Verification of Reactive Programs

Pascal Raymond

Verimag-CNRS

MOSIG - Embedded Systems

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

1. Reactive systems and state machines

Implicit state machines . . . 7
Conservative Abstraction . . . 10
Expressing properties . . . 14
Proving properties . . . 18
Enumerative (forward) algorithm . . . 21

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

```
node b(sec,bea: bool) returns (ontime,late,early: bool);
var diff: int;
let
  diff = 0 -> pre ( diff +
      (if bea then 1 else 0) + (if sec then -1 else 0));
  early = false -> pre (
      (ontime and diff > 3) or (early and diff > 1));
  late = false -> pre(
      (ontime and diff < -3) or (late and diff < -1));
  ontime = not (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

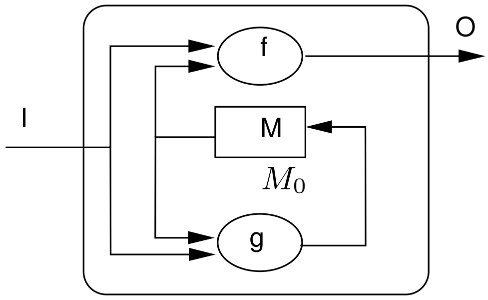
The 3 first ones are obviously safety, while the one is a typical liveness: it refers to unbounded future

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
$\exists M_0, M_1, M_2 \cdots$ s.t. $\forall t \; O_t = f(I_t, M_t)$ and $M_{t+1} = g(I_t, M_t)$

Common model for synchronous programs

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

Implicit vs explicit

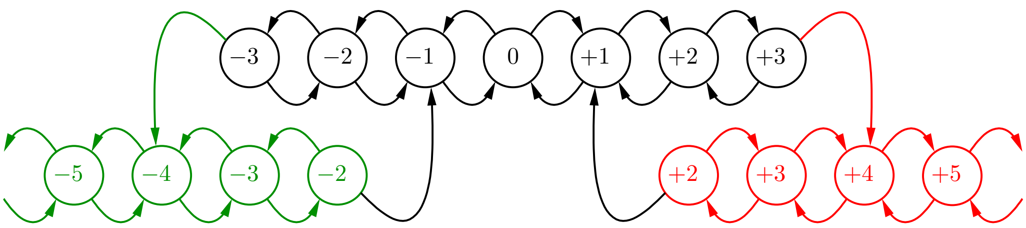
An ISM is equivalent to an explicit state/transition system:

- 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 = $\{$sec, bea$\}$ O = $\{$late, ontime, early$\}$
- A memory for each "`-> pre`" expression,
  (e.g. Plate for "`false -> pre late`"):
  M = $\{$Plate, Pontime, Pearly, Pdiff$\}$
  with M_0 = (*false*, *true*, *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,

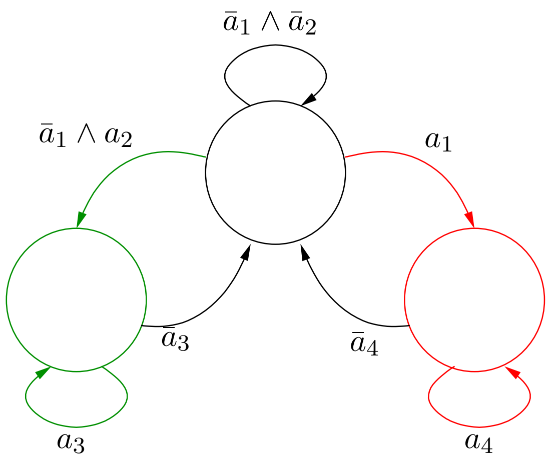
it is impossible to explore it

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 `diff > 3`
- a_2 for `diff < -3`
- a_3 for `diff < -1`
- a_4 for `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 !

$\Rightarrow$ 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

$\Rightarrow$ safeties are preserved or lost, but never introduced

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

- the verification succeeds $\Rightarrow$ property satisfied
- the verification fails $\Rightarrow$ inconclusive
  (it may be a *false negative*)

Expressing properties

Liveness requires complex formalisms (temporal logics)

Safety can be programmed ⇒ 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 = not (late and early);
```
- It’s impossible to directly pass from late to early:
```
ok = true -> not (early and pre late);
```
- It’s impossible to remain late only one instant:
```
Plate = false -> pre late;
PPlate = false -> pre Plate;
ok = not (not late and Plate and not 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 **ok = ontime**, assumption can be:

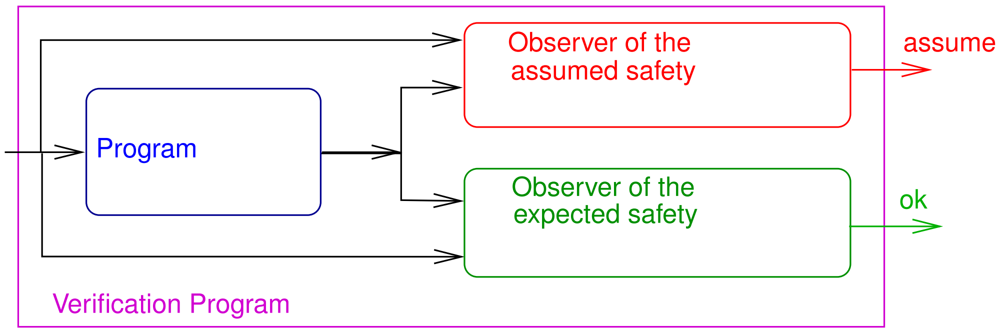
- naive: **assume = (sec = bea);**
- more sophisticated, bea and sec alternate:

```
SF = switch(sec and not bea, bea and not sec);
BF = switch(bea and not sec, sec and not bea);
assume = (SF => not sec) and (BF => not bea);
```

with:

```
node switch(on, off : bool) returns (s: bool);
let s = false -> pre(if s then not off else on); tel
```


General scheme



- We suppose provided such a verification program
- Goal: if assume remains indefinitely true, then ok remains indefinitely true:

  (always $\mathtt{assume}$*)* $\Rightarrow$ *(always* $\mathtt{ok}$*)*

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

  always ((once not $\mathtt{assume}$*) or* $\mathtt{ok}$*)*

  (the problem will be explained later)

Proving properties

Abstracted verification program

Special case of Boolean synchronous program with 2 ”outputs”

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

(N.B. we identify predicates and sets)

Associated explicit automaton

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

We use ”pre” and ”post” functions:

- for $q \in Q$, $\text{post}_H(q) = \{q' / \exists v \;\; q \stackrel{v}{\longrightarrow} q' \;\wedge\; H(q, v)\}$
- for $X \subseteq Q$, $\text{Post}_H(X) = \bigcup_{q \in X} \text{post}_H(q)$
- for $q \in Q$, $\text{pre}_H(q) = \{q' / \exists v \;\; q' \stackrel{v}{\longrightarrow} q \;\wedge\; H(q', v)\}$
- 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: $\text{Err} = \{q/\exists v \quad H(q,v) \wedge \neg\phi(q,v)\}$
- 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 Acc and Bad:

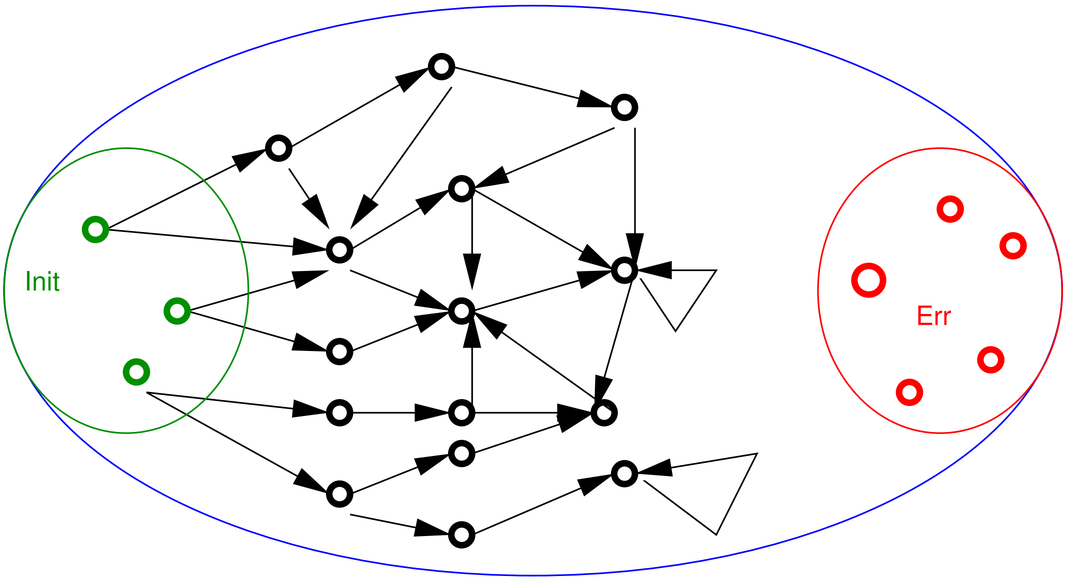
- prove that $\text{Acc} \cap \text{Bad}_0 = \emptyset$ (forward method)
- prove that $\text{Bad} \cap \text{Acc}_0 = \emptyset$ (backward method)

Remark: methods are non symmetric because of determinism

Enumerative (forward) algorithm

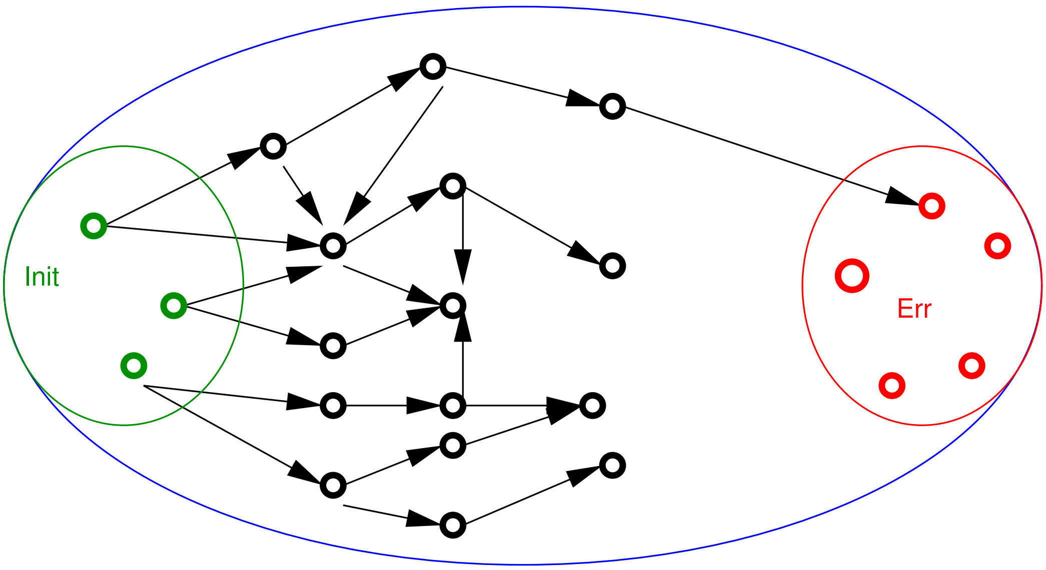
```
CurAcc := Init
Done := empty
while it exits q in CurAcc - Done do {
      (* q ∈ CurAcc \ Done *)
      for all q' in post_H(q) do {
          if q' in Bad0 then EXIT(failed)
          put q' in CurAcc
      }
      put q in Done
}
(* we have CurAcc = Done = Acc *)
EXIT(succeed)
```

Example of success (breadth first)



success

Example of failure (breadth first)



Failure

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:
  $|Acc|$ times the cost of post$_H$(q), with $|Acc| \sim 2^{|S|}$
- backward is even worse: pre$_H$(q) is more complex than post$_H$(q)
  (enumerative backward is never used in practice)

Big problem: computing 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

$\Rightarrow$ Digression on efficient decision techniques

2. Decision techniques (BDD)

Binary Decision Diagrams 28
Operations on BDDs 34
Signed BDD 39

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 $f \in \mathbf{B}^n \to \mathbf{B}$:
  - $\hookrightarrow$ $f(x, y, ..., z) = x.f(1, y, ..., z) + \bar{x}.f(0, y, ..., z)$
- Let's define f_x and $f_{\bar{x}}$ in $\mathbf{B}^{n-1} \to \mathbf{B}$ by:
  - $\hookrightarrow$ $f_x(y, ..., z) = f(1, y, ..., z)$
  - $\hookrightarrow$ $f_{\bar{x}}(y, ..., z) = f(0, y, ..., z)$
- 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 *?*

$f_x = y + (y \oplus z)$

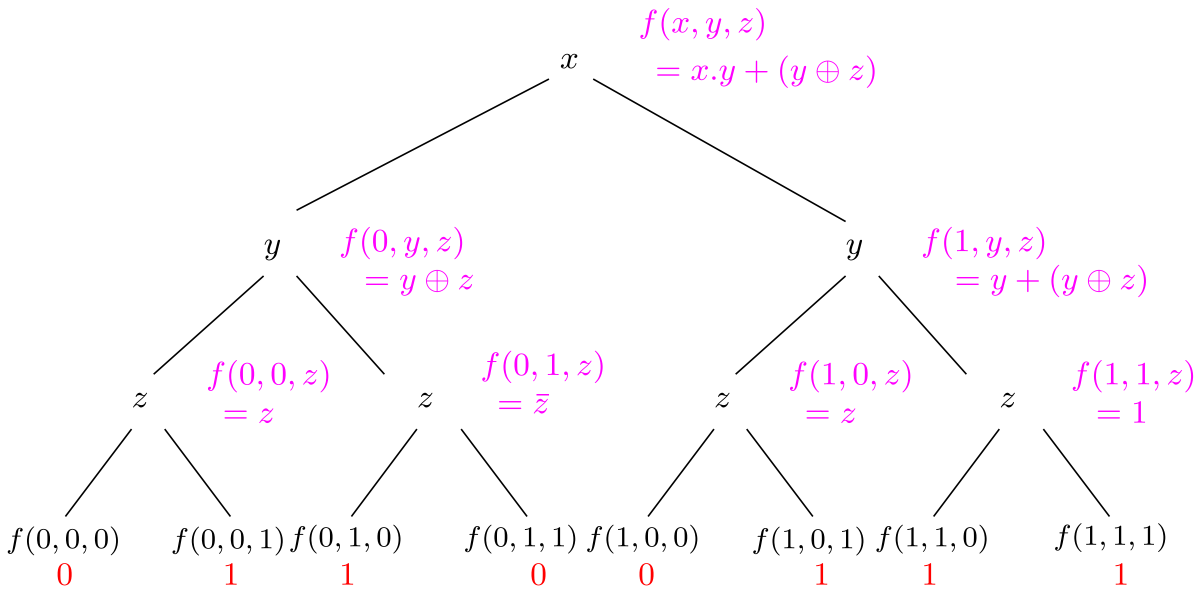
$f_{\bar{y}} = z$

$f_z = x.y + \bar{y} = x + \bar{y}$

Shannon tree

- When applying recursively the S.D. on all variables, one obtains:
  - $\hookrightarrow$ 1 (the always-true function) or
  - $\hookrightarrow$ 0 (the always-false function)
- Example, for $f = x.y + (y \oplus z)$:
  - $\hookrightarrow$ $f_{\bar{x}} = f(0, y, z) = y \oplus z$
  - $\hookrightarrow$ $f_{\bar{x}y} = f(0, 1, z) = \neg z$
  - $\hookrightarrow$ $f_{\bar{x}yz} = f(0, 1, 1) = 0$
