Synchronous program verification with Lustre/Lesar

Pascal Raymond

Apr. 2008, rev. Nov 2018

Synchronous approach has been proposed in the middle of the 80's with the aim of reconcile concurrent programming with determinism. The Lustre language belongs to this approach. It proposes a data-flow programming style, close to classical models like block diagrams or sequential circuits. The semantics of the language is formally defined, and thus formal verification of program functionality is possible.
During the 90's, proof methods based on model exploration (model-checking) have been applied with success in domains like protocol or circuit verification. The model-checking techniques are presented in chapter 3. These methods have been adapted for the validation of programs written in the Lustre language, and a specific model-checker (Lesar) has been developed. The first part of this chapter is dedicated to the presentation of the language and the problems raised by its formal verification: expression of properties and assumptions, conservative abstraction of infinite systems etc. The second part is more general and technical. It details some exploration methods for finite state models. These methods are mainly inspired by previous works on the verification of sequential circuits (symbolic model-checking).

1 Synchronous approach

1.1 Reactive systems

The targeted domain is the one of safety critical reactive systems (embedded systems, control/command). The abstract behavior of a typical reactive system is represented on Figure 1:

- it reacts to the inputs provided by its environment by producing outputs;
- it is itself designed as a hierarchical composition of reactive (sub)systems running in parallel.

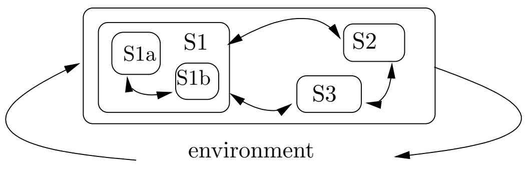


Figure 1: A typical reactive system

1.2 The synchronous approach

A classical approach for implementing a reactive system on a centralized architecture consists in transforming the description-level parallelism into execution-level parallelism:

- atomic processes (or threads) are attached to each sub-system,
- scheduling and communications are realized by the execution platform (in general a real-time oriented operating system).

The main drawbacks of this approach are:

- the overhead due to the executive platform,
- the difficulty to analyze the global behavior of the whole implementation (determinism problem).

The synchronous approach for programming and implementing reactive systems has been proposed to reconcile parallel design with determinism:

- At the design level, the programmer works in an "idealized" world, where both computations and communications are performed in zero-delay (simultaneity between inputs and outputs). He/she can then concentrate only on the *functionality* of the system.
- At the implementation level, the language and its compiler insure that the execution time is bounded; the validity of the synchronous hypothesis can then be validated for a particular hardware platform.

1.3 Synchronous languages

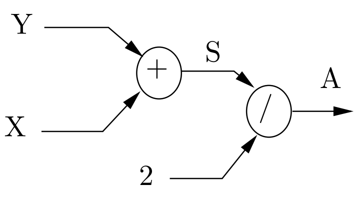
Several languages, based on the synchronous approach, have been proposed:

1. Lustre is a data-flow language; it has been transfered to industry within the tool-suite SCADE[1] tool-suite;
2. Signal[13] is also a data-flow language, which moreover provides a sophisticated notion of clock;
3. Esterel[3] is an imperative language providing "classical" control structures (sequence, loop, concurrency, watch-dog etc). The graphical version is based on hierarchical concurrent automata (SynchCharts [2]).

2 The Lustre language

2.1 Principles

The language [12] adopts the classical data-flow approach, where a system is viewed as a network of operators connected by wires carrying flows of values (Figure 2). Roughly speaking, Lustre is then a textual formalism for describing such diagrams.



```
node Average(X,Y:int)
returns(A:int);
var S:int;
let
   A = S/2;
   S = (X+Y);
tel
```

Figure 2: A data-flow diagram and the corresponding Lustre program

Besides its data-flow style, the main characteristics of the language are the following:

1. Declarative language: the body of a program is a set of equations, whose order is meaningless. This is the substitution principle: if `id = expr` then any occurrence of `id` can be replaced by `expr` without modifying the semantics of the program. For instance, in Figure 2, the intermediate variable `S` can be avoided, so the program is equivalent to the single equation `A = (X+Y)/2` ;
2. Synchronous semantics: the data-flow interpretation is defined according to an implicit discrete time clock. For instance `A` denotes an infinite sequence of integer values $\mathtt{A}_0, \mathtt{A}_1, \mathtt{A}_2, \cdots$. Data operators are operating point-wise on sequences; for instance, `A = (X+Y)/2` means that $\forall t \in \mathbb{N}$, $\mathtt{A}_t = (\mathtt{X}_t + \mathtt{Y}_t)/2$;
3. Temporal operators: in order to describe complex dynamic behaviors, the language provides the `pre` operator (for previous), which shifts flows forward one instant: $\forall t \geq 1 \;\; (\mathtt{pre\ X})_t = \mathtt{X}_{t-1}$, and $(\mathtt{pre\ X})_0 = \perp$. The `pre` operator comes with an initialization operator in order to avoid

[1] `http://www.esterel-technologies.com/products/overview.html`.

```
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
```

Figure 3: The beacon counter

undefined values: $(\mathtt{X\ ->\ Y})_0 = \mathtt{X}_0$ and $\forall t \geq 1\ (\mathtt{X\ ->\ Y})_t = \mathtt{Y}_t$. For instance, `N = 0 -> pre N + 1` defines the sequence of positive integers $(0, 1, 2, 3, \cdots)$;

4. Dedicated language: the language is specifically design to program reactive kernels, not for programming complex data transformation. Basic data types are Boolean (`bool`), integers (`int`) and floating-point values (`real`). Predefined operators are all the classical logical and arithmetics operators. More complex data types, together with the corresponding operators, are kept abstracted in Lustre, and must be implemented in an host language (typically C) [2];

5. Modularity: once defined, a program (called a `node`) can be reused in other programs, just like a predefined operator.

Note that the language, reduced to its Boolean subset, is equivalent to the classical formalism of sequential circuits: (`and`, `or`, `not`) are the logical gates, and the construct `false -> pre` is similar to the notion of register.

2.2 Example: the beacon counter

Before presenting the example, we first define a intermediate node which is particularly useful as a basic brick for building control systems: it is a simple flip/flop system, initially in the state `orig`, and commanded by the buttons `on` and `off`:

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

A train runs on a railway where beacons are deployed regularly. Inside the train, a beacon counter receives a signal each time the train crosses a `bea`con, and also a signal `sec`ond coming from a real time clock.

Knowing that the cruising speed should be *one beacon per second*, the program must compute whether the train is `early`, `on time` or `late`. An hysteresis mechanism prevents oscillation artefacts (dissymmetry between state change conditions).

The corresponding Lustre program is shown on 3.

3 Program verification

The methods presented here are indeed not specific to the Lustre language: they can applied to any formalism relying on a discrete time semantics. This is in particular the case for the other synchronous languages (Esterel, Signal etc), and also to any formalism close to the one of sequential circuits.

[2]This remark mainly concerns the basic Lustre language ; recent versions, in particular the one form the SCADE tool, are much more rich.

3.1 Notion of temporal property

Our goal is functional verification: does the program computes the "right" outputs ? In other terms, the goal is to check whether the program satisfies the expected properties. Since we consider dynamic systems, the expected properties are not simply relations between instantaneous inputs and outputs, but rather between inputs and outputs sequences, thus the term of *temporal properties*.

3.2 Safety and liveness

It exists a whole theory on the classification of temporal properties (and the corresponding logics [14]). For our purpose, we simply need to recall the "intuitive" distinction between:

- *safety* properties, expressing that something (bad) never happens ;
- *liveness* properties, expressing that something (good) may or must happens.

3.3 Beacon counter example

Here are some properties that one may expect form the system:

- It is never both early and late.
- It never switches directly from late to early (and conversely).
- It is never late for a single instant.
- If the train stops, it eventually becomes late.

The first three properties are *safeties*, while the last one is clearly a typical *liveness*.

3.4 State machine

The abstract behavior of a Lustre program (more generally, of a synchronous program) is the one of a deterministic *state machine* (Figure 4):

- the initial state of the memory M is known;
- the values of the outputs, together with the ones of the next state, are functionally defined according to the current values of the inputs and the memory: $f : IM \rightarrow O$ is the output function, $g : IM \rightarrow M$ is the transition function.

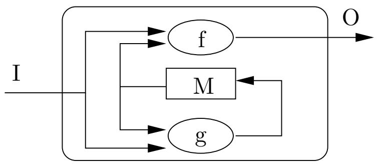


Figure 4: State machine

3.5 Explicit automata

A state machine is also a implicit automata: it is equivalent to a state/transition system (automaton) where each state is a particular configuration of the memory.
Figure 5 shows a small part of the beacon counter automaton. States are representing the values of the memory: the value of `pre diff` is the integer inside the state; the values of the tree Boolean memories (`pre(ontime)`, `pre(early)` and `pre(late)`) are implicitly represented by the position of the state:

- in the top states, `pre(ontime)` is true and the other are false,
- in the bottom-right states, `(pre early)` is true, and the other are false,
- in the bottom-left states, `(pre late)` is true, and the other are false.

