Synchronous Observers and the Verification of Reactive Systems

Nicolas Halbwachs
Verimag Laboratory* and Stanford University†

Fabienne Lagnier, Pascal Raymond
Verimag Laboratory*
Rue Lavoisier, 38330 - Montbonnot St.Martin, France

Introduction

Synchronous programming [20, 14] is a useful approach to design reactive systems. A synchronous program is supposed to *instantly* and *deterministically* react to events coming from its environment. The advantages of this approach have been pointed out elsewhere [20]. Synchronous languages are simple and clean, they have been given simple and precise formal semantics, they allow especially elegant programming style. They conciliate concurrency (at least at the description level) with determinism. They can be compiled into a very efficient sequential code, by means of a specific compiling technique: The control structure of the object code is a finite automaton which is synthesized by an exhaustive simulation of a finite abstraction of the program.

Concerning program verification, it has been argued [8, 16, 29] that the practical goal, for reactive programs, is generally to verify some simple logical safety properties: By a *safety* property, we mean, as usual, a property which expresses that something will never happen, and by a *simple logical* property, we mean a property which depends on logical dependences between events, rather than on complex relations between numerical values.

For the verification of such properties also, the synchronous approach has some advantages: Since the parallel composition is synchronous, the desired properties of a program can be easily and modularly expressed by means of an *observer*, i.e., another program which observes the behavior of the first one and decides whether it is correct. Thus, the same language is used to write the program and its desired properties. The verification then consists in checking that the parallel composition of the program and its observer never causes the observer to complain. This verification can often be performed by traversing the finite control automaton built by the compiler. This automaton is generally much smaller than in the asynchronous case, where non-deterministic interleaving of processes is likely to result in state explosion.

*Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and Vérilog SA associated with IMAG.

†This work was performed while the first 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.

An observer can also be used to express known properties of the program environment. As a reactive system is embedded into an environment with which it tightly interacts, the environment must be strongly taken into account in program design and verification. Generally, the critical properties of a reactive system are only required to hold provided the environment also behaves correctly, that is, under some assumptions about the environment. In [17], we verified a very simple railways control system, and the most important part was the description of the realistic behavior of the trains (they obey the signals, they do not jump from one track to another, etc.). In [16], we used this ability of taking the environment into account in the verification, to propose a modular verification technique: When two processes run in parallel, each of them is part of the other's environment; so any property which is proved about one of them, can be used as an assumption about the other's environment.

So, our verification approach can be summarized by three simple ideas: (1) restriction to safety properties; (2) expression of these properties by means of a synchronous, deterministic observer; (3) taking into account assumptions about the environment. This paper is a survey of our specification and verification techniques, in a very general, language independent, framework. Section 1 introduces a simple model of synchronous input/output machines, which will be used throughout the paper. In section 2, we show how such a machine can be designed to check the satisfaction of a safety property, and we discuss the use of such an observer in program verification. In section 3, we use an observer to restrict the behavior of a machine. This is the basic way for representing assumptions about the environment. Applications to modular and inductive verification are considered. In modular verification, one has to find, by intuition, a property of a subprogram that is strong enough to allow the verification of the whole program without fully considering the subprogram. In section 4, we consider the automatic synthesis of such a property, and in section 5, we investigate the possibility of deducing the subprogram from such a synthesized specification.

1 Synchronous I/O machines

We first define an abstract model of synchronous reactive machines. We could use a synchronous process algebra [27, 28, 1] as a basic formalism. but we will see that *non symmetric communication* is essential for the definition of observer: An observer can see the behavior of the program *without modifying it*, i.e., without additional synchronization. So, we prefer to use a notion of synchronous machine where inputs and outputs do not play a symmetric role. In the following model, as in synchronous languages, outputs are non blocking and synchronously broadcast. Moreover, we will need an explicit notion of state, which lacks in process algebras.

1.1 Definitions

Let us consider a set S of *signals*, and let $E_S = 2^S$ be the set of *events*[1] on S. An *I/O machine* M is a 5-tuple $(Q_M, q0_M, I_M, O_M, \delta_M)$ such that

[1] Events, with the union operation, will play the role of the "monoid of actions" in synchronous process algebras.

- Q_M is a set of states containing $q0_M$, the initial state;
- $I_M \subset S$, $O_M \subset S$ are the disjoint sets of input and output signals, respectively.
- $\delta_M \subseteq Q_M \times E_{I_M} \times E_{O_M} \times Q_M$ is the transition relation. When there is no ambiguity about the considered relation, we will often note "$q \xrightarrow[o]{i} q'$" instead of "$(q, i, o, q') \in \delta_M$".

