Verification of Synchronous Programs

Marc Pouzet

Based on slides by T. Bourke, M. Pouzet and Notes by P. Raymond

January 2020

Introduction

How to model/check properties of synchronous programs? We consider:

- functional properties, i.e., does the output of the system is correct.
- real-time constraints, execution in bounded time and memory, power consumption are examples of non functional properties

Temporal properties are of two kinds:

- Safety property: "something bad never happens", that is, an invariant property satisfied in every accessible state.
  E.g., *The train never crosses a red light* or *doors never open while the train is running*
- Liveness property: "something good eventually happens", that is, the existence of a state satisfying a property and which will eventually occur in any execution.
  E.g., *The train eventually stops*

We only consider safety properties. These are considered as the most critical in practice.

Safety Properties

We do not address the full verification of a system but instead that some bad situations never happen.

- Safety properties can be checked on program abstractions: if P is simplified into P', that is, P' has more behaviors than P and if P' satisfies a safety property so does P.
  E.g., apply a Boolean abstraction replacing a numerical comparison by a Boolean variable.
- Safety properties can be checked on program states rather than execution paths. When the state space is finite, verification is made of a "simple" traversal of the state space.
- Safety properties can be checked modularly. If $\star$ is a composition operator, one can associate an operator $\underline{\star}$ such that, for any processes P_1 and P_2 satisfying ϕ_1 and ϕ_2, their composition $P_1 \star P_2$ satisfies $\phi_1 \underline{\star} \phi_2$.

The Traditional Way

How to express and check/test a safety property? The habit in synchronous design (circuit or software) is to program it: a Lustre program is an invariant.

Program comparison

Two sequential functions `f1` and `f2` (boolean operations + registers) are equivalent if `compare(x1,...,xn)` equals the infinite sequence $\texttt{true}^\omega$.

```
node compare(x1,...,xn:bool) returns (ok:bool);
let
  ok = f1(x1,...,xn) = f2(x1,...,xn);
tel;
```

Verification: a first solution

- Compile the function `compare`; if the automaton only contains transitions labeled with `ok = true`, the property is true.
- Better: directly produce the minimal automaton for `compare`. It should be the trivial automaton with only one state and one transition labeled `ok = true`.

If not, we have built a counter example.

Example: two versions of `switch`

```
node switch1(on, off:bool)
returns (run:bool);
let
  run = if on then true
        else if off then false
        else false -> pre run;
tel;
```

```
node switch2(on, off:bool)
returns (run:bool);
let
  run = if (false -> pre run)
        then not off else on;
tel;
```

```
> lesar switches.lus compare -diag
--Pollux Version 2.4
DIAGNOSIS:
--- TRANSITION 1 ---
on
--- TRANSITION 2 ---
on and off
FALSE PROPERTY
```

```
node compare(on, off:bool)
returns (ok:bool);
let
  ok = (switch1(on, off) = switch2(on, off));
tel;
```

Taking the Environment into Account

Restrict the state space by adding hypothesis on inputs. Add the hypothesis that events on and off are exclusive.

```
node compare(on, off: bool) returns (ok : bool);
  let
    -- on and off never true at the same instant
    assert not(on and off);

    ok = switch1(on, off) = switch2(on, off);
  tel;
```

```
> lesar switches.lus compare -diag
--Pollux Version 2.4

TRUE PROPERTY
```

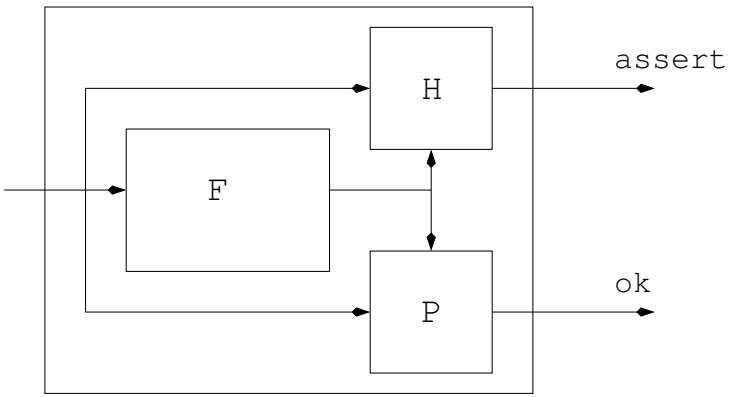
Thus, while the hypothesis stays true, ok stays true

Synchronous Observers

The comparison of programs is a particular case of a synchronous observer.

