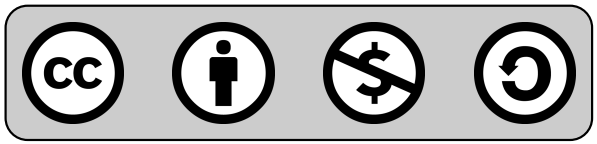
N7PD Declarative Programming

The SAT and SMT problems: theory and solvers

Christophe Garion

ISAE-SUPAERO/DISC

License CC BY-NC-SA 3.0



Outline

Outline

Outline

What is the SAT problem?

SAT is the abbreviation of the **Boolean Satisfiability Problem**: given a propositional formula, is there a propositional interpretation that satisfies it?

SAT is a important **theoretical** problem for CS: first problem to be proved to be NP-complete, phase transition...

But SAT has also lots of **practical applications**:

- scheduling
- planning
- **software verification**
- tooling
- ...

The propositional language

Definition (syntax of $\mathcal{L}_{PL}$)

Let *Var* be a set of propositional variables. The syntax of well-formed formulas of $\mathcal{L}_{PL}$ is given by the following EBNF:

$$\varphi ::= \text{'}A\text{'} \mid \text{'}\top\text{'} \mid \text{'}\bot\text{'} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi$$

where $A \in \mathit{Var}$.

We consider here the semantics of classic propositional logic (cf. MOD first year lecture).

Interpretation, satisfiability, validity

Definition (interpretation)

An **interpretation** is a total function $Var \mapsto \{T, F\}$.

The **truth value** of a formula φ can be evaluated in an intepretation $\mathcal{I}$ using only the truth values of its subformulae.

The truth value of φ in $\mathcal{I}$ is denoted by $[\![\varphi]\!]_{\mathcal{I}}$

Definition (satisfiability)

A wff φ is **satisfiable** iff **there is** an interpretation $\mathcal{I}$ s.t. $[\![\varphi]\!]_{\mathcal{I}} = T$.

Definition (validity)

A wff φ is **valid** iff **for all** interpretations $\mathcal{I}$, $[\![\varphi]\!]_{\mathcal{I}} = T$. This is denoted by $\models \varphi$.

SAT and UNSAT

Notice that if a wff φ is **not satisfiable** (noted **UNSAT**), it means that $[\![\varphi]\!]_{\mathcal{I}} = F$ for **all interpretations** $\mathcal{I}$.

Thus, the following equivalence holds:

$$\varphi \text{ is valid} \Leftrightarrow \neg\varphi \text{ is UNSAT.}$$

Outline

Is SAT a complex problem?

We might wonder first if SAT is **really** a complex problem.

First, let us look at the simpliest algorithm to check if a wff is SAT or not: **truth tables**.

Truth table complexity is clearly $O(2^n)$ where n is the number of propositional variables involved in the wff.

OK, but is there a better algorithm for SAT?

The answer is: **we do not know!**

But if there is an efficient (i.e. polynomial) algorithm for SAT, then it would solve the $P = NP$ question...

NP problems

NP is a class of problems that can decided in **polynomial time** by a non deterministic Turing machine.

Intuitively, a problem in NP is such that verifying if an alternative is a solution is polynomial.

For instance, SAT is in NP: given an interpretation, you can check that the truth value of input formula is T in polynomial time...
... but we do not know if SAT is in P.

Of course, $P \subseteq NP$, but we do not know if $NP \subseteq P$ (this is a 1,000,000$ prize ☺).

NP-complete problems

A **NP-complete** problem is a problem that:

- is in NP
- is such that every problem in NP can be reduced to the problem in polynomial time

Therefore:

- showing that a NP-complete problem is in P implies that **all problems in NP are in P**
- showing that a problem is in NPC gives a good indication on the fact that the problem should be the least likely to be in P...

SAT is NP-complete

Theorem (Cook-Levin)

SAT is NP-complete.

Stephen Cook.
"The complexity of theorem proving procedures".
In: **Proc. of the third annual ACM symposium on Theory of computing**.
1971,
Pp. 151–158.
URL: http://4mhz.de/cook.html.

