Verification of Synchronous Programs

Marc Pouzet
Ecole normale supérieure

Marc.Pouzet@ens.fr

MPRI, October 25, 2016

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 wrong never happen", 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 eventually happen", 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 contain transitions labelled 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 labelled `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;

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

becane.local[3] lesar a.lus compare -diag
--Pollux Version 2.3
DIAGNOSIS:
--- TRANSITION 1 ---
on
--- TRANSITION 2 ---
on and off
FALSE PROPERTY
```

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;

becane.local[4] lesar a.lus compare -diag
--Pollux Version 2.3

TRUE PROPERTY
```

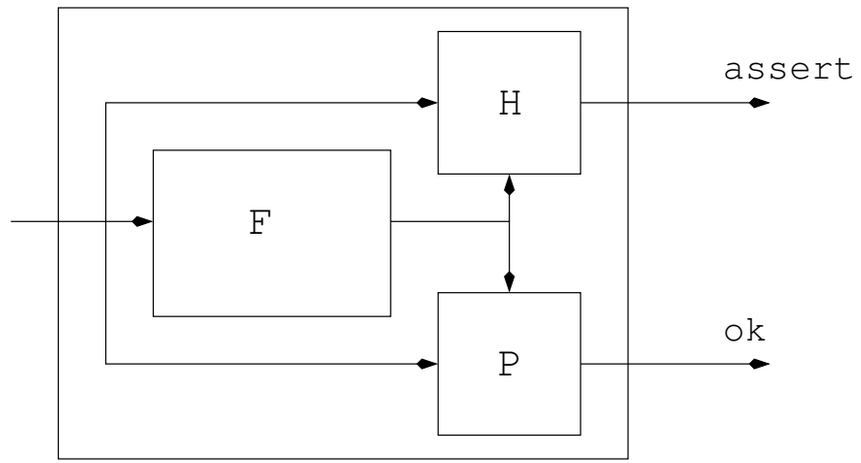
Thus, while the hypothesis stays true, `ok` stays true

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 `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 (`always`$(assert)) \Rightarrow$ (`always`(ok)).

Any temporal safety property can be expressed as a **Lustre** program. No need to introduce a temporal logic in the language [4, 3];