- if $y = F(x)$, we write $ok = P(x, y)$ for the property relating x and y
- and $\texttt{assert}(H(x, y))$ to states an hypothesis on the environment.

```
node check(x:t) returns (ok:bool);
let
  assert H(x,y);
  y = F(x);
  ok = P(x,y);
tel;
```



If *assert* remains indefinitely true then *ok* remains indefinitely true $(\texttt{always}(assert)) \Rightarrow (\texttt{always}(ok))$.

Any temporal safety property can be expressed as a Lustre program. No need to introduce a temporal logic in the language

[Halbwachs, Lagnier, and Raymond (1993): Synchronous observers and the verification of reactive systems] [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE];

Temporal properties are regular Lustre programs

Example of Temporal Properties

- "A is never true twice in a row": `never_twice(A)` where:

```
node never_twice(A:bool) returns (OK:bool);
let
  OK = true -> not(A and pre A);
tel;
```

- "Any event A is followed by an event B before C happen":
  followed_by(A, B) and followed_by(B, C)
  where:

```
node implies(A,B : bool) returns (OK : bool);
let
  OK = not(A) or B;
tel;

node once(A:bool) returns (OK : bool);
let
  OK = A -> A or pre OK;
tel;
```

```
node followed_by(A,B : bool)
returns (OK : bool);
let
  OK = implies(B, once(A));
tel;
```

Example of Temporal Properties (cont.)

Note: Several properties have a sequential nature, e.g., "The temperature should increase for at most 1 min or until the event stop occurs then it must decrease for 2 min".

They can be expressed as regular expressions and then translated into Lustre [Raymond (1996): Recognizing regular expressions by means of dataflow networks]

This is the basis of the language Lutin

[Raymond, Roux, and Jahier (2008): Lutin: A Language for Specifying and Executing Reactive Scenarios]

For an encoding of past-time Linear Temporal Logic (LTL) see:

[Halbwachs, Fernandez, and Bouajjani (1993): An executable temporal logic to express safety properties and its connection with the language Lustre]

Example: beacon counting (by P. Raymond)

Counting beacon do decide whether a train is late, early or on time.

An hysteresis with two thresholds to avoid oscillations;

