An executable temporal logic to express safety properties and its connection with the language Lustre

Nicolas Halbwachs*, Jean-Claude Fernandez†, Ahmed Bouajjani†

Invited paper at the Sixth International Symposium on Lucid and Intensional Programming, ISLIP'93, Quebec, April 1993

Abstract

This paper studies the expressive power of the synchronous data-flow language Lustre as a specification language, and its connection with temporal logic. After a brief overview of Lustre, we define a temporal logic, called SL, which is shown to have exactly the expressive power of regular safety properties. Directly inspired from Boolean Lustre, this logic is executable, in the sense that the accepting automaton of any SL formula can be constructed "on the fly", as the model is read. Then we consider a fragment of SL, called DSL, for the formulas of which the accepting automaton built by the previous technique is deterministic. DSL is shown to have the same expressive power as SL, and to be equivalent to Boolean Lustre.

1 Introduction

Lustre [CPHP87, HCRP91] is a synchronous declarative language designed for programming real-time systems. It is based on the data-flow principle [Kah74, AW85], with a strong restriction which consists in considering that all the variables involved in a program takes their nth value at the same time. In addition to being well-suited to the description of cyclic reactive systems, this restriction allows a specific compiling technique to be applied: A Lustre program can be compiled into an efficient sequential code, the control structure of which is a finite automaton synthesized by the compiler from an exhaustive simulation of Boolean variables appearing in the program [CPHP87].

In [HLR92], we proposed to use Lustre also as a specification language. First of all, we restricted ourselves to the expression of *safety properties*, as it was often argued (see, e.g., [Pnu92]) that almost all the critical properties that are required of a real-time system are safety properties. Such properties can easily be expressed by the invariance of a Boolean Lustre expression. There are several advantages in doing so:

– Both the program and its expected properties are expressed in the same language. The concepts considered in the program and the properties are exactly the same, and there is no "specification language" to learn.

*IMAG Institute. This work was terminated when this author was on leave in Stanford University, partially supported by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1901, and by a grant from the Stanford Office of Technology Licensing. This publication does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement of this work should be inferred.

†IMAG Institute, B.P. 53X, 38041 Grenoble Cedex - France.

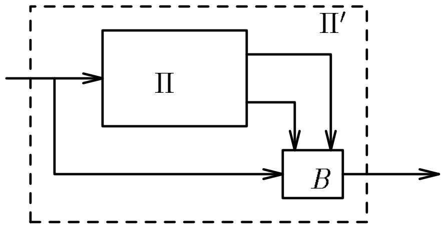

Figure 1: A verification program

- The properties are expressed in an executable language: They can be tested, in order to get a reasonable confidence that they correctly express the informal requirements. Also, the properties can be actually run with the program, implementing a run-time detection of requirement violations.

- This leads to a very simple verification technique: To prove that a program Π satisfies a property P, expressed by the invariance of a Boolean expression B, one builds a new program Π' which is the parallel combination of Π and B, and whose only output is the value of B (cf. Fig 1). Verifying that Π satisfies P amounts to prove that Π' never outputs "*false*". The tool LESAR [Rat92] performs this verification in the finite state case.

This paper studies the expressive power of LUSTRE as a specification language, and its connection with temporal logic. After a brief overview of LUSTRE (Section 2), we will define a temporal logic, called SL, which will be shown to have exactly the expressive power of *regular safety properties* — i.e. safety properties which are regular languages (Section 3). Directly inspired from Boolean LUSTRE, this logic is *executable*, in the sense that the accepting automaton of any SL formula (i.e., the automaton accepting exactly the models of the formula) can be constructed "on the fly", as the model is read (Section 4). In Section 5, we consider a fragment of SL, called DSL, for the formulas of which the accepting automaton provided by the previous technique is *deterministic*. DSL is shown to have the same expressive power as SL, and to be equivalent to Boolean LUSTRE.

2 The language Lustre

2.1 Overview of the language

We do not give here a detailed presentation of the language LUSTRE, which can be found elsewhere [CPHP87, HCRP91]. We only recall the elements which are necessary for understanding the paper.

A LUSTRE program specifies a relation between input and output variables. A variable is intended to be a function of time. Time is assimilated to the set of natural numbers. Variables are defined by means of equations: An equation "`X=E`" , where `E` is a LUSTRE expression, specifies that the variable `X` is always equal to `E`.

Expressions are made of variable identifiers, constants (considered as constant functions), usual arithmetic, Boolean and conditional operators (considered as pointwisely applying to functions) and only two specific operators: the "previous" operator and the "followed-by" operator:

- If E is an expression denoting the function $\lambda n.e(n)$, then "pre(E)" is an expression denoting the function

$$\lambda n.\begin{cases} nil & \text{if } n = 0 \\ e(n-1) & \text{if } n > 0 \end{cases}$$

where *nil* is an undefined value.