- Shannon tree: graphical representation of all the 2^n steps

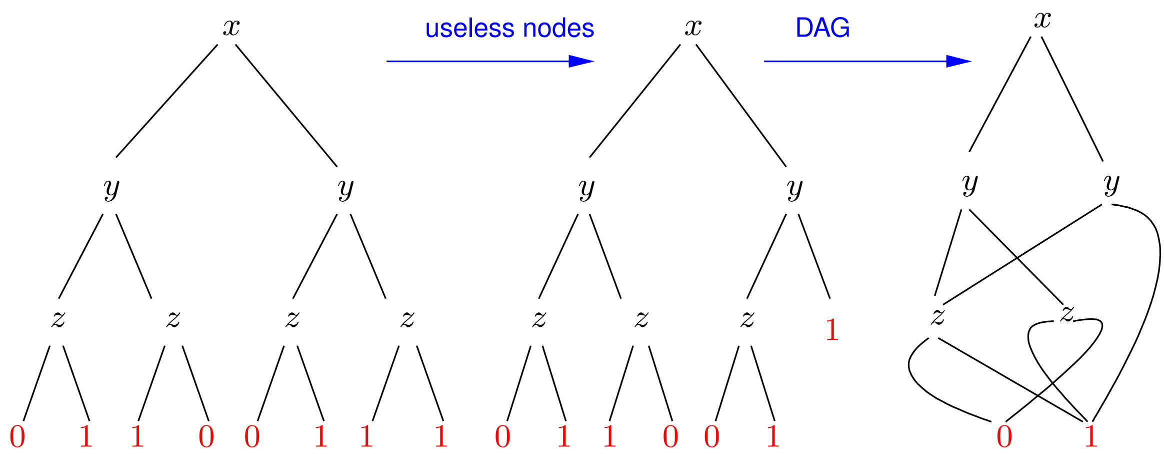
Full decomposition example



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)

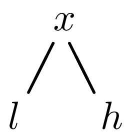


N.B. For a given variable ordering the BDD is *unique*

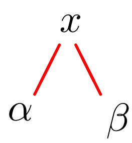
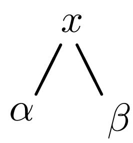
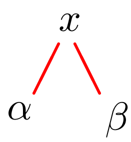
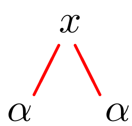
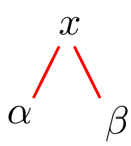
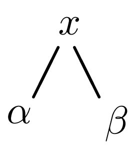
Formal definition

Definition

- *Let* V *be a set of variable, totally ordered by* $\preceq$
- *Let* $V^\star = V \cup \{\infty\}$ *extented with a max value* ($\forall x \in V^\star\ x \preceq \infty$)
- *BDDs are defined, together with their "range" rg* : $BDD \to V^\star$
  - $\hookrightarrow$ 1 *is a BDD with rg*$(1) = \infty$
  - $\hookrightarrow$ 0 *is a BDD with rg*$(0) = \infty$
  - $\hookrightarrow$ *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*

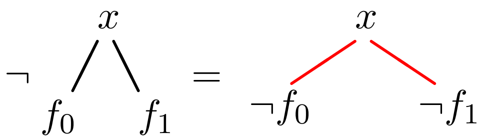
We note  such a triplet

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 $\mathcal{N}(x, \alpha, \beta)$
  - $\hookrightarrow$ we note  for $\mathcal{N}(x, \alpha, \beta)$, and  for a built BDD
-  = ERROR if *rg*$(\alpha) \prec x$ or *rg*$(\beta) \prec x$
-  $= \alpha$
-  =  otherwise

Operations on BDDs

Negation

- $\neg 1 = 0$
- $\neg 0 = 1$
- 

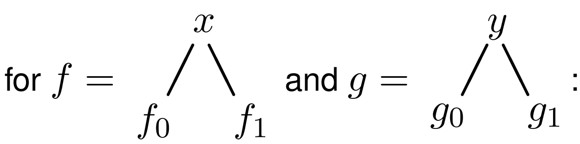
Binary operators

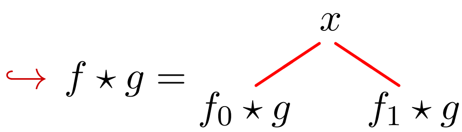
Property: any usual operator $\star$ (in $+$, $\cdot$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$), distribute on Shannon decomposition:

$$(x \cdot f_x + \bar{x} \cdot f_{\bar{x}}) \star (x \cdot g_x + \bar{x} \cdot g_{\bar{x}}) = x \cdot (f_x \star g_x) + \bar{x} \cdot (f_{\bar{x}} \star g_{\bar{x}})$$

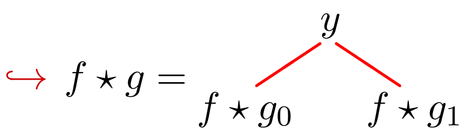
Binary operators (ctd)

- As a consequence, recursive rules are,

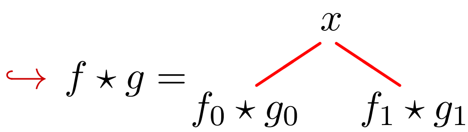




if $x \prec y$ (balance)



if $y \prec x$ (balance)



if $x = y$

Binary operators (ctd)

- Terminal rules apply in priority, for instance:

↪ $(1 + \alpha) = (\alpha + 1) = 1$

$(0 + \alpha) = (\alpha + 0) = \alpha$

↪ $(1 \cdot \alpha) = (\alpha \cdot 1) = \alpha$

$(0 \cdot \alpha) = (\alpha \cdot 0) = 0$

↪ $\alpha \oplus \alpha = 0$

$(0 \oplus \alpha) = (\alpha \oplus 0) = \alpha$

$(1 \oplus \alpha) = (\alpha \oplus 1) = \neg\alpha$

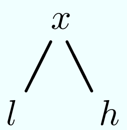
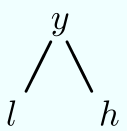
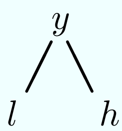
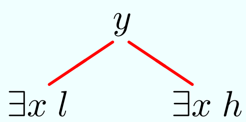
> ***Exercise***
>
> *Terminal rules for "⇒" (implication) ?*
>
> $(0 \Rightarrow \alpha) = (\alpha \Rightarrow 1) = 1$
>
> $(\alpha \Rightarrow 0) = \neg\,\alpha$
>
> $(1 \Rightarrow \alpha) = \alpha$

Quantification

- Boolean quantification is simple

↪ like for any finite domain

↪ unlike infinite domains (e.g. integers) !

> ***Exercise***
>
> *Definition of "$\exists x \;\; \alpha$" ?*
>
> *based on the enumeration of values:* $\exists x \;\; \alpha(x, \vec{w}) = \alpha(0, \vec{w}) \vee \; \alpha(1, \vec{w})$
>
> $\exists x\, 1 \; = \; 1 \qquad \exists x\, 0 \; = \; 0$
>
> $\exists x$  $= h \vee l$
>
>
>
> $\exists x$  $=$ *if* $x \prec y$ *then*  *else* 
>
>
>
> *Same question for "$\forall v \;\; \alpha$" ?*

Notes on complexity

- Cost of $\neg\alpha$: is linear w.r.t to *size*(α)
- Cost of $\alpha \star \beta$: is in "*size*(α) $\times$ *size*(β)