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

node counter (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 = switch(false, ontime and (diff > 3), diff <= 1);
  late = switch(false, ontime and (diff < -3), diff >= -1);
  ontime = not (early or late);
tel
```

Properties

Safety properties:

- "never late and early"; "either late, early or on time"
- "never pass from late to early"
- "it is impossible to remain late only one instant"

Liveness property:

- "if the train stops, it will eventually get late"

Note that: "if the train is ontime and stops for 10 seconds, it will get late" is a safety property.

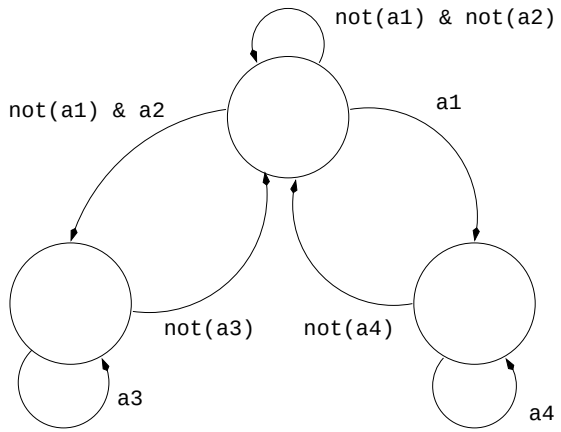
Boolean Abstraction

The explicit automaton is infinite (e.g., pre diff = 0,1,2,...).
Replace numerical comparison by fresh free boolean variables.

a_1 for diff > 3 $\quad$ a_3 for diff $>= -1$

a_2 for diff < -3 $\quad$ a_4 for diff $<= 1$

The resulting automaton is now finite.



- safety properties such as "it is impossible to be late and early" or "it is impossible to directly pass from late to early" are kept.
- some properties cannot be checked anymore: "it is impossible to remain late only one instant" (safety) or "it the train stops, it will eventually get late" (liveness)
- some are introduced: "it is possible to remain late only one instant" (liveness)". True on the abstraction, false on the real program.

Abstraction and Safety

- Boolean abstraction is a special case of over-approximation
- Anything impossible in the abstraction is impossible on the program
- Safety properties are preserved or lost but never introduced

When checking the abstraction:

- if the verification succeeds, the property is satisfied by the initial program
- otherwise, no conclusion can be made ("false negative")

Program Safety Properties as observers

- "it is impossible to be late and early":

  ok = not(late and early);

- "it is impossible to directly pass from late to early":

  ok = true -> (not early and pre late);

- "it is impossible to remain late only one instant":

  plate = false -> pre late;
  pplate = false -> pre late;
  ok = not (not late and plate and not pplate);

- "if the train keeps the right speed, it stays on time"
  - » Naive: assert (sec = bea)
  - » Better: 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);

Symbolic Representation

Using the minimal state automaton generated by the compiler for program verification is expensive.

- no need to explicitly build all the states of the automaton;
- only enumerate them: thus, develop a dedicated tool

Input: an implicit transition system

- A set of input variables I, a set of state variables S
- An initial state: $s_{init} \in \mathbb{B}^{|S|}$
- A property (ok): $\phi(\vec{s}, \vec{i})$
- An hypothesis (assertion): $h(\vec{s}, \vec{i})$
- A transition function g such that: $s'_k = g_k(\vec{s}, \vec{i})$

Notation: write $\vec{s} \xrightarrow{\vec{i}} \vec{s'}$ when $\vec{s'} = (g_1(\vec{s}, \vec{i}), ..., g_k(\vec{s}, \vec{i}))$

Accessible states

- Accessible states:
  A state $\vec{s}$ is accessible w.r.t h iff there exists a sequence:
  $$s_{init} \xrightarrow{\vec{i_1}} \vec{s_2} \xrightarrow{\vec{i_2}} \ldots \xrightarrow{\vec{i_n}} \vec{s} \text{ where } \forall t, h(\vec{s_t}, \vec{i_t})$$
  We write this $\vec{s} \in \mathit{Reach}$, i.e., $\mathit{Reach} = \mu X.(X = \texttt{init} \cup \mathit{Post}_h(X))$.
- Bad states: A state $\vec{s}$ is bad iff:
  $$\exists \vec{i}.h(\vec{s}, \vec{i}) \wedge \neg\phi(\vec{s}, \vec{i})$$
  We write this $\vec{s} \in \mathit{Error}$, i.e., $\mathit{Bad} = \mu X.(X = \mathit{Error} \cup \mathit{Pre}_h(X))$.
- $\mathit{Post}_h(\vec{s})$ is the set of successors of $\vec{s}$:
  $$\mathit{Post}_h(\vec{s}) = \{\vec{s'} \mid \exists \vec{i}.h(\vec{s}, \vec{i}) \wedge \vec{s} \xrightarrow{\vec{i}} \vec{s'}\}$$
- $\mathit{Pre}_h(\vec{s})$ is the set of predecessors of $\vec{s}$:
  $$\mathit{Pre}_h(\vec{s'}) = \{\vec{s} \mid \exists \vec{i}.h(\vec{s}, \vec{i}) \wedge \vec{s} \xrightarrow{\vec{i}} \vec{s'}\}$$
- Goal of the proof: Check that $\mathit{Reach} \cap \mathit{Error} = \emptyset$ or $\mathit{Bad} \cap \texttt{init} = \emptyset$.

Proof by enumeration (forward)