Moreover, transitions are guarded by conditions on inputs that are not represented for the sake of simplicity:

- "`bea and not sec`" guards all the transitions going from left to right,
- "`not bea and sec`" guards all the transitions going from right to left,

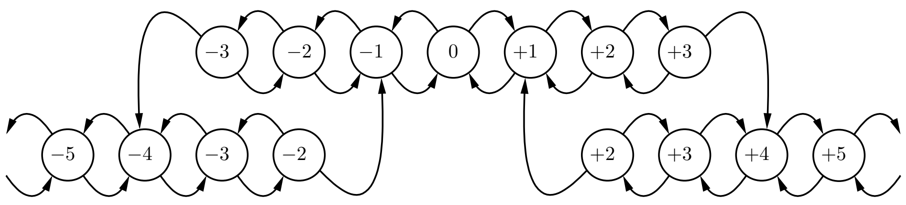


Figure 5: Explicit states of the beacon counter

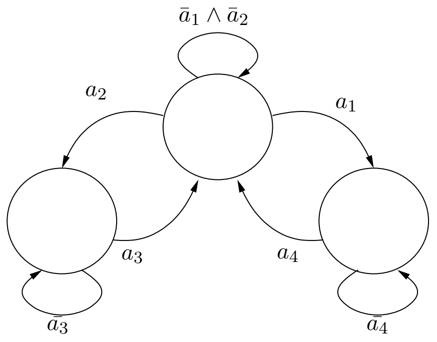
- moreover, there is a loop transition on each state guarded by the condition “`bea = sec`”.

3.6 Principles of model-checking

The explicit automata is a model which synthesizes all the possible behaviors of the program. The exploration of this model allows to study the functional properties of the program. But, in general, the automaton is infinite, or at least enormous, and an exhaustive exploration is impossible. The idea is then to consider a finite (and not too big) approximation of the automaton.
Such an approximation is obtained by forgetting informations: it is an *abstraction* of the program. This abstraction must be conservative for some class of properties, otherwise it would be useless.

3.7 Example of abstraction

The Boolean abstraction consist in abstracting away all computations but the logical ones, that is, for Lustre programs, everything that concerns numerical variables. In the example, it consists in ignoring the internal counter `diff`, and introducing free logical variables at the “frontier” between Boolean and numerical values (Figure 6).



- a_1 replaces `diff > 3`,
- a_2 replaces `diff < -3`,
- a_3 replaces `diff >= -1`,
- a_4 replaces `diff <= 1`,

Figure 6: A Boolean abstraction of the beacon counter

This abstraction does not represent exactly the behaviors of the program, but an over approximation: some behaviors are possible on the abstraction, while they were impossible on the concrete program. Nevertheless, some properties are still satisfied:

- impossible to be both early and late (safety);
- impossible to pass directly from late to early (safety);
- if the train stops, it eventually becomes late (liveness).

Other properties are lost:

- impossible to remain late for a single instant (safety).

More seriously, the converse of this property, not satisfied by the program, has been introduced by the abstraction:

- it is possible to remain late for a single instant (liveness).

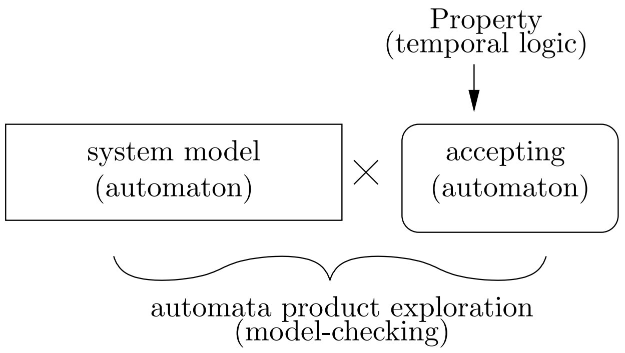


Figure 7: Classical model-checking scheme

It is then very important to precise which conclusions one can take from the study of the abstraction.

3.8 Conservative abstraction and safety

The Boolean abstraction is a particular case of over-approximation: it may introduce new behaviors, but not remove behaviors form the concrete program. As a conclusion, if a behavior does not exist on the abstraction, it certainly does not exist on the real program.
It is then easy to conclude that any over-approximation is conservative for the class of safety properties: safety properties may be kept or lost, but cannot be introduced.
Over-approximation is in fact also conservative for certain subclasses of liveness properties; this is the case for properties stating that an event eventually happens. This kind of properties is often called "unbounded liveness", since the event may occur anytime in the future.
On the contrary, over-approximation is clearly not conservative for liveness properties stating that some behavior is possible.

4 Expressing properties

4.1 Model-checking general scheme

Model-checking is a important domain, with an important literature (see chapter 3). The application domains are (historically) communication protocols and logical circuits. In the case of non-deterministic, deadlocking systems, reasoning in terms of behavior sets is not sufficient: it is necessary to reason in terms of computation trees. This case, related to Computation Tree Logic (CTL), is not presented here. We only consider here the case where the semantics can be expressed in terms of behavior sets (related to Linear Temporal Logic, LTL).
The classical model-checking principle is shown on Figure 7.

- A model of the system is provided, which comes from a high level specification (in the case of protocols), or which has been automatically obtained from a concrete implementation (in the case of logical circuits).
- The expected property is expressed in a particular logic (typically LTL).
- The LTL formula is translated into an operational form, typically a automaton with some accepting criterion. In the case of unbounded liveness, a criterion for accepting infinite, non monotonic sequences is necessary (Bchi automata).
- Model-checking itself consists in exploring the product of the two automata for verifying that any sequence that can be generated by the model is accepted by the formula.

4.2 Model-checking synchronous program

The classical scheme is adapted and simplified for the case of synchronous programs. First, liveness properties are not treated at all, for both theoretical, pragmatic and practical reasons:

- as explained previously, abstractions are required that do not conserve the whole class of liveness properties: the sub-class expressing potentiality cannot be model-checked at all;

- abstraction is conservative for unbounded liveness (expressing eventuality). However, from a pragmatic point of view, such a liveness mat seems too "vague". Moreover, it is usually possible to replace it by a somehow more precise property: for instance, "an alarm is eventually raised in case of problem", can be replaced by "an alarm is raised within 20 second in case of problem", which is a proper safety property.
- at last, from a practical point of view, handling liveness properties requires sophisticated formalism and models (temporal logics, Bchi automata). On the contrary, safety properties (that are nothing than classical program invariants) can be *programmed*, as explained in the next section.

For all these reasons, we restrict ourselves to the verification od safety properties.

4.3 Observers

Since only safety properties are considered, it is not necessary to use (complex) temporal logics for expressing properties: one can express properties by programming a suitable *observer*. Intuitively, an observer is a synchronous program taking as inputs the variables of the system to check, and producing a Boolean output which remains true as long as the expected property is satisfied. If one can establish that this output remains true whatever be the input sequence, then the underlying safety property is proved. An argument in favor of observers is that they can be programmed in the same language than the system to check, and thus, it is not necessary to learn new formalisms. This pragmatic argument is very important for the diffusion of formal methods in industry. Note that it is completely equivalent to observe failures: in this case, the observer outputs an alarm when the property is violated, and the model-checking consists in verifying that the alarm is never emitted, whatever be the inputs sequence.

4.4 Examples

Here are some safety properties expressed in Lustre. By convention, the observer result is called `ok`.

1. impossible to be early and late:
   ```
   ok = not (early and late).
   ```
2. impossible to pass directly form late to early:
   ```
   ok = true -> not ((pre late) and early).
   ```
3. impossible to remain late for a single instant:
   ```
   ok = not Pre(Pre(late)) and Pre(late) => late
   ```
   where `Pre(x) = false -> pre x`.

4.5 Hypothesis

In general, a program is not supposed to work properly if the environment does not satisfies itself some properties. Moreover, it is rather natural to decompose a specification according to the classical duality cause/consequence.

For instance, the following specification: "if the train keeps the cruising speed, it remains on time", can be split according to the cause/consequence scheme:

- the property is simply `ok = ontime`,
- the hypothesis (keeps the cruising speed) is a little bit more complicated. A naive version could be that that beacons and seconds are exactly synchronous (`hypothesis = (sec = bea)`; a more sophisticated (and realistic) one is that beacons and seconds are alternating (`hypothesis = alternate(sec,bea)`, where `alternate` is an accessory program:

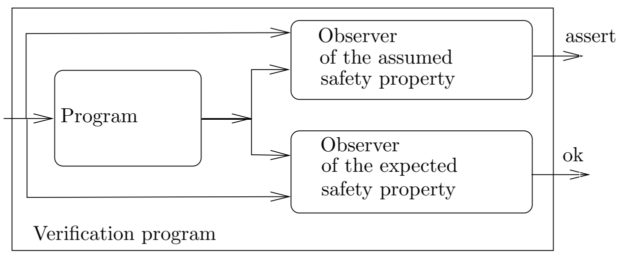


Figure 8: Verification program

```
node alternate(x,y: bool) returns (ok: bool);
var forbid_x, forbid_y : bool;
let
   forbid_x = switch(false, x, y);
   forbid_y = switch(false, y, x);
   ok = not(x and forbid_x or y and forbid_y);
tel
```

Note that one practical advantage of using observers is the ability of defining libraries of generic observers, that can themselves be formally validated.

4.6 Model-checking of synchronous programs

Figure 8 shows the principle of the verification with observers. This principle is used in several verification tools: the Lustre model-checker (Lesar), the Esterel model-checker, and also in the SCADE integrated verification tool.
The user may provide a verification program made of:

- the program under verification itself,
- an observer of the expected property,
- an observer of the assumed property (hypothesis); in Lesar, such assumptions are given through the special construct `assert`.

Given this verification program, the role of the model-checker consists in:

- automatically extracting a finite abstraction (typically a Boolean one, as explained before),
- exploring this abstraction in order to establish that, whatever be the inputs sequence, if `assert` remains true, then `ok` remains true too.