- If E and F are two expressions of the same type, respectively denoting the functions $\lambda n.e(n)$ and $\lambda n.f(n)$, then "E -> F" is an expression denoting the function

$$\lambda n.\begin{cases} e(n) & \text{if } n = 0 \\ f(n) & \text{if } n > 0 \end{cases}$$

A LUSTRE program is structured into *nodes*: a node is a subprogram specifying a relation between its input and output parameters. This relation is expressed by an unordered set of equations, possibly involving local variables. Once declared, a node may be instantiated in any expression, as a basic operator.

For instance the following declaration defines a node which returns *true* whenever its Boolean parameter raises from *false* to *true*:

```
node Edge(x: bool) returns (edge: bool);
let
   edge = false -> (x and not pre(x));
tel
```

Now, the expression "Edge(not C)" is *true* whenever the variable C has a falling edge.

2.2 Automaton generation

Automaton generation from a LUSTRE program has been first used in the LUSTRE compiler to synthesize an efficient control structure for the object code. Roughly speaking, it consists in an exhaustive simulation of the behavior of Boolean variables.

Let us consider the node Edge, as a small example of automaton generation. Initially, its output is clearly *false* (because of the operator ->). Now, the value of x at the first instant provides the value of pre(x) at the second instant. The idea is to code this knowledge in the control structure: according to the value of x, the state of the program at the second instant will be chosen; there will be two possible states, one where it is known that pre(x) is *true*, and one where pre(x) is known to be *false*. Moreover, it is known that these states are not the initial state: so the the result of the -> operator is the result of its second operand. Evaluating these states in the same way, we get the automaton shown in Fig. 2.

2.3 Program specification and verification

Specifying in LUSTRE a temporal property P of a program Π consists in writing a Boolean expression B, involving variables of Π, such that P holds if and only if the expression B is always true during any execution of the program. Obviously, only safety properties can be expressed in that way. Let us show how non trivial properties can be expressed. Consider the following property:

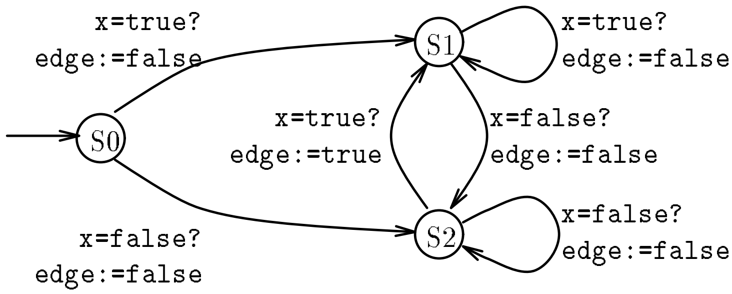

Figure 2: The automaton of the node `Edge`

> "Any occurrence of a critical situation must be followed by an alarm within a five seconds delay"

Such a property relates three events: the critical situation occurrence, the alarm, and the deadline. A general pattern for this property is the following one:

> "Any occurrence of event A is followed by an occurrence of event B before the next occurrence of event C"

However, this formulation cannot be directly translated into LUSTRE since it refers to what happens in the future following an A occurrence, while LUSTRE only allows references to the past with respect to the current instant. That is why we first translate it into the equivalent past expression:

> "Any time C occurs, either A has never occurred before, or B has occurred since the last occurrence of A."

Let us assume that the three events A, B, C are represented by three Boolean variables `A`, `B`, `C`, which are *true* when and only when the corresponding event occurs. Let us define a node, taking `A`, `B`, `C` as input parameters, and returning a Boolean output "`onceBfromAtoC`" which is always true if and only if the property holds:

```
node once_from_to_(B,A,C: bool) returns (onceBfromAtoC: bool);
var never_A, B_since_A: bool;
let
   onceBfromAtoC = not C or (never_A or B_since_A);
   never_A = if A then false else (true -> pre(never_A));
   B_since_A = if A then B
               else if B then true
               else (true -> pre(B_since_A));
tel
```

"`onceBfromAtoC`" is defined to be true if any occurrence of C implies that either A never occurred (represented by the auxiliary variable "`never_A`") or that B has occurred at least once since the last occurrence of A (represented by the variable "`B_since_A`"); "`never_A`" is *true* as long as A does not occur; it becomes *false* at the first occurrence of A and remains *false* forever; "`B_since_A`" becomes *false* when A occurs without B, and becomes *true* when B occurs.

Now to verify the (obvious!) property that if a Boolean variable "`x`" is *false*, and later *true*, the node `Edge(x)` returns at least once *true* in between, one can write the following verification program:

```
node verify(x: bool) returns (ok: bool);
let
  ok = once_from_to_(Edge(x), not x, x);
tel
```

compile it into an automaton, and check on that automaton that the output "`ok`" is never assigned to *false*.

