Introduction

Programs correction

A reactive system is correct if:

- it computes the right outputs (functionality)
- it computes fast enough (real-time)
- here: we focus on functionality

Validation means

- execution-based methods (debug, test, simulation...)
- static-analysis methods: why not "prove" correctness?
Functional verification

- Does the program compute the right outputs?
- Expected relation among time between inputs and outputs: temporal properties

Intuitive partition of temporal properties

- Safety: something (bad) never happens
- Liveness: something (good) may/eventually happens

Contents

1. Reactive systems and state machines ................................................. 4
2. Decision techniques (BDD) ............................................................... 26
3. BDD based methods ................................................................. 44
4. Decision techniques (Sat Solvers) .................................................... 62
5. Sat solver based methods ............................................................. 84
6. Appendix ................................................................. 93
Example: the beacon counter in a train

- Counts the difference between beacons and seconds
- Decides whether the train is late, early or ontime
- Hysteresis to avoid oscillations

```plaintext
define \( b(\text{sec, bea}: \text{bool}) \) returns (ontime, late, early: \text{bool});

var \text{diff}: \text{int};

let
    \text{diff} = 0 \to \text{pre} ( \text{diff} +
        (\text{if bea then 1 else 0}) + (\text{if sec then -1 else 0}));
    \text{early} = \text{false} \to \text{pre} (\n        (\text{ontime and diff > 3}) \text{ or } (\text{early and diff > 1}));
    \text{late} = \text{false} \to \text{pre}(\n        (\text{ontime and diff < -3}) \text{ or } (\text{late and diff < -1}));
    \text{ontime} = \text{not} (\text{early or late});

tel
```
Some properties

- It’s impossible to be late and early
- It’s impossible to directly pass from late to early
- It’s impossible to remain late only one instant
- If the train stops, it will eventually get late

The 3 first ones are obviously safety, while the one is a typical liveness: it refers to unbounded future.

Implicit state machines

Functionality of synchronous program

A synch. prog. is a function from infinite seq. of inputs to infinite seq. of outputs:
\[ \mathcal{P}(I_0, I_1, I_2, \cdots) = O_0, O_1, O_2, \cdots \]
defined via a well initialized internal memory

- Inputs I, outputs O
- Memory M, initial value \( M_0 \)
- Output function: \( O_t = f(I_t, M_t) \)
- Transition function: \( M_{t+1} = g(I_t, M_t) \)

Finally, \( \mathcal{P}(I_0, I_1, I_2, \cdots) = O_0, O_1, O_2, \cdots \) iff
\[ \exists M_0, M_1, M_2 \cdots \text{s.t. } \forall t \ O_t = f(I_t, M_t) \text{ and } M_{t+1} = g(I_t, M_t) \]
Common model for synchronous programs

- Obvious for Lustre (memory = \texttt{pre} operators)
- Less obvious, but still true, for Esterel/SyncCharts (cf. compilation)

Implicit vs explicit

An ISM is equivalent to an explicit state/transition system:

- States are all possible values of \( M \):
  \[ Q = \mathcal{D}(M) \]
- Transition \( q \xrightarrow{i/o} q' \) iff \( q' = g(i, q) \) and \( o = f(i, q) \)
- In general: infinite state machine (numerical)

Example: beacon counter

- \( I = \{ \text{sec, bea} \} \), \( O = \{ \text{late, ontime, early} \} \)
- A memory for each "\( \to \text{pre} \)" expression,
  (e.g. Plate for "\( \text{false} \to \text{pre late} \)"
  \( M = \{ \text{Plate, Pontime, Pearly, Pdiff} \} \)
  with \( M_0 = (\text{false}, \text{true}, \text{false}, 0) \)
- Functions directly given by the Lustre equations

A small part of the explicit automaton:
Conservative Abstraction

Model and verification

The explicit automaton is the set of behavior, so exploring the automaton is checking the program.

Problem: The automaton may be infinite, or at least enormous, it is impossible to explore it.

Idea: work on a finite (not too big) abstraction of the program. N.B. the abstraction must conserve at least some properties (otherwise it’s useless).

Example

Abstraction of numerical comparisons in the beacon counter, they become “free” boolean variables:

- $a_1$ for $\text{diff} > 3$
- $a_2$ for $\text{diff} < -3$
- $a_3$ for $\text{diff} < -1$
- $a_4$ for $\text{diff} > 1$
Conserved properties

- It’s impossible to be late and early (safety)
- It’s impossible to directly pass from late to early (safety)

Lost properties

- It’s impossible to remain late only one instant (safety)
- If the train stops, it will eventually get late (liveness)

More serious: introduced property

- It’s possible to remain late only one instant (liveness): true on the abstraction, false on the real program!

⇒ Important to precisely know what is preserved by the abstraction

Abstraction and safety

- Finite abstraction is a special case of over-approximation
- Anything which is impossible in the abstraction is impossible on the program
- The counterpart is (in general) false

⇒ safeties are preserved or lost, but never introduced

As a consequence, when checking a safety on the abstraction:

- the verification succeeds ⇒ property satisfied
- the verification fails ⇒ inconclusive
  (it may be a false negative)
Expressing properties

Liveness requires complex formalisms (temporal logics)
Safety can be programmed \(\Rightarrow\) observers

Observer

- Observe the inputs and outputs of the program
- Outputs "ok" as long as the behavior meets the property
  (or, equivalently, outputs "ko" when the behavior violate the property)

Example (in Lustre)

- It’s impossible to be late and early:
  \[ ok = \text{not} (\text{late} \text{ and} \text{ early}); \]

- It’s impossible to directly pass from late to early:
  \[ ok = \text{true} \rightarrow \text{not} (\text{early} \text{ and} \text{ pre late}); \]

- It’s impossible to remain late only one instant:
  \[ \text{Plate} = \text{false} \rightarrow \text{pre late}; \]
  \[ \text{PPlate} = \text{false} \rightarrow \text{pre Plate}; \]
  \[ ok = \text{not} (\text{not} \text{ late and} \text{Plate and not} \text{ PPlate}); \]

Let see a quick demo ...
Assumptions

Convenient to split property into assumption/conclusion:

"if the train keeps the right speed, it remains on time"

property is simply \( \text{ok} = \text{ontime} \), assumption can be:

- naive: \( \text{assume} = (\text{sec} = \text{bea}) \);

- more sophisticated, bea and sec alternate:

  \[
  \text{SF} = \text{switch}(\text{sec and not bea}, \text{bea and not sec}) ;
  \]

  \[
  \text{BF} = \text{switch}(\text{bea and not sec}, \text{sec and not bea}) ;
  \]

  \[
  \text{assume} = (\text{SF} \Rightarrow \text{not sec}) \text{ and } (\text{BF} \Rightarrow \text{not bea}) ;
  \]

with:

\[
\text{node switch}(\text{on}, \text{off} : \text{bool}) \text{ returns } (\text{s} : \text{bool}) ;
\]

\[
\text{let } \text{s} = \text{false} \text{ -> pre}(\text{if } \text{s} \text{ then not } \text{off} \text{ else } \text{on}) ; \text{ tel}
\]

Expressing properties 16/101

General scheme

- We suppose provided such a verification program

- Goal: if assume remains indefinitely true, then ok remains indefinitely true:

  \[
  (\text{always assume}) \Rightarrow (\text{always ok})
  \]