In terms of temporal logics, the actual property to check is:

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

This kind of property is not a proper safety, since the negation of the premise is actually a liveness. However, this very special case (invariant implies invariant) does not raise theoretical problems: finite abstraction is also conservative for this kind of property.
The problem is technical, since checking such a property is much more costly than checking a simple invariant. Intuitively, a first computation is necessary to remove from the model all paths that eventually lead to the violation of the assumption. Then, a second exploration is necessary to check whether `ok` remains true on this restricted model.
This is why we make another (conservative) abstraction, consisting in approximating the exact property by:

"always(`so_far_assert` $\Rightarrow$ `ok`) "

Where `so_far_assert` expresses that `assert` *has never been false*, and can be programmed in Lustre:

```
so_far_assert = assert and (true -> so_far_assert);
```

5 Algorithms

5.1 Boolean automaton

Whatever be the high-level programming language, the result of the Boolean abstraction can be defined as a set of logical equations.
A verification program is characterized by:

- a set of free variables (inputs)) V and a set of state variables (registers) S ;
- the initial states are characterized by a condition $Init : \mathbb{B}^{|S|} \rightarrow \mathbb{B}$ (in general, there is a unique initial state, but it has no importance here);
- a set of transition functions $g_k : \mathbb{B}^{|S|}\mathbb{B}^{|V|} \rightarrow \mathbb{B}$, one for each register $k = 1 \cdots |S|$;
- the assumption $H : \mathbb{B}^{|S|}\mathbb{B}^{|V|} \rightarrow \mathbb{B}$;
- and the property $\Phi : \mathbb{B}^{|S|}\mathbb{B}^{|V|} \rightarrow \mathbb{B}$.

5.2 Explicit automaton

The Boolean automaton "encodes" an explicit state/transition system:

- the set of states is $Q = \mathbb{B}^{|S|}$ (all the possible values of the memory);
- initial states are the subset characterized by $Init \subseteq Q$ (we safely mix up sets and characteristic functions);
- the transition relation $R \subseteq Q\mathbb{B}^{|V|}Q$ is defined by:

$$(q, v, q') \in R \quad \Leftrightarrow \quad q'_k = g_k(q, v) \quad k = 1 \cdots |S| \ .$$

  We note $q \xrightarrow{v} q'$ for $(q, v, q') \in R$. In our case, R is in fact a function: q' is unique for any pair (q, v). This is why in the sequel we call "transitions" the pairs from $T = Q\mathbb{B}^{|V|}$.
- $H \subseteq T$ is the subset of transitions satisfying the assumption;
- $\Phi \subseteq T$ is the subset of transitions satisfying the (expected) property.

5.3 The "pre "and "post "functions

In order to simplify the problem, we will reason only in terms of states, by defining the post functions (states that can be reached from somewhere by satisfying the assumption), and the pre functions (states that can lead to somewhere by satisfying the assumption).
For any state $q \in Q$, and any set $X \subseteq Q$ we define:

- $post_H(q) = \{q' \mid \exists v \; q \xrightarrow{v} q' \wedge H(q, v)\}$,
- $POST_H(X) = \bigcup_{q \in X} post_H(q)$,
- $pre_H(q) = \{q' \mid \exists v \; q' \xrightarrow{v} q \wedge H(q', v)\}$,
- $PRE_H(X) = \bigcup_{q \in X} pre_H(q)$.

5.4 Outstanding states

Starting from initial states Init, we define the set of accessible states as the following fix-point:

$$\text{Acc} \;=\; \mu X \;\cdot\; (X = \text{Init} \cup POST_H(X)) \ ,$$

that is, the smallest set containing initial states and kept invariant by the function $POST_H$.
Immediate error states are those such that at least one of the outgoing transition satisfies the assumption but violates the property:

$$\text{Err} \;=\; \{q \mid \exists v \; H(q, v) \wedge \neg\Phi(q, v)\} \ ,$$

By extension, we define the set of "bad" states as those that may lead to an error state:

$$\text{Bad} \;=\; \mu X \;\cdot\; (X = \text{Err} \cup PRE_H(X)) \ ,$$

that is, the smallest set containing (immediate) error states and kept invariant by the function PRE_H.
In the sequel, we note $\text{Acc}_0 = \text{Init}$ and $\text{Bad}_0 = \text{Err}$.

5.5 Principles of the exploration

The goal of the proof is to establish (if possible) that $\mathrm{Acc} \cap \mathrm{Bad} = \emptyset$, that is, no accessible state is also a bad state.
Indeed, it is useless to compute the two fix-points for that, one may try to establish indiscriminately:

- that $\mathrm{Acc} \cap \mathrm{Bad}_0 = \emptyset$ (*forward* method),
- that $\mathrm{Bad} \cap \mathrm{Acc}_0 = \emptyset$ (*backward* method).

In the case of deterministic systems, the two methods are not symmetric: the transition relation is a function (forward method), but its reverse relation has no special property in general (backward method).

6 Enumerative algorithm

The forward enumerative algorithm (Figure 9) is a typical example of fix-point calculus.
Starting form the initial state(s), reachable states are explored one by one, checking for each of them the validity of Φ. If the exploration converges without errors, the property is proved.

```
Known:= Init;     Explored:= ∅
while it exists $q$ in Known \ Explored {
        for all $q'$ in post($q$) {
            if $q' \in$ Err then Output(failure)
            move $q'$ to Known
        }
        move $q$ to Explored
}
Output(success)
```

Figure 9: Enumerative algorithm

Figure 10 pictures, in the left-hand side, an exploration that has succeed: the exploration has converged without reaching any state in Err. In the right-hand side, a state from Err is reached during the exploration and the proof fails.

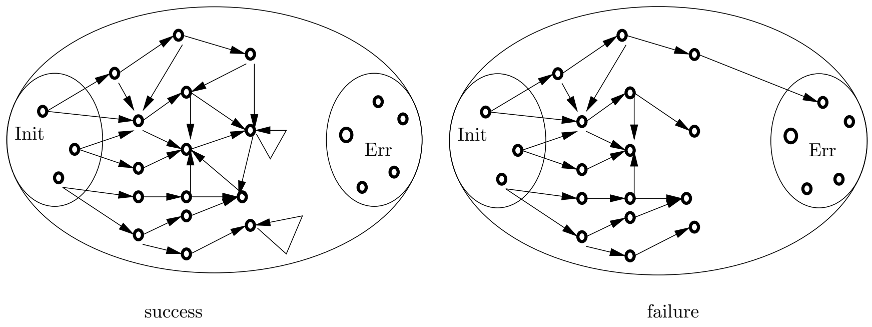


Figure 10: Forward enumeration exploration

In a concrete implementation, a lot of work is necessary to make the algorithm efficient (or at least to limit its inefficiency!).
As a matter of fact, the theoretical complexity is really huge: the number of reachable states is potentially exponential ($2^{|S|}$), and, for each state, an exponential number of transitions must be explored ($2^{|V|}$). Enumerative methods are then limited to relatively small systems (typically a few tens of variables).
By exchanging roles of Init and Err, and replacing *POST* by *PRE*, we trivially obtain a bacward enumerative algorithm. However, such an algorithm is never used because it is likely to be even more inefficient. The problem is related to determinism: forward enumeration mainly consists in applying

functions, while enumerating backward consists in solving relations.

7 Symbolic methods and binary decision diagrams

Symbolic model-checking methods [8, 10, 11, 16] have been proposed during the 90's. They are mainly due to:

- the idea to overpass the limits due to state explosion by adopting implicit representations rather than explicit ones;
- the (re)discover of *Binary Decision Diagrams* (BDD), allowing to (often consisely) represent logical predicates, and operating on them ([1, 7]).

7.1 Notations

Thereafter, f and g denote predicates (i.e. Boolean functions) and :

- 0 denotes the always-false predicate, 1 the always-true one;
- we note either $f \cdot g$ or $f \wedge g$ for the conjunction (logical and);
- we note either $f + g$ or $f \vee g$ for the disjunction (inclusive or);
- $f \oplus g$ represents the difference (exclusive or);
- $\neg f$ denotes the negation, which is also denoted by $\bar{x}$ when the predicate is reduced to a single variable x.

A tuple (of variables, of functions etc) is represented either

- as an explicit sequence $\vec{x} = [x_1, \cdots, x_n]$;
- or as a recursive list $\vec{x} = x_1 :: \vec{x}'$, when the goal is simply to isolate the first element x_1 from the others $\vec{x}'$. The empty tuple/list is denoted by [].

7.2 Handling predicates

Any set of states or transitions can be represented implicitly by a predicate: its characteristic function. For instance, if x, y, z are state variables, the predicate $(x \oplus y).\bar{z}$ encodes all the states where z is false and where either x or y (but not both) is true. If these three variables are the only ones, this set contains 2 elements from $2^3 = 8$ possible states ($(0, 1, 0)$ and $(0, 0, 1)$).
But, supposing that there exist n other variables, whose values are not relevant, then the same predicate encodes a set with $2 * 2^n$ elements. As a consequence, one can consider that the predicate is a concise symbolic representation of the underlying set.
Operations on sets can obviously be defined in terms of predicates: union is isomorphic to disjunction, intersection to conjunction, complement to negation.
It is then possible to define exploration algorithms that directly handle sets (of states, of transitions). In order to implement such algorithms, it is necessary to define some appropriate data-structure for predicates.
In particular, such a data structure must make it possible to check the emptiness of a set, or, equivalently, the equality of two sets. In terms of logics, it is equivalent to check the satisfiability of a predicate, which is nothing but the archetype of the NP-complete decision problem. In other terms, no algorithm is known that solves the problem with a sub-exponential cost.
As a consequence we can already conclude that symbolic methods will be limited to somehow “simple” class of systems. However, it is clear that this class includes is likely to be much more larger than the one that can be treated with enumerative methods.