Idea: given a NP problem $\mathcal{P}$, a NTM $\mathcal{M}$ that solves it, and an entry $\mathcal{I}$ for $\mathcal{P}$, build a formula that is satisfiable iff $\mathcal{M}$ accepts $\mathcal{I}$ for $\mathcal{P}$.

Outline

SAT complexity: end of the game?

Despite its NP-completness, SAT can be used to solve satisfiability problems with thousands, perhaps billions of clauses.
For instance, among the SAT competition 2023 problems you will find:

- Bitcoin mining problems
- verifying floating-points operations
- register allocation for Python functions
- ...

Tomás Balyo et al., eds.
SAT COMPETITION 2023.
2023.
URL: https://researchportal.helsinki.fi/en/publications/proceedings-of-sat-competition-2023-solver-benchmark-and-proof-ch.

SAT solving procedures

SAT solvers are built on one of the following procedures:

- DPLL which explores the decision tree built with the formula to satisfy with backtracking.
  This algorithm is rather simple and you can implement it easily, for instance in OCaml or in Rust.
- CDCL is another algorithm which use backjumping and is more efficient.
  More difficult to implement, but you can find toy implementations in Python or other languages (see for instance `https://github.com/topics/cdcl-algorithm`).

SAT solving procedures

SAT solvers are built on one of the following procedures:

- DPLL which explores the decision tree built with the formula to satisfy with backtracking.
  This algorithm is rather simple and you can implement it easily, for instance in OCaml or in Rust.
- CDCL is another algorithm which use backjumping and is more efficient.
  More difficult to implement, but you can find toy implementations in Python or other languages (see for instance `https://github.com/topics/cdcl-algorithm`).

SAT solving procedures

SAT solvers are built on one of the following procedures:

- DPLL which explores the decision tree built with the formula to satisfy with backtracking.
  This algorithm is rather simple and you can implement it easily, for instance in OCaml or in Rust.
- CDCL is another algorithm which use backjumping and is more efficient.
  More difficult to implement, but you can find toy implementations in Python or other languages (see for instance `https://github.com/topics/cdcl-algorithm`).

Consult Aalto University course CS-E3220 on Declarative Programming to have a good introduction on how to implement a SAT solver.

Modern SAT solvers

Some historical SAT solvers:

- zChaff was on of the first efficient SAT solver
- MiniSAT dedicated to get started on SAT
- SAT4J (written in Java, integrated in Eclipse for libraries dependency)
- Glucose 4.0 (written in C++, winner of several contests)
- Lingeling (written in C, winner of several contests)

Actual best SAT solvers (according to SAT competition):

- Kissat
- MapleSAT which uses machine learning

We will use the Z3 solver with its Java API as both a SAT and SMT during the lab sessions.

Outline

Optimization problems: vocabulary

An optimization problem is composed of:

- a **search space** $\mathcal{S}$: the possible alternatives
- the search space is represented by a set of **variables** $V = \{v_i \ : \ i \in \{1, \ldots, n\}\}$, each variable v_i having a domain D_i
  - an alternative is an instanciation of each v_i
  - the search space is thus $\prod_{i \in \{1,\ldots,n\}} D_i$
- a set Co of **constraints** to be satisfied
  - constraints are **assertions**
  - they may represent physical (real-world) limitations or user prerequisites
  - a **solution** is an alternative that satisfies all constraints in Co
- a set Cr of **criteria** to be satisfied as best
  - a criterion is a function from $\mathcal{S}$ to a totally ordered set ($\mathbb{R}^+$ for instance). This function has to be **minimized or maximized**
  - they represent **user preferences**

Decision problems with SAT

We consider here:

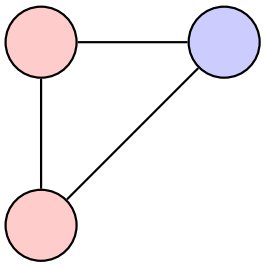
- **decision problems**, i.e. problems whose answer is `YES` or `NO`.
  Notice that there is no criterion to maximize or minimize, but you can model an optimization problem with a corresponding decision problem, see your optimization/operational research courses.
- all variables take their values in $\mathbb{B}$.
  They are called **decision variables**.

So, can we model and solve complex problems using SAT?