- Note: it is NOT a "regular" safety, so in a first step, we approximate it by:

  \[
  \text{always } ((\text{once not assume}) \text{ or } \text{ok})
  \]

  (the problem will be explained later)

Expressing properties 17/101
Abstracted verification program

Special case of Boolean synchronous program with 2 "outputs"

- Free variables $V$, state variables $S$
- Initial state(s): $\text{Init} : B^{\mid S\mid} \rightarrow B$
- Transition functions: $g_k : B^{\mid S\mid} \times B^{\mid V\mid} \rightarrow B$ for $k = 1 \cdots \mid S\mid$
- Assumption: $H : B^{\mid S\mid} \times B^{\mid V\mid} \rightarrow B$
- Property: $\phi : B^{\mid S\mid} \times B^{\mid V\mid} \rightarrow B$

(N.B. we identify predicates and sets)

Associated explicit automaton

We note $Q = B^{\mid S\mid}$ the state space

We use "pre" and "post" functions:

- for $q \in Q$, $\text{post}_H(q) = \{ q' / \exists v \ q \xrightarrow{v} q' \wedge H(q,v) \}$
- for $X \subseteq Q$, $\text{Post}_H(X) = \bigcup_{q \in X} \text{post}_H(q)$
- for $q \in Q$, $\text{pre}_H(q) = \{ q' / \exists v \ q' \xrightarrow{v} q \wedge H(q',v) \}$
- for $X \subseteq Q$, $\text{Pre}_H(X) = \bigcup_{q \in X} \text{pre}_H(q)$
Significant state sets

- Initial state(s): \( \text{Acc}_0 = \{ q/\text{Init}(q) \} \)
- Error states: \( \text{Err} = \{ q/\exists v\ H(q, v) \land \neg \phi(q, v) \} \)
- Reachable states: \( \text{Acc} = \mu X \cdot (X = \text{Init} \cup \text{Post}_H(X)) \)
- Bad states: \( \text{Bad} = \mu X \cdot (X = \text{Err} \cup \text{Pre}_H(X)) \)

Goal

Naive: prove that \( \text{Acc} \cap \text{Bad} = \emptyset \)

No need to compute both \( \text{Acc} \) and \( \text{Bad} \):

- prove that \( \text{Acc} \cap \text{Bad}_0 = \emptyset \) (forward method)
- prove that \( \text{Bad} \cap \text{Acc}_0 = \emptyset \) (backward method)

Remark: methods are non symmetric because of determinism

Proving properties

Enumerative (forward) algorithm

\[
\text{CurAcc} := \text{Init} \\
\text{Done} := \text{empty} \\
\text{while it exits } q \text{ in CurAcc - Done do } \{ \\
\qquad (* q \in \text{CurAcc} \setminus \text{Done} *) \\
\qquad \text{for all } q' \text{ in post}_H(q) \text{ do } \{ \\
\qquad \qquad \text{if } q' \text{ in } \text{Bad}_0 \text{ then EXIT(failed)} \\
\qquad \qquad \text{put } q' \text{ in CurAcc} \\
\qquad \} \\
\qquad \text{put } q \text{ in Done} \\
\} \\
(* \text{we have } \text{CurAcc} = \text{Done} = \text{Acc} *) \\
\text{EXIT(succeed)}
\]
See an example/exercise of exploration: an arithmetic circuit (cf. 6.1)

Notes on implementation

• depth first, breath first, or other
• compact encoding of states
• very costly:
  \[ |Acc| \text{ times the cost of post}_H(q), \text{ with } |Acc| \sim 2^{|S|} \]
• backward is even worse: \( \text{pre}_H(q) \) is more complex than \( \text{post}_H(q) \)
  (enumerative backward is never used in practice)

Big problem: computing \( \text{post}_H(q) \)

• For a given \( q \), find all \( v \) s.t. \( H(q, v) \)
• Typical decision problem (NP-complete)
• Naive method: try all \( 2^{|V|} \) possible values
• Need for non trivial, efficient decision procedure
  \[ \Rightarrow \text{Digression on efficient decision techniques} \]
2. Decision techniques (BDD)

Decision techniques

Problem: let $F$ be a formula on $V$, find all $v \in 2^{|V|}$ s.t. $F(v)$

- Mainly two kind of solutions:
  - Enumeration of the solutions, related to Sat-Solving, reference algo is Davis-Putnam
  - Construction of the solution set, related to canonical form, reference method is Binary Decision Diagrams (BDD)

- We first study BDDs:
  - Used with a certain success
  - Address also the problem of state explosion
  - Ad hoc algorithms: Symbolic Model Checking
Binary Decision Diagrams

Shannon decomposition

- For any \( f \in B^n \rightarrow B \):
  \[ f(x, y, ..., z) = x.f(1, y, ..., z) + \bar{x}.f(0, y, ..., z) \]

- Let’s define \( f_x \) and \( f_{\bar{x}} \) in \( B^{n-1} \rightarrow B \) by:
  \[ f_x(y, ..., z) = f(1, y, ..., z) \]
  \[ f_{\bar{x}}(y, ..., z) = f(0, y, ..., z) \]

- For any \( f \) and any \( x \), \( f_x \) and \( f_{\bar{x}} \) are unique

**Exercise**

Let \( f(x, y, z) = x.y + (y \oplus z) \), compute \( f_x, f_{\bar{y}}, f_z \)?

\[ f_x = y + (y \oplus z) \]
\[ f_{\bar{y}} = z \]
\[ f_z = x.y + \bar{y} = x + \bar{y} \]

Shannon tree

- When applying recursively the S.D. on all variables, one obtains:
  \[ 1 \] (the always-true function) or
  \[ 0 \] (the always-false function)

- Example, for \( f = x.y + (y \oplus z) \):
  \[ f_{\bar{x}} = f(0, y, z) = y \oplus z \]
  \[ f_{\bar{xy}} = f(0, 1, z) = \neg z \]
  \[ f_{\bar{xyz}} = f(0, 1, 1) = 0 \]

- Shannon tree: graphical representation of all the \( 2^n \) steps
Full decomposition example

\[ f(x, y, z) = x.y + (y \oplus z) \]

N.B. For a given variable ordering the tree is unique

Binary Decision Diagram

- Concise representation of the Shannon tree
- No useless nodes (if x then g else g ⇔ g)
- Share common sub-graph (DAG)

N.B. For a given variable ordering the BDD is unique
Formal definition

Definition

- Let $V$ be a set of variable, totally ordered by $\preceq$
- Let $V^* = V \cup \{\infty\}$ extened with a max value $(\forall x \in V^* \ x \preceq \infty)$
- BDDs are defined, together with their "range" $rg : \text{BDD} \rightarrow V^*$
  - $\leftarrow 1$ is a BDD with $rg(1) = \infty$
  - $\leftarrow 0$ is a BDD with $rg(0) = \infty$
  - $\leftarrow$ let $x \in V$, let $h$ and $l$ be two BDDs with $x \prec rg(h)$ and $x \prec rg(l)$, then $\alpha = (x, l, h)$ is also a BDD