7.3 Representation of the predicates

7.3.1 The Shannon's decomposition

Any Boolean function $f(x, y, \cdots, z)$ can be decomposed according to the value of x into a pair of functions, f_x and $f_{\bar{x}}$, not depending on x:

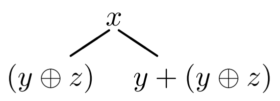
$$f(x, y, \cdots, z) \quad = \quad x.f_x(y, \cdots, z) \; + \; \bar{x}.f_{\bar{x}}(y, \cdots, z)$$

$$\begin{aligned}\text{with: } f_x(y,\cdots,z) &= f(1,y,\cdots,z)\\ \text{and: } f_{\bar{x}}(y,\cdots,z) &= f(0,y,\cdots,z)\end{aligned}$$

For instance, for $f(x,y,z) = x \cdot y + (y \oplus z)$, one gets:

$$\begin{aligned}f(x,y,z) &= x \cdot f(1,y,z) + \bar{x} \cdot f(0,y,z)\\ &= x \cdot (1 \cdot y + (y \oplus z)) + \bar{x} \cdot (0 \cdot y + (y \oplus z))\\ &= x \cdot (y + (y \oplus z)) + \bar{x} \cdot (y \oplus z)\end{aligned}$$

This decomposition can be represented by a binary tree whose root is labelled by the variable, and whose right branch leads to f_x (called the high branch), while its left branch leads to $f_{\bar{x}}$ (called the low branch):



By recursively applying the decomposition for all the variables, one gets a complete binary tree whose leaves are not depending in any variable, that is, the constants 1 (always true) or 0 (always false). Figure 11 shows the Shannon's tree of f.

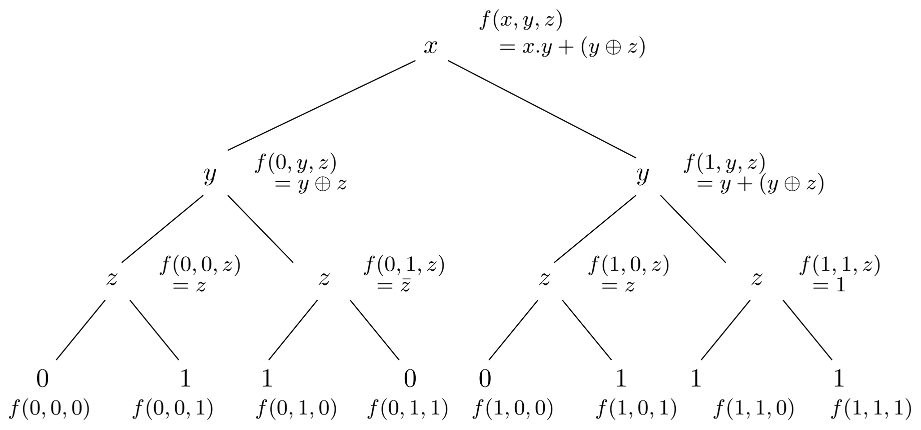


Figure 11: The Shannon's tree of $f(x,y,z) = x.y + (y \oplus z)$

For a fixed variable order, the complete binary tree is a canonical representation of the function, which means that any function semantically equivalent to f gives the same tree. For instance, $g(x,y,z) = y.(x + \bar{z}) + \bar{y}.z$ is syntactically different from f, but leads to the same Shannon tree when decomposed according to the same variable order: it is then is fact the same function.

7.3.2 Binary Decision Diagrams

Some nodes in the Shannon's tree are useless: this is the case for any binary node whose high and low branches are leading to isomorphic sub-trees. Such a node can be eliminated by directly connecting the sub-tree to its father node. By eliminating all useless node, one obtains the "reduced Shannon tree", which is still a canonical representation of the corresponding function.
At last, isomorphic sub-trees can be shared in order to obtain a directed acyclic graph (DAG), as shown in Figure 12. This graph is called the "Binary Decision Diagram" of the function, for the considered variable order.

7.4 Typical interface of a BDD library

Figure 12 illustrates the principle of BDD, but in no case the way BDDs are actually built: it is not necessary to build an exponential binary tree as an intermediate structure for a graph that we expect

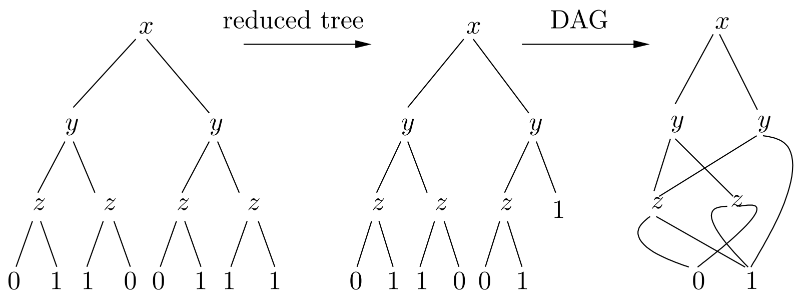


Figure 12: From Shannon's tree to BDD

to be concise! In practice, BDDs operations are available through an abstract interface allowing the user to manipulate predicates without considering the details of the implementation (in particular the variable order). A typical interface provides:

- the equivalence (which is actually the equality),
- constant functions (leaves 0 and 1),
- identity, for all x,
- all the classical logical operators (and, or, not, etc.),
- quantifiers, which, in the Boolean case, can be defined in terms of logical operations, for instance: $\exists x : f(x, y, \cdots, z) = f(1, y, \cdots, z) + f(0, y, \cdots, z)$.

7.5 Implementation of BDDs

Internally, a BDD library handles a set of variables $\mathcal{V}$, totally ordered by a a relation $\leq$; we classically note $<$ (respectively $\geq$) the induced strict relation (respectively the reverse relation). The set of BDDs over $(\mathcal{V}, <)$ is denoted $\mathcal{B}$. In order to define it formally, the structure $(\mathcal{V}, <)$ is extended with a special value ∞ such that $\forall x \in \mathcal{V} : x < \infty$. B is defined together with the function *range* : $\mathcal{B} \rightarrow \mathcal{V} \cup \{\infty\}$ which, intuitively, gives the least significant variable appearing in a BDD, or ∞ in the case of a leaf. The set of BDDs contains:

- the leaves 0 and 1, with $range(0) = range(1) = \infty$,
- the binary graphs $\gamma = \overset{x}{\alpha \diagup\diagdown \beta}$, such that $x < range(\alpha)$ and $x < range(\beta)$, with $range(\gamma) = x$.

The internal procedure builds, if necessary, a binary node, given one variable and two existing BDDs. A call to this procedure is represented by $\overset{x}{\alpha \,/\!\!/\backslash\!\!\backslash\, \beta}$, in order to distinguish it from actual nodes that will be represented with simple lines:

- $\overset{x}{\alpha \,/\!\!/\backslash\!\!\backslash\, \beta}$ undefined if $x \not< range(\alpha)$ or $x \not< range(\beta)$;
- $\overset{x}{\alpha \,/\!\!/\backslash\!\!\backslash\, \alpha} = \alpha$ (never build useless node);
- otherwise, $\overset{x}{\alpha \,/\!\!/\backslash\!\!\backslash\, \beta} = \overset{x}{\alpha \diagup\diagdown \beta}$.

7.6 Operations on BDDs

7.6.1 Negation

Building the negation is achieved by a simple recursive descent:

- case of leaves: $\neg 1 = 0$ et $\neg 0 = 1$,
- case of a binary node: $\neg \overset{x}{\alpha_0 \diagup\diagdown \alpha_1} = \overset{x}{\neg\alpha_0 \,/\!\!/\backslash\!\!\backslash\, \neg\alpha_1}$.

7.6.2 Binary operators

All classical operators are distributing over the Shannon's decomposition: let $\star$ be any operator (among $.$, $+$, $\oplus$, $\Leftrightarrow$):

$$\begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array} \star \begin{array}{c}x\\ \diagup\ \diagdown\\ \beta_0\quad\beta_1\end{array} = \begin{array}{c}x\\ /\!\!/\quad\backslash\!\!\backslash\\ \alpha_0\star\beta_0\ \ \alpha_1\star\beta_1\end{array}$$

If the arguments have different root variables, a "balance" should be performed in the recursive descent in order to meet the range constraints. If some variable $x < \mathit{range}(\beta)$ does not appear within a β, that means that β is (virtually) equivalent to $\begin{array}{c}x\\ /\!\!/\ \backslash\!\!\backslash\\ \beta\quad\beta\end{array}$.

Let $\alpha = \begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}$ and $\beta = \begin{array}{c}y\\ \diagup\ \diagdown\\ \beta_0\quad\beta_1\end{array}$ be two BDDs with $x \neq y$::

- if $x < y$, $\alpha \star \beta = \begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array} \star \begin{array}{c}x\\ /\!\!/\ \backslash\!\!\backslash\\ \beta\quad\beta\end{array} = \begin{array}{c}x\\ /\!\!/\quad\backslash\!\!\backslash\\ \alpha_0\star\beta\ \ \alpha_1\star\beta\end{array}$,
- if $y < x$, $\alpha \star \beta = \begin{array}{c}y\\ /\!\!/\ \backslash\!\!\backslash\\ \alpha\quad\alpha\end{array} \star \begin{array}{c}y\\ \diagup\ \diagdown\\ \beta_0\quad\beta_1\end{array} = \begin{array}{c}y\\ /\!\!/\quad\backslash\!\!\backslash\\ \alpha\star\beta_0\ \ \alpha\star\beta_1\end{array}$.

These generic rules must be completed by terminal rules specific to each particular operator, for instance:

- $\alpha + 0 = 0 + \alpha = \alpha$ and $\alpha + 1 = 1 + \alpha = 1$,
- $\alpha \cdot 0 = 0 \cdot \alpha = 0$ and $\alpha \cdot 1 = 1 \cdot \alpha = \alpha$,
- $\alpha \oplus 0 = 0 \oplus \alpha = \alpha$ and $\alpha \oplus 1 = 1 \oplus \alpha = \neg\alpha$, etc.

Since the representation is canonical, it is also possible (and better) to exploit the known properties of the operators, for instance, for any BDD α:

- $\alpha + \alpha = \alpha$, $\alpha \cdot \alpha = \alpha$, $\alpha \oplus \alpha = 0$.