- Algebraic formula to BDD: exponential (worst case)
- Variable ordering is very important:
  $(x_1 \oplus x_2) \cdot (x_3 \oplus x_4) \cdot \cdots \cdot (x_{2n-1} \oplus x_{2n})$
  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

$\Rightarrow$ 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)

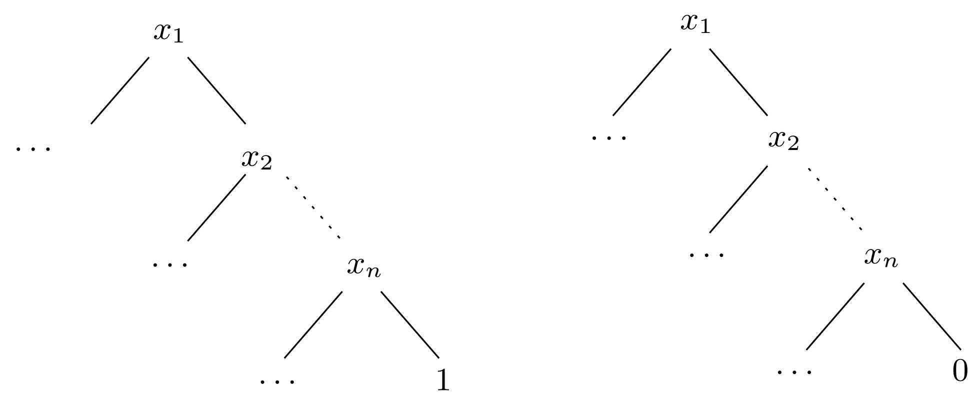
Sharing structure

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

Positive functions

Definition

$f \in \mathbf{B}^n \to \mathbf{B}$ *is positive iff* $f(1, 1, \cdots, 1) = 1$



Positive function   Negative function

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

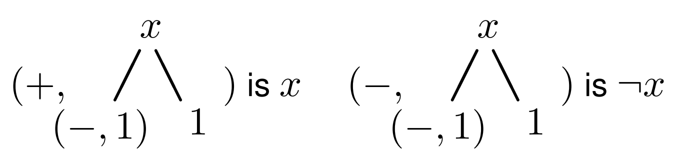
SBDD

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*

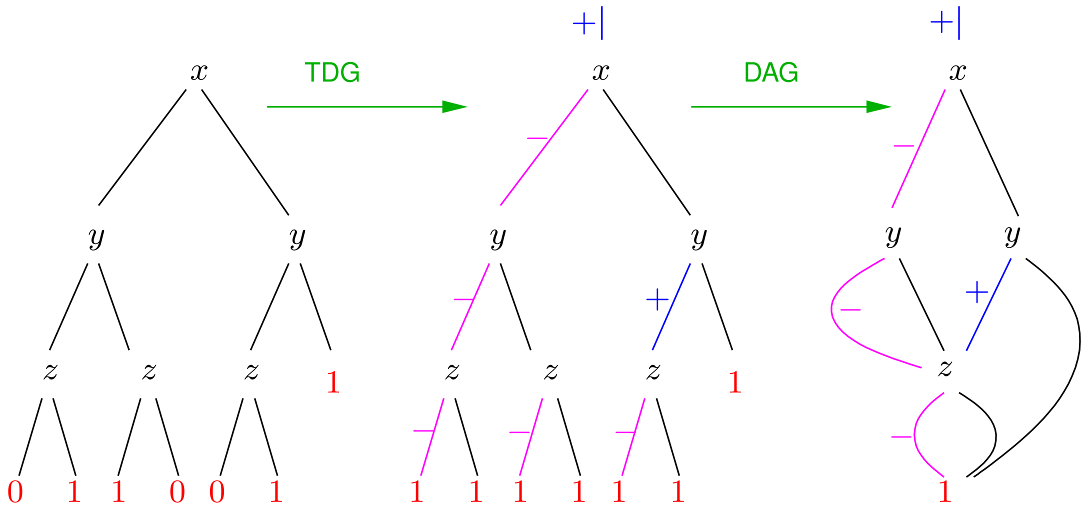
Examples:

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



Full SBDD example

$x \cdot y + (y \oplus z)$



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 $\Rightarrow$ 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 $x + y \cdot \neg t$:
  - $\hookrightarrow$ 10 concrete states
  - $\hookrightarrow$ 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: $\text{Post}_H : 2^Q \to 2^Q$
  $\text{Post}_H(X) = \{q' / \exists q \in X, v \in 2^V \ \ H(q,v) \wedge q \xrightarrow{v} q'\}$
  (implementation is presented later)

Algorithm

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

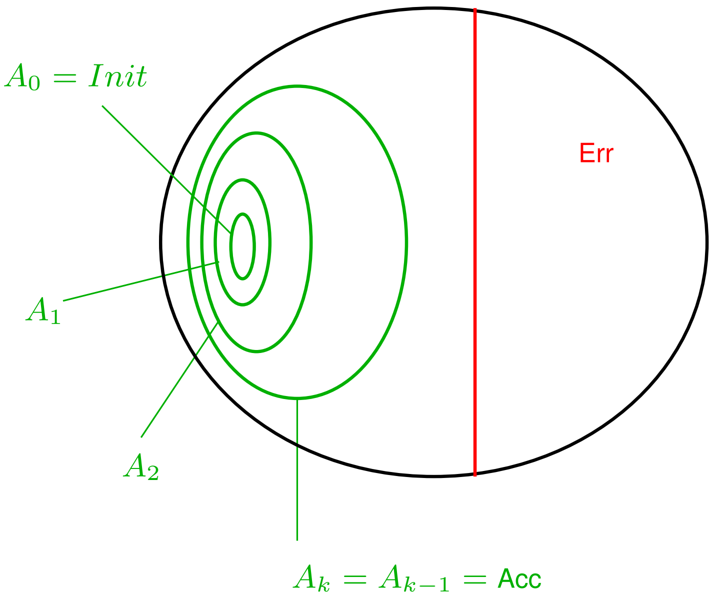
- Initially: $A := \text{Init}$
- Repeat:

↪ if $A \wedge \text{Err} \neq 0$ then EXIT(failed)

↪ else let $A' := A \vee \text{Post}_H(A)$
  if $A' = A$ then EXIT(succeed)
  else $A := A'$, and continue

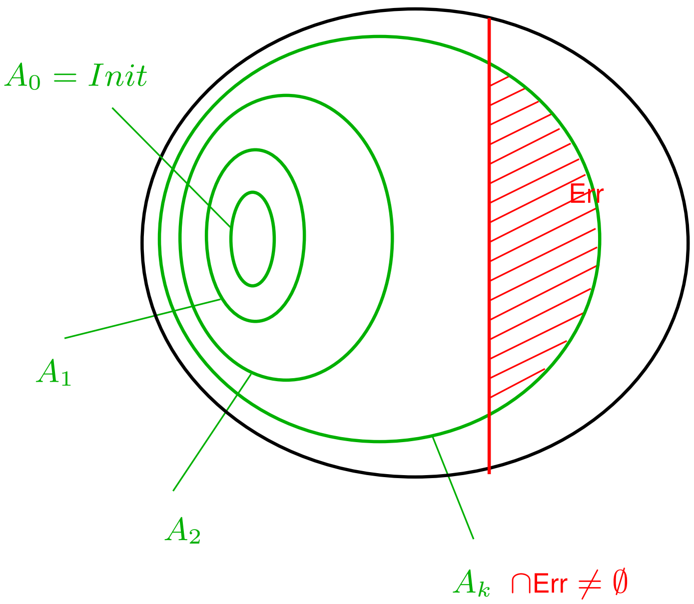
When the proof succeeds, we have $A = A' = \text{Acc}$

Execution



Proof succeeds

Execution (cntd)



Proof fails

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')

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

$\rightarrow$ 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 $X(l) \wedge H(l)$
  (Remark: $Y \neq 0$, otherwise it's trivial Post$_H(0) = 0$)
- $Img[g_1 ... g_n](Y)$ the expected formula over s', defined by:

$$Img[g_1 ... g_n](Y) = \exists l \; Y(l) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(l)$$

Let us study the Shannon decomposition of this formula ...

Decomposition on s'_1:

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

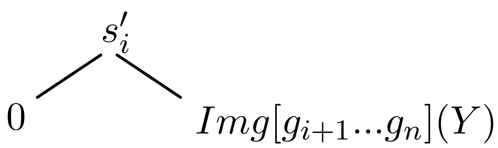
We consider 3 cases:

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

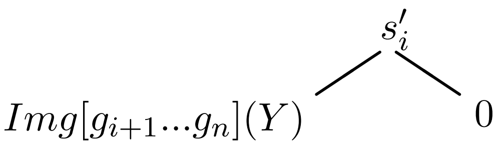
Conclusion: recursive definition of Img, where s'_i variables are never merged with the other

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

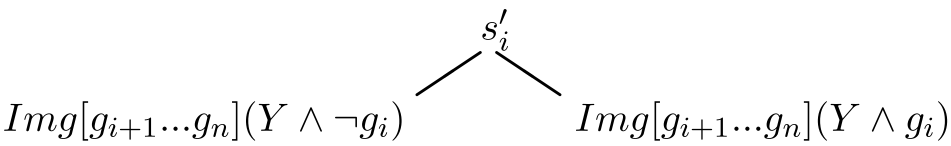
if $Y \Rightarrow g_i$ then



if $Y \Rightarrow \neg g_i$ then



else

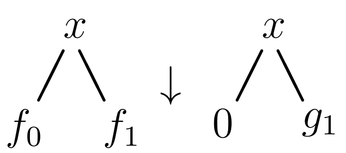


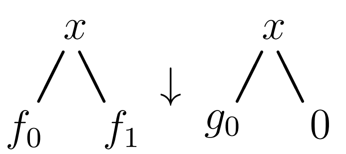
Optimization of image computing

- How to define a ”Knowing that” operator ?
- intuitively, $h = f$ knowing that g must be
  - ↪ equivalent to f if g is true ($f.g \Rightarrow h \Rightarrow f + \bar{g}$)
  - ↪ such that $h = 1$ if $g \Rightarrow f$
  - ↪ such that $h = 0$ if $g \Rightarrow \neg f$
  - ↪ as *simple* as possible otherwise
- Remarks:
  - ↪ Depend on a particular representation (not strictly logical)
  - ↪ There are many of such operators
  - ↪ Some of them have *interesting* extra properties

Constrain operator

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

- $f \downarrow 1 = f$
