnitPropagate, clause 3} \\
p^d \, r & \implies \text{Backtrack, clause 4} \\
\neg p & \implies \text{UnitPropagate, clause 1} \\
\neg p \, q & \implies \text{Fail, clause 2} \\
fail & 
\end{align*}
\]
Optimizations

Backjump: if the contradiction found is independent of the last chosen decision literal, one may backtrack to an earlier decision literal, in this way pruning part of the search tree.

Learn: in using backjump, new clauses are derived, which are added to the CNF.

Forget: by adding new clauses, old clauses may be redundant and are removed.

Restart: after having changed the original CNF by learn and forget, at some time start anew with the adjusted CNF, for which the heuristics make better choices.
Optimizations

- **Backjump**: if the contradiction found is independent of the last chosen decision literal, one may backtrack to an earlier decision literal, in this way pruning part of the search tree.
Optimizations

- **Backjump**: if the contradiction found is independent of the last chosen decision literal, one may backtrack to an earlier decision literal, in this way pruning part of the search tree.

- **Learn**: in using backjump, new clauses are derived, which are added to the CNF.
Optimizations

- **Backjump**: if the contradiction found is independent of the last chosen decision literal, one may backtrack to an earlier decision literal, in this way pruning part of the search tree.

- **Learn**: in using backjump, new clauses are derived, which are added to the CNF.

- **Forget**: by adding new clauses, old clauses may be redundant and are removed.
Optimizations

- **Backjump**: if the contradiction found is independent of the last chosen decision literal, one may backtrack to an earlier decision literal, in this way pruning part of the search tree.

- **Learn**: in using backjump, new clauses are derived, which are added to the CNF.

- **Forget**: by adding new clauses, old clauses may be redundant and are removed.

- **Restart**: after having changed the original CNF by learn and forget, at some time start anew with the adjusted CNF, for which the heuristics make better choices.
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions.
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions.

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF.
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF

Straightforward: Transform to logically equivalent CNF

Claim: Every CNF equivalent to this formula has at least $2^{n-1}$ clauses

This exponential blow-up often occurs in practice, so we need something else
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions.

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF.

Straightforward: Transform to logically equivalent CNF.

This is always possible, however, it often happens that every CNF logically equivalent to a given proposition is unacceptably big.

Example:

\[(\cdots (p_1 \leftrightarrow p_2) \leftrightarrow p_3 \cdots) \leftrightarrow p_n \]

Claim: Every CNF equivalent to this formula has at least \(2^{n-1}\) clauses.

This exponential blow-up often occurs in practice, so we need something else.
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF

Straightforward: Transform to logically equivalent CNF

This is always possible, however, it often happens that every CNF logically equivalent to a given proposition is unacceptably big

*Example:*

\[(\cdots ( (p_1 \iff p_2) \iff p_3 ) \cdots \iff p_n )\]
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions.

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF.

Straightforward: Transform to logically equivalent CNF.

This is always possible, however, it often happens that every CNF logically equivalent to a given proposition is unacceptably big.

*Example:*

\[(\cdots ((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)\]

*Claim:* Every CNF equivalent to this formula has at least \(2^{n-1}\) clauses.
Until now we only considered CNFs, and we are not yet able to decide SAT for arbitrary propositions.

We want to be able to decide satisfiability of arbitrary propositions by first transforming the proposition to CNF.

Straightforward: Transform to logically equivalent CNF.

This is always possible, however, it often happens that every CNF logically equivalent to a given proposition is unacceptably big.