```
reached := {s⃗_init};
explored := ∅;
while reached\explored ≠ ∅ do
  let s⃗ ∈ reached\explored in
  if s⃗ ∈ Error then raise Stop
  else
    begin
      explored := explored ∪ {s⃗};
      reached := reached ∪ Post_h(s)
    end
done;
```

The algorithm is very expensive: $2^{|S|}$ states to explore; $2^{|I|}$ in every state.

In the same way, we can build a backward version of the algorithm (far more complex; never used in practice).

This is the basis of model checking

[Queille and Sifakis (1982): Specification and Verification of Concurrent Systems in CESAR]

[E. M. Clarke, Emerson, and Sistla (1986): Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications]

Its direct implementation is too inefficient to treat large systems (e.g., several millions of states).

Symbolic Algorithms

- manage directly sets (of states, of transitions) symbolically
- a set of states = a Boolean formula on state variables
- Example: $x \wedge \neg y$ = the set of states where x is true and y is false.
- ϕ and h are formulas on $S \times I$, thus they define a set of pairs $(\vec{s}, \vec{i})$

Set operations: a set as a boolean formula

- let $A \subseteq \mathbb{B}^n$ and ϕ_A be the corresponding formula
- union: $\phi_{A \cup B} = \phi_A \vee \phi_B$; intersection: $\phi_{A \cap B} = \phi_A \wedge \phi_B$
- complement: $\phi_{\mathbb{B}^n \setminus A} = \neg \phi_A$

Problem: efficient implementation of formula and the corresponding decision (does $A = B$?)
$\Longrightarrow$ Binary Decision Diagrams (BDDs).

Binary Decision Diagrams

Introduced by R. Bryant in the 80s for the verification of hardware.

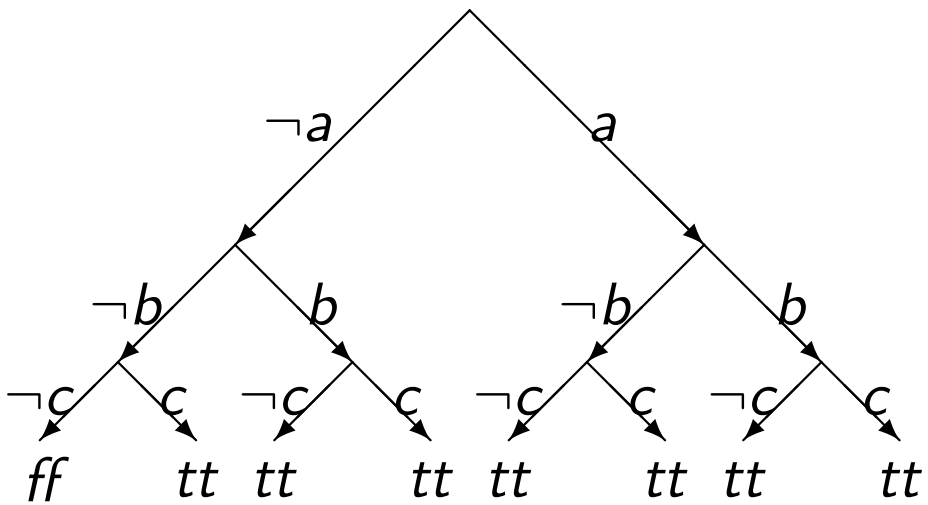
[Bryant (1986): Graph-Based Algorithms for Boolean Function Manipulation]

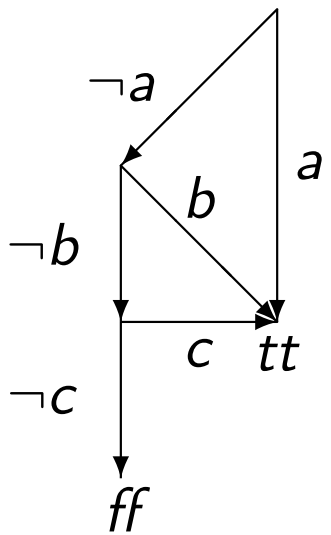
(See also: [Andersen (1999): An Introduction to Binary Decision Diagrams])

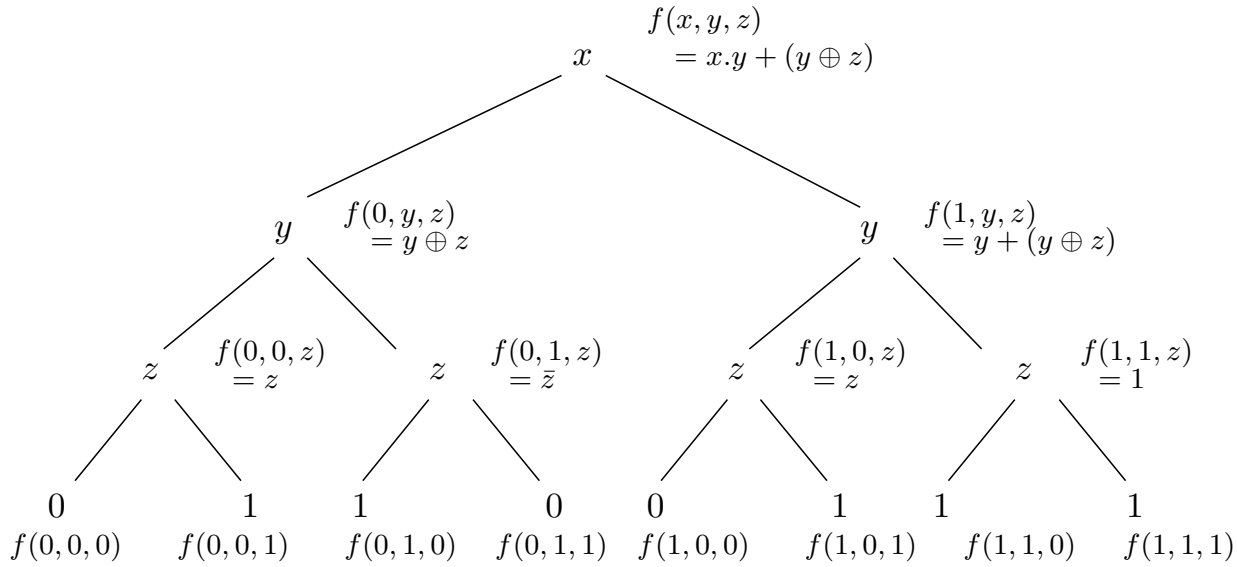
- reduced and canonical representation of boolean functions