Intuitively, in response to a sequence $(i_1, i_2, \ldots, i_n, \ldots)$ of input events, such a machine returns a sequence $(o_1, o_2, \ldots, o_n, \ldots)$ of output events, such that there exists a sequence $(q_0, q_1, \ldots, q_n, \ldots)$ of states, with $q_0 = q0_M$ and for all $n \geq 1$, $q_{n-1} \xrightarrow[o_n]{i_n} q_n$. The sequence $((i_1 \cup o_1), (i_2 \cup o_2), \ldots, (i_n \cup o_n), \ldots)$ will then be called a *trace* of the machine.

If $\sigma = ((i_1 \cup o_1), (i_2 \cup o_2), \ldots, (i_n \cup o_n))$ is a finite trace, and $(q_0, q_1, \ldots, q_n)$ is a corresponding sequence of states, we will note $q0_M \xrightarrow{\sigma} q_n$. For any state q, we will note $traces(q)$ the set $\{\sigma \mid q0_M \xrightarrow{\sigma} q\}$ of traces leading to q. This notation is extended to sets of states: For any $X \subseteq Q_M$, $traces(X) = \bigcup_{q \in X} traces(q)$.

Let us note δ^r_M the *reaction function* from $Q_M \times E_{I_M}$ into $2^{E_{O_M} \times Q_M}$, defined by

$$\delta^r_M = \lambda(q, i).\{(o, q') \mid (q, i, o, q') \in \delta_M\}$$

A *reactive* machine cannot refuse a non-empty input event, and thus satisfies the following property: $\forall q \in Q_M, \forall i \subseteq I_M,\ i \neq \emptyset \implies \delta^r_M(q, i) \neq \emptyset$.

A *deterministic* machine has at most one possible reaction to a given input event, and thus satisfies: $\forall q \in Q_M, \forall i \subseteq I_M,\ |\delta^r_M(q, i)| \leq 1$. For a deterministic machine, we will note δ^O_M (respectively δ^Q_M) the *function* giving, for a state q and an input event i, the output event o (resp. the next state q') such that (q, i, o, q') belongs to δ_M.

We will use the usual *precondition* and *postcondition* functions, from 2^{Q_M} to 2^{Q_M}: For any $X \subseteq Q_M$,

- $post_M(X)$ is the set of successors of states belonging to X:

$$post_M(X) \;=\; \{q' \mid \exists q \in X, \exists i, o,\ q \xrightarrow[o]{i} q'\}$$

- $pre_M(X)$ is the set of states having a successor state in X:

$$pre_M(X) \;=\; \{q \mid \exists q' \in X, \exists i, o,\ q \xrightarrow[o]{i} q'\}$$

- $\widetilde{pre}_M(X)$ is the set of states having *all* their successors in X:

$$\begin{aligned}\widetilde{pre}_M(X) &= \{q \mid \forall i, \forall o, \forall q' \text{ such that } q \xrightarrow[o]{i} q',\ q' \in X\} \\ &= Q_M \setminus pre_M(Q_M \setminus X)\end{aligned}$$

1.2 Operations on I/O machines

Projection: Let M be an I/O machine, and $O' \subseteq O_M$. The *projected machine* $M \downarrow O'$ is $(Q_M, q0_M, I_M, O', \delta')$, where $\delta' = \{(q, i, o \cap O', q') \mid (q, i, o, q') \in \delta_M\}$.

Obviously, if M is reactive (respectively, deterministic), so is $M \downarrow O'$.

Synchronous product: Let M_1 and M_2 be two I/O machines, with $O_{M_1} \cap O_{M_2} = \emptyset$[2]. We define their *synchronous product* $M_1||M_2$ to be the I/O machine M where

- $Q_M = Q_{M_1} \times Q_{M_2}\ ,\quad q0_M = (q0_{M_1}, q0_{M_2})$
- $I_M = (I_{M_1} \setminus O_{M_2}) \cup (I_{M_2} \setminus O_{M_1})\ ,\quad O_M = O_{M_1} \cup O_{M_2}$
- $((q_1, q_2), i, o, (q_1', q_2')) \in \delta_M \iff \quad (q_1, (i \cup o) \cap I_{M_1}, o \cap O_{M_1}, q_1') \in \delta_{M_1}$ and $(q_2, (i \cup o) \cap I_{M_2}, o \cap O_{M_2}, q_2') \in \delta_{M_2}$

In other words, a transition of the product involves a transition of each machine, triggered by the global input signals *and the signals emitted by the other machine*.