**Example:**

\[
(\cdots ((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)
\]

**Claim:** Every CNF equivalent to this formula has at least \(2^{n-1}\) clauses.

This exponential blow-up often occurs in practice, so we need something else.
Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

A satisfying assignment for $T(A)$ restricting to original variables is a satisfying assignment for $A$. The size of $T(A)$ is linear in the size of $A$. The standard approach to investigate satisfiability of $A$ is applying a modern CNF based SAT solver on $T(A)$.
Solution:

Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

- $A$ is satisfiable if and only if $T(A)$ is satisfiable.
Solution:

Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

- $A$ is satisfiable if and only if $T(A)$ is satisfiable
- A satisfying assignment for $T(A)$ restricting to original variables is a satisfying assignment for $A$
Solution:

Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

- $A$ is satisfiable if and only if $T(A)$ is satisfiable
- A satisfying assignment for $T(A)$ restricting to original variables is a satisfying assignment for $A$
- The size of $T(A)$ is linear in the size of $A$
Solution:

Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

- $A$ is satisfiable if and only if $T(A)$ is satisfiable
- A satisfying assignment for $T(A)$ restricting to original variables is a satisfying assignment for $A$
- The size of $T(A)$ is linear in the size of $A$

The standard approach to investigate satisfiability of $A$ is applying a modern CNF based SAT solver on $T(A)$.
Use *Tseitin transformation*, that is, introduce fresh variable names for all subformulas of $A$, and then build CNF $T(A)$ using these fresh variables such that

- $A$ is satisfiable if and only if $T(A)$ is satisfiable
- A satisfying assignment for $T(A)$ restricting to original variables is a satisfying assignment for $A$
- The size of $T(A)$ is linear in the size of $A$

The standard approach to investigate satisfiability of $A$ is applying a modern CNF based SAT solver on $T(A)$

This typically works well for formulas over thousands of variables
For instance, this is done by calling

```
yices -e -smt test.smt
```

where `test.smt` contains the formula

```example
(benchmark test.smt
:extrapreds ((A) (B) (C) (D))
:formula (and
  (iff A (and D B))
  (implies C B)
  (not (or A B (not D)))
  (or (and (not A) C) D)
))
yields

(= A false)
(= B false)
(= D true)
(= C false)
```

Hans Zantema
SAT solving, SMT solving and Program Verification
For instance, this is done by calling

\texttt{yices -e -smt test.smt}

where \texttt{test.smt} contains the formula

\textit{Example:} (\texttt{and and or} have any number of arguments)

\begin{verbatim}
(benchmark test.smt:extrapreds ((A) (B) (C) (D)):
:formula (and (iff A (and D B))
  (implies C B)
  (not (or A B (not D)))
  (or (and (not A) C) D))

yields

sat

(= A false)
(= B false)
(= D true)
(= C false)
\end{verbatim}
For instance, this is done by calling
\texttt{yices -e -smt test.smt}
where \texttt{test.smt} contains the formula

\textit{Example:} (\texttt{and and or} have any number of arguments)

\begin{verbatim}
(benchmark test.smt
:extrapreds ((A) (B) (C) (D))
:formula (and
(iff A (and D B))
(implies C B)
(not (or A B (not D)))
(or (and (not A) C) D)
))
\end{verbatim}
For instance, this is done by calling
\texttt{yices -e -smt test.smt}
where \texttt{test.smt} contains the formula

\textit{Example:} (\texttt{and and or} have any number of arguments)

\begin{verbatim}
(benchmark test.smt
 :extrapreds ((A) (B) (C) (D))
 :formula (and
 (iff A (and D B))
 (implies C B)
 (not (or A B (not D)))
 (or (and (not A) C) D)
 ))

sat
 (= A false)
 (= B false)
 (= D true)
 (= C false)
\end{verbatim}

Hans Zantema
SAT solving, SMT solving and Program Verification
A wide range of problems involving
  - binary arithmetic
  - program correctness
  - termination of rewriting
  - puzzles like Sudoku
can be encoded as SAT problems
A wide range of problems involving
- binary arithmetic
- program correctness
- termination of rewriting
- puzzles like Sudoku
can be encoded as SAT problems

Typically, a program is written in which an instance of a problem is entered, and a corresponding SAT problem is produced, after which a plain SAT solver is applied to solve the problem
Extensions

Constraint problems, optimization

Linear optimization: given \( n \) real valued variables \( x_1, \ldots, x_n \), find the highest (or lowest) value of a linear combination \( \sum_{i=1}^{n} a_i x_i \) satisfying a given number of constraints all of the shape \( \sum_{i=1}^{n} b_i x_i \leq c \).

If the variables are integer valued, this is called integer optimization.