We note $\text{AA}_{h \leftarrow l}(x)$ such a triplet

Implementation

- uniqueness of leaves 0 and 1 is built-in
- uniqueness of binary nodes guaranteed by hash-coding
- the creation of binary nodes is implemented by a function $\mathcal{N}(x, \alpha, \beta)$

  - we note $\bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\beta}$ for $\mathcal{N}(x, \alpha, \beta)$, and $\bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\beta}$ for a built BDD

  - $\bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\beta} =$ ERROR if $rg(\alpha) \prec x$ or $rg(\beta) \prec x$

  - $\bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\alpha} = \alpha$

  - $\bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\beta} = \bigwedge \limits^{x} \bigwedge \limits^{\alpha} \bigwedge \limits^{\beta}$ otherwise
Operations on BDDs

Negation

- \( -1 = 0 \)
- \( -0 = 1 \)

\[ - \bigwedge_{f_0}^x f_1 = \bigwedge_{-f_0} f_1 \]

Binary operators

Property: any usual operator \( \star \) (in \( +, \cdot, \oplus, \Rightarrow, \leftrightarrow \)), distribute on Shannon decomposition:

\[
(x \cdot f_x + \bar{x} \cdot \bar{f}_{\bar{x}}) \star (x \cdot g_x + \bar{x} \cdot g_{\bar{x}}) = x \cdot (f_x \star g_x) + \bar{x} \cdot (f_{\bar{x}} \star g_{\bar{x}})
\]

Binary operators (ctd)

As a consequence, recursive rules are,

for \( f = \bigwedge_{f_0}^x f_1 \) and \( g = \bigwedge_{g_0}^y g_1 \):

- \( f \star g = \bigwedge_{f_0 \star g_0}^{f_1 \star g_1} \) if \( x < y \) (balance)

- \( f \star g = \bigwedge_{f \star g_0}^{f \star g_1} \) if \( y < x \) (balance)

- \( f \star g = \bigwedge_{f_0 \star g_0} f_1 \star g_1 \) if \( x = y \)
Binary operators (ctd)

- Terminal rules apply in priority, for instance:

  \[ (1 + \alpha) = (\alpha + 1) = 1 \]
  \[ (0 + \alpha) = (\alpha + 0) = \alpha \]

  \[ (1 \cdot \alpha) = (\alpha \cdot 1) = \alpha \]
  \[ (0 \cdot \alpha) = (\alpha \cdot 0) = 0 \]

  \[ \alpha \oplus \alpha = 0 \]
  \[ (0 \oplus \alpha) = (\alpha \oplus 0) = \alpha \]
  \[ (1 \oplus \alpha) = (\alpha \oplus 1) = \neg \alpha \]

**Exercise**

Terminal rules for “⇒” (implication)?

\[ (0 \Rightarrow \alpha) = (\alpha \Rightarrow 1) = 1 \]
\[ (\alpha \Rightarrow 0) = \neg \alpha \]
\[ (1 \Rightarrow \alpha) = \alpha \]

Quantification

- Boolean quantification is simple

  \[ \leftarrow \text{like for any finite domain} \]

  \[ \leftarrow \text{unlike infinite domains (e.g. integers)!} \]

**Exercise**

Definition of “∃x α”? Based on the enumeration of values:

\[ \exists x \quad \alpha(x, \vec{w}) = \alpha(0, \vec{w}) \lor \alpha(1, \vec{w}) \]

\[ \exists x \quad 1 = 1 \quad \exists x \quad 0 = 0 \]

\[ \exists x \quad \begin{array}{cc}
  l & \hline \\
  h & \hline \\
\end{array} = h \lor l \]

\[ \exists x \quad \begin{array}{cc}
  l & \hline \\
  y & \hline \\
\end{array} = \text{if } x < y \text{ then } \begin{array}{cc}
  l & \hline \\
  y & \hline \\
\end{array} \text{ else } \exists x \quad \begin{array}{cc}
  l & \hline \\
  \hline \\
\end{array} \quad \exists x \quad \begin{array}{cc}
  l & \hline \\
  y & \hline \\
\end{array} \quad \exists x \quad \begin{array}{cc}
  h & \hline \\
  \hline \\
\end{array} \]

Same question for “∀v α”?
Notes on complexity

- Cost of $\neg \alpha$: is linear w.r.t to $\text{size}(\alpha)$
- Cost of $\alpha \ast \beta$: is in $\text{size}(\alpha) \times \text{size}(\beta)$
- Algebraic formula to BDD: exponential (worst case)
- Variable ordering is very important:
  $$\left( x_1 \oplus x_2 \right) \cdot \left( x_3 \oplus x_4 \right) \cdot \ldots \cdot \left( x_{2n-1} \oplus x_{2n} \right)$$

  - Size in $O(n)$ for $x_1 \prec x_2 \prec x_3 \prec \cdots \prec x_{2n}$
  - Size in $O(2^n)$ for $x_1 \prec x_3 \prec \cdots \prec x_{2n-1} \prec x_2 \prec x_4 \prec \cdots \prec x_{2n}$

Lots of variants/implementations

⇒ an interesting variant: Signed BDD

Signed BDD

Note on negation

- BDDs for $f$ and $\neg f$ are very similar: same structure, only leaves are different
- They don’t share any node (costly in space)
- Computing $\neg$ costs (a little)

Sharing structure

- Concretely represent only one of $f$ or $\neg f$
- Define the other as the negation
- Problem: how to keep it canonical?
Positive functions

**Definition**

\( f \in \mathbb{B}^n \rightarrow \mathbb{B} \) is positive iff \( f(1, 1, \ldots, 1) = 1 \)

Idea: Nodes are reserved for positive functions, negative ones are defined by adding a sign flag

Signed BDD 40/101

---

**SBDD**

**Recursive definition of SBDD and FPOS**

- A SBDD is a couple \((s, f) \in \{+, -\} \times FPOS\)
  
  i.e. (sign + positive func)

- 1 is a FPOS (the unique leaf)

- A triplet in \(V \times SBDD \times FPOS\) is a FPOS, with the same range constraints than classical BDD

Examples:

- \((+, 1)\) is "always true"  \((- , 1)\) is "always false"

- \((+, x)\) is \(x\)  \((- , x)\) is \(\neg x\)

Signed BDD 41/101
Notes on complexity

- Negation is free
- Always better than "classical" BDD (space and time)

Using a BDD library

- Even when not explicit, they are always SBDD
- Variable ordering is hidden (dynamic reordering)
- High level Boolean functions are provided
  (true-bdd, false-bdd, idy-bdd(v), and-bdd(f,g) etc)
- Some other ad hoc procedures (depending on Shannon decomposition)
3. BDD based methods

Forward Symbolic algorithms ......................................................... 45
Backward symbolic algorithm ......................................................... 58

Forward Symbolic algorithms

Encoding sets with formulas
- Enumerative algo ⇒ complexity is related to number of states/transitions
- Idea: encoding sets (states, transitions) by Boolean formula (BDD)
- Example: \( S = \{x, y, z, t\} \), states such that \( x + y \cdot \neg t \):
  - 10 concrete states
  - small formula (3 BDD nodes)
- this family of method is called Symbolic Model Checking
Reachable states computation