- based on the Shannon decomposition

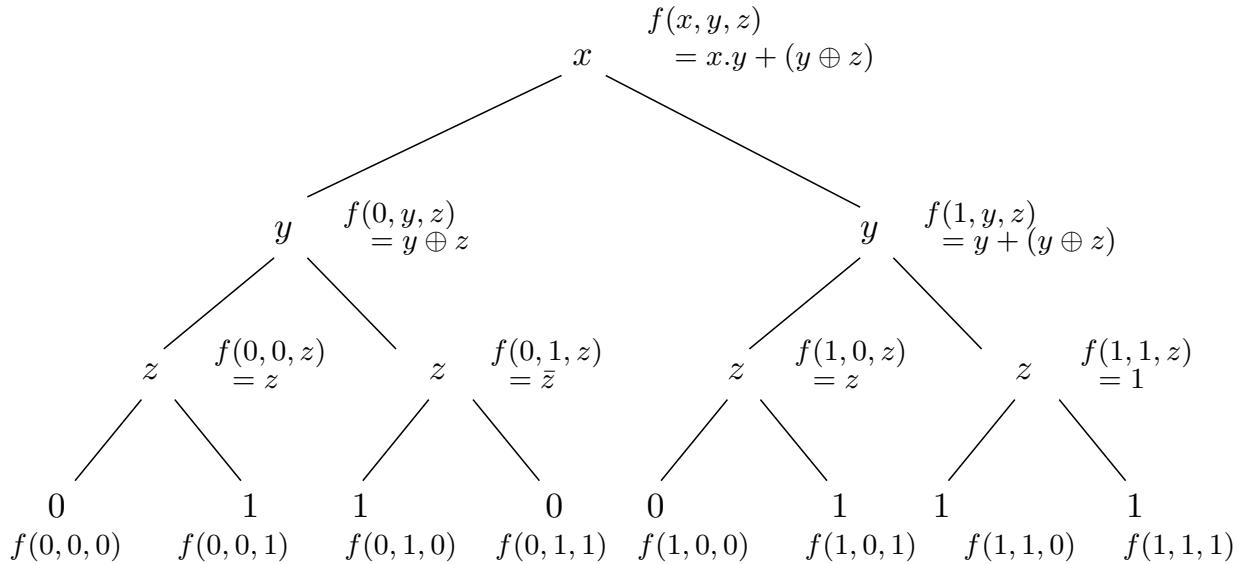
$$f(a,b,c) = a \wedge f(tt,b,c) \vee \neg a \wedge f(ff,b,c)$$

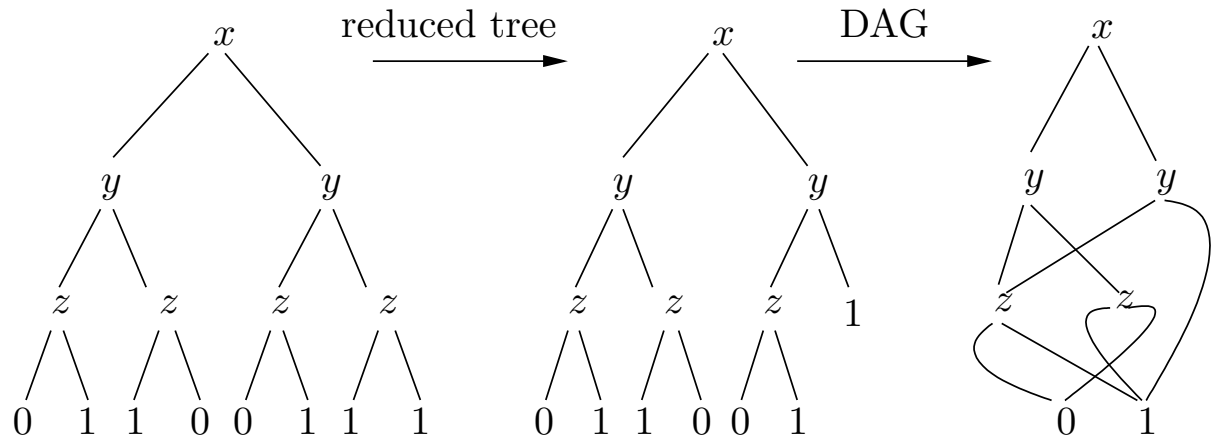
Example: $f(a,b,c) = a \vee b \vee c$






$f(x,y,z)$
$= x.y + (y \oplus z)$
x
y
$f(0,y,z)$
$= y \oplus z$
y
$f(1,y,z)$
$= y + (y \oplus z)$
z
$f(0,0,z)$
$= z$
z
$f(0,1,z)$
$= \bar{z}$
z
$f(1,0,z)$
$= z$
z
$f(1,1,z)$
$= 1$
0
$f(0,0,0)$
1
$f(0,0,1)$
1
$f(0,1,0)$
0
$f(0,1,1)$
0
$f(1,0,0)$
1
$f(1,0,1)$
1
$f(1,1,0)$
1
$f(1,1,1)$


$f(x,y,z)$
$= x.y + (y \oplus z)$
x
y
$f(0,y,z)$
$= y \oplus z$
y
$f(1,y,z)$
$= y + (y \oplus z)$
z
$f(0,0,z)$
$= z$
z
$f(0,1,z)$
$= \bar{z}$
z
$f(1,0,z)$
$= z$
z
$f(1,1,z)$
$= 1$
0 1 1 0 0 1 1 1
$f(0,0,0)$ $f(0,0,1)$ $f(0,1,0)$ $f(0,1,1)$ $f(1,0,0)$ $f(1,0,1)$ $f(1,1,0)$ $f(1,1,1)$


reduced tree
DAG
x
y y
z z z z
0 1 1 0 0 1 1 1
x
y y
z z z 1
0 1 1 0 0 1
x
y y
z z
0 1

Binary Decision Diagrams

- logical operations
  - » $\vee, \wedge, \neg, \forall$ *bounded*, $\exists$ *bounded* can be expressed
- symbolic computation of boolean functions

Basic principle:

- first choose an order between variables $x_1 < ... < x_n$;
- for every boolean operation op, compute $apply(op)(b_1, b_2)$

$$
\begin{aligned}
&apply(op)(ite(x, b_1, b_2), ite(y, b_1', b_2')) = \\
&\quad ite(x, apply(op)(b_1, b_1'), apply(op)(b_2, b_2')) \text{ if } x = y \\
&\quad ite(x, apply(op)(b_1, ite(y, b_1', b_2')), apply(op)(b_2, ite(y, b_1', b_2'))) \text{ if } x < y \\
&\quad ite(y, apply(op)(b_1, ite(x, b_1, b_2)), apply(op)(b_2, ite(x, b_1, b_2))) \text{ otherwise}
\end{aligned}
$$

- + memoization tables to build the ROBDD (Reduced Ordered BDD).

Problem: The size of BDD depends on the chosen order between variables. This choice is difficult.

Symbolic Model Checking

Encoding set of states by Boolean formula, e.g., $x \vee \neg y$ for the union of sets where x is true or y is false.

A BDD A to represents reachable states in less than n transitions.