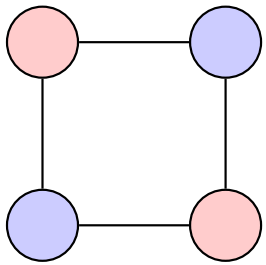
The 2-coloring problem

Exercise

Let us consider the following problem: considering a graph G with n nodes and a set E of edges, can we color the nodes with only two colors such that adjacent nodes have different colors?



✘



✔

The Sudoku problem

Let n be an integer such that $n \geq 3$. A Sudoku is a problem in which we consider:

- a $n^2 \times n^2$ grid
- each cell of the grid contains a **unique number** choosen betwen 1 and n^2
- a number appears exactly one time in a row
- a number appears exactly one time in a column
- a number appears exactly one time in a subgrid

The Sudoku problem

4	1	5	6	3	8	9	7	2
3	6	2	4	7	9	1	6	5
7	8	9	2	1	5	3	6	4
9	2	5	3	4	1	7	5	8
1	3	8	7	5	6	4	2	9
5	7	4	9	8	2	6	3	1
2	5	7	1	6	4	8	9	3
8	4	3	5	9	7	2	1	6
6	9	1	8	2	3	5	4	7

4	1	5	6	3	8	1	7	2
3	6	2	4	7	9	1	6	5
7	8	9	2	1	5	3	6	4
9	2	5	3	4	1	7	5	8
1	3	8	7	5	6	4	2	9
5	7	4	9	8	2	6	3	1
2	5	7	1	6	4	8	9	3
8	5	3	4	9	7	2	1	6
6	9	1	8	2	4	5	3	7

Can you model and solve the Sudoku problem with a SAT problem?

- what are the **decision variables**?
- what are the **constraints**?

The Sudoku problem

Decision variables: n^3 boolean variables

For $r \in \{1, \ldots, n\}$, $c \in \{1, \ldots, n\}$ and $v \in \{1, \ldots, n\}$, $p_{r,c,v}$ means "it is true that value v is in the cell (r, c)"

The Sudoku problem

- **existence constraints**: for all $r \in \{1, \ldots, n\}$ and $c \in \{1, \ldots, n\}$
  $$\bigvee_{v \in \{1,\ldots,n\}} p_{r,c,v}$$
- **uniqueness constraints**: for all $r \in \{1, \ldots, n\}$, $c \in \{1, \ldots, n\}$ and $v \in \{1, \ldots, n\}$
  $$p_{r,c,v} \rightarrow \bigwedge_{v' \neq v} \neg p_{r,c,v'}$$
- **row constraints**: for all $r \in \{1, \ldots, n\}$, $c \in \{1, \ldots, n\}$ and $v \in \{1, \ldots, n\}$
  $$p_{r,c,v} \rightarrow \bigwedge_{c' \neq c} \neg p_{r,c',v}$$
- **column constraints**: for all $r \in \{1, \ldots, n\}$, $c \in \{1, \ldots, n\}$ and $v \in \{1, \ldots, n\}$
  $$p_{r,c,v} \rightarrow \bigwedge_{r' \neq r} \neg p_{r',c,v}$$
- **subgrid constraints**: find how to express them (there are several possible models).

Some remarks on the Sudoku model

Important: there are no quantifiers in propositional logic!

Therefore, if you must add each constraint **one by one**.

For instance, if you use the Java programming language:

```
for (int r = 1; r <= n; r++) {
  for (int c = 1; c <= n; c++) {
    for (int v = 1; v <= n; v++) {
      // add formula for uniqueness of value v in (r, c)
    }
  }
}
```

Outline

Conclusion

SAT can be used to model lots of decision problems and you will see during the lab sessions that solvers are really efficient!

But:

- you have to find a **propositional encoding** of the problem
- SAT suffers from a **lack of expressiveness**, particularly the lack of quantifiers

Outline

Outline

Sudoku using SAT: really?

Using SAT to solve the Sudoku problem is **exhausting**: nobody wants to explicitely translate a problem using integers into a problem using booleans 😠

How to escape from the unexpressiveness of SAT?

This is possible with Satisfiability Modulo Theories (SMT)!

Clark Barrett et al.
"Satisfiability Modulo Theories".
In:
Handbook of Satisfiability.
Ed. by Armin Biere et al.
Vol. 185.
Frontiers in Artificial Intelligence and Applications.
IOS Press, Feb. 2009.
Chap. 26, pp. 825–885.
ISBN: 978-1-58603-929-5.