For these problems, linear optimization and integer optimization, extremely powerful techniques are available, unrelated to SAT solving.

Our focus is on finding just a solution, rather than finding an optimal solution.

An important technique is the Simplex method, in which sets of inequalities are reduced by Gauss elimination.
Extensions

**Constraint problems, optimization**

Linear optimization: given \( n \) real valued variables \( x_1, \ldots, x_n \), find the highest (or lowest) value of a linear combination \( \sum_{i=1}^{n} a_i x_i \) satisfying a given number of constraints all of the shape \( \sum_{i=1}^{n} b_i x_i \leq c \).

If the variables are integer valued, this is called integer optimization. For these problems, linear optimization and integer optimization extremely powerful techniques are available, unrelated to SAT solving.

Our focus is on finding just a solution, rather than finding an optimal solution. An important technique is the Simplex method, in which sets of inequalities are reduced by Gauss elimination.
Constraint problems, optimization

Linear optimization: given \( n \) real valued variables \( x_1, \ldots, x_n \), find the highest (or lowest) value of a linear combination \( \sum_{i=1}^{n} a_i x_i \) satisfying a given number of constraints all of the shape \( \sum_{i=1}^{n} b_i x_i \leq c \)
Extensions

*Constraint problems, optimization*

*Linear optimization:* given $n$ real valued variables $x_1, \ldots, x_n$, find the highest (or lowest) value of a linear combination $\sum_{i=1}^{n} a_i x_i$ satisfying a given number of constraints all of the shape $\sum_{i=1}^{n} b_i x_i \leq c$

If the variables are integer valued, this is called *integer optimization*
Extensions

Constraint problems, optimization

Linear optimization: given $n$ real valued variables $x_1, \ldots, x_n$, find the highest (or lowest) value of a linear combination

$$\sum_{i=1}^{n} a_i x_i$$

satisfying a given number of constraints all of the shape

$$\sum_{i=1}^{n} b_i x_i \leq c$$

If the variables are integer valued, this is called integer optimization

For these problems linear optimization and integer optimization extremely powerful techniques are available, unrelated to SAT solving
Extensions

Constraint problems, optimization

Linear optimization: given $n$ real valued variables $x_1, \ldots, x_n$, find the highest (or lowest) value of a linear combination $\sum_{i=1}^{n} a_i x_i$ satisfying a given number of constraints all of the shape $\sum_{i=1}^{n} b_i x_i \leq c$

If the variables are integer valued, this is called integer optimization

For these problems linear optimization and integer optimization extremely powerful techniques are available, unrelated to SAT solving

Our focus is on finding just a solution, rather than finding an optimal solution
Constraint problems, optimization

Linear optimization: given $n$ real valued variables $x_1, \ldots, x_n$, find the highest (or lowest) value of a linear combination
\[ \sum_{i=1}^{n} a_i x_i \]
satisfying a given number of constraints all of the shape
\[ \sum_{i=1}^{n} b_i x_i \leq c \]

If the variables are integer valued, this is called integer optimization.

For these problems linear optimization and integer optimization extremely powerful techniques are available, unrelated to SAT solving.

Our focus is on finding just a solution, rather than finding an optimal solution.

An important technique is the Simplex method, in which sets of inequalities are reduced by Gauss elimination.
Find natural numbers $a$, $b$, $c$, $d$ such that:

- $2a > b + c$
- $2b > c + d$
- $2c > 3d$
- $3d > a + c$

**Approach 1:**

Choose $n$ boolean variables for each of the numbers $a$, $b$, $c$, $d$, $2a$, $b+c$, $2b$, $c+d$, $2c$, $2d$, $3d$, $a+c$ representing their binary encodings, and express the constraints with '+' and '>' in the standard way for expressing binary arithmetic, using several extra boolean variables for carries. Then apply a SAT solver on the resulting formula. The formula will be satisfiable; transform the satisfying assignment to the desired numbers $a$, $b$, $c$, $d$.