- $0 \downarrow g = 0$
- $1 \downarrow g = 1$
- 

$= f_1 \downarrow g_1$
- 

$= f_0 \downarrow g_0$
- 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 $\wedge$ under $\exists$ quantifier:
    $\exists x \;\; (f \wedge g)(x) \equiv \exists x \;\; (f \downarrow g)(x)$
  - ↪ in particular:

$$\exists l \quad Y(l) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(l) \quad \equiv \quad \exists l \quad \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(l))$$

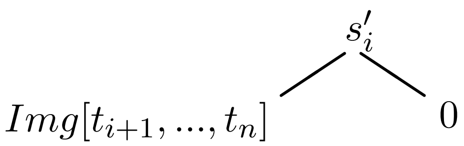
- Constrain and image computing:
  - ↪ $Img[g_1 ... g_n](Y) = Img[(g_1 \downarrow Y)...(g_n \downarrow Y)](1)$
    $\Rightarrow$ second argument useless, only compute universal images

Optimized image computing

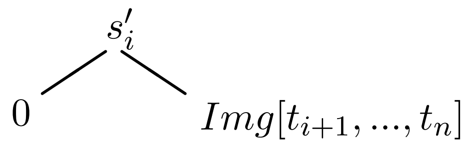
- Compute all $t_i = g_i \downarrow (X \downarrow H)$
- Then $Img[t_1, ..., t_n]$ with:

$Img[] = 1$

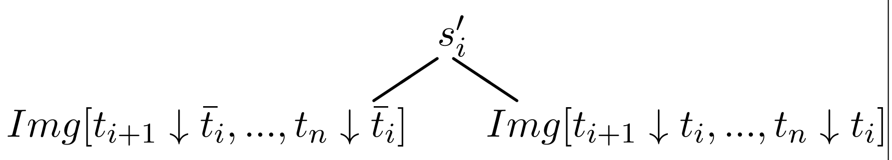
$Img[0, t_{i+1}, ..., t_n] =$



$Img[1, t_{i+1}, ..., t_n] =$



$Img[t_i, t_{i+1}, ..., t_n] =$



Backward symbolic algorithm

How it works

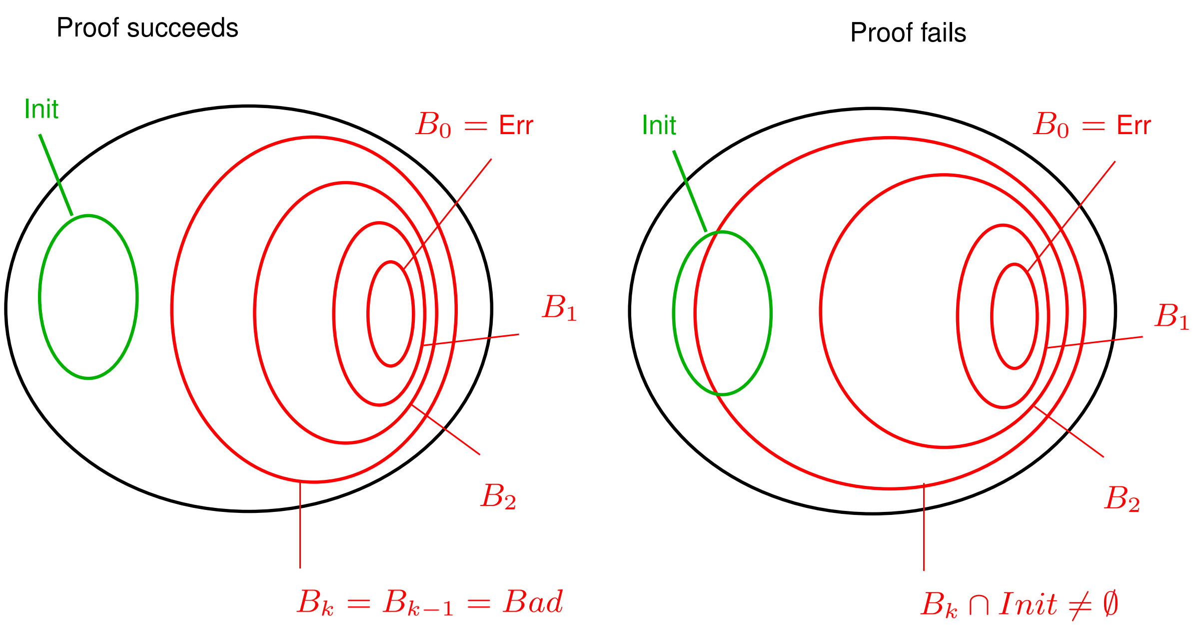
- Very similar to forward
- Uses reverse image computing $\text{Pre}_H : 2^Q \rightarrow 2^Q$
  $\text{Pre}_H(X) = \{q \;/\; \exists\, q' \in X, v \in 2^V \;\; H(q,v) \wedge q \xrightarrow{v} q'\}$
- Uses B = *states leading to Err in less than n transitions*
- Initially: $B :=$ Err
- Repeat:

↪ if $B \wedge$ Init $\neq 0$ then EXIT(failed)

↪ else let $B' := B \vee \text{Pre}_H(B)$
  if $B' = B$ then EXIT(succeed)
  else $B := B'$, and continue

When the proof succeeds, we have $B = B' =$ Bad

Backward symbolic



Implementation of Pre$_H$(X)

No need to merge s_i and s'_i in BDD

Similar to function composition

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

with:

- $\textit{Revim}[0] = 0$
- $\textit{Revim}[1] = 1$
- $\textit{Revim}\left[\begin{array}{c} s'_i \\ \diagup \;\; \diagdown \\ X_0 \quad\; X_1 \end{array}\right] = g_i(s,v) \cdot \textit{Revim}[X_1] + \neg g_i(s,v) \cdot \textit{Revim}[X_0]$

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 63
CNF transformation 66
Davis-Putnam Algorithm 70
Recursive learning 79
SAT modulo theory 82

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 ($\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\oplus$ 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 $Sat(\phi \vee \psi)$ iff $Sat(\phi)$ OR $Sat(\psi)$

↪ Normal Form: for *simplicity*

Terminology

- A literal l is either a variable x, or the negation of a variable $\bar{x}$.
- 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 $\wedge$ or $\cdot$
- “logical OR ” is $\vee$ or $+$
- “logical NOT ” is $\neg$ or $^{-}$

CNF transformation

Naive method

De Morgan's law to push "¬" the leaves

$$\text{CNF}(x) = x \qquad \text{CNF}(\bar{x}) = \bar{x}$$
$$\text{CNF}(f.g) = \text{CNF}(f).\text{CNF}(g)$$
$$\text{CNF}(\neg(f+g)) = \text{CNF}(\neg f).\text{CNF}(\neg g)$$
$$\text{CNF}(f+g) = \text{Merge}(\text{CNF}(f), \text{CNF}(g))$$
$$\text{CNF}(\neg(f.g)) = \text{Merge}(\text{CNF}(\neg f), \text{CNF}(\neg g))$$

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: CNF$(x.y + \bar{x}.(z+t))$ = ?

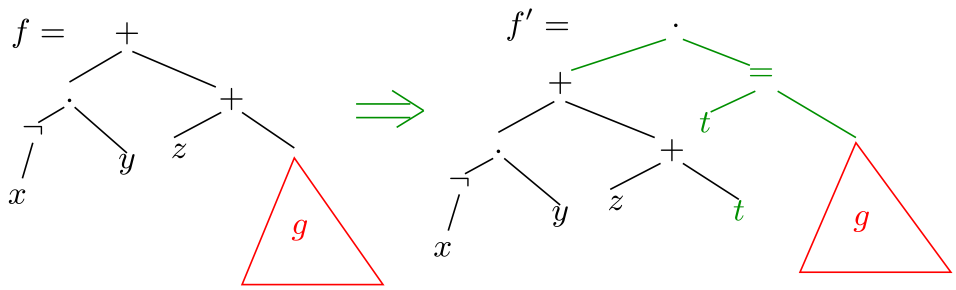
$(\bar{x} + y).(x + y + z)$

Problem

- Naive algo is *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 formlulas, example:



N.B. doest not change the SAT problem: $Sat(f)$ iff $Sat(f')$

Classical CNF construction, aka 3-SAT construction

- One (extra) variable per (binary) operator.