1.3 Causality

With this very loose definition of the synchronous product, it can happen that the product of two deterministic (respectively reactive) machines is not deterministic (resp. reactive). This is the well-known problem of *causality paradoxes* in synchronous languages [6, 26]. For instance, let $I_{M_1} = \{x, y\}, I_{M_2} = \{x, z\}, O_{M_1} = \{z\}$ and $O_{M_2} = \{y\}$. Then:

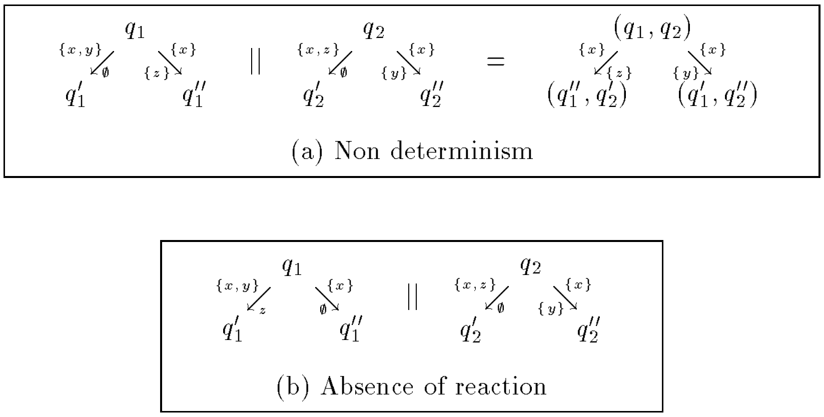
• Assume (see Fig.1.a) that $q_1 \xrightarrow[\emptyset]{\{x,y\}} q_1'$ and $q_1 \xrightarrow[\{z\}]{\{x\}} q_1''$ are the only transitions in δ_{M_1} from state q_1, and that $q_2 \xrightarrow[\emptyset]{\{x,z\}} q_2'$ and $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$ are the only transitions in δ_{M_2} from state q_2. If the input event $\{x\}$ occurs when the product machine $M_1||M_2$ is in the state (q_1, q_2), two different transitions can take place:

- either M_1 performs $q_1 \xrightarrow[\{z\}]{\{x\}} q_1''$ and then the emission of z forces the transition $q_2 \xrightarrow[\emptyset]{\{x,z\}} q_2'$ in M_2. So the compound transition is $(q_1, q_2) \xrightarrow[\{z\}]{\{x\}} (q_1'', q_2')$;
- or, conversely, M_2 performs $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$, forcing the transition $q_1 \xrightarrow[\emptyset]{\{x,y\}} q_1'$ in M_1, and the resulting global transition is $(q_1, q_2) \xrightarrow[\{y\}]{\{x\}} (q_1', q_2'')$.

So, in that case, the product of two deterministic machines is non deterministic.

• Assume now (Fig. 1.b) that $q_1 \xrightarrow[\{z\}]{\{x,y\}} q_1'$ and $q_1 \xrightarrow[\emptyset]{\{x\}} q_1''$ are the only transitions in δ_{M_1} from state q_1, and that δ_{M_2} is as before. Now, if the input event $\{x\}$ occurs in the state (q_1, q_2), the global system has no legal behavior, since:

[2] The restriction that parallel machines don't share common output signals is for simplicity only. It does not exist in Esterel [6] and Argos [26].



(a) Non determinism

(b) Absence of reaction

Figure 1: Synchronous product

- if M_2 performs $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$, then the emission of y forces the transition $q_1 \xrightarrow[\{z\}]{\{x,y\}} q_1'$ in M_1. But now, since z is emitted, M_2 should not have made its transition.
- Conversely if M_1 performs $q_1 \xrightarrow[\emptyset]{\{x\}} q_1''$, since z is not emitted, M_2 must perform $q_2 \xrightarrow[\{y\}]{\{x\}} q_2''$ and the emission of y forbids the transition of M_1.

So, in that case, the product of two reactive machines is not reactive.

An important feature of synchronous languages is that their parallel composition operator (synchronous product) introduces neither non-determinism nor deadlock. Compile-time consistency checks insure that the compound machine has a *unique, minimal,* reaction to each input event (see for instance [14] for details): Let M_1 and M_2 be two deterministic and reactive I/O machines, let $\delta^O_{M_1}$, $\delta^O_{M_2}$ be their respective output functions. When $M_1 || M_2$ is in the state (q_1, q_2) and receives an input event i, the output event o must satisfy

$$o = \delta^O_{M_1}(q_1, (i \cup o) \cap I_{M_1}) \;\cup\; \delta^O_{M_2}(q_2, (i \cup o) \cap I_{M_2})$$

i.e., be a fixpoint of the function

$$\lambda x.\, \delta^O_{M_1}(q_1, (i \cup x) \cap I_{M_1}) \;\cup\; \delta^O_{M_2}(q_2, (i \cup x) \cap I_{M_2})$$

Causality problems come from the fact that this function is not always monotone, and thus, may admit zero or several minimal fixpoints. Compile-time consistency checks insure the existence and unicity of a least fixpoint, and the synchronous product is defined by

$$\delta^O((q_1, q_2), i) = \mu x.\, \delta^O_{M_1}(q_1, (i \cup x) \cap I_{M_1}) \;\cup\; \delta^O_{M_2}(q_2, (i \cup x) \cap I_{M_2})$$

$$\begin{aligned}\delta^Q((q_1, q_2), i) = \Big(\;& \delta^Q_{M_1}(q_1, (i \cup \delta^O((q_1, q_2), i)) \cap I_{M_1}) \,, \\ & \delta^Q_{M_2}(q_2, (i \cup \delta^O((q_1, q_2), i)) \cap I_{M_2}) \;\Big)\end{aligned}$$

(where, as usual, $\mu x.f$ denotes the least fixpoint of the function $\lambda x.f$).

