Verification of Synchronous Programs

Marc Pouzet
Ecole normale supérieure
Marc.Pouzet@ens.fr

MPRI, November 4, 2014
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 $\star_\phi$ such that, for any processes $P_1$ and $P_2$ satisfying $\phi_1$ and $\phi_2$, their composition $P_1 \star_\phi P_2$ satisfies $\phi_1 \star_\phi \phi_2$. 
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 \( f_1 \) and \( f_2 \) (boolean operations + registers) are equivalent if \( \text{compare}(x_1,\ldots,x_n) \) equals the infinite sequence \( \text{true}^\omega \).

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

**Verification: a first solution**

- **Compile the function** \( \text{compare} \); if the automaton only contain transitions labelled with \( \text{ok} = \text{true} \), the property is true.

- **Better**: directly produce the minimal automaton for \( \text{compare} \). It should be the **trivial automaton** with only one state and one transition labelled \( \text{ok} = \text{true} \).

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.

```plaintext
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;
```

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 $\text{assert}(H(x, y))$ to states an hypothesis on the environment.

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

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

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.
**Boolean abstraction:** The explicit automaton is infinite (e.g., \( \text{pdiff}=0,1,-1,... \)). Replace numerical comparison by fresh free boolean variables. E.g., \( a_1 \) for \( \text{diff} > 3 \), \( a_2 \) for \( \text{diff} > 1 \), \( a_3 \) for \( \text{diff} < -3 \), \( a_4 \) for \( \text{diff} < -1 \).

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 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”:
  \[ \text{ok} = \neg (\text{late and early}) \];

- “it is impossible to directly pass from late to early”:
  \[ \text{ok} = \text{true} \rightarrow (\neg \text{early and pre late}) \];

- “it is impossible to remain late only one instant”:
  \[
  \begin{align*}
  \text{plate} &= \neg \text{false} \rightarrow \text{pre late}; \\
  \text{pplate} &= \neg \text{false} \rightarrow \text{pre late}; \\
  \text{ok} &= \neg (\neg \text{late and plate and not pplate});
  \end{align*}
  \]

- “if the train keeps the right speed, it stays on time”
  - Naive: \text{assert (sec = bea)}
  - Better: \text{bea and sec alternate}:
    \[
    \begin{align*}
    \text{sf} &= \text{switch(} \text{sec and not bea, bea and not sec} \text{)}; \\
    \text{bf} &= \text{switch(} \text{bea and not sec, sec and not bea} \text{)}; \\
    \text{assume} &= (\text{sf} \Rightarrow \neg \text{sec}) \text{ and } (\text{bf} \Rightarrow \neg \text{bea});
    \end{align*}
    \]
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: $s_{init} \in \mathcal{B}^{|S|}$
- A property (ok): $\phi(s, i)$
- An hypothesis (assertion): $h(s, i)$
- A transition function $g$ such that: $s'_k = g_k(s, i)$

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

• **Accessible states:**
  A state \( \vec{s} \) is accessible w.r.t \( h \) iff there exists a sequence:
  \[
  s_{\text{init}} \xrightarrow{i_1} s_2 \xrightarrow{i_2} \ldots \xrightarrow{i_n} \vec{s}
  \]
  where \( \forall t, h(s_t, \vec{i}_t) \)

  We write this \( \vec{s} \in \text{Reach} \), i.e., \( \text{Reach} = \mu X. (X = \text{init} \cup \text{Post}_h(X)) \).

• **Bad states:** A state \( \vec{s} \) is bad iff:
  \[
  \exists \vec{i}. h(\vec{s}, \vec{i}) \land \neg \phi(\vec{s}, \vec{i})
  \]

  We write this \( \vec{s} \in \text{Error} \), i.e., \( \text{Bad} = \mu X. (X = \text{Error} \cup \text{Pre}_h(X)) \).