SMT: definition

Roughly speaking, a SMT problem is a **satisfaction problem** for a **first-order logic** formula.

The expressions used in the formulas are interpreted **modulo** some theories, for instance the theory of integers, the theory of uninterpreted functions etc.

For instance, consider the formula $\forall x \forall y\ x < y \rightarrow x < y + y$.

We are not interested in verifying that the previous formula is valid in **all FOL models**, but only for interpretations in which $<$ is the order on integers and $+$ is addition.

As SAT in NP-complete

- SMT is generally **NP-hard**
- SMT can be undecidable (depending on the theory you use)

Theories: vocabulary

A theory is a set of **first-order formulae** closed under derivability and generally defined by on of its subsets called **axioms**.

A theory can be

- **consistent** if there is at least an interpretation of the theory
- **complete** if for every formula φ of the theory, either $\models \varphi$ or $\models \neg\varphi$
- **decidable** if there is a decision procedure for checking validity of formulae of the theory

Sometimes you only consider a **fragment** of the theory, e.g. to gain decidability.

You may consider for instance the **quantifier-free** fragment (QF) of a theory.

A theory example: equality and uninterpreted functions

Let us consider the theory of **equality** and **uninterpreted functions**:

- $=$ is the only interpreted symbol (defined by axioms)
- constant, functions and other predicate symbols are uninterpreted

Axioms:

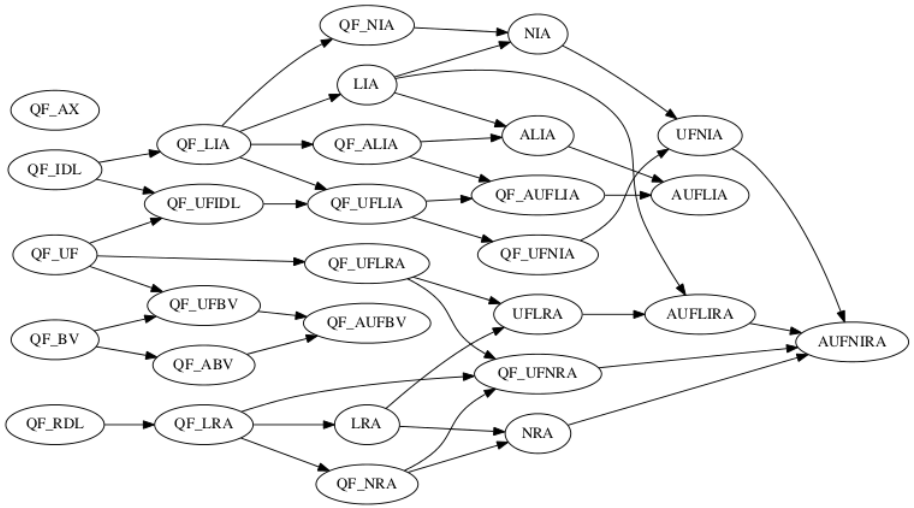
reflexivity	$\forall x\ x = x$
symmetry	$\forall x \forall y\ x = y \rightarrow y = x$
transitivity	$\forall x \forall y \forall z\ x = y \wedge y = z \rightarrow x = z$
congruence (functions)	$\forall \vec{x} \forall \vec{y}\ x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow$ $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$
congruence (predicates)	$\forall \vec{x} \forall \vec{y}\ x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow$ $P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n)$

This theory is **undecidable**, but its QF fragment is decidable.