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

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;
```

Notice: 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, ICALP 96].

This is the basis of the language **Lutin** [Raymond, EURASIP 08].

Example: beacon counting (by P. Raymond)

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

An hysteresis with two thresholds to avoid oscillations;

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

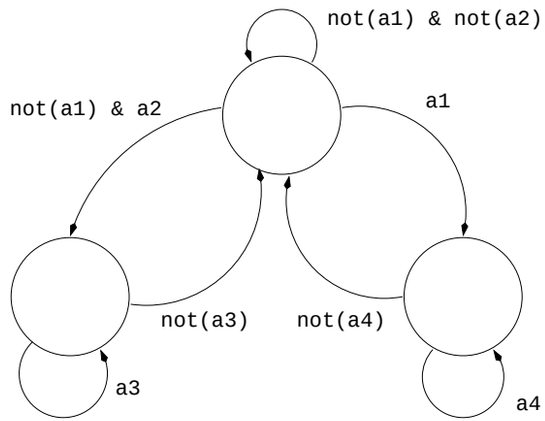
- "if the train stops, it will eventually get late"

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

Abstraction

Boolean abstraction: The explicit automaton is infinite (e.g., `pdiff`=0,1,-1,...). Replace numerical comparison by fresh free boolean variables. E.g., a_1 for `diff > 3`, a_2 for `diff > 1`, a_3 for `diff < -3`, 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 abtraction is impossible on the program
- Safety properties are preserved or lost but never introduced

When checking the abstraction:

- if the verification succeed, the property is verified 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 explicitely 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: $\vec{s_{init}} \in \mathbb{B}^{|S|}$
- 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:

  $$s_{\vec{init}} \xrightarrow{\vec{i_1}} \vec{s_2} \xrightarrow{\vec{i_2}} \ldots \xrightarrow{\vec{i_n}} \vec{s} \text{ where } \forall t, h(\vec{s_t}, \vec{i_t})$$

  We write this $\vec{s} \in Reach$, i.e., $Reach = \mu X.(X = \texttt{init} \cup Post_h(X))$.

- **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 $\vec{s} \in Error$, i.e., $Bad = \mu X.(X = Error \cup Pre_h(X))$.

- $Post_h(\vec{s})$ is the set of successors of $\vec{s}$:

  $$Post_h(\vec{s}) = \{\vec{s'} / \exists \vec{i}.h(\vec{s}, \vec{i}) \wedge \vec{s} \xrightarrow{\vec{i}} \vec{s'}\}$$

- $Pre_h(\vec{s})$ is the set of predecessors of $\vec{s}$:

  $$Pre_h(\vec{s'}) = \{\vec{s} / \exists \vec{i}.h(\vec{s}, \vec{i}) \wedge \vec{s} \xrightarrow{\vec{i}} \vec{s'}\}$$

- **Goal of the proof:** Check that $Reach \cap Error = \emptyset$ or $Bad \cap \texttt{init} = \emptyset$.

Proof by enumeration (forward)

Two sets: `reach` (reached states), `exp` (explored states). Initialy: `reach` = $\{\vec{s_{init}}\}$ and `exp` = $\emptyset$.

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

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

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

Symbolic Algorithms

This is the basis of **model checking** [Queille and Sifakis, 82; Clarke, Emerson, Sistla, 86]. Its direct implementation is too inefficient to treat large systems (e.g., several millions of states).

Solution:

- manage directly sets (of states, of transitions) symbolically
- a set of state = 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 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 $A = B$?). $\Rightarrow$ Binary Decision Diagrams (BDD).

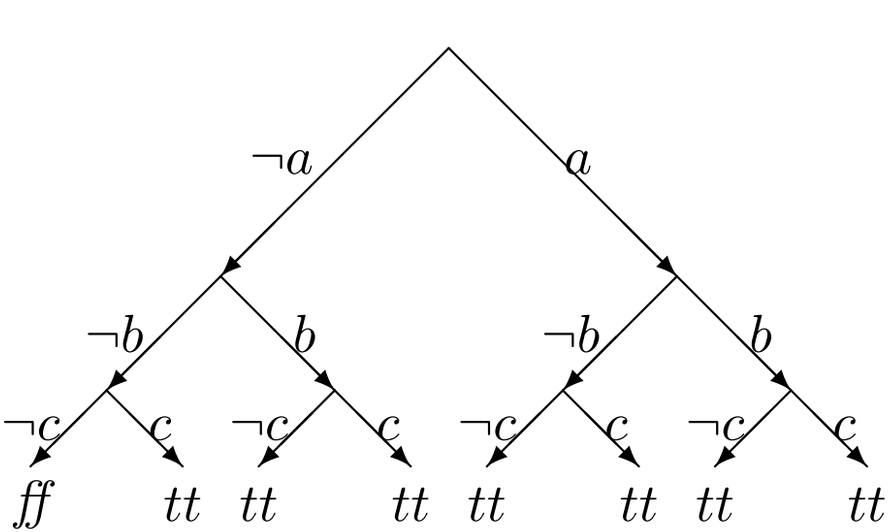
Binary Decision Diagrams

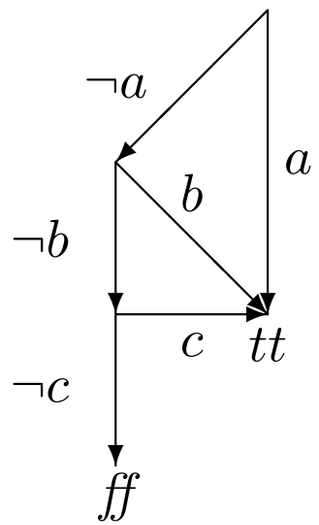
Introduced by Randy Bryant in the 80's for the verification of hardware.

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





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 $build(op)(b_1, b_2)$

$$build(op)(ite(x, b_1, b_2), ite(y, b'_1, b'_2)) =$$
$$ite(x, build(op)(b_1, b'_1), build(op)(b_2, b'_2)) \text{ if } x = y$$
$$ite(x, build(op)(b_1, ite(y, b'_1, b'_2)), build(op)(b_2, ite(y, b'_1, b'_2))) \text{ if } x < y$$
$$ite(y, build(op)(b_1, ite(x, b_1, b_2)), build(op)(b_2, ite(x, b_1, b_2))) \text{ otherwise}$$

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

Problem: The size of BDD depend on the chosen order between variables. This choice is difficult. *(One lesson by Jean Vuillemin on BDDs)*.

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.

Initially: $A =$ `init` .

Algorithm:

```
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
```

A the end, $A = A' = Reach$.

Implementation of $Post_h(X)$

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

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

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.

Initially: $B = Error$. Algorithm:

```
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
```

A the end, $B = B' = Bad$.

Implementation of $Pre_h(X)$

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

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

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 $V : \{x_1, ..., x_n\} \to \{0, 1\}$ such that $V(b) = 1$.

- initial algorithm by Davis-Putnam-Logemann-Loveland (DPLL); various heuristic. Generalisation 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 BDD fail): find "counter-examples" by using a SAT solver. Originaly proposed by Clarke et all. [TACAS 99].

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 make $Init(s_0) \wedge Path(s[0..k]) \wedge \neg P(s_k)$ 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, Stalmark and Singh, FMCAD 2000].

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

- $(\texttt{init}\ s_0 \wedge P(s_0) \wedge (\forall s.(P(s) \wedge T(s,s') \Rightarrow P(s')))) \Rightarrow \forall s \in Reach.P(s)$
- $(\texttt{init}\ s_0 \wedge P(s_0) \wedge P(s_1) \wedge T(s_0,s_1) \wedge$
  $(\forall s.(P(s) \wedge P(s') \wedge T(s,s') \wedge T(s',s'')) \Rightarrow P(s''))) \Rightarrow \forall s \in Reach.P(s)$
- etc.

Stop when there is an accessible state which does not verify P, i.e., for some n, the formula $(Path(s[0..n]) \wedge \neg all.P(s[0..n]))$ 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 [George Hagen and Cesare Tinelli, FMCAD 2008]

References

[1] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[2] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, pages 109–117. IEEE, 2008.

[3] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[4] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[5] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, Paderborn, Germany, July 1996. LNCS 1099, Springer Verlag.

[6] Pascal Raymond. *Synchronous Program Verification with Lustre/Lesar*, chapter 6, pages 171–206. ISTE/Wiley, 2008.

[7] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: a language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008, 2008. Article ID 753821.

[8] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.