• **Post}_h(\vec{s})** is the set of successors of \( \vec{s} \):
  \[
  \text{Post}_h(\vec{s}) = \{ \vec{s}' / \exists \vec{i}. h(\vec{s}, \vec{i}) \land \vec{s} \xrightarrow{i} \vec{s}' \}
  \]

• **Pre}_h(\vec{s})** is the set of predecessors of \( \vec{s} \):
  \[
  \text{Pre}_h(\vec{s}') = \{ \vec{s} / \exists \vec{i}. h(\vec{s}, \vec{i}) \land \vec{s} \xrightarrow{i} \vec{s}' \}
  \]

• **Goal of the proof:** Check that \( \text{Reach} \cap \text{Error} = \emptyset \) or \( \text{Bad} \cap \text{init} = \emptyset \).
Proof by enumeration (forward)

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

while \( \text{reach}\setminus\text{exp} \neq \emptyset \) do
  let \( \vec{s} \in \text{reach}\setminus\text{exp} \) in
  if \( \vec{s} \in \text{Error} \) then raise Stop
  else
    begin
      exp := exp \cup \{\vec{s}\};
      reach := reach \cup 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:** \( x \land \neg y \) = the set of states where \( x \) is true and \( y \) is false.
- \( \phi \) 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 \( \phi_A \) the corresponding formula
- union: \( \phi_{A \cup B} = \phi_A \lor \phi_B \); intersection: \( \phi_{A \cap B} = \phi_A \land \phi_B \)
- 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

\[ f(a, b, c) = a \land f(tt, b, c) \lor \neg a \land f(ff, b, c) \]

**Example:** \( f(a, b, c) = a \lor b \lor c \)
Binary Decision Diagrams

- logical operations
  - \( \lor, \land, \neg, \forall \text{ bounded}, \exists \text{ bounded} \) can be expressed

- symbolic computation of boolean functions

**Basic principle:**

- first choose an order between variables \( x_1 < \ldots < x_n \);

- for every boolean operation \( op \), compute \( \text{build}(op)(b_1, b_2) \)

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

- + 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., \( x \lor \neg y \) 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: \( A = \text{init} \).

Algorithm:

\[
\text{while true do}
\]
\[
\text{if } A \land \text{Error} \neq 0 \text{ then raise Error}
\]
\[
\text{else let } A' = A \lor Post_h(A) \text{ in}
\]
\[
\text{if } A' = A \text{ then raise Success}
\]
\[
\text{else } A := A'
\]

\[
\text{done}
\]

A the end, \( A = A' = \text{Reach} \).
Implementation of $Post_h(X)$

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

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

The same algorithm except that we compute $\text{Pre}_h(X)$.

A BDD $B$ to represent states leading to $Error$ in less than $n$ transitions.

Initially: $B = Error$. Algorithm:

while true do
  if $A \land \text{init} \neq 0$ then raise Error
  else let $B' = B \lor \text{Pre}_h(B)$ 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 $s_1, ..., s_n$.

$$\exists s', v. (X(s') \land h(s, v) \land (\bigwedge_{i=1}^{n} s'_i = g_i(s, v)))$$
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 \( V : \{ x_1, ..., x_n \} \rightarrow \{ 0, 1 \} \) 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 $Path(s[0..k]) = Init(s_0) \land T(s_0, s_1) \land ... \land T(s_{i-k}, s_k)$.
• let $P(s[0..k]) = P(s_0) \land P(s_1) \land ... \land P(s_k)$
• let $Trace(c[0..k])$ be an assignment to variables $s[0..k]$ that make $Init(s_0) \land Path(s[0..k]) \land \neg P(s_k)$ true.

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

It lacks an induction. $k$-induction is the iteration of a BMC [Sheeran, Stalmark and Singh, FMCAD 2000].

\textbf{Basic Principle} Prove it by induction, either in one step, two steps, etc.

- \((\text{init } s_0 \land P(s_0) \land (\forall s. (P(s) \land T(s, s') \Rightarrow P(s')))) \Rightarrow \forall s \in \text{Reach}.P(s)\)

- \((\text{init } s_0 \land P(s_0) \land P(s_1) \land T(s_0, s_1) \land (\forall s. (P(s) \land P(s') \land T(s, s') \land T(s', s'')) \Rightarrow P(s''))) \Rightarrow \forall s \in \text{Reach}.P(s)\)

- etc.

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

Basic algorithm

\[ i := 0; \]
\[ \text{while true do} \]
\[ \quad \text{let } path = \text{Path}(s[0..i]) \text{ in} \]
\[ \quad \text{if } (path \land \neg \forall \text{all.} P(s[0..i])) \text{ then raise Error(Trace(c[0..i]))} \]
\[ \quad \text{else let } base = path \Rightarrow P(s[0..i]) \text{ in} \]
\[ \quad \quad \text{let } ind = path \land P(s[0..i]) \land T(s_i, s_{i+1}) \Rightarrow P(s_{i+1}) \text{ in} \]
\[ \quad \quad \text{if } base \land ind \text{ then raise Success; } \]
\[ \quad i := i + 1; \]
\[ \text{done} \]

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