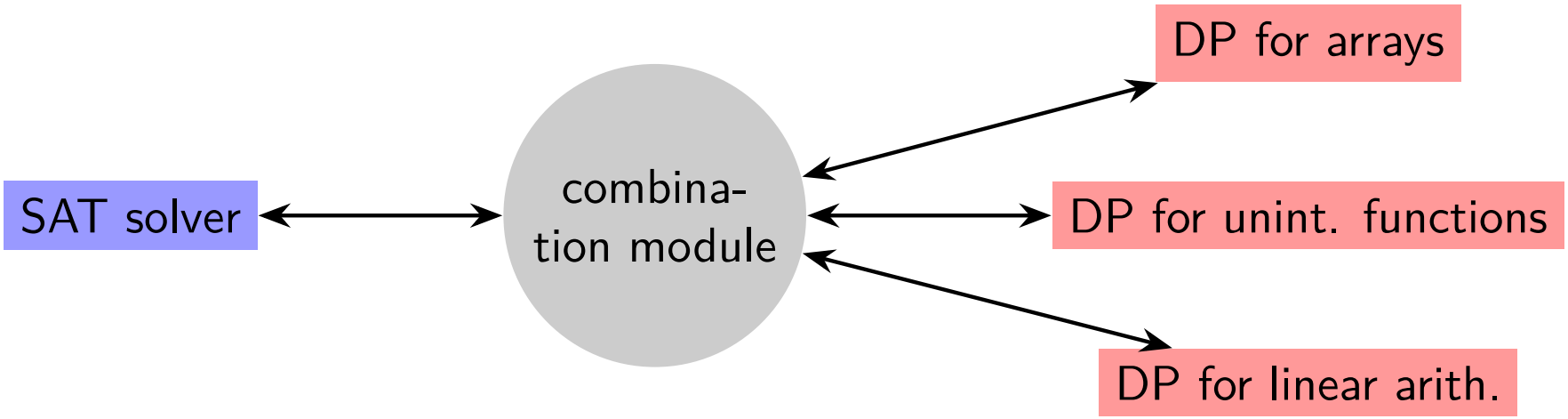
Available theories

The SMT-LIB initiative provides a common languages for SMT solvers and provides several theories and associated logics:

- integer numbers
- real numbers
- strings
- functional arrays
- bitvectors
- floating-point numbers



SMT solvers: principles



- the formula is abstracted in SAT and send to the SAT solver
- the solver returns a model or UNSAT
- the model is checked with theory decision procedures

SMT solvers: example

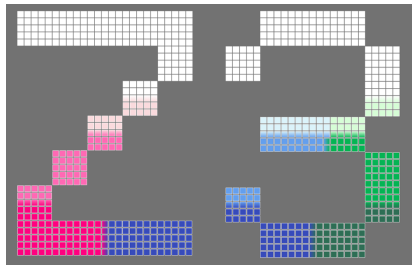
$$\underbrace{g(a) = c}_{x_1} \wedge \underbrace{f(g(a)) \neq f(c)}_{\neg x_2} \vee \underbrace{g(a) = d}_{x_3} \wedge \underbrace{c \neq d}_{\neg x_4}$$

1. send $\{x_1, \neg x_2 \vee x_3, \neg x_4\}$ to SAT solver
2. SAT solver returns a model: $\{x_1, \neg x_2, \neg x_4\}$
3. DP for uninterpreted function finds $\{x_1, \neg x_2, \neg x_4\}$ UNSAT
4. send $\{x_1, \neg x_2 \vee x_3, \neg x_4, \neg x_1 \vee x_2 \vee x_4\}$ to SAT solver
5. SAT solver returns a model: $\{x_1, x_3, \neg x_4\}$
6. DP for uninterpreted function finds $\{x_1, x_3, \neg x_4\}$ UNSAT
7. send $\{x_1, \neg x_2 \vee x_3, \neg x_4, \neg x_1 \vee x_2 \vee x_4, \neg x_1 \vee \neg x_3 \vee x_4\}$ to SAT solver
8. SAT solver return UNSAT

Real-world SMT solvers

There are several SMT solvers that are used for research and industrial projects:

- Alt-Ergo
- CVC5
- Z3



We will use Z3 during the lab sessions.

Outline

The Ints theory

SMT-LIB provides a theory for integers in which you can find:

- the `+`, `-`, `*`, `div` and `mod` operations
- the `=`, `<=`, `<`, `>=` and `>` relations

You can choose different logics:

- quantifier free linear arithmetic (`QF_LIA`)
- non-linear arithmetic (`NIA`)
- ...

Remark: you will not have to explicitely choose a logic during the lab sessions.

Back to Sudoku

Exercise

Can you now model the Sudoku problem using QFLIA?

Beware of equality!

Important: you will interact with the Z3 SMT solver through its Java API.

Do not mix **Java expressions** and **Z3 expressions**!