Depends on $n$; $n = 7$ gives a solution $a = 30$, $b = 27$, $c = 32$, $d = 21$.
Find natural numbers $a, b, c, d$ such that
$2a > b + c, 2b > c + d, 2c > 3d$ and $3d > a + c$
Find natural numbers $a, b, c, d$ such that
$2a > b + c, 2b > c + d, 2c > 3d$ and $3d > a + c$

**Approach 1:**
Choose $n$ boolean variables for each of the numbers $a, b, c, d, 2a, b + c, 2b, c + d, 2c, 2d, 3d, a + c$ representing their binary encodings, and express the constraints with ‘+’ and ‘>’ in the standard way for expressing binary arithmetic, using several extra boolean variables for carries.
Find natural numbers $a, b, c, d$ such that
$2a > b + c, 2b > c + d, 2c > 3d$ and $3d > a + c$

*Approach 1:*
Choose $n$ boolean variables for each of the numbers $a, b, c, d, 2a, b + c, 2b, c + d, 2c, 2d, 3d, a + c$ representing their binary encodings, and express the constraints with '+' and '>' in the standard way for expressing binary arithmetic, using several extra boolean variables for carries.

Then apply a SAT solver on the resulting formula.
Find natural numbers \( a, b, c, d \) such that
\[
2a > b + c, \quad 2b > c + d, \quad 2c > 3d \quad \text{and} \quad 3d > a + c
\]

**Approach 1:**
Choose \( n \) boolean variables for each of the numbers \( a, b, c, d \), \( 2a, b + c, 2b, c + d, 2c, 2d, 3d, a + c \) representing their binary encodings, and express the constraints with '+' and '>' in the standard way for expressing binary arithmetic, using several extra boolean variables for carries.

Then apply a SAT solver on the resulting formula.

The formula will be satisfiable; transform the satisfying assignment to the desired numbers \( a, b, c, d \).
Find natural numbers \( a, b, c, d \) such that
\[
2a > b + c, \quad 2b > c + d, \quad 2c > 3d \quad \text{and} \quad 3d > a + c
\]

**Approach 1:**
Choose \( n \) boolean variables for each of the numbers \( a, b, c, d, 2a, b + c, 2b, c + d, 2c, 2d, 3d, a + c \) representing their binary encodings, and express the constraints with '+' and '>' in the standard way for expressing binary arithmetic, using several extra boolean variables for carries.

Then apply a SAT solver on the resulting formula.

The formula will be satisfiable; transform the satisfying assignment to the desired numbers \( a, b, c, d \).

Depends on \( n \); \( n = 7 \) gives a solution
\[
a = 30, \quad b = 27, \quad c = 32, \quad d = 21
\]
Approach 2:
Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables.
Approach 2:
Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables

In the mechanism with the derivation rules, the central access to the formula is checking whether

\[ M \models \neg C \]

for both \( M \) being a list of literals and \( C \) being a clause
Approach 2:
Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables.

In the mechanism with the derivation rules, the central access to the formula is checking whether

\[ M \models \neg C \]

for both \( M \) being a list of literals and \( C \) being a clause.

That is, we have to check whether for every literal \( \ell \) in \( C \), the conjunction of \( \ell \) and all literals in \( M \) gives rise to a contradiction.
Approach 2: Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables.

In the mechanism with the derivation rules, the central access to the formula is checking whether

\[ M \models \neg C \]

for both \( M \) being a list of literals and \( C \) being a clause.

That is, we have to check whether for every literal \( \ell \) in \( C \), the conjunction of \( \ell \) and all literals in \( M \) gives rise to a contradiction.

For basic SAT this means that \( \neg \ell \) occurs in \( M \).
Approach 2:
Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables.

In the mechanism with the derivation rules, the central access to the formula is checking whether

\[ M \models \neg C \]

for both \( M \) being a list of literals and \( C \) being a clause.

That is, we have to check whether for every literal \( \ell \) in \( C \), the conjunction of \( \ell \) and all literals in \( M \) gives rise to a contradiction.

For basic SAT this means that \( \neg \ell \) occurs in \( M \).

The machinery is also correct if we have another mechanism to check whether a conjunction of literals is contradictory.
**Approach 2:**
Extend the SAT solver in such a way that it can deal with the inequalities directly, rather than only on boolean variables.