- operates on a verification program \((S, V, \text{Init}, G, \phi, H)\),
  (we note \(Q = 2^{|S|}\) the state space),
- manipulates sets of states (formulas on \(S\)) and transitions (formulas on \(S \times V\)),
- uses set (i.e. logical) operators \((\cup, \cap, \setminus\) etc),
- uses image computing: \(\text{Post}_H : 2^Q \rightarrow 2^Q\)
  \[\text{Post}_H(X) = \{q' / \exists q \in X, v \in 2^V \ H(q, v) \land q \xrightarrow{v} q'\}\]
  (implementation is presented later)

Algorithm

Manipulates a BDD \(A = \) states reachable in less than \(n\) transitions

- Initially: \(A := \text{Init}\)
- Repeat:
  \(\Leftarrow\) if \(A \land \text{Err} \neq 0\) then EXIT(failed)
  \(\Leftarrow\) else let \(A' := A \lor \text{Post}_H(A)\)
    if \(A' = A\) then EXIT(succeed)
    else \(A := A'\), and continue

When the proof succeeds, we have \(A = A' = \text{Acc}\)
Execution

\[ A_0 = \text{Init} \]

\[ A_1 \]

\[ A_2 \]

\[ A_k = A_{k-1} = \text{Acc} \]

Proof succeeds

Forward Symbolic algorithms 48/101

Execution (cntd)

\[ A_0 = \text{Init} \]

\[ A_1 \]

\[ A_2 \]

\[ A_k \cap \text{Err} \neq \emptyset \]

Proof fails

Forward Symbolic algorithms 49/101
Naive implementation of $Post_H(X)$

Using only logical operators, one build a (huge) formula over:

- source state variables $s_1, s_2, \cdots s_n$ (or $s$)
- free variables $v_1, v_2, \cdots v_m$ (or $v$)
- target state variables $s'_1, s'_2, \cdots s'_n$ (or $s'$)

$\exists s, v \left( X(s) \land H(s, v) \land \bigwedge_{i=1}^{n} s'_i = g_i(s, v) \right)$

$\rightarrow s$ is a source state
$\rightarrow (s, v)$ satisfies the assumption
$\rightarrow$ each $s'_i$ is the image of $g_i$
$\rightarrow$ elimination of all $s_i$ and all $v_j$

Result: the formula $N(s')$ characterizing the target states

Efficient implementation of $Post_H(X)$

- **Problem:** naive method merges $s_i$ and $s'_j$ in BDD
- **Idea:** using the fact that we have transition *functions*
- **How:** Define $Post_H(X)$ by induction on transition functions

In order to simplify, we note:

- $l$ for $(s, v)$
- $Y(l)$ for $X(l) \land H(l)$
  (Remark: $Y \neq 0$, otherwise it's trivial $Post_H(0) = 0$)
- $Img[g_1\cdots g_n](Y)$ the expected formula over $s'$, defined by:
  $Img[g_1\cdots g_n](Y) = \exists l Y(l) \land \bigwedge_{i=1}^{n} s'_i = g_i(l)$

Let us study the Shannon decomposition of this formula ...
Decomposition on $s'_1$:

- $s'_1 = 1$ gives $I_1 = \exists l \ (Y \land g_1)(l) \land (\bigwedge_{i=2}^{n} s'_i = g_i(l))$
- $s'_1 = 0$ gives $I_0 = \exists l \ (Y \land \neg g_1)(l) \land (\bigwedge_{i=2}^{n} s'_i = g_i(l))$

We consider 3 cases:

- $Y \land g_1$ is identically false (i.e. $Y \land \neg g_1 = Y$):
  
  $I_1 = 0$
  
  $I_0 = (\exists l \ Y(l) \land \bigwedge_{i=2}^{n} s'_i = g_i(l)) = Img[g_2...g_n](Y)$

- $Y \land \neg g_1$ is identically false (i.e. $Y \land g_1 = Y$):
  
  $I_1 = (\exists l \ Y(l) \land \bigwedge_{i=2}^{n} s'_i = g_i(l)) = Img[g_2...g_n](Y)$
  
  $I_0 = 0$

- otherwise:
  
  $I_1 = \exists l \ (Y \land g_1)(l) \land (\bigwedge_{i=2}^{n} s'_i = g_i(l)) = Img[g_2...g_n](Y \land g_1)$
  
  $I_0 = \exists l \ (Y \land \neg g_1)(l) \land (\bigwedge_{i=2}^{n} s'_i = g_i(l)) = Img[g_2...g_n](Y \land \neg g_1)$

Conclusion: recursive definition of $Img$, where $s'_i$ variables are never merged with the other

- $Img[](Y) = 1$
- $Img[g_i...g_n](Y) =$

  if $Y \Rightarrow g_i$ then
  
  $Img[g_{i+1}...g_n](Y)$

  if $Y \Rightarrow \neg g_i$ then
  
  $Img[g_{i+1}...g_n](Y)$

  else
  
  $Img[g_{i+1}...g_n](Y \land \neg g_i)$

  $Img[g_{i+1}...g_n](Y \land g_i)$
Optimization of image computing

- How to define a "Knowing that" operator?
- intuitively, $h = f$ knowing that $g$ must be
  \[ \Leftarrow \text{ equivalent to } f \text{ if } g \text{ is true } (f \cdot g \Rightarrow h \Rightarrow f + \bar{g}) \]
  \[ \Leftarrow \text{ such that } h = 1 \text{ if } g \Rightarrow f \]
  \[ \Leftarrow \text{ such that } h = 0 \text{ if } g \Rightarrow \neg f \]
  \[ \Leftarrow \text{ as simple as possible otherwise} \]

- Remarks:
  \[ \Rightarrow \text{ Depend on a particular representation (not strictly logical)} \]
  \[ \Rightarrow \text{ There are many of such operators} \]
  \[ \Rightarrow \text{ Some of them have interesting extra properties} \]

Constrain operator

$f \downarrow g$, is defined for $g \neq 0$ by:
- $f \downarrow 1 = f$
- $0 \downarrow g = 0$
- $1 \downarrow g = 1$

\[ \begin{array}{c}
\frac{x}{f_0}
\downarrow
\frac{x}{g_1}
= f_1 \downarrow g_1
\end{array} \]

\[ \begin{array}{c}
\frac{x}{f_0}
\downarrow
\frac{x}{g_0}
= f_0 \downarrow g_0
\end{array} \]

- otherwise, classical "balance" rules
Constrain operator (cntd)

- Extra properties of constrain:
  - distributes on negation:
    \[ (\neg f) \downarrow g \equiv \neg (f \downarrow g) \]
  - substitutes to \( \land \) under \( \exists \) quantifier:
    \[ \exists x \ (f \land g)(x) \equiv \exists x \ (f \downarrow g)(x) \]
  - in particular:
    \[ \exists \ Y \ (l \land \bigwedge_{i=1}^{n} s'_i = g_i(l)) \equiv \exists \ (g_i \downarrow Y)(l)) \]

- Constrain and image computing:
  - \[ Img[g_1\ldots g_n](Y) = Img[(g_1 \downarrow Y)\ldots(g_n \downarrow Y)](1) \]
    \( \Rightarrow \) second argument useless, only compute universal images