- Example:
  - ↪ $f = (x.y + \neg(x + \bar{y} + \bar{z}))$ gives $f = a$ where
    - ∗ $a = b + c$ and
    - ∗ $b = x \cdot y$ and
    - ∗ $c = \neg(x + d) = \bar{x} \cdot \bar{d}$ and
    - ∗ $d = \bar{y} + \bar{z}$
  - ↪ each equation gives exactly 3 clauses, e.g.:
    - ∗ $(a = b + c) \quad \Leftrightarrow \quad (\bar{a} + b + c) \cdot (a + \bar{b}) \cdot (a + \bar{c})$
    - ∗ $(b = x \cdot y) \quad \Leftrightarrow \quad (b + \bar{x} + \bar{y}) \cdot (\bar{b} + x) \cdot (\bar{b} + y)$
  - ↪ Finally: f gives 1 unit clause + 4 equations (binary ops.) that each gives 3 clauses:
    - ∗ 13 clauses
    - ∗ LINEAR: size of f' = $1 + 3\times$ size of f

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:
  - ↪ $CNF(a = (x \oplus y)) \quad = \quad (\bar{a} + \bar{x} + \bar{y}).(\bar{a} + x + y).(a + \bar{x} + y).(a + x + \bar{y})$
  - ↪ $CNF(a = (x = y)) \quad = \quad (\bar{a} + \bar{x} + y).(\bar{a} + x + \bar{y}).(a + \bar{x} + \bar{y}).(a + x + y)$
- 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 !)

Davis-Putnam Algorithm

History

- More a *general method*, with lots of derived algorithms
- The very first Davis-Putnam is NOT the right one:

$\hookrightarrow$ it's a space exploration algo (that may explode in memory)

- The "right one " should be refered as Davis-Putnam-Logemann-Loveland (DPLL):

$\hookrightarrow$ this is where the idea of linear memory cost appear

General structure

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

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

$Sat(\phi)$ =

  ϕ := *Simplify*(ϕ)
  if *Tautology*(ϕ) returns SAT
  if *Contradiction*(ϕ) returns UNSAT
  chose ONE literal x
  if $Sat(\phi \wedge x)$ returns SAT
  else if $Sat(\phi \wedge \neg x)$ returns SAT
  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 $\bar{x}$ by 0
  - ↪ i.e. clauses containing x are erased
  - ↪ i.e. $\neg x$ is erased from the other clausese
  - ↪ $\equiv$ 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

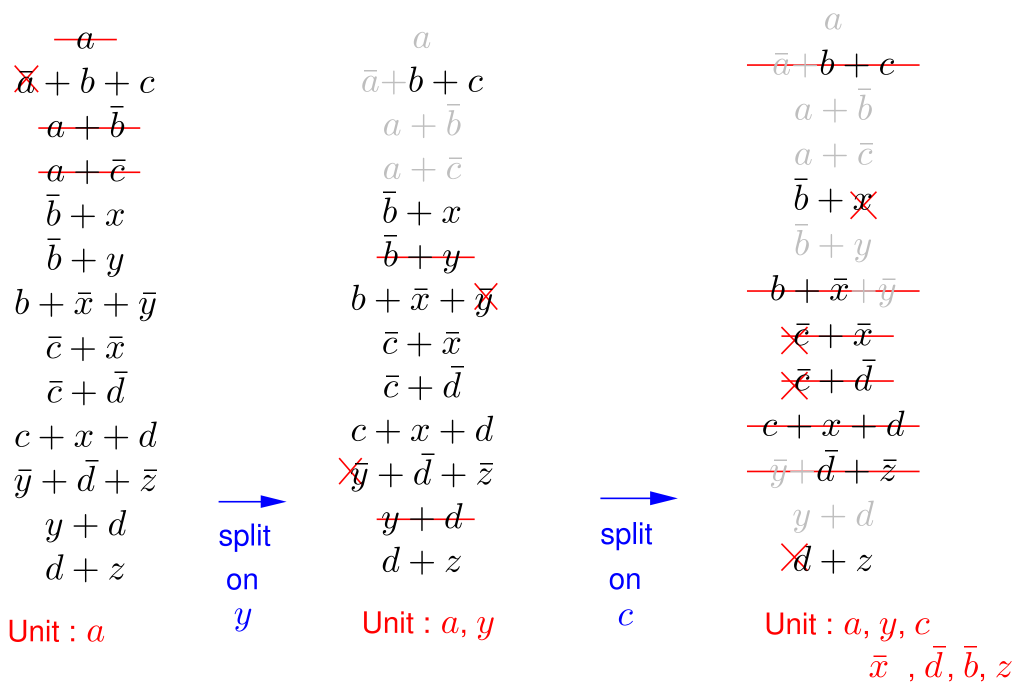
- How it works ?
  - ↪ If x is pure, alors $\phi \equiv (x + \alpha).\beta$, where neither α nor β are containing x ou $\bar{x}$
  - ↪ Conclusion: $\exists x((x + \alpha).\beta) \equiv (\beta + \alpha.\beta) \equiv \beta$
  - ↪ i.e. ϕ has solutions iff it has solutions for $x = 1$
- Problem: what about 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(LL)

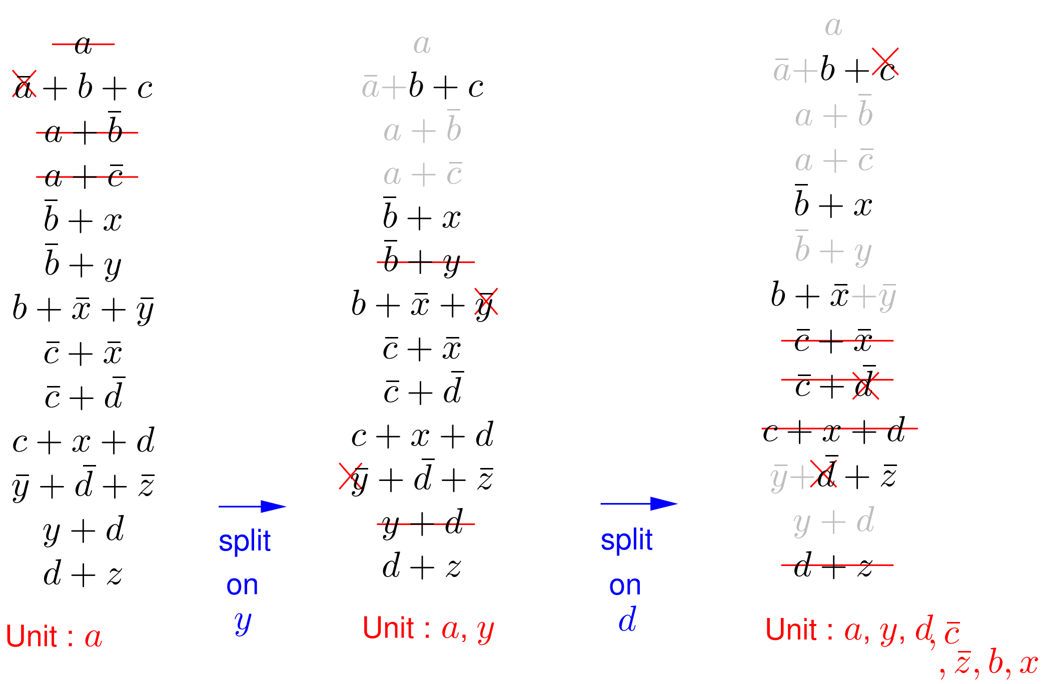
- extended SAT (enumerate solutions) with unit propagation and split
- arguments:
  - ↪ the (CNF) formula to solve f
  - ↪ the inherited partial candidate solution (monomial) m
- Starting call: DPLL(f, 1)

DPLL (f, m)

    while it exists a unit clause l in f do
        f := Eliminate(f, l); m := $m \cdot l$
    if f is identically true then PrintSolution(m); return
    else if f is identically false then return
    else chose some literal x in f
        DPLL(f, $m \cdot x$)
        DPLL(f, $m \cdot \bar{x}$)

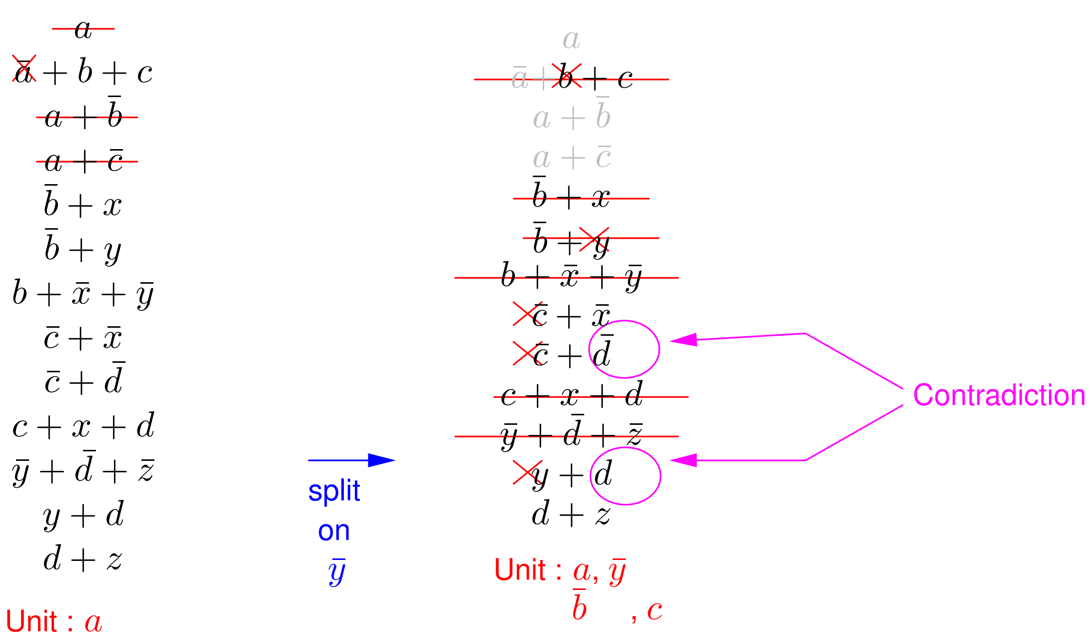


Solution : $\bar{x}$, y, z



$\rightarrow$ Split d gives $\bar{z}$: solution : $x, y, \bar{z}$

$\rightarrow$ Split $\bar{d}$ gives z : solution : x, y, z



NO solution

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 $k < n$

$x + y + z$	$x + y + z$	$x + y + z$
$\bar{x} + t + v$	$\bar{x} + t + v$	$\bar{x} + t + v$
$y + \bar{t} + w$	$y + \bar{t} + w$	~~$y + \bar{t} + w$~~
$\bar{y}$ unit of range n	$\bar{t}$ unit	contradiction

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:
  - ↪ immediately back-track to level k and add unit $\neg p_n$ to P_k (not so smart);
  - ↪ 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: $\psi_1 = (x \geq y - 1) \wedge (x + y \leq 1) \wedge (y \geq 0) \wedge (x \leq -2)$

↪ Ask the host (Linear Algebra) solver for the satisfiability of ψ_1: answer UNSAT, continue Boolean SAT solving ...

- 2nd (Boolean) solution found: $a \cdot b \cdot c \cdot e$

↪ Corresponding Host Theory formula is: $\psi_2 = (x \geq y - 1) \wedge (x + y \leq 1) \wedge (y \geq 0) \wedge (x \geq 2)$

↪ Ask the host (Linear Algebra) solver for the satisfiability of ψ_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 $f(x_1, \cdots, x_n)$ 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:

  ↪ if f is unsat, then $\neg f$ is sat for any valuations of the variables

  ↪ i.e. $\neg(\exists x\ \neg f(x)) \Leftrightarrow \forall x\ f(x)$

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

Shortcuts

- Transition relation:
  - ↪ $T(s',s) \ =_{def} \ \exists v \ s' \xrightarrow{v} s \wedge \ H(s',v)$
- Reachable states:
  - ↪ $A_0(s) = \text{Init}(s)$
  - ↪ $A_{n+1}(s) = \exists s_n \ A_n(s_n) \wedge T(s_n,s)$
  - ↪ i.e. $A_n(s)$ are the states reachable in n steps
- Property succesors:
  - ↪ $\psi^{-1}(s) \ =_{def} \ \exists s' \ \psi(s') \ \wedge \ T(s',s)$
  - ↪ $\psi^{-n-1}(s) \ =_{def} \ \exists s' \ \psi^{-n}(s') \ \wedge \ T(s',s)$
  - ↪ i.e. $\psi^{-n}(s)$ are the states reachable by a path of length n from a state satisfying ψ

A trivial case ...

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

A less trivial case ...

- if property holds for all initial states
  i.e. $A_0(s) => \psi(s)$ is a tautology
- and moreover $\psi^{-1}(s) \;\Rightarrow\; \psi(s)$
- then, by induction, ψ 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: $A_i(s) \Rightarrow \psi(s)$ is a tautology for any $i = 1 \cdots n$
- and if $\psi^{-1}(s) \wedge \psi^{-2}(s) \wedge \cdots \wedge \psi^{-n}(s) \;\Rightarrow\; \psi(s)$,
- then, by induction, ψ holds for any state

Completness 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

6. Appendix

Example/exercice: arithmetic circuit 94

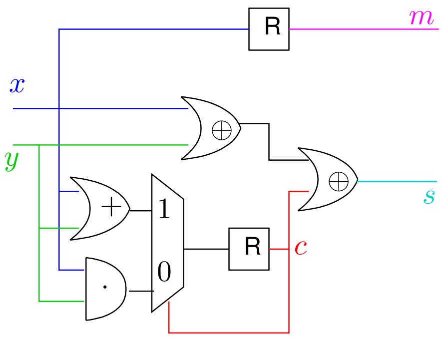
Example/exercice: arithmetic circuit

Serial adder:

- inputs x, y
- outputs s(um), c(arry)

Shift:

- m encodes $2 \times x$



time (most significant bits) →

c	0	0	1	0	
x	0	1	0		(2)
y	1	1	0		(3)
s	1	0	1		(5)
m	0	0	1		(4)

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*
  $c_{init} = 0, g_c = c.(x + y) + \bar{c}.x.y$
  $m_{init} = 0, g_m = x$
- $H \equiv (x = y)$, *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*

Starting state		Inputs		Output/Prop		Next state	
c	m	x	y	s	ϕ	c'	m'
0	0	0	0	0	1	0	0
		1	1	0	1	1	1
1	1	0	0	1	1	0	0
		1	1	1	1	1	1

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
  ↪ `x or (y xor z);` outputs: `x + y.-z + -y.z`
  ↪ `x => (y => x);` outputs: `1` (canonical form)
- Assign a formula to a "variable"
  ↪ `s := c xor x xor y;`
- Define a function over formulae
  ↪ `Implique(X,Y) := not X or Y;`
- Usefull commands: `help` and `syntax`
- ... quick demo.

Boolean model (reminder)

- 2 inputs $V = \{x, y\}$
- 2 memories $S = \{c, m\}$ with

  $c_{init} = 0$, $g_c = c.(x + y) + \bar{c}.x.y$

  $m_{init} = 0$, $g_m = x$
- $H \equiv (x = y)$, and $\phi \equiv (m = s)$, where $s = (c \oplus x \oplus y)$
- Error states: Err $\equiv (\exists x, y \;\; H \wedge \neg\Phi) \;\; \equiv (c \oplus m)$

In bddc syntax ...