3 A temporal logic of safety

3.1 Definitions and notations

Throughout the paper, we will consider *trace semantics*: A trace τ on a set of *observables* O is a (finite or infinite) sequence of elements of O. A *property* is a mapping from traces to $\{true, false\}$. P is a *safety property* if and only if the following equivalence holds:

$$P(\tau) = true \iff P(\tau') = true \quad \text{for any finite prefix } \tau' \text{ of } \tau$$

In other words, the set $\mathcal{L}(P)$ of traces satisfying a safety property P is a prefix-closed (as expressed by the "$\Longrightarrow$" implication above) and limit-closed (as expressed by the "$\Longleftarrow$" implication) language. A safety property P is *regular* if an only if $\mathcal{L}(P)$ is the closure of a regular language $\mathcal{R}$ — i.e., the set of traces any finite prefix of which belongs to $\mathcal{R}$.

A *transition system* Π is a quadruple (Q, q_0, O, λ) where Q is a set of states, $q_0 \in Q$ is the initial state, O is a set of observables, and λ is a ternary relation on $Q \times O \times Q$ (transition relation). A trace of Π is a sequence $(\omega_0, \omega_1, \ldots, \omega_n, \ldots)$ in O^∞ (the set of finite or infinite sequences of O), such that there exists a sequence of states $(q_0, q_1, \ldots, q_n, \ldots)$ and $(q_n, \omega_n, q_{n+1}) \in \lambda$ for any n. Let $\mathcal{L}(\Pi)$ be the set of traces of Π.

We say that a transition system Π satisfies a safety property P (noted $\Pi \models P$), if and only if all of its traces satisfy P, i.e., iff $\mathcal{L}(\Pi) \subseteq \mathcal{L}(P)$.

As far as safety properties are concerned, a finite state transition system will often be considered as an automaton, accepting both finite and infinite words, and all states of which is accepting. Such an automaton will be called a *safety automaton*. The class of languages on a given vocabulary, which can be characterized by a safety automaton, is clearly the class of languages characterized by regular safety properties.

3.2 The logic SL

In this section, we define a temporal logic whose expressive power is exactly the set of regular safety properties. This logic, called SL, is a fragment of the quantified temporal logic QTL [Sis83, LPZ85].

3.2.1 Syntax

Let us consider two disjoint finite alphabets, *Prop* of propositional symbols, and *Aux* of auxiliary symbols. SL is defined by means of a two-level syntax:

– SL formulas, also called **safety formulas**, noted ψ, are of the form

$$\exists x_1, x_2, \ldots, x_k \, \Box\varphi$$

where $x_1, x_2, \ldots, x_k$ belong to Aux and φ is a *past formula*.

– **past formulas**, noted φ, obey the following syntax:

  – Any symbol $a \in Prop \cup Aux$ is a past formula

  – If φ_1 and φ_2 are past formulas, so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$ and $\bullet\varphi_1$

The set of past formulas is included in the temporal logic of the past TLP [LPZ85].

3.2.2 Semantics

Models of formulas are traces, and, as usual, we note "$\tau \models \phi$" the fact that the trace τ is a model of the formula ϕ. Let us successively define the semantics of past formulas and safety formulas.

Past formulas: Let $O = 2^{Prop \cup Aux}$. Models of past formulas are finite traces over O. The satisfaction of a formula by such a trace is defined as follows:

$$\begin{array}{lll}
(\omega_0, \ldots, \omega_n) \models a & \text{iff} & a \in \omega_n \\
(\omega_0, \ldots, \omega_n) \models \neg\varphi & \text{iff} & (\omega_0, \ldots, \omega_n) \not\models \varphi \\
(\omega_0, \ldots, \omega_n) \models \varphi_1 \vee \varphi_2 & \text{iff} & \text{either } (\omega_0, \ldots, \omega_n) \models \varphi_1 \text{ or } (\omega_0, \ldots, \omega_n) \models \varphi_2 \\
(\omega_0, \ldots, \omega_n) \models \bullet\varphi & \text{iff} & n > 0 \text{ and } (\omega_0, \ldots, \omega_{n-1}) \models \varphi
\end{array}$$

Usual abreviations are introduced:

$$\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2) \qquad \varphi_1 \supset \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2 \qquad \varphi_1 \equiv \varphi_2 \triangleq (\varphi_1 \supset \varphi_2) \wedge (\varphi_2 \supset \varphi_1)$$

The operator $\bullet$ is the "previous" operator of TLP. We will often use its dual $\neg \bullet \neg$. Notice that their only difference is that $\bullet\varphi$ is *false* on a trace of only one element, whereas $\neg \bullet \neg\varphi$ is *true*. On any longer trace, $\bullet\varphi$ and $\neg \bullet \neg\varphi$ have the same value.