7.6.3 Cofactors and quantifiers

Instantiating a variable x within a BDD α with the value true or false is achieved by computing the cofactors $\mathit{high}(x, \alpha)$ or $\mathit{low}(x, \alpha)$:

- $\mathit{high}(x, \alpha) = \mathit{low}(x, \alpha) = \alpha$ if $x < \mathit{range}(\alpha)$, i.e. x does not appear within α ;
- $\mathit{high}\left(x, \begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}\right) = \alpha_1$ and $\mathit{low}\left(x, \begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}\right) = \alpha_0$;
- at last, if $y < x$, $c\left(x, \begin{array}{c}y\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}\right) = \begin{array}{c}y\\ /\!\!/\quad\backslash\!\!\backslash\\ c(x,\alpha_0)c(x,\alpha_1)\end{array}$, whatever be the cofactor c (*high* or *low*).

Quantifiers can be defined in terms of cofactors:

- $(\exists x : \alpha) = \mathit{high}(x, \alpha) + \mathit{low}(x, \alpha)$ et
- $(\forall x : \alpha) = \mathit{high}(x, \alpha) \cdot \mathit{low}(x, \alpha)$.

However, for the sake of efficiency, it is better to propose a more direct algorithm, based on a single recursive descent over α. We only give the algorithm for $\exists$, the one for $\forall$ is similar:

- $(\exists x : \alpha) = \alpha$ if $x < \mathit{range}(\alpha)$,
- $\left(\exists x : \begin{array}{c}x\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}\right) = \alpha_0 + \alpha_1$
- at last, if $y < x$ $\left(\exists x : \begin{array}{c}y\\ \diagup\ \diagdown\\ \alpha_0\quad\alpha_1\end{array}\right) = \begin{array}{c}y\\ /\!\!/\quad\backslash\!\!\backslash\\ (\exists x : \alpha_0)(\exists x\ \alpha_1)\end{array}$.

7.7 Notes on the complexity

The rules presented in the previous sections are corresponding to high level algorithms. An efficient implementation must use some memory cache mechanism in order to store (and retrieve) intermediate results.

If we suppose this memory cache perfect (no intermediate result is lost, and the overhead can be neglected), we can give a theoretical bound for the cost of operations on BDDs.

The complexity mainly depend on the size of the arguments, expressed in number of BDD nodes (that is, taking into account the sharing of common sub-graphs). The size is noted $|\alpha|$.
The complexity of a binary operation between α and β is trivially bounded by $|\alpha| * |\beta|$: if the memory cache is perfect, each pair (node from α, node from β) is visited at most once.
For a quantification, for instance $\exists x : \alpha$, a rough approximation gives $|\alpha|^3$: for any occurrence of the variable x (roughly bounded by $|\alpha|$), a binary operation is performed between the corresponding branches (each of them being also roughly bounded by $|\alpha|$).
The bounds presented here are obviously very pessimistic, however, they are sufficient to state that the complexity of one operation is of polynomial magnitude.
This results holds for one single operation, and then, it is not meaningful for a typical use of the BDDs. As a matter of fact, the typical use of a BDD library consists in building the BDD of predicate given as classical Boolean expression. Let e be such an algebraic expression, and n the number of operators appearing in it (and, or, not, exists etc). The whole BDD construction cost is exponential in n in the worst case. The worst case is reached, for instance, with the following expression, with $n = 2k$:
$(x_1 \oplus x_{k+1}) \cdot (x_2 \oplus x_{k+2}) \cdot ... \cdot (x_{k-1} \oplus x_{n-1}) \cdot (x_k \oplus x_n)$
With the variable order $x_1 < x_2 < \cdots < x_n$, it is easily established that the size of the BDD is of order 2^n (exactly $3(2^k - 1)$).
This example also illustrates the dramatic influence of the variables ordering: the same expression with the ordering $x_1 < x_{k+1} < x_2 < x_{k+2} < ... < x_k < x_n$ gives a BDD whose size is linear in n (exactly $3(k - 1)$).
For any expression, some optimal ordering exists, which gives a BDD of minimal size. However, finding such optimal ordering is in practice intractable because of the complexity.
Efficient BDDs implementations only try to avoid "clearly bad orderings": if the memory size grows too fast at run-time, it is probably because the current ordering is bad chosen. The BDDs manager thus tries to modify the order in order to make the memory consuming decrease. This king of techniques is known as dynamic reordering ([15]).

7.8 Typed decision diagrams

Efficient implementations are using a "trick" that allows to share the graphs corresponding to opposite functions f and $\neg f$.
Intuitively, only one of these functions is actually represented in the memory, while the other is defined, thanks to a single sign bit, to be its opposite. The first definition of this method is, as far as we know, due to J.P. Billon (*Typed Decision Graph* or TDG [5, 6]).

7.8.1 Positive functions

In order to guarantee canonicity, the set of logical functions $\mathcal{F}$ (whatever be the arity) is split into:

- the set of positive functions, $\mathcal{F}^+$, which evaluate to true when all their arguments are true,
- the set of negative functions, $\mathcal{F}^-$, which evaluate to false when all their arguments are true.

For instance, the always-true function 1 is positive, and so are all the functions x, $x + y$, etc. On the contrary, 0, $\bar{x}$, $x \oplus y$, are examples of negative functions.

7.8.2 TDG

Typed decision graphs are a representation exploiting the partition between positive and negative functions.
TDG are pairs (a, α), where a is the sign $\in \{+, -\}$, and α is a binary decision graph corresponding to a positive function. We use the schematic notation $\overset{|+}{\alpha}$ or $\overset{|-}{\alpha}$.
Within a graph, edges are also labelled with a sign, according to the following rules:

- the unique leaf is the always-true function 1;
- binary nodes are either of the form $\begin{array}{c} x \\ {}^{+}/\ \backslash^{+} \\ \alpha_0 \quad \alpha_1 \end{array}$, or $\begin{array}{c} x \\ {}^{-}/\ \backslash^{+} \\ \alpha_0 \quad \alpha_1 \end{array}$, where α_1 and α_0 are themselves positive function graphs. Range rules, related to the variable ordering, are the same as those of classical

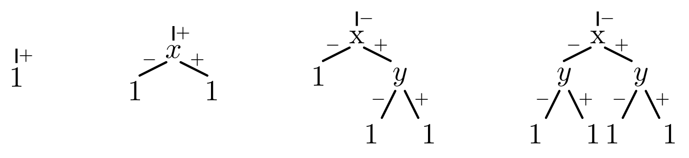


Figure 13: From left to right, TDGs of 1, x, $x \cdot \bar{y}$, $x \oplus y$.

BDDs.
Figure 13shows some TDG examples, for the ordering $x \prec y$ (sub-graph sharing is not represented for the sake of readability).

7.8.3 TDG implementation=

Handling TDGs only requires a negligible overhead in terms of internal operations. All the rules defined for classical BDDs are still valid modulo a simple decoding/encoding of the sign bit:

- $\overset{|+}{1}$ is interpreted as 1, $\overset{|-}{1}$ as 0;
- $\begin{array}{c} |+ \\ {}_{+}\!\diagup\!\overset{x}{}\!\diagdown_{+} \\ \alpha_0 \quad \alpha_1 \end{array}$ is equivalent to $\begin{array}{c} x \\ /\ \backslash \\ \alpha_0 \quad \alpha_1 \end{array}$ and $\begin{array}{c} |+ \\ {}_{-}\!\diagup\!\overset{x}{}\!\diagdown_{+} \\ \alpha_0 \quad \alpha_1 \end{array}$ to $\begin{array}{c} x \\ /\ \backslash \\ \neg\alpha_0 \quad \alpha_1 \end{array}$;
- $\begin{array}{c} |- \\ {}_{+}\!\diagup\!\overset{x}{}\!\diagdown_{+} \\ \alpha_0 \quad \alpha_1 \end{array}$ is equivalent to $\begin{array}{c} x \\ /\ \backslash \\ \neg\alpha_0 \quad \neg\alpha_1 \end{array}$ and $\begin{array}{c} |- \\ {}_{-}\!\diagup\!\overset{x}{}\!\diagdown_{+} \\ \alpha_0 \quad \alpha_1 \end{array}$ to $\begin{array}{c} x \\ /\ \backslash \\ \alpha_0 \quad \neg\alpha_1 \end{array}$.

7.8.4 Interest of TDGs

The main result is that negation on TDGs has a constant (thus "null") cost. In particular, one can exploit new properties in order to speed up computations, for instance: $\alpha + \neg\alpha = 1, \quad \alpha \cdot \neg\alpha = 0, \quad \alpha \oplus \neg\alpha = 1$.
For memory consuming concern, one can establish that a TDG is always smaller than the equivalent BDD (for the same ordering), and that the gain is 1/2 in the best case.
Finally, it appears that TDGs are strictly better than classical BDDs. As a matter of fact, even if it is not always explicit, all efficient implementations are using this representation (sometimes called signed BDDs). From the user point of view, this optimization is completely transparent, and one can continue to reason in terms of Shannon decomposition, knowing that negation has a null cost and that it can be widely used.

7.9 Care set et generalized cofactor

7.9.1 "'Knowing that" operators

When dealing with a logical function f, it is likely that the value of f has no importance outside some known subset of its argument. This *care set* can itself be described by its characteristic function g. In order to optimize algorithms, one may want to exploit this information for simplifying the representation of the function f. The idea is then to define some "knowing that" operator such that the result "h = $f$sachant-que g "satisfies:

- h is equivalent to f when g is true, that is when $g \Rightarrow (h \Leftrightarrow f)$
- h is "as simple as possible", in particular, if it appears that $g \Rightarrow f$, one expects to get $h = 1$, and, conversely, if $g \Rightarrow \neg f$, one expects to get $h = 0$.