In the mechanism with the derivation rules, the central access to the formula is checking whether

\[ M \models \neg C \]

for both \( M \) being a list of literals and \( C \) being a clause.

That is, we have to check whether for every literal \( \ell \) in \( C \), the conjunction of \( \ell \) and all literals in \( M \) gives rise to a contradiction.

For basic SAT this means that \( \neg \ell \) occurs in \( M \).

The machinery is also correct if we have another mechanism to check whether a conjunction of literals is contradictory.

For instance, \( x > y + 1 \land y > z \land z > x + 2 \) is contradictory.
Fruitful approach for *Satisfiability Modulo Theories* (SMT) for the case where the theory consists of linear inequalities over integers or reals:
Fruitful approach for *Satisfiability Modulo Theories* (SMT) for the case where the theory consists of linear inequalities over integers or reals:

Use same approach for SAT solving with derivation rules, only for checking whether a set of literals \(\equiv\) linear inequalities is contradictory apply techniques like simplex method.
Fruitful approach for *Satisfiability Modulo Theories* (SMT) for the case where the theory consists of linear inequalities over integers or reals:

Use same approach for SAT solving with derivation rules, only for checking whether a set of literals $=\text{linear inequalities}$ is contradictory apply techniques like simplex method

Tools like *Yices* and *Barcelogic* exploit these ideas and are strong tools for SMT
Example: rectangle placement

Can you put 12 squares of sizes

\[5 \times 5, \ 6 \times 6, \ 7 \times 7, \ \ldots, \ 16 \times 16\]

in a \(39 \times 39\) square?
Example: rectangle placement

Can you put 12 squares of sizes

\[5 \times 5, \ 6 \times 6, \ 7 \times 7, \ldots, \ 16 \times 16\]

in a \(39 \times 39\) square?
How was this solution found?

By encoding the problem in an SMT formula and then call Yices
Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$:

$$x_1 + w_1 \leq x_2 \lor x_2 + w_2 \leq x_1 \lor y_1 + h_1 \leq y_2 \lor y_2 + h_2 \leq y_1$$

Apply SMT solver to conjunction of requirements: every small rectangle fits in big rectangle every two distinct small rectangles do not overlap

Contact with NXP where this approach is exploited for chip design

SAT solving, SMT solving and Program Verification
How was this solution found?

By encoding the problem in an SMT formula and then call Yices.
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width \( w \) and height \( h \) on \((x, y)\) fit in big rectangle of width \( W \) and height \( H \):

\[
x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H
\]
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$: 
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$:

$$x_1 + w_1 \leq x_2$$
How was this solution found?

By encoding the problem in an SMT formula and then call Yices.

Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$:

$$x_1 + w_1 \leq x_2 \lor x_2 + w_2 \leq x_1$$
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width \( w \) and height \( h \) on \((x, y)\) fit in big rectangle of width \( W \) and height \( H \):

\[
x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H
\]

Rectangles \((x_i, y_i, w_i, h_i)\) do not overlap for \( i = 1, 2 \):

\[
x_1 + w_1 \leq x_2 \lor x_2 + w_2 \leq x_1 \lor y_1 + h_1 \leq y_2 \lor y_2 + h_2 \leq y_1
\]
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$:

$$x_1 + w_1 \leq x_2 \lor x_2 + w_2 \leq x_1 \lor y_1 + h_1 \leq y_2 \lor y_2 + h_2 \leq y_1$$

Apply SMT solver to conjunction of requirements:

- every small rectangle fits in big rectangle
- every two distinct small rectangles do not overlap
How was this solution found?