Safety formulas: Models of safety formulas are finite or infinite traces over 2^{Prop}. Such a trace $(\omega_0, \ldots, \omega_n, \ldots)$, $\omega_i \subseteq Prop$, satisfies an SL formula $\exists x_1, x_2, \ldots, x_k \Box\varphi$, if and only if there exists a trace $(\omega'_0, \ldots, \omega'_n, \ldots)$, $\omega'_i \subseteq Aux$, such that,

$$\forall n, \; ((\omega_0 \cup \omega'_0), (\omega_1 \cup \omega'_1), \ldots, (\omega_n \cup \omega'_n)) \models \varphi$$

Examples: The SL formula $\exists x \Box((x \equiv \bullet(a \vee x)) \;\wedge\; (b \supset x))$ expresses that the first time a is *true* strictly precedes the first time b is *true*. Here, the quantified variable x stands for "a has been *true* at least once in the past": It is initially *false*, becomes *true* just after a is *true*, and then remains *true* forever.

In the same way, the "*since*" operator $\mathcal{S}$ of TLP can be expressed: Let us recall that the TLP formula $\varphi_1 \mathcal{S} \varphi_2$ is true if and only if φ_1 has been continuously *true* since the last time φ_2 was *true*. Such a formula will be handled in SL as an existentially quantified variable, say x, such that $\Box(x \equiv (\varphi_2 \vee (\varphi_1 \wedge \bullet x)))$. This expression results from the TLP axiom $\varphi_1 \mathcal{S} \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \bullet(\varphi_1 \mathcal{S} \varphi_2))$.
[end of examples]

3.3 Expressive power

Proposition 1 *The class of properties expressible in SL is exactly the class of regular safety properties.*

Proof: We show that with any safety automaton can be associated an SL formula, such that any word accepted by the automaton is a model of the formula. The reverse direction is a special case of the result established in [LPZ85], concerning the power of quantified TLP. It will be considered again in the next section.

Let $(Q, q_0, \Sigma, \lambda)$ be a safety automaton, where $Q = \{q_0, \ldots, q_k\}$. We take Σ as proposition alphabet (a is *true* when the automaton receives an a), Q as auxiliary alphabet (q is *true* when the automaton is in state q). The words accepted by the automaton are the models of the formula $\exists q_0, \ldots, q_k \quad \Box\varphi$, where

$$\begin{aligned}
\varphi \;=\; & \Big(q_0 \supset \bigvee_{(q,a,q_0)\in\lambda} \neg \bullet \neg(q \wedge a)\Big) & (1)\\
& \wedge \bigwedge_{i=1}^{k} \left(q_i \supset \bigvee_{(q,a,q_i)\in\lambda} \bullet(q \wedge a) \right) & (2)\\
& \wedge \bigvee_{i=0}^{k} q_i \wedge \bigwedge_{\substack{i,j=0\\ i\neq j}}^{k} \neg(q_i \wedge q_j) \wedge \bigvee_{a\in\Sigma} a \wedge \bigwedge_{\substack{a,b\in\Sigma\\ a\neq b}} \neg(a \wedge b) & (3)
\end{aligned}$$

Implication 1 expresses that the automaton is initially in state q_0 (because of the use of $\neg \bullet \neg$) and comes back to q_0 only if it was previously in a state q and received a symbol a, such that $(q, a, q_0) \in \lambda$. Implications 2 deal with other states (using now the operator $\bullet$), and the line 3 expresses that the automaton is always in one and only one state, and receives always one and only one symbol. *[end of proof]*

Example: Let us consider the automaton shown in Figure 3. From the proof of proposition 1, the following SL formula characterizes the same language:

$$\begin{aligned}
\exists q_0, q_1, q_2, \quad \Box \quad & q_0 \supset \neg \bullet \neg(q_1 \wedge b) \vee \neg \bullet \neg(q_2 \wedge b)\\
\wedge \quad & q_1 \supset \bullet\,(q_0 \wedge a)\\
\wedge \quad & q_2 \supset \bullet\,(q_0 \wedge a) \vee \bullet(q_1 \wedge c)\\
\wedge \quad & (q_0 \vee q_1 \vee q_2) \wedge \neg(q_0 \wedge q_1) \wedge \neg(q_1 \wedge q_2) \wedge \neg(q_2 \wedge q_0)\\
\wedge \quad & (a \vee b \vee c) \wedge \neg(a \wedge b) \wedge \neg(b \wedge c) \wedge \neg(c \wedge a)
\end{aligned}$$

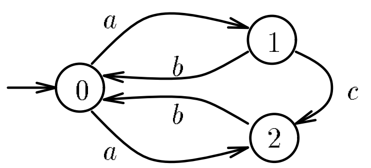

Figure 3: A finite automaton

[end of example]

In conclusion, SL has exactly the power needed for expressing safety properties of finite state systems.