```
A := init
while true do
  if A ∧ Error ≠ 0 then raise Error
  else let A' = A ∨ Post_h(A) in
       if A' = A then raise Success
       else A := A'
done
```

On success, $A = A' = \mathit{Reach}$.

Implementation of $Post_h(X)$

Build a formula over $s_1, ..., s_n$ (state variables), input variables $v_1, ..., v_k$, new state variables $s'_1, ..., s'_n$.

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

Backward Symbolic Algorithm

The same algorithm except that we compute $Pre_h(X)$.
A BDD B to represent states leading to *Error* in less than n transitions.

```
B := Error
while true do
  if A ∧ init ≠ 0 then raise Error
  else let B' = B ∨ Pre_h(B) in
       if B' = B then raise Success
       else B := B'
done
```

On success, $B = B' = Bad$.

Implementation of $Pre_h(X)$

Build a formula over $s'_1, ..., s'_n$ (state variables), input variables $v_1, ..., v_k$, new state variables $s_1, ..., s_n$.

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

Kind 2 Model Checker

- http://kind2-mc.github.io/kind2/ (or use web interface: http://kind.cs.uiowa.edu:8080/app/)
- SMT-based Model Checker for Lustre:
  Bounded Model Checking, k-induction, IC3, ...
- Specify properties to check as comments:
  --%PROPERTY ok;

```
> kind2 switches.lus
kind2 v1.1.0-214-g00b3d21d

==========================================================
Analyzing compare
  with First top: "compare"
             subsystems
               | concrete: switch2, switch1

<Success> Property ok is valid by inductive step after 0.164s.