By encoding the problem in an SMT formula and then call Yices

Rectangle of width $w$ and height $h$ on $(x, y)$ fit in big rectangle of width $W$ and height $H$:

$$x \geq 0 \land x + w \leq W \land y \geq 0 \land y + h \leq H$$

Rectangles $(x_i, y_i, w_i, h_i)$ do not overlap for $i = 1, 2$:

$$x_1 + w_1 \leq x_2 \lor x_2 + w_2 \leq x_1 \lor y_1 + h_1 \leq y_2 \lor y_2 + h_2 \leq y_1$$

Apply SMT solver to conjunction of requirements:
- every small rectangle fits in big rectangle
- every two distinct small rectangles do not overlap

Contact with NXP where this approach is exploited for chip design
Program verification

For a program doing \( m \) steps, like

\[
\text{for } j := 1 \text{ to } m \text{ do } \cdots
\]

introduce \( m + 1 \) copies \( a_0, \ldots, a_m \) for every variable \( a \), where

\( a_i \) means: the value of \( a \) after \( i \) steps

Assignment \( a := e \) in step \( i \) can be expressed as

\[
(a_i + 1 \leftrightarrow e_i) \land \bigwedge c (c_i + 1 \leftrightarrow c_i)
\]

where \( c \) runs over all variables \( \neq a \)

Hans Zantema SAT solving, SMT solving and Program Verification
For a program doing $m$ steps, like

\[
\text{for } j := 1 \text{ to } m \text{ do } \cdots
\]

introduce $m + 1$ copies $a_0, \ldots, a_m$ for every variable $a$, where $a_i$ means: the value of $a$ after $i$ steps
For a program doing $m$ steps, like

for $j := 1$ to $m$ do · · ·

introduce $m + 1$ copies $a_0, \ldots, a_m$ for every variable $a$, where $a_i$ means: the value of $a$ after $i$ steps

Assignment $a := e$ in step $i$ can be expressed as

$$(a_{i+1} \leftrightarrow e_i) \land \bigwedge_{c}(c_{i+1} \leftrightarrow c_i)$$

where $c$ runs over all variables $\neq a$
Required property to be proved = specification of the program
Required property to be proved = specification of the program
Typically given by a *Hoare triple*:

\[
\{P\} S \{Q\}
\]

Here

\(S\) is the program
\(P\) is the precondition: the property assumed to hold initially
\(Q\) is the postcondition: the property that should hold after the program has finished

For proving \(\{P\} S \{Q\}\) add the formula

\(P_0 \land \neg Q_m\)

to the formula expressing the semantics of the program, and prove that the resulting formula is unsatisfiable
Required property to be proved = specification of the program

Typically given by a Hoare triple:

\[ \{ P \} S \{ Q \} \]

Here

- \( S \) is the program
- \( P \) is the \textit{precondition}: the property assumed to hold Initially
- \( Q \) is the \textit{postcondition}: the property that should hold after the program has finished
Required property to be proved = specification of the program

Typically given by a Hoare triple:

\[ \{ P \} S \{ Q \} \]

Here

- \( S \) is the program
- \( P \) is the \textit{precondition}: the property assumed to hold initially
- \( Q \) is the \textit{postcondition}: the property that should hold after the program has finished

For proving \( \{ P \} S \{ Q \} \) add the formula

\[ P_0 \land \neg Q_m \]

to the formula expressing the semantics of the program, and prove that the resulting formula is unsatisfiable
Simple example: boolean array $a[1..m]$

CLAIM: After doing for $j := 1$ to $m - 1$ do 

$$a[j + 1] := a[j]$$

we have $a[1] = a[m]$

postcondition

Precondition $=$ true, may be ignored

$a_{ij}$ represents value $a[i]$ after $j$ iterations

Semantics of $j$th iteration: $(a[j + 1], j ↔ a[j], j - 1) \land \bigwedge_{i \in \{1, \ldots, m\}, i \neq j + 1} (a_{ij} ↔ a[i], j - 1)$

Negation of postcondition: $\neg (a_1, m - 1 ↔ a_m, m - 1)$

For a fixed $m$, prove by a SAT solver that conjunction of all of these claims is unsatisfiable
Simple example: boolean array $a[1..m]$

CLAIM: After doing
for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]$
Simple example: boolean array $a[1..m]$

CLAIM: After doing
for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]$
we have

$$a[1] = a[m]$$

postcondition
Simple example: boolean array \( a[1..m] \)

CLAIM: After doing
for \( j := 1 \) to \( m - 1 \) do \( a[j + 1] := a[j] \)

we have

\[
\begin{align*}
\underbrace{a[1] = a[m]} \\
\text{postcondition}
\end{align*}
\]