Remark: If we consider formulas of the form $\exists x_1, x_2, \ldots, x_k, \varphi_1 \wedge \Box\varphi_2$, where φ_1 and φ_2 are past formulas, we get the power of regular languages on finite traces: If $(Q, q_0, \Sigma, \lambda, F)$ is a finite state automaton, where $F \subseteq Q$ is the set of final states, and if $\exists q_0, \ldots, q_k \Box\varphi$ is the SL formula associated with $(Q, q_0, \Sigma, \lambda)$, as shown above, then the formula

$$\exists q_0, \ldots, q_n, \bigvee_{q_i \in F} q_i \ \wedge \ \Box\varphi$$

characterizes the language recognized by the automaton. So, on finite models and in spite of its restricted syntax, this logic has the same power as the fully quantified logic of the past QTLP.

4 Executing safety formulas

In this section, we consider the problem of building an automaton accepting exactly the language of models of a given SL formula. This problem has been considered in several papers (e.g. [MW84, VW86, PR89]), in a more general case. However, our presentation consists of giving the operational semantics of the formulas, that will allow a progressive construction of the automaton.

The automaton $\mathcal{A}(\psi)$ associated with a safety formula ψ is defined on the alphabet 2^{Prop}, and accepts a word $(\omega_0, \ldots, \omega_n, \ldots)$ if and only if it is a model of ψ. Since the behavior of this automaton only depends on the current state and the current input, the construction consists in translating a property of traces into a property of states. Each state of the automaton will correspond to a formula, and the transitions will correspond to formula *rewriting*, according to the scheme:

$$(\omega_0, \ldots, \omega_n, \ldots) \models \psi \iff \psi \xrightarrow{\omega_0} \psi_1 \text{ and } (\omega_1, \ldots, \omega_n, \ldots) \models \psi_1$$

Formula rewriting is a way of recording the past inputs for future evaluations. It is the basic mechanism for translating trace properties into state properties.

Moreover, with each past formula φ, we will associate an input/output automaton $\mathcal{A}(\varphi)$, with input alphabet $2^{Prop \cup Aux}$, and whose Boolean output is *true* if and only if the sequence of inputs received so far is a model of φ.

4.1 Past formulas

The inputs of the automaton $\mathcal{A}(\varphi)$ are subsets ω of the finite alphabet $A = Prop \cup Aux$. The transitions (rewritings) will be defined by means of a predicate $\varphi \xrightarrow{\omega:b} \varphi'$, where $\omega \in 2^A$ and $b \in \{true, false\}$, which means "on the trace reduced to the singleton (ω), the value of φ is b; on a longer trace $\omega.\tau'$, the value of φ is the same as the one of φ' on τ' ". In other words, this predicate will be defined so that the following rules hold:

$$\frac{\varphi \xrightarrow{\omega_0:b_0} \varphi_1 \xrightarrow{\omega_1:b_1} \cdots \varphi_{n-1} \xrightarrow{\omega_n:true} \varphi_n}{(\omega_0, \ldots, \omega_n) \models \varphi} \qquad \frac{\varphi \xrightarrow{\omega_0:b_0} \varphi_1 \xrightarrow{\omega_1:b_1} \cdots \varphi_{n-1} \xrightarrow{\omega_n:false} \varphi_n}{(\omega_0, \ldots, \omega_n) \not\models \varphi}$$

The transition predicate is defined by means of structural inference rules [Plo81]:

– A formula consisting of a basic proposition always evaluates in the same way: its value is found in the input.

$$\frac{a \in \omega}{a \xrightarrow{\omega:true} a} \quad , \quad \frac{a \notin \omega}{a \xrightarrow{\omega:false} a}$$

– Boolean operators always evaluate in the same way, according to the values of their operands:

$$\frac{\varphi_1 \xrightarrow{\omega:b_1} \varphi_1' \ , \ \varphi_2 \xrightarrow{\omega:b_2} \varphi_2'}{\varphi_1 \vee \varphi_2 \xrightarrow{\omega:b_1 \vee b_2} \varphi_1' \vee \varphi_2'} \quad , \quad \frac{\varphi \xrightarrow{\omega:b} \varphi'}{\neg\varphi \xrightarrow{\omega:\neg b} \neg\varphi'}$$

– A $\bullet$ operator is always evaluated as if the current input was the first one. So it always evaluates to *false*. However, if its operand evaluates to *true*, the operator is rewritten into its dual $\neg \bullet \neg$, in order to return *true* in the next state. We get the following rules:

$$\frac{\varphi \xrightarrow{\omega:false} \varphi'}{\bullet\varphi \xrightarrow{\omega:false} \bullet\,\varphi'} \quad , \quad \frac{\varphi \xrightarrow{\omega:true} \varphi'}{\bullet\varphi \xrightarrow{\omega:false} \neg \bullet \neg\varphi'}$$

Example: The evaluation of the formula $\bullet a$ on a trace $(\{a\}, \{b\}, \{a\}, \{b\})$ provides:

$$\bullet a \xrightarrow{\{a\}:false} \neg \bullet \neg a \xrightarrow{\{b\}:true} \bullet\, a \xrightarrow{\{a\}:false} \neg \bullet \neg a \xrightarrow{\{b\}:true} \bullet\, a$$

and, since the last output is *true*, $(\{a\}, \{b\}, \{a\}, \{b\}) \models \bullet a$.
[end of example]