For instance, if you want to add the constraint $x \leq 3$ as a Z3 formula, you have to write

```
IntExpr  x = context.mkIntConst("x");
BoolExpr c = context.mkLe(x, context.mkInt(3));

solver.add(c);
```

not something like

```
int     x;
boolean c = x <= 3;

solver.add(c);
```

Outline

What are bit vectors?

Bit vectors are arrays that stores bits in a compact way.

When defining a bit vector, you have to choose its **size**.

Available operations and relations:

- natural number conversions `bv2nat` and `nat2bv`
- `concat` and `extract`
- bit-wise operations `not`, `and` and `or`
- arithmetic operations `neg`, `add`, `mul`, `div` and `rem`
- shift operations `shl`, `shr`
- extensions `s_extend` and `extend`
- `=`, `<=`, `<`, `>=` and `>` relations

Use of bit vectors

Bit vectors are extensively used for software verification, as in most programming languages

- integer values are implemented with fixed size bit vectors
- floating-point arithmetic uses IEEE754 standard

Lots of SMT solvers, particularly Z3, use bit-blasting to represent floating-point arithmetic, i.e. they reduce floating-point arithmetic to bit vectors arithmetic.

But even if the theory of bit vectors is decidable, it is quite expensive to use (reduction to SAT for instance)!

Overflow detection

Exercise

Let us suppose that you work with integers implemented as bit vectors of size 4. You have to add two values. How can you check that there is no overflow during the operation?

Outline

Definition of arrays

John Mac Carthy first defined in 1961 a theory for arrays that is implemented in Z3 for instance.

Leonardo de Moura and Nikolaj Bjorner.
Generalized, efficient array decision procedures.
Technical Report MST-TR-2009-121.
Microsft Research, 2009.
URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/fmcad09.pdf.

Definition of arrays

John Mac Carthy first defined in 1961 a theory for arrays that is implemented in Z3 for instance.

In the theory, arrays are **purely functional** (you cannot modify them) and **infinite**.

Two operations are available:

- $select(a, i)$ represents the value stored at index i in array a
- $store(a, i, v)$ represents the array obtained from a by changing the value at index i with value v

They respect the following axioms:

$$\begin{aligned} \forall a \forall i \forall v \quad & select(store(a, i, v), i) = v \\ \forall a \forall i \forall j \forall v \quad & i \neq j \rightarrow select(store(a, i, v), j) = select(a, j) \\ \forall a \forall b \quad & (\forall i \; select(a, i) = select(b, i)) \rightarrow (a = b) \end{aligned}$$

Example

For instance, the following formula expresses the fact that after storing the value 3 at index 1 in an array, then storing the value 4 at the same index, then the value at index 1 in the resulting array cannot be 3:

$$\forall a \; select(store(store(a, 1, 3), 1, 4), 1) \neq 3$$

Beware, the $\neq$ symbols in the previous equation is **inequality on integers**!

Swapping values

Exercise

Let us consider two arrays a and b. Can you write constraints expressing that b is the result of swapping two values in a?

Outline

How to mix theories?

You may often encounter formulas such as

$$x = y + 2 \ \wedge \ a = store(b, x + 1, 4) \ \wedge$$
$$(select(a, y + 3) = 2 \ \vee \ f(x - 1) \neq f(b + 1))$$

which mixes

- linear arithmetic
- arrays
- uninterpreted functions

Is it possible to combine them **modularly**?

The Nelson-Opper method

Nelson and Opper gave in 1979 conditions with which such theories can be mixed.

Greg Nelson and Derek C. Oppen.
"Simplification by cooperating decision procedures".
In: **ACM Trans. on Programming Languages and Systems** 1.2 (1979), pp. 245–257.

For instance, SMT-LIB defines `QF_AUFLIA`, the quantifier free theory of arrays, uninterpreted functions and linear arithmetic.

Outline

Conclusion

SMT solvers are now powerful tools to model and solve real-world problems such as

- program verification
- logic/mathematical puzzles
- capture the flag contests
- ...

The main difficulty is (as always) to **correctly model the problem**.

Outline

Outline

Transition systems

Transition systems can be used to model complex systems, software etc.