Precondition = true, may be ignored
Simple example: boolean array $a[1..m]$

CLAIM: After doing for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]

we have

$$a[1] = a[m]$$

postcondition

Precondition $= true$, may be ignored

$a_{ij}$ represents value $a[i]$ after $j$ iterations
Simple example: boolean array $a[1..m]$

**CLAIM:** After doing
for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]$

we have

$$a[1] = a[m]$$

postcondition

Precondition $= true$, may be ignored

$a_{ij}$ represents value $a[i]$ after $j$ iterations

Semantics of $j$th iteration:

$$(a_{j+1,j} \leftrightarrow a_{j,j-1}) \land \bigwedge_{i \in \{1,\ldots,m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1})$$
Simple example: boolean array \( a[1..m] \)

**CLAIM:** After doing
for \( j := 1 \) to \( m - 1 \) do \( a[j + 1] := a[j] \)
we have

\[
a[1] = a[m]
\]

postcondition

Precondition = true, may be ignored

\( a_{ij} \) represents value \( a[i] \) after \( j \) iterations

Semantics of \( j \)th iteration:

\[
(a_{j+1,j} \leftrightarrow a_{j,j-1}) \land \bigwedge_{i \in \{1,\ldots,m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1})
\]

Negation of postcondition:

\[
\neg(a_{1,m-1} \leftrightarrow a_{m,m-1})
\]
Simple example: boolean array $a[1..m]$

**CLAIM:** After doing
for $j := 1$ to $m - 1$ do $a[j + 1] := a[j]$
we have

$$a[1] = a[m]$$

postcondition

Precondition = true, may be ignored

$a_{ij}$ represents value $a[i]$ after $j$ iterations

Semantics of $j$th iteration:

$$(a_{j+1,j} \leftrightarrow a_{j,j-1}) \land \bigwedge_{i \in \{1,...,m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1})$$

Negation of postcondition: $\neg(a_{1,m-1} \leftrightarrow a_{m,m-1})$

For a fixed $m$, prove by a SAT solver that conjunction of all of these claims is unsatisfiable
Example

CLAIM:

After doing

\[
\begin{align*}
a &:= 0; \\
& \text{for } i := 1 \text{ to } m \\
& \quad a := a + k \\
\end{align*}
\]

we have

\[
a = m \cdot k
\]

For fixed \(m\) this is proved by proving unsatisfiability of the SMT formula

\[
a_0 = 0 \land m - 1 \land i = 0 \land a_i + 1 = a_i + k \land \neg (a = m \cdot k)
\]

Hans Zantema

SAT solving, SMT solving and Program Verification
**CLAIM:** After doing
CLAIM: After doing

\[ a := 0; \]
\[ \text{for } i := 1 \text{ to } m \text{ do } a := a + k \]
**CLAIM:** After doing

\[
a := 0; \\
\text{for } i := 1 \text{ to } m \text{ do } a := a + k
\]

we have \( a = m \times k \)
**CLAIM:** After doing

\[
a := 0; \\
\text{for } i := 1 \text{ to } m \text{ do } a := a + k
\]

we have \( a = m \times k \)

For fixed \( m \) this is proved by proving unsatisfiability of the SMT formula

\[
a_0 = 0 \land \bigwedge_{i=0}^{m-1} a_{i+1} = a_i + k \land \neg(a = m \times k)
\]
If statement

if \( b \) then \( S_1 \) else \( S_2 \)

In step \( i \) can be expressed as \((b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)\) where formulas \( F_1 \), \( F_2 \) express \( S_1 \), \( S_2 \) in step \( i \). In this way verification of a rich class of imperative programs can be expressed in SMT.

Restrictions: Only works when number of steps can be established statically, so no recursion or while loops.

Features: No restriction on number of initial states, all kinds of non-determinism allowed.

For instance: prove that with rules of alternating bit protocol, if three numbers are sent, and three numbers have been received, these are the same.