Such an operator is obviously not unique; the first condition just states that the expected result belongs to a logical interval: $(f \cdot g) \;\Rightarrow\; h \;\Rightarrow\; (f + \neg g)$, Finding, within this interval, the function whose BDD is minimal is far too expensive. We can just propose knowing-that operators based on heuristics, and such that the complexity is reasonable (with the same order as the one of classical binary operators).

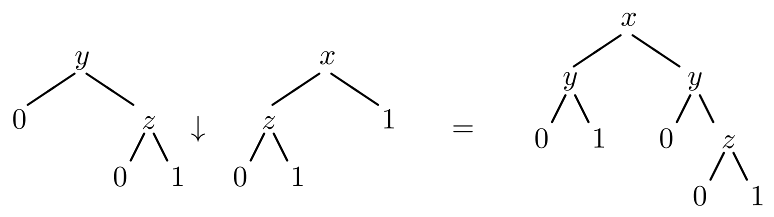


Figure 14: With the ordering $x \prec y \prec z$, $(y \cdot z) \downarrow (x + z) = y \cdot (\bar{x} + z)$

7.9.2 Generalized cofactor

We present here a “knowing-that” operator, denoted by $f \downarrow g$, and usually called *constrain*, or “generalized cofactor”:

- $\alpha \downarrow 0$ is undefined , $\alpha \downarrow 1 = \alpha$,
- $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{/\backslash}} \downarrow \underset{0 \quad \beta}{\overset{x}{/\backslash}} = \alpha_1 \downarrow \beta$ and $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{/\backslash}} \downarrow \underset{\beta \quad 0}{\overset{x}{/\backslash}} = \alpha_0 \downarrow \beta$,
- otherwise $\underset{\alpha_0 \quad \alpha_1}{\overset{x}{/\backslash}} \downarrow \underset{\beta_0 \quad \beta_1}{\overset{x}{/\backslash}} = \underset{\alpha_0 \downarrow \beta_0 \quad \alpha_1 \downarrow \beta_1}{\overset{x}{/\!\!/\backslash\!\!\backslash}}$,
- the other rules are the classical balancing rules (see 7.6.2).

The name generalized cofactor comes from the fact that, for any function f and any no-null function g we have: $f = g \cdot (f \downarrow g) + \neg g \cdot (f \downarrow \neg g)$. In particular, if g is a single variable x, one gets the classical Shannon cofactors: $f = x \cdot (f \downarrow x) + \bar{x} \cdot (f \downarrow \bar{x})$.

7.9.3 Restriction

The generalized cofactor does not have particularly good properties in terms of minimization; it may appear that the result of $\alpha \downarrow \beta$ is bigger than the argument α, and, even worse, than the result contains variables that where not present in α (see Figure 14). In this example, the problem is due to the balancing rules that are introducing the variable x in the result, while it was not present in the argument.

In order to avoid this problem, we define another operator, denoted by $\Downarrow$ and usually called “restriction”. It only differs from the constraint in the way balancing rules are defined:

- if $x \prec y$ then $\alpha \Downarrow \beta = \underset{\alpha_0 \Downarrow \beta \quad \alpha_1 \Downarrow \beta}{\overset{x}{/\!\!/\backslash\!\!\backslash}}$,
- if $y \prec x$ then $\alpha \Downarrow \beta = \alpha \Downarrow (\exists y : \beta) = \alpha \Downarrow (\beta_0 + \beta_1)$.

The name restriction comes from the fact that the graph of the result $f \Downarrow g$ is always a sub-graph of the one of f.

7.9.4 Algebraic properties of the generalized cofactor

Conversely to the restriction, the generalized cofactor does not satisfy simple properties in terms of minimization: $f \downarrow g$ is “often” simpler than f, but not always.

The counterpart is that generalized cofactor satisfies very interesting algebraic properties that restriction do not have:

- it left-distributes on negation (see demonstration 11.1):

$$(\neg\alpha) \downarrow \beta \quad = \quad \neg(\alpha \downarrow \beta) \tag{1}$$

- under existential quantification, it is equivalent to the conjunction (see demonstration 11.1):

$$(\exists \vec{x} : (\alpha \downarrow \beta)(\vec{x})) \quad = \quad (\exists \vec{x} : (\alpha \cdot \beta)(\vec{x})) \tag{2}$$

Being, in general, less costly than the conjunction, It is then interesting to use generalized cofactor whenever it is possible.

8 Forward symbolic exploration

The goal is still the one of the section 5 : explore the whole state space of a finite automaton. But the granularity has changed: instead of exploring each state one by one, we directly reason on state subsets.
Operating on subsets is made possible by the use of BDDs. As explained before, BDDs allow to represent (in general concisely) the characteristic functions of the subsets.

8.1 General scheme

The algorithm is represented in picture 15: at the end of the i^{th} iteration, the variable A holds the (characteristic function of) the set A_i (states that can be reached from the initial ones in at most i transitions). If, during the iteration number k, A_k intersects Err, then the proof fails. Otherwise the exploration goes on, slice by slice, and converges to the set Acc (states reachable in any number of transitions). Figure 16 illustrates the two cases: success (left) and failure (right).

```
$A$ := Init
**repeat** {
        **if** $A \cap \text{Err} \neq \emptyset$ **then** Exit(failure)
        $A'$ := $A \cup POST_H(A)$
        **if** $A' = A$ **then** Exit(success)
        **else** $A$ := $A'$
}
```

Figure 15: Forward symbolic algorithm

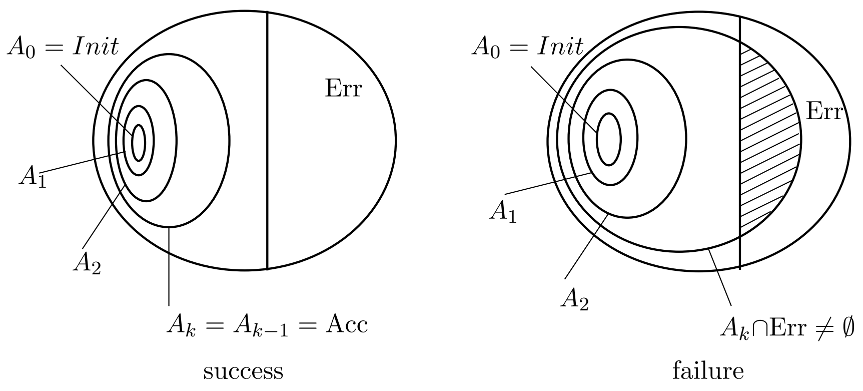

$A_0 = Init$
Err
A_1
A_2
$A_k = A_{k-1} = \text{Acc}$
success
$A_0 = Init$
Err
A_1
A_2
$A_k \cap \text{Err} \neq \emptyset$
failure

Figure 16: Forward symbolic exploration

8.2 Detailed implementation

Concretely, sets are identified to their characteristic functions, themselves implemented with BDDs: union is implemented by disjunction, intersection by conjunction, empty set is implemented by the "always false" BDD, etc.
Complexity is mainly due to the computation of $POST_H(A)$, which takes as argument the BDD encoding the source states, and returns the BDD encoding the corresponding target states. The implementation of this function is presented in the sequel, but it is already clear that its complexity mainly depends on the size of the argument (i.e. the number of nodes in the BDD encoding A).
From this point of view, the presented algorithm makes no optimization at all. States reachable in at most $n + 1$ transitions (A_{n+1}) are computed as the *post* of all states reachable in at most n transitions:

$$A_{n+1} = A_n \cup POST_H(A_n)$$

But in fact, only the computation of the states reachable in exactly $n+1$ is necessary: $POST_H(A_n \setminus A_{n-1})$. As a matter of fact, the other states ($POST_H(A_{n-1})$) already belong, by construction, to A_n. However, even if $A_n \setminus A_{n-1}$ is a smaller set than A_n, it is not necessarily a good idea to compute $POST_H(A_n \setminus A_{n-1})$: one important characteristic of BDDs is that their size is not related to the number of elements they encode.
A good solution is then to use the restriction operator ($\Downarrow$,i cf. 7.9.1):

$$A_{n+1} = A_n \cup POST_H(A_n \Downarrow \neg A_{n-1})$$

This method guarantees that the argument of $POST_H$:

- contains at least $A_n \cap \neg A_{n-1}$,
- contains at most $A_n \cup A_{n-1} = A_n$,
- and its BDD is smaller than the ones of those two bounds.

Figure 17 shows the optimized version of the algorithm. At each iteration n, A holds the states reachable in at most n transitions and A_p the states reachable in at most $n-1$ transitions.

```
$A_p$ := $\emptyset$;    $A$ := Init
repeat {
        if $A \cap \mathrm{Err} \neq \emptyset$ then Exit(failure)
        $A'$ := $A \cup POST_H(A \Downarrow \neg A_p)$
        if $A' = A$ then Exit(success)
        $A_p$ := $A$;     $A$ := $A'$
}
```

Figure 17: Optimized symbolic forward algorithm

8.3 Symbolic image computing

The symbolic image computing, $POST_H(X)$, consists in building the characteristic function of the target states knowing the characteristic function of the source states. This computation is extremely costly; sophisticated, non-trivial algorithms are necessary to reach a (relative) efficiency. However, we present first a "naive" algorithm, in order to illustrate the feasibility of the computation. The necessary optimizations are presented in a second part.
The naive algorithm consists in strictly following the formal definition of *POST*. For that purpose, one builds a "huge" formula involving:

- the source state variables $\vec{s} = (s_1, \cdots, s_n)$,
- the free variables $\vec{v} = (v_1, \cdots, v_m)$,
- the target state variables $\vec{s}' = (s'_1, \cdots, s'_n)$.