Transitions systems are defined by:

- a set of **state variables** representing the state of the system.
  An instantiation of state variables is called a **state**.
- an **initial predicate** I s.t. for a given state s, $I(s)$ is true iff s is the initial state of the system.
- a **relation transition** represented by a FOL predicate T s.t. for two given states s and s', $T(s, s')$ is true iff s' is a state that can be obtained by executing a valid transition from s.

See your previous lectures on transition systems and model-checking.

The array swap example

Exercise

Consider the previous problem in which you have modelled what swapping values in an array means.
Let us consider an initial array of size n. We want to define a transition system modelling the evolution of the initial array through successive swapping action.

- what are the states of the system?
- given a state s, how many states are directly accessible from s?

Model-checking

Given a transition system representing a system, we want to verify if the system verifies some interesting **properties**, e.g. liveness properties or safety properties.

Properties are often expressed as temporal logic formulas and you have seen algorithms and techniques to prove them.

We are interested here in **reachability properties**: is a particular state or set of states reachable from the initial state using the transition relation?

Model-checking and SMT

How can we use SMT solvers for model-checking?

- define states as a set of variables (arrays, integers etc)
- define **decision variables** representing the possible actions firing transitions
- express transition relation between states by formulas **guarded by decision variables**.
  Such formulas will be as "if action a is executed from state s, then the next state s' will verify $T(s, s')$".
- the **expected property** can be modeled as a formula over states $P(s)$

Therefore if we want to check if property P is verified after two transitions, **we may ask a SMT solver if the following formula is SAT or not**:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge P(s_2)$$

where $T(.,.)$ is the transition formula guarded by decision variables.

The swap array example cont.

Exercise

We want to check with a SMT solver if it is possible to sort an existing array in **exactly three swaps** (of course, you can swap an array index with itself).

- how many states are there? What do you choose to model them?
- what is the initial predicate?
- what are the decision variables for this problem?
- what are the transition formulas?
- how do you express the property to check?
- what is the final formula to check with the SMT solver to solve the problem?

Bounded model-checking

The previous example was rather easy as we only **unroll** the transition relation a given number of times.

How can we check properties without knowing how many times we must unroll the transition relation?

We will use **bounded model-checking**: given a "temporal horizon" k, we will try to verify the property by unrolling the transition relation 0 times, and then 1 time if the property is not verified etc until unrolling the relation k times.

Outline

BMC with SMT

Let us suppose that we have a bound k for verification of property P on a transition system defined by states s_i, an initial predicate I and a transition relation T.

The BMC algorithm can be defined as follows:

- check if the following formula is SAT or not:

$$I(s_0) \wedge P(s_0)$$

  If SAT, exit with `verified in 0 steps`.
- check if the following formula is SAT or not:

$$I(s_0) \wedge T(s_0, s_1) \wedge P(s_1)$$

  If SAT, exit with `verified in 1 step`.
- ...
- check if the following formula is SAT or not:

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots T(s_{k-1}, s_k) \wedge P(s_k)$$

  If SAT, exit with `verified in k steps`, if UNSAT exit with `cannot be verified in k steps`.

A more precise algorithm

Algorithm 3.1: A BMC algorithm using a SMT solver

Data: a transition system, a property P to verify and a bound $k \in \mathbb{N}$
Result: `verified in n steps` or `cannot be verified in k steps`

```
1  S := {I(s_0)};
2  i := 0;
3  while i ≠ k do
4  |   S := S ∪{P(s_i)};
5  |   if S is SAT then
6  |   |   return verified in i steps;
7  |   end
8  |   S := S \{P(s_i)};
9  |   S := S ∪{T(s_i, s_{i+1})};
10 |   i := i + 1;
11 end
12 return cannot be verified in k steps;
```

How to use Z3 for BMC?

You may wonder how to write the previous algorithm using Z3's Java API.