----------------------------------------------------------
Summary of properties:
----------------------------------------------------------
ok: valid (at 1)
==========================================================
> kind2 --enable BMC --enable IND --lus_main compare switches.lus
```

Encoding Lustre in SMT [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

- Represent streams as uninterpreted functions $\mathbb{N} \to \tau$
- Examples:

x = y + z	$\forall n : \mathbb{N},\ x(n) = y(n) + z(n)$
x = y -> y + pre z	$\forall n : \mathbb{N},\ x(n) = \mathit{ite}(n = 0, y(0), y(n) + z(n-1))$

Encoding Lustre in SMT [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

- Represent streams as uninterpreted functions $\mathbb{N} \to \tau$
- Examples:

x = y + z	$\forall n : \mathbb{N},\ x(n) = y(n) + z(n)$
x = y -> y + pre z	$\forall n : \mathbb{N},\ x(n) = \mathit{ite}(n = 0, y(0), y(n) + z(n-1))$

- Let N be a node with stream variables $\mathbf{x} = \langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ ($x_1, \ldots, x_p$ are inputs, and $y_1, \ldots, y_q$ are outputs)
- $$\Delta(n) = \begin{cases} y_1(n) = t_1[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \\ \qquad \vdots \\ y_q(n) = t_q[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \end{cases}$$

```
node thermostat (actual_temp, target_temp, margin: real)
returns (cool, heat: bool);
let
  cool = (actual_temp − target_temp) > margin;
  heat = (actual_temp − target_temp) < −margin;
tel

node therm_control (actual: real; up, down: bool) returns (heat, cool: bool);
var target, margin: real;
let
  margin = 1.5;
  target = 70.0 −> if down then (pre target) − 1.0
                  else if up then (pre target) + 1.0
                  else pre target;
  (cool, heat) = thermostat (actual, target, margin);
tel
```

$$\Delta(n) = \begin{cases} m(n) = 1.5 \\ t(n) = ite(n = 0, 70.0, ite(d(n), t(n-1) - 1.0, \ldots)) \\ c(n) = (a(n) - t(n)) > m(n) \\ h(n) = ((a(n) - t(n)) < -m(n) \end{cases}$$

Model-checking Safety Property with a SAT Solver

Given a boolean formula b with free variables $x_1, ..., x_n$ from propositional logic, find a valuation $V : \{x_1, ..., x_n\} \rightarrow \{0, 1\}$ such that $V(b) = 1$.

- initial algorithm by Davis-Putnam-Logemann-Loveland (DPLL); various heuristics. Generalization of SAT to QBF (Quantified Boolean Formula)
- a very active/competitive research/industrial topic (see http://www.satlive.org/)
- Now, more interest for SMT (Satisfiability Modulo Theory) for first-order logic (quantified formula + interpreted/non-interpreted functions)
- close interaction between a SAT solver and ad-hoc solvers (e.g., simplex. method for linear arithmetic constraints)

Bounded Model-checking (BMC)

A BDD is a compact representation of the truth of a boolean formula and represents the whole behavior of the transition function. It may be too large when the system gets bigger.

Alternative approach (when BDDs fail): find "counter-examples" by using a SAT solver. Originally proposed by Clarke et al. [Biere, Cimatti, E. Clarke, and Zhu (1999): Symbolic Model Checking without BDDs]

Notation

- A property P to check for every accessible state.
- $Init(s_0)$ if s_0 is an initial state; $T(s_i, s_{i+1})$ if s_{i+1} is a successor of s_i.
- let $Path(s[0..k]) = Init(s_0) \wedge T(s_0, s_1) \wedge ... \wedge T(s_{i-k}, s_k)$.
- let $P(s[0..k]) = P(s_0) \wedge P(s_1) \wedge ... \wedge P(s_k)$
- let $Trace(c[0..k])$ be an assignment to variables $s[0..k]$ that makes $Init(s_0) \wedge Path(s[0..k]) \wedge \neg P(s_k)$ true.

Prove that $Path(s[0..k]) \Rightarrow P(s[0..k])$ or (equivalently), find a counter example with some k such that: $Path(s[0..k]) \wedge \neg P(s_k)$

k-Induction

It lacks an induction. k-induction is the iteration of a BMC

[Sheeran, Singh, and Stålmarck (2000): Checking Safety Properties Using Induction and a SAT-Solver]

Basic Principle Prove it by induction, either in one step, two steps, etc.

- $(\texttt{init}\ s_0 \wedge P(s_0) \wedge (\forall s.(P(s) \wedge T(s,s') \Rightarrow P(s')))) \Rightarrow \forall s \in \mathit{Reach}.P(s)$
- $\left(\begin{array}{c} \texttt{init}\ s_0 \wedge P(s_0) \wedge P(s_1) \wedge T(s_0,s_1) \wedge \\ (\forall s,\ (P(s) \wedge P(s') \wedge T(s,s') \wedge T(s',s'')) \Rightarrow P(s'')) \end{array}\right)$
  $\Rightarrow \forall s \in \mathit{Reach}.\ P(s)$
- etc.

Stop when there is an accessible state which does not verify P, i.e., for some n, the formula $(\mathit{Path}(s[0..n]) \wedge \neg \mathit{all}.P(s[0..n]))$ is satisfiable.

Algorithm

Basic algorithm