This formula is the conjunction of the following constraints:

- $\vec{s}$ is a source state, i.e. $X(\vec{s})$;
- the transition satisfies the hypothesis, i.e. $H(\vec{s}, \vec{v})$;
- each target state variable s'_i is the image of the corresponding transition function, i.e. $s'_i = g_i(\vec{s}, \vec{v})$ for all $i = 1 \cdots n$.

Note that this formula is nothing else than the *transition relation* relating the source states from X with their target states. Once this formula built, it remains to:

- quantify existentially inputs and source state variables ($\vec{s}$ and $\vec{v}$), in order to obtain a formula on the next state variables only ($\vec{s}'$),
- rename the $\vec{s}'$ variables into $\vec{s}$ to allow BDD operations with other state formula (e.g. $A \cup POST_H(A)$ in algorithm Fig. 15).

$$POST_H(X) \quad = \quad \left(\exists \vec{s}, \vec{v} : \left(X(\vec{s}) \cdot H(\vec{s}, \vec{v}) \cdot \bigwedge_{i=1}^{n} (s'_i = g_i(\vec{s}, \vec{v})) \right) \right) [\vec{s}/\vec{s}']$$

This definition only uses "simple" Boolean operations, and thus it can be literally implemented with BDDs operators.

8.4 Optimized image computing

8.4.1 Principles

The main default of the naive algorithm is that it requires to build a huge intermediate formula: state variables are appearing twice, in their source (s_i) and target (s'_i) versions.
In the worst case, the intermediate BDD has a size of order 2^{2n+m}, while the expected result is "only" of order 2^n. It may appear that the intermediate BDD can not be build, even if the result would have finally been of "reasonable" size.
The idea of optimized algorithm is mainly to avoid building the transition relation (intermediate formula) by exploiting the fact that the transition is actually a function, not a whatever relation.
The algorithm we present here is due to Berthet, Madre and Coudert ([9, 10, 11]).

8.4.2 Universal image

In order to simplify the notations, we note $\vec{t} = (\vec{s}, \vec{v})$ the pairs of source state variables and input (free) variables. We also note $Y = X \wedge H$ the conjunction of the source states and the hypothesis.
The expected result $POST_H(X)$ is the image of the states satisfying Y by the (vector of) transition functions $[g_1, \cdots, g_n]$.
For the sake of clarity, we suppose that $\mathcal{I}$ generates a formula on the $\vec{v}\,'$ variables (next state). Just like for the naive algorithm, a final renaming of these variable is performed to allow the combination of the result with other state formula. We note:

$$POST_H(X) \;=\; \left(\mathcal{I}^{Y}_{[g_1,\cdots,g_n]}\right)[\vec{s}/\vec{s}\,']$$

where

$$\mathcal{I}^{Y}_{[g_1,\cdots,g_n]} \;=\; \exists \vec{t}:\; Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t})$$

The first step consists in integrating the source constraint Y within the transition functions. This integration is based on the algebraic properties of the generalized cofactor, from which the following theorem can be established (see demonstration 11.1):

$$\exists \vec{t}:\; Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t}) \;\Leftrightarrow\; \exists \vec{t}:\; \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(\vec{t})) \tag{3}$$

It follows that computing the image of a set Y by a vector of transition functions is equivalent to computing the universal image of this vector of functions constrained by Y:

$$\mathcal{I}^{Y}_{[g_1,\cdots,g_n]} \;=\; \exists \vec{t}:\; \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(\vec{t})) \;=\; = \mathcal{I}^{1}_{[g_1\downarrow Y,\cdots,g_n\downarrow Y]}$$

From now on, when computing a universal image ($\mathcal{I}^{1}_{[f_1,\cdots,f_n]}$) we simply note $\mathcal{I}_{[f_1,\cdots,f_n]}$.

8.4.3 Case of a single transition function

In order to understand the algorithm, we first present how to compute the image of a single transition function.
The predicate $\mathcal{I}[f_n]$ depends on a single variable: the (target) state variable s'_n. It denotes the possible values of the result of f_n.
The Shannon decomposition, according to s'_n, gives:

$$\mathcal{I}_{[f_n]} = \exists \vec{t}: (s'_n = f_n(\vec{t})) = s'_n \cdot (\exists \vec{t}:\; f_n(\vec{t})) \;+\; \bar{s}'_n \cdot (\exists \vec{t}:\; \neg f_n(\vec{t}))$$

There are three possible cases:

1. f_n is identically true, thus $(\exists\vec{t} : f_n(\vec{t})) = 1$ and $(\exists\vec{t} : \neg f_n(\vec{t})) = 0$; it results that the target variable s'_n is eventually true:
$$\mathcal{I}_{[1]} = s'_n \quad ;$$
2. f_n is identically false, thus $(\exists\vec{t} : f_n(\vec{t})) = 0$ and $(\exists\vec{t} : \neg f_n(\vec{t})) = 1$; the target variable s'_n is eventually false:
$$\mathcal{I}_{[0]} = \bar{s}'_n \quad ;$$
3. otherwise, f_n can be either true or false depending on the quantified variables, thus $(\exists\vec{t} : f_n(\vec{t})) = (\exists\vec{t} : \neg f_n(\vec{t})) = 1$; the target variable s'_n can take any value:
$$\mathcal{I}_{[f_n]} = s'_n + \bar{s}'_n = 1 \ .$$

8.4.4 Shannon decomposition of the image

We consider now the case of several transition functions: $\mathcal{I}_{[f_k,\cdots,f_n]}$ with $k < n$. The Shannon decomposition according to the first target variable s'_k gives:

$$\begin{aligned}\mathcal{I}_{[f_k,f_{k+1},\cdots,f_n]} &= \exists\vec{t} : \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t}) \;=\; \exists\vec{t} : (s'_k = f_k(\vec{t})) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t}) \\ &= s'_k \cdot \left(\exists\vec{t}\ f_k(\vec{t}) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) + \bar{s}'_k \cdot \left(\exists\vec{t}\ \neg f_k(\vec{t}) \wedge \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right)\end{aligned}$$

There are, once again, three cases:

1. f_k is identically true (and $\neg f_k$ is identically false):
$$\mathcal{I}_{[1,f_{k+1},\cdots,f_n]} = s'_k \cdot \left(\exists\vec{t} \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) = s'_k \cdot \mathcal{I}_{[f_{k+1},\cdots,f_n]} \quad ;$$
2. f_k is identically false (and $\neg f_k$ is identically true):
$$\mathcal{I}_{[0,f_{k+1},\cdots,f_n]} = \bar{s}'_k \cdot \left(\exists\vec{t} \bigwedge_{i=k+1}^{n} s'_i = f_i(\vec{t})\right) = \bar{s}'_k \cdot \mathcal{I}_{[f_{k+1},\cdots,f_n]} \quad ;$$
3. otherwise, f_k can be either true or false, and one can recognize the definition of the image for the transitions vector $[f_{k+1}, \cdots, f_n]$ applied to f_k (right-hand side) and $\neg f_k$ (left-hand side):
$$\mathcal{I}_{[f_k,f_{k+1},\cdots,f_n]} = s'_k \cdot \mathcal{I}^{f_k}_{[f_{k+1},\cdots f_n]} + \bar{s}'_k \cdot \mathcal{I}^{\neg f_k}_{[f_{k+1},\cdots f_n]} \ .$$

We use the properties of the $\downarrow$ operator in order to give an equivalent definition which uses only universal image computation. Moreover, one can note that the case of a single transition function can fit in the general case as soon as we define that $\mathcal{I}_{[]} = 1$.

8.4.5 Recursive computation of the image

Finally, we obtain a purely recursive definition of the universal image computing. This definition leads to an algorithm that recursively builds the Shannon decomposition of the universal image, that is, its BDD. This algorithm (Figure 18) makes heavy use of the generalized cofactor. For the sake of clarity, the presented algorithm builds a BDD on the $\vec{s}'$ variables, but note that it can directly produce a BDD on the $\vec{s}$ ones, making the final renaming useless.

- $\mathcal{I}_{[]} = 1$
- $\mathcal{I}_{[1,f_{k+1},\cdots,f_n]} =$
- $\mathcal{I}_{[0,f_{k+1},\cdots,f_n]} =$
- $\mathcal{I}_{[f_k,f_{k+1},\cdots,f_n]} =$

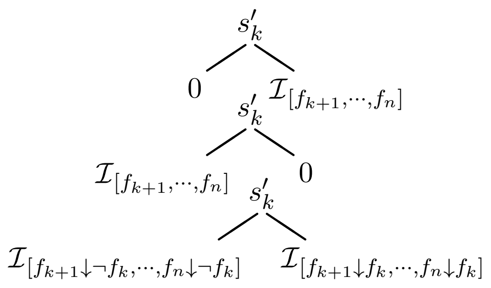

s'_k
0
$\mathcal{I}_{[f_{k+1},\cdots,f_n]}$
s'_k
$\mathcal{I}_{[f_{k+1},\cdots,f_n]}$
0
s'_k
$\mathcal{I}_{[f_{k+1}\downarrow\neg f_k,\cdots,f_n\downarrow\neg f_k]}$
$\mathcal{I}_{[f_{k+1}\downarrow f_k,\cdots,f_n\downarrow f_k]}$

Figure 18: Recursive computation of the universal image

```
$B :=$ Err;  $B_p = \emptyset$
repeat {
        if $B \cap$ Init $\neq \emptyset$ then Exit(failure)
        $B' := B \cup PRE_H(B \Downarrow \neg B_p)$
        if $B' = B$ then Exit(success)
        else $B_p = B$;   $B := B'$
}
```

Figure 19: Symbolic backward algorithm

9 Backward symbolic exploration

9.1 General scheme

The backward exploration algorithm is the dual of the forward one (Figure 19): we simply have to swap the roles of Init and Err, and to replace the image computation by the "reverse-image" computation (PRE_H function).
When the algorithm succeeds, the variable B holds the set of states that can lead to an error, which is exactly the set Bad defined in 5.4.
The complexity is mainly due to the reverse image computation (PRE), whose formal definition is:

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

Just like for the $POST$ computation, a naive implementation consists in applying this definition:

- build a (huge) BDD depending on source state variables ($\vec{s}$), inputs ($\vec{v}$) and target state variables ($\vec{s}'$);
- quantify existentially $\vec{s}'$ and $\vec{v}$ in order to obtain the expected BDD, depending only on $\vec{s}$.