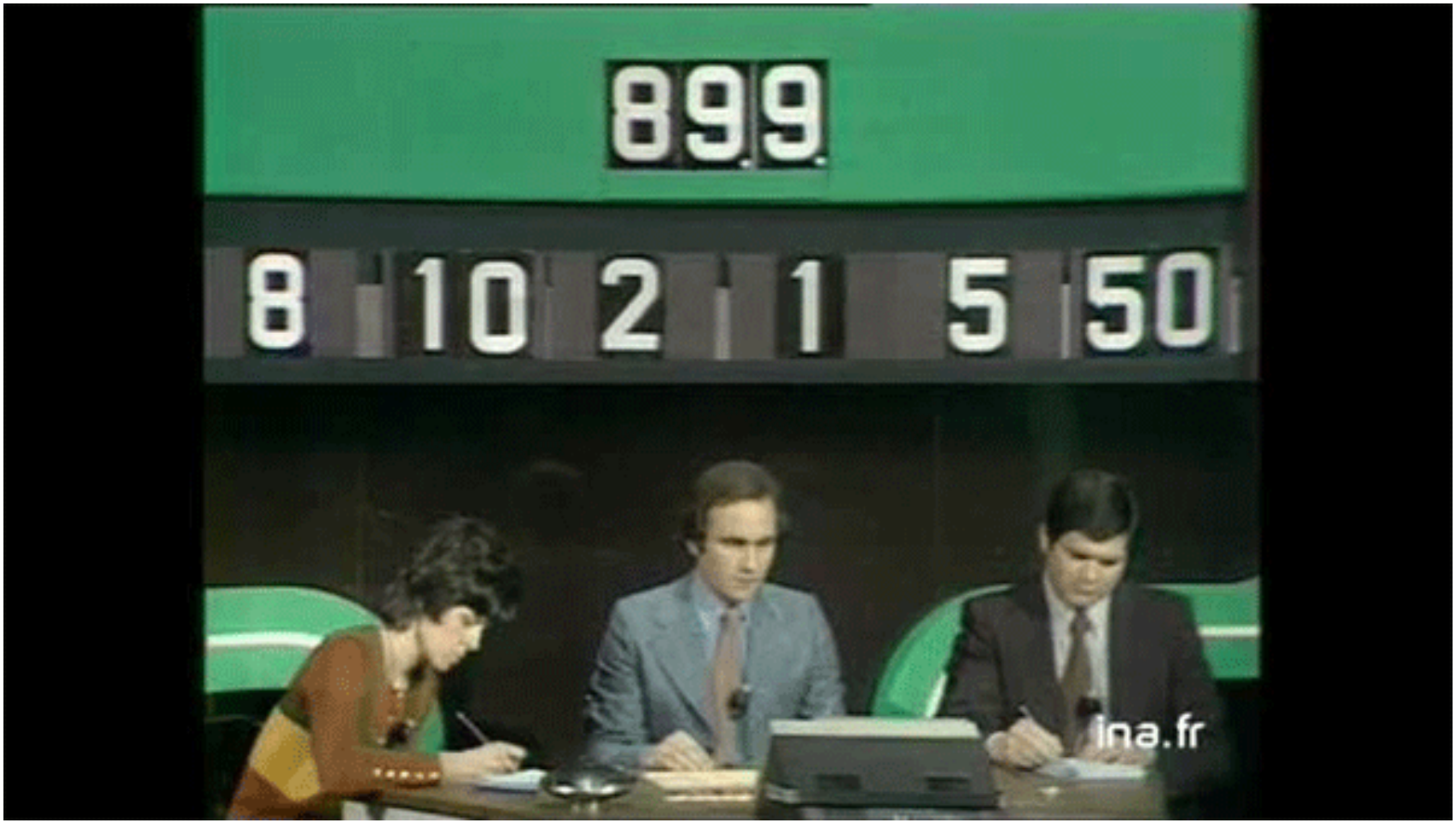
In particular, do you have to built the set S of formulae to check at each iteration.

Z3 basic solver is **incremental**, you can use:

- `solver.push()` to "save" the current state of the solver, particularly the formulae you have asserted
- `solver.pop()` to go back to the previously pushed state

Outline

Last century amusement...



Last century amusement...

- you have n integer constants
- you can use $+$, $*$, $-$, $/$
- you may use each integer constant only one time
- you must find the sequence of operations to get as a result the expected number
  If not possible, you must find the nearest integer.

Here:

$$(10 + 8) \times 50 - 1 = 899$$

Model for the project

Exercise

You will model the previous problem by a transition system.

1. what are the states of the system? How do you model them?
2. what is the meaning of a transition? Is the graph of reachable states finite? What is its diameter?
3. what are the decision variables of the problem?
4. how do you model the constraints? Are actions always feasible?