---

Optimized image computing

- Compute all \( t_i = g_i \downarrow (X \downarrow H) \)
- Then \( Img[t_1, \ldots, t_n] \) with:
  \[ Img[] = 1 \]
  \[ Img[0, t_{i+1}, \ldots, t_n] = Img[t_{i+1}, \ldots, t_n] \]
  \[ Img[1, t_{i+1}, \ldots, t_n] = Img[t_{i+1}, \ldots, t_n] \]
  \[ Img[t_i, t_{i+1}, \ldots, t_n] = Img[t_{i+1} \downarrow \bar{t}_i, \ldots, t_n \downarrow \bar{t}_i] Img[t_{i+1} \downarrow t_i, \ldots, t_n \downarrow t_i] \]
Backward symbolic algorithm

How it works

• Very similar to forward

• Uses reverse image computing $\text{Pre}_H : 2^Q \rightarrow 2^Q$
  $$\text{Pre}_H(X) = \{q \mid \exists q' \in X, v \in 2^V \quad H(q, v) \land q \rightarrow v \rightarrow q'\}$$

• Uses $B = \text{states leading to Err in less than } n \text{ transitions}$

• Initially: $B := \text{Err}$

• Repeat:
  $$\text{if } B \land \text{Init} \neq 0 \text{ then } \text{EXIT(failed)}$$
  $$\text{else let } B' := B \lor \text{Pre}_H(B)$$
  $$\text{if } B' = B \text{ then } \text{EXIT(succeed)}$$
  $$\text{else } B := B', \text{ and continue}$$

When the proof succeeds, we have $B = B' = \text{Bad}$
Implementation of $\text{Pre}_H(X)$

No need to merge $s_i$ and $s'_i$ in BDD

Similar to function composition

- $\text{Pre}_H(X) = \exists v \ H(s, v) \land \text{Revim}[X](s, v)$

with:

- $\text{Revim}[0] = 0$
- $\text{Revim}[1] = 1$
- $\text{Revim}[\bigwedge_{X_0}^{s'} \bigvee_{X_1}^i ] = g_i(s, v) \cdot \text{Revim}[X_1] + \neg g_i(s, v) \cdot \text{Revim}[X_0]$

Conclusion

- Approach limited to safety (i.e. program invariants)
- Exhaustive (but symbolic) finite state machine exploration
- Inspired/derived from methods designed for circuit verification (90’s)
- Despite the "untractable" theoric complexity, works well for a large class of programs:
  - control programs, few numerical aspects (otherwise abstraction may be too rough)
  - small size, but note that complexity is not directly related to the number of variables (symbolic)
4. Decision techniques (Sat Solvers)

The SAT problem

Definition and complexity

• Is a propositional formula satisfiable?

• More generally: find all solutions.

• This is “THE” NP-complete problem,
  i.e. combinatorial explosion in time and/or space (worst case)

Restriction

• Implicitly: only consider methods with low-cost in memory,

• i.e. memory cost is polynomial,

• i.e. may explode in time but not in space

• It excludes methods like BDD
SAT input data

• For the user: formula in algebraic form (¬, ∨, ∧, ⇒, ⇔, ⊕ etc.)

• For the algorithms: Conjunctive Normal Form (CNF)

  → Conjunctive because it is the hard form

  → The dual (Disjunctive Normal Form) is “simple”: it can be linearly reduced

  Sat(ϕ ∨ ψ) iff Sat(ϕ) OR Sat(ψ)

  → Normal Form: for simplicity

Terminology

• A literal l is either a variable x, or the negation of a variable ¯x.

• A clause is a disjunction of literals c = \bigvee_{i \in I} l_i.

• A (CNF) formula is a conjunction of clauses f = \bigwedge_{j \in J} c_j

Notations

• “logical AND ” is ∧ or ·

• “logical OR ” is ∨ or +

• “logical NOT ” is ¬ or ¯
CNF transformation

Naive method

De Morgan’s law to push “¬” the leaves

\[
\begin{align*}
\text{CNF}(x) &= x \\
\text{CNF}(\overline{x}) &= \overline{x} \\
\text{CNF}(f.g) &= \text{CNF}(f).\text{CNF}(g) \\
\text{CNF}(\overline{(f + g)}) &= \text{CNF}(\overline{f}).\text{CNF}(\overline{g}) \\
\text{CNF}(f + g) &= \text{Merge}(\text{CNF}(f), \text{CNF}(g)) \\
\text{CNF}(\overline{(f.g)}) &= \text{Merge}(\text{CNF}(\overline{f}), \text{CNF}(\overline{g}))
\end{align*}
\]

where “merge” is the clause cross-product:

\[
\text{Merge}(\bigwedge_{i \in I} \phi_i, \bigwedge_{j \in J} \psi_j) = \bigwedge_{i,j \in I \times J} (\phi_i + \psi_j)
\]

Example: \(\text{CNF}(x.y + \overline{x}.(z + t)) = ?\)

\((\overline{x} + y).(x + y + z)\)

Problem

- Naive algo is \textit{exponential} in the worst case:
  \(f = (x_0.x_1) + (x_2.x_3) + \cdots + (x_{2k}.x_{2k+1})\)
  \(\Rightarrow 2^{k+1}\) clauses.

- Not surprising: as complex as DNF, that is, as complex as SAT itself!

Indirect method

- Idea: add extra variables to “split” big formulas, example:

  \(f = \overline{x} + y + z\)
  \(f' = \overline{x} + y + z + t\)

  N.B. does not change the SAT problem: \(Sat(f)\) iff \(Sat(f')\)
Classical CNF construction, aka 3-SAT construction

- One (extra) variable per (binary) operator.

- Example:

  \[ f = (x \cdot y + \neg(x + \bar{y} + \bar{z})) \]
  gives \( f = a \) where
  \[
  \begin{align*}
  * & \ a = b + c \quad \text{and} \\
  * & \ b = x \cdot y \quad \text{and} \\
  * & \ c = \neg(x + d) = \bar{x} \cdot \bar{d} \quad \text{and} \\
  * & \ d = \bar{y} + \bar{z}
  \end{align*}
  \]

- Each equation gives exactly 3 clauses, e.g.:
  \[
  \begin{align*}
  * & \ (a = b + c) \iff (\bar{a} + b + c) \cdot (a + \bar{b}) \cdot (a + \bar{c}) \\
  * & \ (b = x \cdot y) \iff (b + \bar{x} + \bar{y}) \cdot (\bar{b} + x) \cdot (\bar{b} + y)
  \end{align*}
  \]

- Finally: \( f \) gives 1 unit clause + 4 equations (binary ops.) that each gives 3 clauses:
  \[
  \begin{align*}
  * & \ 13 \text{ clauses} \\
  * & \ \text{LINEAR: size of } f' = 1 + 3 \times \text{size of } f
  \end{align*}
  \]

Note on 3-SAT formulation

- As seen in the example, \(+\) and \(\cdot\) operators give 3 clauses,

- Exclusive or (difference) and equivalence are “more complex” and give 4 clauses:

  \[
  \begin{align*}
  * & \ CNF(a = (x \oplus y)) = (\bar{a} + x + \bar{y}).(\bar{a} + x + y).(a + \bar{x} + \bar{y}).(a + x + \bar{y}) \\
  * & \ CNF(a = (x = y)) = (\bar{a} + x + \bar{y}).(\bar{a} + x + y).(a + \bar{x} + \bar{y}).(a + x + y)
  \end{align*}
  \]

- However, 3-SAT transformation of any problem is linear

- Important: each clause contains at most 3 literals

  - Terminology: 3-SAT problem = solve a CNF where clauses have at most 3 literals,
  - Terminology: K-SAT problem = solve a CNF where clauses have at most K literals ...