Hans Zantema  SAT solving, SMT solving and Program Verification
If statement

if $b$ then $S_1$ else $S_2$

in step $i$ can be expressed as

$$(b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)$$

where formulas $F_1, F_2$ express $S_1, S_2$ in step $i$
If statement

if $b$ then $S_1$ else $S_2$

in step $i$ can be expressed as

$$(b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)$$

where formulas $F_1, F_2$ express $S_1, S_2$ in step $i$

In this way verification of a rich class of imperative programs can be expressed in SMT
If statement

if \( b \) then \( S_1 \) else \( S_2 \)

in step \( i \) can be expressed as

\[(b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)\]

where formulas \( F_1, F_2 \) express \( S_1, S_2 \) in step \( i \)

In this way verification of a rich class of imperative programs can be expressed in SMT

**Restrictions:** Only works when number of steps can be established statically, so no recursion or while loops
If statement

if \( b \) then \( S_1 \) else \( S_2 \)

in step \( i \) can be expressed as

\[
(b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)
\]

where formulas \( F_1, F_2 \) express \( S_1, S_2 \) in step \( i \)

In this way verification of a rich class of imperative programs can be expressed in SMT

Restrictions: Only works when number of steps can be established statically, so no recursion or while loops

Features: No restriction on number of initial states, all kinds of non-determinism allowed
If statement

\[
\text{if } b \text{ then } S_1 \text{ else } S_2
\]

in step \( i \) can be expressed as

\[
(b_i \rightarrow F_1) \land (\neg b_i \rightarrow F_2)
\]

where formulas \( F_1, F_2 \) express \( S_1, S_2 \) in step \( i \)

In this way verification of a rich class of imperative programs can be expressed in SMT

**Restrictions:** Only works when number of steps can be established statically, so no recursion or while loops

**Features:** No restriction on number of initial states, all kinds of non-determinism allowed

For instance: prove that with rules of alternating bit protocol, if three numbers are sent, and three numbers have been received, these are the same
Five integer variables $a_1, a_2, a_3, a_4, a_5$ are given, for which the initial value of $a_i$ is $i$ for $i = 1, \ldots, 5$. For $i = 2, 3, 4$ it is possible to execute the step

\[ a_i := a_{i-1} + a_{i+1} \]

Establish the minimum number of these steps required for one of the $a_i$'s having exactly the value 300.
Five integer variables $a_1, a_2, a_3, a_4, a_5$ are given, for which the initial value of $a_i$ is $i$ for $i = 1, \ldots, 5$

For $i = 2, 3, 4$ it is possible to execute the step

$$a_i := a_{i-1} + a_{i+1}$$

Establish the minimum number of these steps required for one of the $a_i$'s having exactly the value 300

Apply the approach just sketched for various values of $m$, until the postcondition $\bigvee_{i=1}^{5} a_{im} = 300$ can be reached
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems
- There are several instances of SAT/SMT problems over thousands of variables that are easily solved by current solvers

Never believe that NP-hard is hard.

Hans Zantema
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems.
- There are several instances of SAT/SMT problems over thousands of variables that are easily solved by current solvers.
- Such instances include encodings of all kinds of puzzles, like sudoku, kakuro, killer sudoku, (whole range of Japanese puzzles).
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems.

- There are several instances of SAT/SMT problems over thousands of variables that are easily solved by current solvers.

- Such instances include encodings of all kinds of puzzles, like sudoku, kakuro, killer sudoku, (whole range of Japanese puzzles).

- Such instances include problems of placement and routing in chip design.
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems.

- There are several instances of SAT/SMT problems over thousands of variables that are easily solved by current solvers.

- Such instances include encodings of all kinds of puzzles, like sudoku, kakuro, killer sudoku, (whole range of Japanese puzzles).

- Such instances include problems of placement and routing in chip design.

- Such instances include several verification / reachability problems.
Conclusions

- SAT solving and extensions like SMT solving are NP-hard problems.
- There are several instances of SAT/SMT problems over thousands of variables that are easily solved by current solvers.
- Such instances include encodings of all kinds of puzzles, like sudoku, kakuro, killer sudoku, (whole range of Japanese puzzles).
- Such instances include problems of placement and routing in chip design.
- Such instances include several verification / reachability problems.
- Never believe that NP-hard is hard.