Proposition 2 *For any past formula φ on the alphabet A, for any $\omega \in 2^A$, there is a unique Boolean b and a unique formula φ' such that $\varphi \xrightarrow{\omega:b} \varphi'$.*

Proof: By induction on the rules. *[end of proof]*

The unique formula φ' such that $\varphi \xrightarrow{\omega:b} \varphi'$ is called the *derivative* of φ with respect to ω, and is noted $\partial\varphi/\partial\omega$. More generally, for any finite trace $\tau = (\omega_0, \ldots, \omega_n)$ on 2^A, we define the derivative of φ with respect to τ, noted $\partial\varphi/\partial\tau$, to be the unique formula φ_n such that

$$\varphi \xrightarrow{\omega_0:b_0} \varphi_0 \xrightarrow{\omega_1:b_1} \varphi_1 \xrightarrow{\omega_2:b_2} \ldots \xrightarrow{\omega_n:b_n} \varphi_n$$

Proposition 3 *The set $Q_\varphi = \{\partial\varphi/\partial\tau \mid \tau \in (2^A)^*\}$ of all derivatives of a past formula φ, is finite.*

Proof: From the rules, it appears that a derivative $\partial\varphi/\partial\tau$ can only differ from φ by the fact that some occurrences of the "$\bullet$" operator are replaced by its dual "$\neg \bullet \neg$". As a consequence, a formula φ has at most 2^{n_φ} distinct derivatives, where n_φ is the number of "$\bullet$" operators appearing in φ. *[end of proof]*

As a consequence of the above propositions, the transition system $(Q_\varphi, \varphi, 2^A \times \{true, false\}, \longrightarrow)$ is a finite, complete, deterministic, automaton. This result is well-known in temporal logic [VW86], but in the above presentation, it is also closely related with Brzozowski's theorem [Brz64, BS87] which expresses the termination of the algorithm of the "residual" on regular expressions, and which is the basis of the automaton generation in the ESTEREL compiler [BG92]. An important point is that a formula can be "interpreted on the fly": the rules can be used to check the satisfaction of a formula by a sequence of inputs, without building the whole automaton first.

4.2 Safety formulas

The rewriting $\psi \xrightarrow{\omega} \psi'$ of safety formulas means that there exists a trace $\omega.\tau$ satisfying ψ, and that such a trace satisfies ψ if and only if τ satisfies ψ'. In other words, a sequence $(\omega_0, \ldots, \omega_n, \ldots)$ satisfies ψ if and only if there exists a sequence $(\psi_0, \ldots, \psi_n, \ldots)$ such that $\psi = \psi_0$, and for every n, $\psi_n \xrightarrow{\omega_n} \psi_{n+1}$.

Let $\psi = \exists\vec{x}\Box\varphi$ (where $\vec{x}$ denotes a vector $(x_0, \ldots, x_k)$ of auxiliary variables), with basic alphabet *Prop*. An automaton for ψ can be deduced from the automaton $\mathcal{A}(\varphi)$ associated with φ, as follows:

- All the inputs are projected onto *Prop*;
- Only the transitions giving the output *true* are retained.

There is only one rule: $$\frac{\varphi \xrightarrow{\omega:true} \varphi'}{\exists\vec{x}\Box\varphi \xrightarrow{\omega\cap Prop} \exists\vec{x}\Box\varphi'}$$

Here again, we get a finite automaton accepting the models of the formula. Notice that this automaton can be non deterministic because of the projection onto *Prop*.

Example: Figure 4 shows the automaton associated with the safety formula

$\exists x\Box((x \equiv \bullet(a \vee x)) \ \wedge \ b\supset x)$

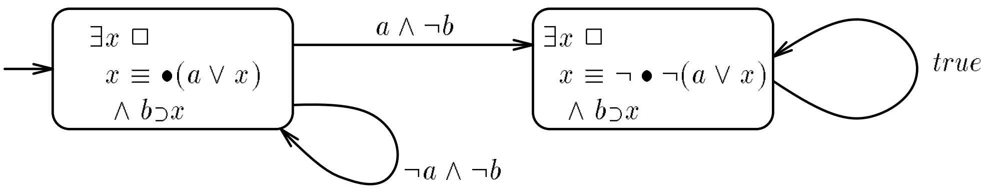

Figure 4: The automaton of the formula $\exists x\Box((x \equiv \bullet(a \vee x)) \ \wedge \ b\supset x)$

We also show below the "execution" (on-line interpretation) of this formula according to the sequence of inputs $\{\}, \{a\}, \{b\}$:

- Step 1: input $\{\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{a \vee x \xrightarrow{\{\}:false} a \vee x}{\bullet(a \vee x) \xrightarrow{\{\}:false} \bullet\,(a \vee x)}}{x \equiv \bullet(a \vee x) \xrightarrow{\{\}:true} x \equiv \bullet(a \vee x)} \qquad b\supset x \xrightarrow{\{\}:true} b\supset x}{x \equiv \bullet(a \vee x) \ \wedge \ b\supset x \xrightarrow{\{\}:true} x \equiv \bullet(a \vee x) \ \wedge \ b\supset x}}{\exists x\Box(x \equiv \bullet(a \vee x) \ \wedge \ b\supset x) \xrightarrow{\{\}} \exists x\Box(x \equiv \bullet(a \vee x) \ \wedge \ b\supset x)}$$

– Step 2: input $\{a\}$

$$\dfrac{\dfrac{\dfrac{a \vee x \stackrel{\{a\}:true}{\longrightarrow} a \vee x}{\bullet(a \vee x) \stackrel{\{a\}:false}{\longrightarrow} \neg \bullet \neg(a \vee x)}}{x \equiv \bullet(a \vee x) \stackrel{\{a\}:true}{\longrightarrow} x \equiv \neg \bullet \neg(a \vee x)} \quad b \supset x \stackrel{\{a\}:true}{\longrightarrow} b \supset x}{\dfrac{x \equiv \bullet(a \vee x) \ \wedge \ b \supset x \stackrel{\{a\}:true}{\longrightarrow} x \equiv \neg \bullet \neg(a \vee x) \ \wedge \ b \supset x}{\exists x \Box(x \equiv \bullet(a \vee x) \ \wedge \ b \supset x) \stackrel{\{a\}}{\longrightarrow} \exists x \Box(x \equiv \neg \bullet \neg(a \vee x) \ \wedge \ b \supset x)}}$$

– Step 3: input $\{b\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \vee x \stackrel{\{b,x\}:true}{\longrightarrow} a \vee x}{\neg(a \vee x) \stackrel{\{b,x\}:false}{\longrightarrow} \neg(a \vee x)}}{\bullet\neg(a \vee x) \stackrel{\{b,x\}:false}{\longrightarrow} \bullet \neg(a \vee x)}}{\neg \bullet \neg(a \vee x) \stackrel{\{b,x\}:true}{\longrightarrow} \neg \bullet \neg(a \vee x)}}{x \equiv \neg \bullet \neg(a \vee x) \stackrel{\{b,x\}:true}{\longrightarrow} x \equiv \neg \bullet \neg(a \vee x)} \quad b \supset x \stackrel{\{b,x\}:true}{\longrightarrow} b \supset x}{\dfrac{x \equiv \bullet(a \vee x) \ \wedge \ b \supset x \stackrel{\{b,x\}:true}{\longrightarrow} x \equiv \neg \bullet \neg(a \vee x) \ \wedge \ b \supset x}{\exists x \Box(x \equiv \neg \bullet \neg(a \vee x) \ \wedge \ b \supset x) \stackrel{\{b\}}{\longrightarrow} \exists x \Box(x \equiv \neg \bullet \neg(a \vee x) \ \wedge \ b \supset x)}}$$

Since it is accepted, the sequence $\{\}, \{a\}, \{b\}$ is a model of the formula.
[end of example]

5 The deterministic fragment

As noticed before, the automaton of a safety formula may be non deterministic, because of the projection onto *Prop*. Let us identify now a syntactic fragment of SL, for the formulas of which the above process produces deterministic automata.

Let DSL be the fragment of SL consisting of formulas of the form

$$\exists x_1, x_2, \ldots, x_k \ \Box(\varphi \wedge (x_1 \equiv \varphi_1) \wedge \ldots \wedge (x_k \equiv \varphi_k))$$

where
- φ is a past formula
- For each $i = 1 \ldots k$, φ_i is a past formula, where auxiliary variables may only appear under a $\bullet$ operator.

Proposition 4

1. *For any DSL formula ψ, $\mathcal{A}(\psi)$ is deterministic.*
2. *The class of properties expressible in DSL is again the class of regular safety properties.*

Proof: As noticed before, non determinism in the automaton can only appear during the elimination of auxiliary variables. Since auxiliary variables only appear under a $\bullet$ operator in φ_i ($i = 1 \ldots k$), the value of φ_i in each state is completely determined by the value of propositional symbols. Since the value of φ_i determines the value of x_i, the projection onto *Prop* doesn't introduce non determinism. This establishes the point (1).