- 3-SAT is as general as SAT, thus NP-complete

- 2-SAT is strictly simpler, proved polynomial (in fact linear !)
David-Putnam Algorithm

History

- More a **general method**, with lots of derived algorithms
- The very first Davis-Putnam is NOT the right one:
  - It’s a space exploration algo (that may explode in memory)
- The “right one ” should be referred as **Davis-Putnam-Logemann-Loveland (DPLL)**:
  - This is where the idea of linear memory cost appears

General structure

Parameterized by 3 functions **Simplify, Tautology, Contradiction** such that:

- \( \text{Sat}(\text{Simplify}(\phi)) \) iff \( \text{Sat}(\phi) \)
- \( \text{Simplify}(\phi) \) is simpler (i.e. smaller)
- \( \text{Tautology}(\phi) \), resp. \( \text{Contradiction}(\phi) \) detect whether \( \phi \) is a trivial tautology, resp. contradictory (i.e. for a neglectable cost)

\[
\text{Sat}(\phi) = \\
\phi := \text{Simplify}(\phi) \\
\text{if } \text{Tautology}(\phi) \text{ returns SAT} \\
\text{if } \text{Contradiction}(\phi) \text{ returns UNSAT} \\
\text{chose ONE literal } x \\
\text{if } \text{Sat}(\phi \land x) \text{ returns SAT} \\
\text{else if } \text{Sat}(\phi \land \neg x) \text{ returns SAT} \\
\text{else returns UNSAT}
\]
Original Simplify procedure

- Based on two principles:
  - Propagation of unit clauses.
  - Elimination of pure literals.

- A clause is unit if it contains a single literal:
  - \( x \) is replaced by 1 and \( \overline{x} \) by 0
  - i.e. clauses containing \( x \) are erased
  - i.e. \( \neg x \) is erased from the other clauses
  - \( \equiv \) constant propagation

- A literal \( l \) is pure if its negation does not appear in any clause
  - we can arbitrary chose to set \( l \) to 1,
  - which leads to simplify the problem (“erase ” clauses containing \( l \))

Note on pure literals

- How it works?
  - If \( x \) is pure, alors \( \phi \equiv (x + \alpha) \beta \), where neither \( \alpha \) nor \( \beta \) are containing \( x \)
    ou \( \overline{x} \)
  - Conclusion: \( \exists x ((x + \alpha) \beta) \equiv (\beta + \alpha \beta) \equiv \beta \)
  - i.e. \( \phi \) has solutions iff it has solutions for \( x = 1 \)

- Problem: what about the (potential) solutions where \( x = 0 \)?
  - it is possible to perform “basic” SAT: answer yes/no
  - but not “extended” SAT: iterate all solutions
  - In practice: pure literal rule is not used (even if rather smart)
“Classical” DP(\(\text{LL}\))

- extended SAT (enumerate solutions) with unit propagation and split
- arguments:
  \(\rightarrow\) the (CNF) formula to solve \(f\)
  \(\rightarrow\) the inherited partial candidate solution (monomial) \(m\)
- Starting call: DPLL\((f, 1)\)

\[
\text{DPLL}\left(f, m\right)
\]

while it exists a unit clause \(l\) in \(f\) do

\[
f := \text{Eliminate}\left(f, l\right); m := m \cdot l
\]

if \(f\) is identically true then \(\text{PrintSolution}\left(m\right); \text{return}\)
else if \(f\) is identically false then \(\text{return}\)
else chose some literal \(x\) in \(f\)

\[
\text{DPLL}\left(f, m \cdot x\right)
\]

\[
\text{DPLL}\left(f, m \cdot \bar{x}\right)
\]

Davis-Putnam Algorithm
\( \overline{a} \)

\( \overline{a} + b + c \)

\( \overline{a} + \overline{b} \)

\( \overline{a} + \overline{c} \)

\( \overline{b} + x \)

\( \overline{b} + y \)

\( b + \overline{x} + \overline{y} \)

\( \overline{c} + \overline{x} \)

\( \overline{c} + \overline{d} \)

\( c + x + d \)

\( \overline{y} + \overline{d} + \overline{z} \)

\( y + d \)

\( d + z \)

Unit : \( a \)

\( \overline{a} \)

\( \overline{a} + b + c \)

\( \overline{a} + \overline{b} \)

\( \overline{a} + \overline{c} \)

\( \overline{b} + x \)

\( \overline{b} + y \)

\( b + \overline{x} + \overline{y} \)

\( \overline{c} + \overline{x} \)

\( \overline{c} + \overline{d} \)

\( c + x + d \)

\( \overline{y} + \overline{d} + \overline{z} \)

\( y + d \)

\( d + z \)

Unit : \( a, y \)

\( \overline{d} \)

\( \overline{d} + \overline{z} \)

\( \overline{d} \)

\( \overline{a} + b + \overline{c} \)

\( \overline{a} + b \)

\( \overline{a} + \overline{c} \)

\( b + \overline{x} \)

\( \overline{b} + y \)

\( b + \overline{x} + \overline{y} \)

\( \overline{c} + \overline{x} \)

\( \overline{c} + \overline{d} \)

\( c + x + d \)

\( \overline{y} + \overline{d} + \overline{z} \)

\( y + d \)

\( d + z \)

Unit : \( a, y, d, \overline{c}, \overline{z}, b, x \)

\( \rightarrow \) Split \( d \) gives \( \overline{z} \) : solution : \( x, y, \overline{z} \)

\( \rightarrow \) Split \( \overline{d} \) gives \( z \) : solution : \( x, y, z \)

\[ \text{Davis-Putnam Algorithm } 76/101 \]

\( \overline{a} \)

\( \overline{a} + b + c \)

\( \overline{a} + \overline{b} \)

\( \overline{a} + \overline{c} \)

\( \overline{b} + x \)

\( \overline{b} + y \)

\( b + \overline{x} + \overline{y} \)

\( \overline{c} + \overline{x} \)

\( \overline{c} + \overline{d} \)

\( c + x + d \)

\( \overline{y} + \overline{d} + \overline{z} \)

\( y + d \)

\( d + z \)

Unit : \( a \)

\( \overline{a} \)

\( \overline{a} + b + c \)

\( \overline{a} + \overline{b} \)

\( \overline{a} + \overline{c} \)

\( \overline{b} + x \)

\( \overline{b} + y \)

\( b + \overline{x} + \overline{y} \)

\( \overline{c} + \overline{x} \)

\( \overline{c} + \overline{d} \)

\( c + x + d \)

\( \overline{y} + \overline{d} + \overline{z} \)

\( y + d \)

\( d + z \)

Unit : \( a, \overline{y} \)

\( \overline{b} \), \( c \)

\( \text{Contradiction} \)

\( \text{NO solution} \)

\[ \text{Davis-Putnam Algorithm } 77/101 \]
Implementation elements

- pivot (branching literal) choice very important (heuristics).
- Data structures as “light” as possible.
- Idem for the control structure (“stack-free”).