```
i := 0;
while true do
  let path = Path(s[0..i]) in
  if (path ∧ ¬all.P(s[0..i])) then raise Error(Trace(c[0..i]))
  else let base = path ⇒ P(s[0..i]) in
       let ind = path ∧ P(s[0..i]) ∧ T(s_i, s_{i+1}) ⇒ P(s_{i+1}) in
       if base ∧ ind then raise Success;
  i := i + 1;
done
```

Remarks:

- only add real successors in n steps to reduce the size of the formula (s_n such that $T(s_{n-1}, s_n)$ and s_{n-1} is not reachable in $n-1$ steps). This appear in case of loops in the transition system.

See the use of SMT solver to model-check Lustre programs

[Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

Summary

- Express programs, (safety) properties, and assumptions on the environment in a single language.
- Model-checking ideal:
  - » 'push-button' verification gives ok or counter-example;
  - » no need to understand why (i.e., write invariants).
- Automata minimization and forward enumeration are usually too expensive.
- BDD-based symbolic verification works well for purely boolean programs.
  - » Abstract for programs with integers or reals.
  - » 'Bit blast' for programs with integers.
  - » Already less 'push-button'...
- SAT-based techniques for BMC, complete with k-induction.
- Extend SAT to SMT to handle integers and apply sophisticated optimizations.
- (Not treated: automata-based techniques, ITPs.)

Project

Two possibilities

- lmoch: model-checker with k-induction (and IC3).
- minilucy: compiler to sequential code (e.g., C or Rust).

Details

- Work alone or with someone else.
- Implement or experiment with the aspect that most interests you.
- Due: 16 December 2018.
- Presentations: 17 to 21 December.

References I

- Andersen, H. R. (1999). *An Introduction to Binary Decision Diagrams*. Lecture Notes.
- Biere, A., A. Cimatti, E. Clarke, and Y. Zhu (Mar. 1999). "Symbolic Model Checking without BDDs". In: *5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*. Ed. by W. R. Cleaveland. Vol. 1579. LNCS. Amsterdam, The Netherlands: Springer, pp. 193–207.
- Bryant, R. E. (Aug. 1986). "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8, pp. 677–691.
- Clarke, E. M., A. Emerson, and A. P. Sistla (Apr. 1986). "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Programming Languages and Systems (TOPLAS)* 8.2, pp. 244–263.
- Hagen, G. and C. Tinelli (Nov. 2008). "Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques". In: *Proc. 8th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2008)*. Ed. by A. Cimatti and R. B. Jones. IEEE. Portland, OR, USA, Article 15.

References II

- Halbwachs, N., F. Lagnier, and P. Raymond (June 1993). "Synchronous observers and the verification of reactive systems". In: *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*. Ed. by M. Nivat, C. Rattray, T. Rus, and G. Scollo. Twente: Workshops in Computing, Springer Verlag.
- Halbwachs, N., J.-C. Fernandez, and A. Bouajjani (Apr. 1993). "An executable temporal logic to express safety properties and its connection with the language Lustre". In: *Proc. 6th Int. Symp. Lucid and Intensional Programming (ISLIP'93)*. Quebec, Canada.
- Halbwachs, N., F. Lagnier, and C. Ratel (Sept. 1992). "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE". In: *IEEE Trans. Software Engineering* 18.9, pp. 785–793.
- Queille, J.-P. and J. Sifakis (Apr. 1982). "Specification and Verification of Concurrent Systems in CESAR". In: *Proc. 5th Int. Symp. Programming*. Ed. by M. Dezani-Ciancaglini and U. Montanari. Vol. 137. LNCS. Turin, Italy: Springer, pp. 337–351.

References III

- Raymond, P. (n.d.). "Synchronous Program Verification with Lustre/Lesar". In: chap. 6, pp. 171–206.
- — (July 1996). "Recognizing regular expressions by means of dataflow networks". In: *Proc. 23rd Int. Colloq. on Automata, Languages and Programming*. Ed. by F. Meyer auf der Heide and B. Monien. LNCS 1099. Paderborn, Germany: Springer, pp. 336–347.
- Raymond, P., Y. Roux, and E. Jahier (2008). "Lutin: A Language for Specifying and Executing Reactive Scenarios". In: *EURASIP Journal of Embedded Systems*.
- Sheeran, M., S. Singh, and G. Stålmarck (Nov. 2000). "Checking Safety Properties Using Induction and a SAT-Solver". In: *Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*. Ed. by W. A. Hunt Jr. and S. D. Johnson. IEEE. Austin, TX, USA, pp. 127–144.