2 Observers of safety properties

In this section, we show how a safety property can be specified by means of a synchronous observer. Such an observer is an I/O machine, taking as inputs both the input and the output signals of the machine under observation, and emitting an "alarm" signal as soon as the observed signals do not satisfy the property.

2.1 Safety properties

A *trace* σ on a set of signals S is a finite or infinite sequence of events on S. A *property* on S is a set of traces on S. An I/O machine M satisfies a property P if and only if each trace of M belongs to P. A property P on S is a *safety property* if and only if:

$$\sigma \in P \Longleftrightarrow \sigma' \in P \text{ for any finite prefix } \sigma' \text{ of } \sigma$$

In other words, a safety property is a prefix-closed (as expressed by the "$\Longrightarrow$" implication above) and limit-closed (as expressed by the "$\Longleftarrow$" implication) language on the vocabulary 2^S.

2.2 Observer

Let P be a safety property on S. Let α (read "alarm") be a signal not in S. An *observer* of P is a *deterministic* and *reactive* I/O machine $\Omega_P = (Q_{\Omega_P}, q0_{\Omega_P}, S, \{\alpha\}, \delta_{\Omega_P})$, returning a sequence of empty output events as long as it receives a sequence of input events which belongs to P. More precisely, let σ be a finite trace on S belonging to P (notice that the empty trace belongs to any safety property). Let q_σ be the state that Ω_P reaches after reading σ (if σ is the empty trace, q_σ is the initial state of Ω_P). Then, for any event $e \in 2^S$,

$$\delta^O_{\Omega_P}(q_\sigma, e) = \begin{cases} \emptyset & \text{if } \sigma.e \in P \\ \{\alpha\} & \text{otherwise} \end{cases}$$

Let us assume also that any transition emitting α leads to a distinguished state q_α.

Now, a machine M satisfies a safety property P if and only if the compound machine $M||\Omega_P$ never returns any event containing α; or, equivalently, never reaches an *erroneous* state belonging to $Q_M \times \{q_\alpha\}$. We will note Q_P^M the set $Q_M \times (Q_{\Omega_P} \backslash \{q_\alpha\})$ of non erroneous states of $M||\Omega_P$.

A practical advantage of this approach, is that the properties are written in the same language as the programs, and in fact, properties are programs. As such, they can be executed and tested. An observer can be actually run with the program, thus detecting any violation of the property (run-time checks).

Notice that this approach cannot be used with only an asynchronous composition, or at least, that it cannot be applied *modularly*. For instance, consider the following property: "*the signal* `b` *is emitted at least once between every* *two successive emissions of the signal* a ". If this property is checked by an asynchronous observer, since the observer is not guaranteed to catch all the signals, it can miss any occurrence of b. So, even if the property is satisfied, the observer can emit an alarm. To check such a property of an asynchronous program, one must add some synchronization code all along the transitions of the observed program, since otherwise, the asynchronous product does not ensure that all the transitions will be observed. When verifying a program, such modifications are of course harmful, since one cannot be sure that the verified program behaves the same once the additional code is removed. This contradicts G. Berry's "WYPIWYE" principle ("*what you prove is what you execute*") which fully applies in the synchronous case.

2.3 Application to program verification

The verification that a machine M satisfies a safety property P now amounts to proving that the machine $M' = M || \Omega_P$ never returns any event containing α. So, any safety property has been translated into an *invariant*. More precisely, one has to prove that the set $Reach(M')$ of M' *reachable states* is included in the set Q_P^M of non erroneous states of M'. $Reach(M')$ is classically defined as a least fixpoint:

$$Reach(M') = \mu X.\{q0_{M'}\} \cup post_{M'}(X)$$

Let us list the advantages of this expression of the verification problem, according to various verification methods:

State enumeration: For finite state systems, state enumeration techniques (enumerative model-checking) have been widely experimented [31, 11]. In general, these techniques involve the construction of the whole state graph of the program, and its memorization for the analysis of *trace properties*. Now, since the problem has been reduced to the analysis of a *state property* (an invariant), the state graph needs only to be *traversed*. Particularly efficient techniques are available (e.g., [18]) for such a traversal.