Recursive learning

Principles: range and contradictions

- **State of the algo** during the execution:
  - units with range 0 \((L_0)\) = initial units and their consequences,
  - units with range 1 \((L_1)\) = 1st pivot \(p_1\) and its consequences,
  - etc.

- If a **contradiction occurs at range** \(n\) (pivot \(p_n\)), then:
  - it exists at least 2 clauses \(x + a + b + c + ...\) and \(\bar{x} + \alpha + \beta + \gamma + ...\)
  - with \(\bar{x}\) and \(x\) are of range \(n\) (contradiction)
  - and \(\bar{a}, \bar{b}, \bar{\alpha}, \bar{\beta}, ...\) of some range \(k \leq n\)

- **Property**: let \(k\) be the greatest range (different from \(n\)):
  - choices (pivots) made between ranges \(k\) and \(n\) have NO influence on the contradiction
  - i.e. same contradiction would have occur if \(p_n\) have been chosen just after range \(k\)
  - i.e. \(\bigwedge_{i=1}^{k} p_i \Rightarrow \bar{p}_n\)
Example

$\bar{z}, \bar{v}$ et $\bar{w}$ are literals of range max $k < n$

\[
\begin{align*}
  x + y + z & \quad x + y + z & \quad x + y + z \\
  \bar{x} + t + v & \quad \bar{x} + t + v & \quad \bar{x} + t + v \\
  y + \bar{t} + w & \quad y + \bar{t} + w & \quad y + \bar{t} + w \\
  \bar{y} \text{ unit of range } n & \quad \bar{t} \text{ unit} & \quad \text{contradiction}
\end{align*}
\]

Conclusion

- If the choice $p_n$ has been made just after range $k$, the same contradiction would have occur
- thus: $\bigwedge_{i=1}^{k} p_i \Rightarrow \neg p_n$
- Particularly interesting when $k < n - 1$
- $\Rightarrow$ recursive learning

Recursive learning

- How to exploit contradictions sources
- If we found that $\bigwedge_{i=1}^{k} p_i \Rightarrow \neg p_n$, we can:
  - $\leftrightarrow$ immediately back-track to level $k$ and add unit $\neg p_n$ to $P_k$ (not so smart);
  - $\leftrightarrow$ continue normally with the extra info that $\neg p_n$ must be considered as unit as long as the level is greater than $k$.

Conclusion on (basic) SAT-solver

- Cost (potentially) exponential in time, but polynomial in space
- Lots of efficient (relative!) implementations
- Important extension: SAT Modulo Theory
SAT modulo theory

Principles

• Most of (modern) solvers ARE SMT solvers
• Extension of Boolean SAT Solver
• First order logic + decidable embedded theory (e.g. linear algebra)
• Data: a first-order (i.e. Boolean) formula, where variables are sentences in the host theory
• How it works:
  ↦ a classical SAT solver enumerate the Boolean solutions (conjunction of host formula)
  ↦ the host solver checks the satisfiability of the Boolean solution in the host theory

Example: SMT with Linear Algebra theory

• First order formula (in CNF): \( \phi = (a \cdot b \cdot c \cdot (d + e)) \)
• Where: \( a = (x \geq y - 1), \ b = (x + y \leq 1), \ c = (y \geq 0), \ d = (x \leq -2), \ e = (x \geq 2) \)
• 1st (Boolean) solution found: \( a \cdot b \cdot c \cdot d \)
  ↦ Corresponding Host Theory formula is:
  \( \psi_1 = (x \geq y - 1) \land (x + y \leq 1) \land (y \geq 0) \land (x \leq -2) \)
  ↦ Ask the host (Linear Algebra) solver for the satisfiability of \( \psi_1 \):
    answer UNSAT, continue Boolean SAT solving ...
• 2nd (Boolean) solution found: \( a \cdot b \cdot c \cdot e \)
  ↦ Corresponding Host Theory formula is:
  \( \psi_2 = (x \geq y - 1) \land (x + y \leq 1) \land (y \geq 0) \land (x \geq 2) \)
  ↦ Ask the host (Linear Algebra) solver for the satisfiability of \( \psi_2 \):
    answer UNSAT, continue Boolean SAT solving ...
• No more Boolean solution, the SMT problem is UNSATISFIABLE
5. Sat solver based methods

Sat solvers ............................................................................................................... 85
Sat solver vs state machines .............................................................. 88

Sat solvers

What is a sat solver ?
- deals with first order formulas
- answer wether a (Boolean) formula $f(x_1, \ldots, x_n)$ is:
  - unsatisfiable (i.e. it is a false assertion)
  - satisfiable, with, in general, one solution of the formula
  - alternatively, a sat solver is also able to enumerate all the solution

Examples
- for $(x \cdot y + (y \oplus z))$, answer "sat", with, e.g. $x = 0, y = 1, z = 0$,
  (or $x = 0, y = 0, z = 1$, or $x = 1, y = 1, z = 1$ etc)
- for $(x = y). (\neg y \cdot z \cdot x)$ answer "unsat"
Sat solver and tautologies

- can be used to check tautologies:
  \[ \text{if } f \text{ is unsat, then } \neg f \text{ is sat for any valuations of the variables} \]
  \[ \text{i.e. } \neg (\exists x \neg f(x)) \iff \forall x f(x) \]
- example: \( \neg (x \Rightarrow (y \Rightarrow x)) \) is unsat, thus \( (x \Rightarrow (y \Rightarrow x)) \) is a tautology

Theoretical facts

The (Boolean) satisfiability problem is:

- decidable, thus complete decision algorithm exist,
- untractable (it is the NP-complete reference problem)

Note SMT Solvers

- Most of the (modern) existing tools do more than Boolean decision.
- They integrate extra "knowledge" on other domains, like linear arithmetics, ordered sets, etc.
- They are called Sat Modulo Theory Solvers (SMT-solver).

Depending on the integrated theory, the SMT problem:

- decidable, e.g. Boolean + Presburger arithmetics,
- or just semi-decidable (full arithmetics) the tool may answer sat, unsat, or inconclusive.
Sat solver vs state machines

Reminder: a verification program is...

- a set of (free) variables $v$, a set of state variables $s$
- a set of initial state characterized by $\text{Init}(s)$
- a transition function characterized by $s' = \text{Post}_H(s)$
- a (state) property $\psi(s) = (\forall h(s, v) \Rightarrow \phi(s, v))$

Shortcuts

- Transition relation:
  \[ T(s', s) \overset{\text{def}}{=} \exists v s' \xrightarrow{v} s \land H(s', v) \]
- Reachable states:
  \[ A_0(s) = \text{Init}(s) \]
  \[ A_{n+1}(s) = \exists s_n A_n(s_n) \land T(s_n, s) \]
  \[ \text{i.e. } A_n(s) \text{ are the states reachable in } n \text{ steps} \]
- Property successors:
  \[ \psi^{-1}(s) \overset{\text{def}}{=} \exists s' \psi(s') \land T(s', s) \]
  \[ \psi^{-n-1}(s) \overset{\text{def}}{=} \exists s' \psi^{-n}(s') \land T(s', s) \]
  \[ \text{i.e. } \psi^{-n}(s) \text{ are the states reachable by a path of length } n \text{ from a state satisfying } \psi \]
A trivial case ...