```
Gm := x;
Gc := if c then (x or y) else (x and y);
s := x xor y xor c;
Init := not c and not m;
H := (x = y);
Phi := (m = s) ;
Err := exist x,y (H and not Phi);
Acc0 := Init;
```


Step 0

- Check that $Acc_0 = Init \cap Err = \emptyset$

```
Acc0 and Err;
```

gives 0, ok, continue and compute Acc_1

Step 1

- $Acc_1 = Acc_0 \cup$ post$_H(Acc_0)$
- Recall the definition of Post$_H$ (slide 50)

```
Post(A) := exist x,y,m,c (A and H and (xm = Gm) and (xc = Gc));
```

- Computes:

```
Post(Acc0);
```

gives: `xm.xc + -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) := exist a (F and (a=b));
```

- *The "right" definition of Post:*

```
Postbis(X) := Rnm(xc,c, Rnm(xm,m, Post(X)));
```

- *Check that:*

`Postbis(Acc0);` *gives:* $m.c + -m.-c$, *i.e.* $m = c$

- *Compute:*

```
Acc1 := Acc0 or Postbis(Acc0);
```

- *Are* Acc_0 *and* Acc_1 *the same ?*

```
compare(Acc1,Acc0);
```

answers 0 (not the same), fixpoint not reached...

- *Check:*

```
Acc1 and Err;
```

gives empty, no error yet ...

Step 2

- *Compute* Acc_2*:*

```
Acc2 := Acc1 or Postbis(Acc1);
```

- *Check:*

```
compare(Acc2,Acc1);
```

answers 1 (same), fixpoint is reached !

Property satisfied,

we have proven formally *that*

$\forall x, y \in \mathbb{Z} \quad (x = y) \Rightarrow (x + y = 2x)$