Reduction techniques: The drawback of state enumeration techniques is the explosion of the number of states, as the size of the program increases[3]. Other approaches [7] consist in building a reduced state graph, according to some observation criteria. Now, in our approach, the machine of interest is not really $M||\Omega_P$, but rather $(M||\Omega_P) \downarrow \alpha$, since we are only interested in the presence of the signal α. This is an obvious observation criterion. So, in contrast with classic model-checking, the property is taken into account in the state graph generation. Assume the property is satisfied, then the minimal state graph of $(M||\Omega_P) \downarrow \alpha$ has only one state (it is the "always silent" automaton). Algorithms for generating a minimal state graph have been proposed [5, 25]. When applied to our simple verification problem, these algorithms amount to proving that the initial state belongs to the greatest invariant $Invar(Q_P^M)$ included in Q_P^M, i.e., the greatest part of Q_P^M from which the transition relation

[3]Notice that the state explosion is more important in an asynchronous system, because of the non deterministic interleaving of asynchronous transitions.

does not permit to go out. This greatest invariant is wellknown to be a greatest fixpoint:

$$Invar(Q_P^M) = \nu X. Q_P^M \cap \widetilde{pre}_{M||\Omega_P}(X)$$

Approximate analysis: When infinite state systems are considered, approximate methods (and, in particular, *abstract interpretation techniques* [9, 10]) can be applied to compute approximations of the set $Reach((M||\Omega_P) \downarrow \alpha)$. If an upper approximation of this set is included in Q_P^M, this proves that the erroneous states cannot be reached (see [13] for an application of such a method). If a lower approximation intersects the complement of Q_P^M, an error is detected.

In the remainder of the paper, we will essentially consider finite state machines, so all the considered fixpoints will be (theoretically) computable. In the following section, we will see that property observers can also be used to take into account known properties of the program environment.

3 Taking the environment into account

The main feature of reactive systems is that they tightly interact with their environment. As a consequence, the properties of the environment must be carefully taken into account in the design and verification of such a system. A reactive system is not intended to work in an *arbitrary* environment. In general, system specifications contain a lot of informations about the behavior of the environment, which are the hypotheses under which the design must take place. These known properties about the environment can involve not only the inputs of the system, but also its outputs, since the environment responds to the system. So, in general, among the set of traces of an I/O machine, only some of them are "realistic", i.e., satisfy the assumptions about the environment. In this section, we show how the behavior of an I/O machine can be restricted by a safety property, in order to take such assumptions into account in the verification process.

3.1 Behavior restriction

Given a safety property A (assumption) of the environment of M, our goal is to define a *restricted* machine M' having exactly the same behaviors as M composed with any environment satisfying A: the set of traces of M' must be the intersection with A of the set of traces of M.

Restriction: Let M be an I/O machine, and Ω_A be an observer of a safety property A on the set $S = I_M \cup O_M$ of input/output signals of M. Let $M' = M||\Omega_A$. We define the restriction M/Ω_A to be the I/O machine $(Q_{M'}, q0_{M'}, I_M, O_M, \delta')$, where $\delta' = \{(q, i, o, q') \in \delta_{M'} \mid \alpha \notin o\}$

Obviously, the restricted machine M/Ω_A is generally not reactive, even if M is reactive: The restriction takes into account a property of the environment, and thus, refuses some unrealistic inputs. However, it can happen that in some states of the restricted machine, *all the input events* are refused. So, the restricted machine deadlocks, a highly undesirable situation in reactive systems. One can consider this as an error in the expression of the assumption A. However, we adopt another point of view: When restricting a machine M with an assumption A, the user intends to consider all the *infinite traces* of M that satisfy A. So, the machine must not enter any path in M/Ω_A which *inevitably* leads to a deadlock state. We define now another restriction, called *non-blocking restriction*, which has the intended behavior:

Non-blocking restriction: Let M be an I/O machine, and Ω_A be an observer of a safety property A on the set $S = I_M \cup O_M$ of input/output signals of M. Let $M' = M||\Omega_A$. Let us call $sink_A$ the set of states of M' leading inevitably to the violation of A:

$$sink_A = \mu X.\widetilde{pre}_{M'}((Q_M \times \{q_\alpha\}) \cup X)$$

Then, if $q0_{M'} \notin sink_A$, we define $M/_{\!\infty}\,\Omega_A$ to be the I/O machine $(Q_{M'} \setminus sink_A, q0_{M'}, I_M, O_M, \delta'')$, where

$$\begin{aligned} \delta'' &= \delta_{M'} \cap ((Q_{M'} \setminus sink_A) \times E_{I_M} \times E_{O_M} \times (Q_{M'} \setminus sink_A)) \\ &= \{(q, i, o, q') \in \delta_{M'} \mid q, q' \notin sink_A \text{ and } \alpha \notin o\} \end{aligned}$$

One can notice that, if M is deterministic, $M/_{\!\infty}\,\Omega_A = M/\Omega_{traces(Q_{M'} \setminus sink_A)}$. So, the property A has been strengthened into the other property $A' = traces(Q_{M'} \setminus sink_A)$ which cannot block the machine M: Any finite trace satisfying A' leads to state of M which has at least one outgoing transition preserving A'.