- a sat solver knows nothing about automata and states, however:
  \[ \leftrightarrow \text{if it appears that } \psi(s) \text{ is a tautology, then the property is checked!} \]
  \[ \leftrightarrow \text{i.e. it does not depend on states (lucky case)} \]

A less trivial case ...

- if property holds for all initial states
  \[ \text{i.e. } A_0(s) \implies \psi(s) \text{ is a tautology} \]
- and moreover \( \psi^{-1}(s) \implies \psi(s) \)
- then, by induction, \( \psi \) holds for any state
- the property is 1-inductive
- otherwise: inconclusive, try 2-induction, 3-induction etc?

N-induction principle

- If the property holds for any \( n \)-reachable states: \( A_i(s) \implies \psi(s) \) is a tautology for any \( i = 1 \cdots n \)
- and if \( \psi^{-1}(s) \land \psi^{-2}(s) \land \cdots \land \psi^{-n}(s) \implies \psi(s) \),
- then, by induction, \( \psi \) holds for any state

Completeness of the method

- any safety property that holds for a finite automaton is \( k \)-inductive for some \( k \)
- this \( k \) is bounded by the diameter of the automaton
Complexity of the method

- the size of formulas (and variables) grows linearly with the induction degree $n$...
- ... but sat-solving cost grows exponentially with the number of variables!
- in practice, the method is limited to 1 or 2 induction
- alternative:
  - check the n-basis $(\bigwedge_{i=0}^{n} A_i(s) \Rightarrow \psi(s))$ ...
  - ... but not the induction rule
  - more tractable in practice (may work for a few hundreds of step)
  - but indeed not complete: not a proof, rather a super-test
  - often call bounded model checking

Sat solver vs state machines

6. Appendix

Example/exercice: arithmetic circuit
## Example/exercice: arithmetic circuit

### Serial adder:
- **Inputs** $x, y$
- **Outputs** $s$ (um), $c$ (arry)

### Shift:
- $m$ encodes $2 \times x$

<table>
<thead>
<tr>
<th>Time (most significant bits)</th>
<th>$c$</th>
<th>$x$</th>
<th>$y$</th>
<th>$s$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Property: if always $x = y$ then always $s = m$

### Serial adder, questions ...

- Give the (implicit) automaton of the system
- Explore the system with the enumerative method (by “hand”) (and prove that “always($x=y$) $\Rightarrow$ always($s = m$)”)
**Boolean model**

- 2 inputs $V = \{x, y\}$
- 2 memories $S = \{c, m\}$ with
  
  \[
  c_{\text{init}} = 0, \quad g_c = c.(x + y) + \bar{c}.x.y \\
  m_{\text{init}} = 0, \quad g_m = x
  \]
- $H \equiv (x = y)$, and $\phi \equiv (m = s)$, where $s = (c \oplus x \oplus y)$

**Enumerative exploration (the “tabular method”)**

Note: we have “pre-computed” that $x = y$ are the only possible inputs

<table>
<thead>
<tr>
<th>Starting state</th>
<th>Inputs</th>
<th>Output/Prop</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$m$</td>
<td>$x$</td>
<td>$y$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example/exercice: arithmetic circuit

---

**Serial adder, questions (cntd)**

- Explore the system with the symbolic method
  
  n.b. hardly feasible by hand, need an helper: bddc
- Use bddc (basic BDD calculator)
  - How it works: reads formula, build (and echo if possible) the corresponding BDD
    
    $\leftarrow x \; \text{or} \; (y \; \text{xor} \; z) \; ; \; \text{outputs: } x + y.z + \bar{y}.z$
    
    $\leftarrow x \; \Rightarrow \; (y \; \Rightarrow \; x) \; ; \; \text{outputs: } 1 \; \text{(canonical form)}$
- Assign a formula to a “variable”
  
    $\leftarrow s := c \; \text{xor} \; x \; \text{xor} \; y;$
- Define a function over formulae
  
    $\leftarrow \text{Implique}(X, Y) := \text{not } X \; \text{or} \; Y;$
- Usefull commands: help and syntax
- ... quick demo.

Example/exercice: arithmetic circuit
Boolean model (reminder)

• 2 inputs $V = \{x, y\}$
• 2 memories $S = \{c, m\}$ with
  - $c_{\text{init}} = 0$, $g_c = c.(x + y) + \overline{c}.x.y$
  - $m_{\text{init}} = 0$, $g_m = x$
• $H \equiv (x = y)$, and $\phi \equiv (m = s)$, where $s = (c \oplus x \oplus y)$
• Error states: $\text{Err} \equiv (\exists x, y \ H \land \neg \Phi) \equiv (c \oplus m)$

In bddc syntax ...

$G_m := x$;
$G_c := \text{if } c \text{ then } (x \text{ or } y) \text{ else } (x \text{ and } y)$;
$s := x \text{ xor } y \text{ xor } c$;
$\text{Init} := \text{not } c \text{ and not } m$;
$H := (x = y)$;
$\Phi := (m = s)$;
$\text{Err} := \exists x, y \ (H \land \neg \Phi)$;
$\text{Acc0} := \text{Init}$;

Example/exercice: arithmetic circuit

Step 0
- Check that $\text{Acc}_0 = \text{Init} \cap \text{Err} = \emptyset$

$\text{Acc0 and Err}$;
gives 0, ok, continue and compute $\text{Acc}_1$

Step 1
- $\text{Acc}_1 = \text{Acc}_0 \cup \text{post}_H(\text{Acc}_0)$
- Recall the definition of $\text{Post}_H$ (slide 50)

$\text{Post}(A) := \exists x, y, m, c (A \land H \land (xm = G_m) \land (xc = G_c))$;
- Computes:

$\text{Post}(\text{Acc0})$;
gives: $xm.xc + \overline{xm}.-xc$, i.e. $xm = xc$
- Warning, technical problem: we need a formula on $c$ and $m$ (not $xc$ and $xm$)
Step 1 (cntd)

- Trick, use a “rename” function:
  \[
  Rnm(a, b, F) := \text{exist } a \ (F \text{ and } (a = b));
  \]

- The “right” definition of Post:
  \[
  Postbis(X) := Rnm(xc, c, Rnm(xm, m, Post(X)))
  \]

- Check that:
  \[
  Postbis(Acc0);
  \]
  gives: \(m \cdot c + -m \cdot -c\), i.e. \(m = c\)

- Compute:
  \[
  Acc1 := Acc0 \text{ or } Postbis(Acc0);
  \]
  Are \(Acc0\) and \(Acc1\) the same?

- Check:
  \[
  \text{compare}(Acc1, Acc0);
  \]
  answers 0 (not the same), fixpoint not reached...

- Check:
  \[
  Acc1 \text{ and } Err;
  \]
  gives empty, no error yet ...

Example/exercice: arithmetic circuit 100/101

Step 2

- Compute \(Acc2\):
  \[
  Acc2 := Acc1 \text{ or } Postbis(Acc1);
  \]

- Check:
  \[
  \text{compare}(Acc2, Acc1);
  \]
  answers 1 (same), fixpoint is reached!

Property satisfied,
we have proven formally that
\[
\forall x, y \in \mathbb{Z} \ (x = y) \Rightarrow (x + y = 2x)
\]