To prove the point (2), we show that any prefix-closed and limit-closed regular language can be characterized by a DSL formula: Any such language can be characterized by a deterministic safety automaton. Now, consider the SL formula associated with a safety automaton $\mathcal{A}$ by the construction defined in the proof of the proposition 1. One can easily see that, if $\mathcal{A}$ is deterministic, all the implications appearing in the formula can be replaced by equivalences, and thus the formula belongs to DSL.
[end of proof]

Example: The automaton of Figure 5 is the result of determinizing the automaton of Figure 3. A DSL formula characterizing the same language is:

$$\exists x_0, x_1, x_2 \text{ such that } \quad \Box \begin{array}{lcl} x_0 & \equiv & \neg \bullet \neg(x_1 \wedge b) \vee \neg \bullet \neg(x_2 \wedge b) \\ \wedge \quad x_1 & \equiv & \bullet(x_0 \wedge a) \\ \wedge \quad x_2 & \equiv & \bullet(x_1 \wedge c) \\ \multicolumn{3}{l}{\wedge \neg(a \wedge b) \wedge \neg(b \wedge c) \wedge \neg(c \wedge a) \wedge (a \vee b \vee c)} \\ \multicolumn{3}{l}{\wedge (x_0 \vee x_1 \vee x_2)} \end{array}$$

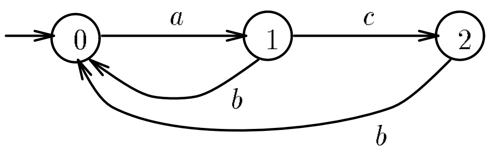

Figure 5: A deterministic automaton

[end of example]

Connection with Boolean Lustre

With each DSL formula ψ can be associated a Boolean LUSTRE program $\Pi(\psi)$, with only one Boolean output which is always *true* if and only if the sequence of inputs is a model of ψ. The translation is just a matter of syntax, defined by the following rules:

- *Past formulas*:

  $\Pi(a) =$ `a` , ($a \in Prop \cup Aux$) $\qquad \Pi(\neg\phi) =$ `not` $\Pi(\varphi)$

  $\Pi(\varphi_1 \vee \varphi_2) = \Pi(\varphi_1)$ `or` $\Pi(\varphi_2)$ $\qquad \Pi(\bullet\varphi) =$ `false -> pre(`$\Pi(\varphi)$`)`

- *DSL formulas*: Let $\psi = \exists x_1, x_2, \ldots, x_k\ \Box(\varphi \wedge (x_1 \equiv \varphi_1) \wedge \ldots \wedge (x_k \equiv \varphi_k))$ be a DSL formula, with $Prop = \{p_1, p_2, \ldots, p_n\}$. Then $\Pi(\psi)$ is the following program

```
node PSI (p1,p2,...,pn:bool) returns (psi:bool);
var x1,x2,...,xk:bool;
let
   psi = Π(φ);
   x1 = Π(φ1);
   x2 = Π(φ2);
     ...
   xk = Π(φk);
tel
```

The inverse translation is slightly more complicated. First of all, the program must be expanded, i.e., each node call must be replaced by the node body, after suitable parameter passing and local variable renaming. The result is a single node, whose body is a "flat" system of equations. This expansion is the first step performed by the LUSTRE compiler. In order to translate the Boolean LUSTRE expressions into past formulas, two problems remain, which both are due to initializations: In LUSTRE, the result of a "`pre`" operator is undefined ("*nil*") at the initial instant, while in SL, the result of "$\bullet$" is initially *false*. We have also to translate the LUSTRE initialization operator, "`->`".

- *The problem of "nil"*: A correct LUSTRE program may never output an undefined value. The compiler checks that no "*nil*" value can influence the outputs of the program, and otherwise rejects the program. So, for a correct program, "*nil*" can be replaced by any value, without changing the meaning of the program: The "`pre`" operator can be indifferently translated either into "$\bullet$" or into "$\neg \bullet \neg$".

- *The "`->`" operator*: The initialization operator can be straightforwardly translated, using an auxiliary variable, say "*init*", whose value is *true* only at the first instant: In DSL, this variable is defined by the equation "$init \equiv \neg \bullet \neg false$". Then, if we note $\Phi(\mathtt{E})$ the formula associated with the LUSTRE expression `E`, we have

$$\Phi(\mathtt{E1\ ->\ E2}) = (init \wedge \Phi(\mathtt{E1})) \vee (\neg init \wedge \Phi(\mathtt{E2}))$$

References

[AW85] E. A. Ashcroft and W. W. Wadge. LUCID*, the data-flow programming language.* Academic Press, 1985.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[Brz64] J. A. Brzozowski. Derivative of regular expressions. *JACM*, 11(4), 1964.

[BS87] G. Berry and R. Sethi. From regular expressions to deterministic automata. *TCS*, 25(1), 1987.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, Munchen*, January 1987.

[HCRP91] 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.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7), 1992.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs, LNCS 194*. Springer Verlag, 1985.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS*, 6(1), January 1984.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.

[Pnu92] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. In *CONCUR'92, Stony Brook, LNCS 630*, August 1992.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th Conference on Principles of Programming Languages*. ACM, 1989.

[Rat92] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thesis, Université Joseph Fourrier, Grenoble, June 1992.

[Sis83] A. P. Sistla. *Theoretical issues in the design and verification of distributed systems*. PhD thesis, Harvard University, 1983.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, June 1986.