3.2 Application

As before, a direct use of this way of expressing assumptions by an observer, is to execute the observer with the program, thus checking at run-time that the assumptions are satisfied. The restriction can also be used for program testing, to use only testcases corresponding to realistic scenarios. We consider now the use of restriction in the verification process:

Verification under assumptions: Given an I/O machine M, a safety assumption A about its environment, and a safety property P, one can prove that M satisfies P provided the environment satisfies A, by

1. proving that $(M/_{\!\infty}\,\Omega_A)$ has some behaviors, i.e., that the initial state of $M||\Omega_A$ does not belong to $sink_A$. Otherwise, the assumption and the program are contradictory.
2. verifying that the machine $((M/_{\!\infty}\,\Omega_A)||\Omega_P) \downarrow \{\alpha\}$ emits only empty events (Of course, here, α is the alarm signal of Ω_P).

Modular verification: Any sub-process of a compound system sees the remainder of the system as a part of its own environment. The ability to take the environment into account allows modular verification: Having proved a property about a sub-process, one can use this property in the verification of the remainder of the system. More precisely, let M_1, M_2 be two machines, and let P be a safety property we want to prove about $M_1||M_2$. Assume another safety property P' has been proven about M_2 alone. Then if $M_1 /_{\infty} \Omega_{P'}$ satisfies P, so does $M_1 || M_2$. This amounts to considering M_2 as the environment of M_1. Of course, assumptions about the global environment can also be taken into account. With a little additional hypothesis (see [2] and the "decomposition theorem" of [23]), which amounts to the absence of causality problems, one can even use a seemingly circular reasoning, which consists first in proving a property P_2 of M_2 under the assumption that M_1 satisfies P_1, and then in proving that M_1 satisfies P_1 assuming M_2 satisfies P_2.

Inductive proofs: Moreover, the modular verification technique can be extended to the inductive verification of regular networks of processes [34, 16]. Assume one wants to prove a safety property P of the machine

$$\underbrace{M || M || \ldots || M}_{n \text{ times}}$$

for any $n \geq 1$. This can be done by finding a property P' such that (1) M satisfies P', (2) $(M /_{\infty} \Omega_{P'})$ satisfies P', and (3) P' implies P. (1) proves that P' holds for $n = 1$, (2) proves that, if P' holds for n, then it holds for $n + 1$. So, P' holds for any n, and from (3), so does P. Point (3) can be established by proving that the machine $\chi(I, O) /_{\infty} \Omega_{P'}$ satisfies P, where

$$\chi(I, O) = (\{q\}, q, I, O, \{q\} \times E_I \times E_O \times \{q\})$$

is the "chaos" machine which completely non deterministically returns any event of E_O whatever be its input event from E_I. Of course, as for modular verification, a crucial problem is the choice of the property P'. It is considered in the next section.

4 Specification synthesis

Let us come back to modular verification: Given two machines M_1 and M_2, and a safety property P on $S = I_{M_1} \cup O_{M_1} \cup I_{M_2} \cup O_{M_2}$, one must find a property P' on $S_2 = I_{M_2} \cup O_{M_2}$ such that

1. M_2 satisfies P', and
2. $M_1 /_{\infty} \Omega_{P'}$ satisfies P

Moreover, the proof of each of the above points is expected to be easier than the global proof that $M_1 || M_2$ satisfies P.

This section deals with the synthesis of such a property P', satisfying the point (2) above by construction, when all the involved machines are finite state.

First, we need some additional definitions: Let $\sigma = (e_1, e_2, \ldots, e_n, \ldots)$ be a trace on S. We define the *projection* of σ on a set S' of signals to be the trace $\sigma \downarrow S' = (e_1 \cap S', e_2 \cap S', \ldots, e_n \cap S', \ldots)$. The projection on S' of a set T of traces is $T \downarrow S' = \{\sigma \downarrow S' \mid \sigma \in T\}$. If T is a set of *finite* traces on S, we note $\mathcal{C}(T)$ the set of traces on S which do not have any prefix in T. Obviously, $\mathcal{C}(T)$ is a safety property.

The intuitive method to find P' is the following: Replace M_2 by the "chaos" machine $\chi(I_{M_2}, O_{M_2})$. If $M_1 || \chi(I_{M_2}, O_{M_2})$ satisfies P, the machine M_2 does not influence the satisfaction of P (i.e. we can take $P' = true$) and we are done. Otherwise, $M_1||\chi(I_{M_2}, O_{M_2})$ can reach some erroneous states, and the role of M_2 is to forbid the traces leading to those states. But, for doing so, M_2 can only restrict its own signals (P' cannot involve signals that M_2 cannot see).