9.2 Reverse image computing

Just like for the $POST$, it is possible to avoid the construction of the huge intermediate formula by reasoning on the Shannon decomposition of the result.
First, since the hypothesis H does not depend on the variables $\vec{s}'$, it is possible to keep it out of the corresponding quantifier:

$$PRE_H(X) = \exists \vec{v} : H(\vec{s}, \vec{v}) \cdot \mathcal{R}_{\vec{g}}(X) \ ,$$

where $\mathcal{R}_{\vec{g}}(X)$ is the reverse image of X for the transition functions $\vec{g}$, i.e. the pairs $(\vec{s}, \vec{v})$ leading to some state in X:

$$\mathcal{R}_{\vec{g}}(X) \ = \ \exists \vec{s}' : X(\vec{s}') \cdot \bigwedge_{i=1}^{n} s'_i = g_i(\vec{s}, \vec{v}) \ .$$

Let us note $X = s'_1 \cdot X_1 \; + \; \bar{s}'_1 \cdot X_0$, it is easily shown that:

$$\mathcal{R}_{g_1::\vec{g}'}\left(\begin{array}{c} s'_1 \\ / \quad \backslash \\ X_0 \quad X_1 \end{array}\right) \; = \; g_1 \cdot \mathcal{R}_{\vec{g}'}(X_1) \; + \; \neg g_1 \cdot \mathcal{R}_{\vec{g}'}(X_0) \; .$$

The reverse image computation is then equivalent to a recursive composition of transition function. In order to obtain the complete algorithm (Figure 20), we use the following properties:

- $\mathcal{R}_{\vec{g}}(0) = 0$ and $\mathcal{R}_{\vec{g}}(1) = 1$, since $\vec{g}$ is a total function;
- the computation can be optimized by the use of the restriction, which trivially satisfies (just like any other "knowing-that" operator, see 7.9.1):

$$f \cdot g \; = \; f \cdot (g \Downarrow f)$$

- $\mathcal{R}^0_{\vec{g}} = 0 \quad$ et $\quad \mathcal{R}^1_{\vec{g}} = 1$
- $\mathcal{R}_{g_k::\vec{g}}\left(\begin{array}{c} s'_k \\ / \quad \backslash \\ X_0 \quad X_1 \end{array}\right) \; = \; \begin{array}{c} g_k \\ \diagup \quad \diagdown \\ \mathcal{R}_{\vec{g}\Downarrow\neg g_k}(X_0) \quad \mathcal{R}_{\vec{g}\Downarrow g_k}(X_1) \end{array}$

Figure 20: Recursive computation of the reverse-image

9.3 Comparison between the two methods

It is rather difficult to compare the two methods since their efficiency dramatically depends on the "nature" of the explored model. One can simply state that, empirically, the forward symbolic method is "very often" more efficient. One possible explanation is that image computation is somehow intrinsically simpler than reverse-image computation. In fact, this explanation is not really convincing, since both algorithms are clearly of the same order of complexity (over-exponential), and that the efficiency factor (if it even exists) is likely to be negligible.

In fact, experience tends to show that this is the actual "nature" of the property to check that makes the big difference. Roughly speaking, the properties can be classified in two families:

- Accessibility properties are those that strongly depend on the actual initial state(s), in the sense that slightly modifying the initial states may lead either to success or failure. For this kind of properties, the forward method appears to be the more efficient.
- Inductive invariant properties are those that not (strongly) depend on the initial states. Intuitively, such a property Φ satisfies $PRE_H(\Phi) \Rightarrow \Phi$, or more generally $\bigwedge_{i=1}^{k} PRE^i_H(\Phi) \Rightarrow \Phi$, for some (small) constant k. In this case the backward method is likely to converge much more quickly than the forward one.

Finally, the empirical "folk-theorem" stating that the forward method is better than the backward method seems to mainly reflect the fact that model-checking is generally used to prove accessibility properties rather than inductive properties.

10 Conclusion and related works

Finite state systems verification is theoretically decidable, but practically limited by the combinational explosion of state space.

With symbolic methods based on BDDs, the intrinsic limitation due to the actual number of states can be overpassed: the complexity is no longer related to the number of variables, but rather to the relations between the variables. However, the use of BDDs remains extremely costly both in time and memory.

Sat decision methods (for satisfiability) are consisting in enumerating the solutions of a formula without storing them in memory. Comparing to BDDs, such methods have an exponential cost in time, but only a polynomial cost in memory. Model-checking algorithms can be adapted to use a decision engine based on Sat methods rather than BDDs. In this case, iterations are likely to be more efficient (at least in term of memory), but the counterpart is that checking the convergence becomes dramatically costly. This is why this kind of method is used to check properties for bounded-time executions: *bounded model-checking*[4].

11 Demonstrations

11.1 Lemme: co-factor main property

For a given order of the variables, and thus an operator $\downarrow$ perfectly defined, for any function $g : \mathbb{B}^n :\rightarrow \mathbb{B}$ not identically false, it exists a function $\pi_g : \mathbb{B}^n \rightarrow \mathbb{B}^n$ such that:

- for any $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and any $\vec{x} \in \mathbb{B}^n$:

$$(f \downarrow g)(\vec{x}) = f(\pi_g(\vec{x})) \qquad \text{i.e. } f \downarrow g = f \circ \pi_g \tag{4}$$

- π_g sends any point from $\mathbb{B}^n$ into g:

$$\forall \vec{x} \;:\; g(\pi_g(\vec{x})) \qquad \text{i.e.} \qquad (\exists \vec{y} : \vec{x} = \pi_g(\vec{y})) \;\Leftrightarrow\; g(\vec{x}) \tag{5}$$

The recursive definition of π_g directly comes from the one of $\downarrow$. Let $g = x \cdot g_1 + \bar{x} \cdot g_0$ be the Shannon decomposition of g according the least variable in the considered order:

- $\pi_g(\vec{x}) = \vec{x} \quad \text{if} \;\; g(\vec{x})$
- $\pi_g(0 :: \vec{x}) = \text{if } (g_0 = 0) \text{ then } (1 :: \pi_{g_1}(\vec{x})) \text{ else } (0 :: \pi_{g_0}(\vec{x}))$
- $\pi_g(1 :: \vec{x}) = \text{if } (g_1 = 0) \text{ then } (0 :: \pi_{g_0}(\vec{x})) \text{ else } (1 :: \pi_{g_1}(\vec{x}))$

Theorem 1

$((\neg f) \downarrow g)(\vec{x}) = (\neg f)(\pi_g(\vec{x}))$ [Lemme 4]
$= \;\neg f(\pi_g(\vec{x})) = \neg(f \downarrow g)(\vec{x})$

Theorem 2

$\exists \vec{x} \;\; (f \downarrow g)(\vec{x}) \quad\Leftrightarrow\; \exists \vec{x} \;\; f(\pi_g(\vec{x}))$ [Lemme 4]
$\Leftrightarrow \;\; \exists \vec{x}\, \exists \vec{y} \;\; f(\vec{y}) \wedge (\vec{y} = \pi_g(\vec{x}))$
$\Leftrightarrow \;\; \exists \vec{y} \;\; f(\vec{y}) \wedge (\exists \vec{x} \; (\vec{y} = \pi_g(\vec{x})))$
$\Leftrightarrow \;\; \exists \vec{y} \;\; f(\vec{y}) \wedge g(\vec{y})$ [Lemme 5]
$\Leftrightarrow \;\; \exists \vec{y} \;\; (f \wedge g)(\vec{y})$

Theorem 3

$$\exists \vec{t} \;\; Y(\vec{t}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{t})$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} (Y(\vec{t}) \wedge (s'_i = g_i(\vec{t}))$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} ((s'_i \wedge g_i(\vec{t}) \wedge Y(\vec{t})) \vee (\neg s'_i \wedge \neg g_i(\vec{t}) \wedge Y(\vec{t})))$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \wedge Y)(\vec{t})) \vee (\neg s'_i \wedge (\neg g_i \wedge Y)(\vec{t})))$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \downarrow Y)(\vec{t})) \vee (\neg s'_i \wedge (\neg g_i \downarrow Y)(\vec{t}))) \quad \text{[Thorme 1]}$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} ((s'_i \wedge (g_i \downarrow Y)(\vec{t})) \vee (\neg s'_i \wedge \neg(g_i \downarrow Y)(\vec{t}))) \quad \text{[Thorme 2]}$$

$$\Leftrightarrow \; \exists \vec{t} \;\; \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(\vec{t}))$$

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[2] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, July 1996.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In *Advances in Computers*, volume 58. Academic press, 2003.

[5] J.P. Billon. Perfect normal forms for discrete functions. Report 87019, BULL, March 1987.

[6] J.P. Billon and J.C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In G. Milne, editor, *IFIP WG 10.2 Int Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. North Holland.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, June 1990.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification.* North-Holland, November 1989.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble.* LNCS 407, Springer Verlag, 1989.

[11] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In R. Kurshan, editor, *International Workshop on Computer Aided Verification*, Rutgers, New Jersey, June 1990.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[14] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, pages 377–410, 1990.

[15] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[16] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer Aided Design (ICCAD)*, November 1990.