More precisely: If $Reach(M_1||\Omega_P)$ does not intersect $Q_{M_1} \times \{q_\alpha\}$, let $P' = true$. Otherwise let $T_{\mathrm{err}} = traces(Q_{M_1} \times \{q_\alpha\})$ be the set of erroneous traces. The following proposition states that $\mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is a necessary and sufficient property that M_2 must satisfy so that $M_1||M_2$ satisfies P:

Proposition: Let $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$. Then $M_2 \models P' \iff M_1||M_2 \models P$.

Proof: Let $\sigma[n]$ denote the nth prefix of a trace σ.
($\Longrightarrow$): If $M_2 \models P'$, then every trace σ of $M1||M2$ satisfies $\sigma \downarrow S_2 \in \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$. So, $\forall n, (\sigma \downarrow S_2)[n] \notin T_{\mathrm{err}} \downarrow S_2$, and since $(\sigma \downarrow S_2)[n] = (\sigma[n] \downarrow S2)$, $\forall n, \sigma[n] \notin T_{\mathrm{err}}$. This means that $\sigma \in P$.
($\Longleftarrow$): Assume M_2 does not satisfy P', and let σ be a trace of M_2 not belonging to P'. Then, there exists n such that $\sigma[n] \in (T_{\mathrm{err}} \downarrow S_2)$, and there exists a trace $\sigma' \in T_{\mathrm{err}}$ such that $\sigma[n] = (\sigma'[n]) \downarrow (S_2)$. So, the finite trace $\sigma'[n]$ is compatible with both M_1 and M_2 and leads to the violation of P. □

Remark: $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is stronger than $P'' = \mathcal{C}(T_{\mathrm{err}}) \downarrow S_2$. A trace σ of M_2 can be the common projection of two traces σ' and σ'' of $M_1||M_2$, with $\sigma' \in \mathcal{C}(T_{\mathrm{err}})$ and $\sigma'' \notin \mathcal{C}(T_{\mathrm{err}})$. In that case, σ belongs to P'' (as the projection of σ') and not to P'.

Stronger specifications: However, the necessary and sufficient property $P' = \mathcal{C}(T_{\mathrm{err}} \downarrow S_2)$ is sometimes too complicated to be interesting: As a matter of fact, an observer of P' can be as complicated as $M_1||\Omega_P$. In that case the proof that M_2 satisfies P' is not easier than the proof that $M_1||M_2$ satisfies P, so nothing is gained with modular proof. Now, any stronger property than P' can be tried. Such a stronger property P'' will still ensure that $M_1/\!\!\!\!\angle_\infty\Omega_{P''}$ satisfies P, but, since it is no longer a necessary property, one cannot conclude that $M_1||M_2$ does not satisfy P if M_2 does not satisfy P'' .

The basic technique to build such a stronger property P'' is the following: Let us note the function $\lambda T.\, \mathcal{C}(T \downarrow S_2)$ by "*avoid*". Thus, $P' = avoid(T_{\mathrm{err}})$. Then, for any set T of traces containing T_{err} (i.e., for any upper approximation of T_{err}), $avoid(T)$ is stronger than P'.

5 Module synthesis

In the preceding section, we have outlined a method to find a property P' such that, for any machine M_2 satisfying P', $M_1||M_2$ satisfies P. P' has been only deduced from M_1 and P, so, it could be built even before M_2 is designed. So, the next question is: can M_2 be *synthesized* from P', considered as a specification? In the finite state case, this is theoretically possible: The specification must be strengthened to become *executable*. P' has been constructed so as to concern only the input/output signals of M_2. Now, an additional constraint is that M_2 must preserve P' by controlling only its output signals. In each reachable state, and whatever be the received input event (possibly satisfying an input assumption), M_2 must be able to perform a transition preserving P'.

Executability: A property P on a set of signals $S = I \cup O$ is *executable* with respect to (I, O), if and only if for any finite trace $\sigma \in P$, for any input event $i \in E_I$, there exists an output event $o \in E_O$ such that $\sigma.(i \cup o) \in P$. For any safety property P, there exists a weakest executable safety property, implying P. It will be noted $\mathcal{E}(P)$.

Relative precondition: Let P be a safety property on $I \cup O$ and Ω_P be an observer of P. For any $X \subseteq Q_{\Omega_P}$, we define

$$pre^I_{\Omega_P}(X) = \{q \mid \forall i \subseteq I, \exists o \subseteq O, \delta^Q_{\Omega_P}(q, i \cup o) \in X\}$$

In other words, $pre^I_{\Omega_P}(X)$ is the set of states which can lead into X (in one step) *whatever be the input event* received in these states.

Executable strengthening: Let $Exe = \nu X.\, pre^I_{\Omega_P}(X) \setminus \{q_\alpha\}$. Then Exe does not contain the erroneous state q_α, and

$$\forall q \in Exe,\ \forall i \subseteq I,\ \exists o \subseteq O, \text{ such that } \delta^Q_{\Omega_P}(q, i \cup o) \in Exe$$

Moreover, Exe is the largest set of states satisfying this property. As a consequence, a restriction of Ω_P which detects any trace going out of Exe is an observer of $\mathcal{E}(P)$. Another consequence is that $\chi(O, I)/\Omega_{\mathcal{E}(P)}$ is the most general reactive machine satisfying P. Notice that Exe can be empty, which means that P is not *realizable* in the sense of [3]: There is no machine on (I, O) preserving P against any environment.

Conclusion

Many ideas that have been presented are specializations and simplifications of previous works. For instance:

- The specification of properties by means of a synchronous observer is very close to the approach of COSPAN [24], which takes also into account liveness, both in the program and the properties.
- Several verification approaches take into account the environment, e.g., [21] [2] [22], and some of them propose modular methods. The "don't care sets" considered in hardware design and verification [4, 12] are also a way of expressing assumptions about the environment.
- The synthesis problems considered in Sections 4 and 5 have been dealt with in several papers — both in control theory [32, 33, 19], and in computer science [30, 3] — and often extended to cope with liveness properties.

Our simplifications consist in considering *safety properties* of *synchronous systems*. They are suggested by the application field we have in mind: The synchronous model has been shown to be very convenient for the design of reactive systems. In general, most liveness properties are introduced for one of the following reasons:

- To abstract a real-time constraint: For instance, one replace a deadline property by the requirement that something "eventually occurs". Now, in reactive systems, such real-time constraints may not be abstracted, in general: the constraint "*an alarm must be sent within 2 milliseconds after the detection of a dangerous situation*" may not be replaced by "*the alarm must eventually occur*"!

- To restrict the asynchronous semantics: In asynchronous models, concurrency is modelled by non-deterministic interleaving, and this non-determinism must be restricted by fairness constraints. Obviously, this problem does not exist in the synchronous model. In asynchronous systems, compositionality is also achieved by allowing arbitrary (but fair) "stuttering" of processes. The synchronous model is obviously compositional thanks to zero-time, simultaneous, reactions.

Now, these simplifications are certainly fruitful, from a practical point of view. They increase the performances of the automatic tools: For instance, for finite state methods, the synchronous model drastically reduces the size of the considered state graphs; safety properties can be checked by a graph traversal, without storing any path. To specify a safety property by means of an observer, one doesn't need to use — and to learn – any other language than the programming language used to write the program. All these ideas are under implementation in the Lustre- Saga software development system [15], and actual industrial experimentations are going on.

References

[1] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *TCS*, 30, April 1984.

[2] M. Abdi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*. LNCS 430, Springer Verlag, May 1989.

[3] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *16th ICALP*, pages 1–17. LNCS 372, Springer Verlag, July 1989.

[4] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on CAD/ICAS*, CAD-7(6):723–739, June 1988.

[5] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

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

[7] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1990.

[8] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, January 1977.

[10] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. Research Report LIX/RR/92/08, Ecole Polytechnique, March 1992. (to appear in the Journal of Logic Programming, special issue on Abstract Interpretation).

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[12] M. Damiani and G. DeMicheli. Don't care set specifications in combinational and synchronous logic circuits. Technical Report CSL-TR-92-531, Computer Systems Laboratory, Stanford University, 1992.

[13] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, July 1993.

[14] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[15] 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.

[16] 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.

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

[18] G. J. Holzmann. Automated protocol validation in Argos : Assertion proving and scatter searching. *IEEE Trans. on Software Ingineering*, SE-13(6):683–696, June 1987.

[19] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *American Control Conference, Chicago*, June 1992.

[20] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[21] B. Josko. MCTL - An extension of CTL for modular verification of concurrent systems. In *Workshop on Temporal Logic in Specification, Manchester.* LNCS 398, Springer Verlag, 1987.

[22] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29, February 1992.

[23] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, July 1993.

[24] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness.* LNCS 430, Springer Verlag, May 1989.

[25] D. Lee and M. Yanakakis. Online minimization of transition systems. In *24th ACM Symp. on the Theory of Computing, STOC'92, Vancouver, B.C.*, 1992.

[26] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92, Stony Brook.* LNCS 630, Springer Verlag, August 1992.

[27] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[28] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

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

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

[31] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *International Symposium on Programming.* LNCS 137, Springer Verlag, April 1982.

[32] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1), January 1987.

[33] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), January 1989.

[34] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble.* LNCS 407, Springer Verlag, 1989.