Automatic Verification of Parameterized Linear Networks of Processes

David Lesens, Nicolas Halbwachs, Pascal Raymond
VERIMAG*
Miniparc – ZIRST, F-38330 Montbonnot, France
e-mail: {David.Lesens|Nicolas.Halbwachs|Pascal.Raymond}@imag.fr

Abstract

This paper describes a method to verify safety properties of parameterized linear networks of processes. The method is based on the construction of a network invariant, defined as a fixpoint. Such invariants can often be automatically computed using heuristics based on Cousot's widening techniques. These techniques have been implemented and some non-trivial examples are presented.

1 Introduction

After the very first success of automatic verification of finite-state systems [QS82, CES86], Apt and Kozen [AK86] have shown that these techniques cannot be applied to regular networks of processes of unknown size. More formally, let $\{P_1, \ldots, P_k\}$ be a finite multi-set of processes, and $\{\times_1, \ldots, \times_k\}$ be a multi-set of binary composition operators over processes, and consider the family $\mathcal{F}$ of processes generated by:

$$\begin{array}{ll} \forall i = 1 \ldots k, & (P_i \in \mathcal{F}) \\ \text{and} & (P \in \mathcal{F} \Longrightarrow P \times_i P_i \in \mathcal{F}) \end{array}$$

The problem of verifying that every $P \in \mathcal{F}$ satisfies a property φ is undecidable [AK86], even if it is decidable for each $P_i, i = 1 \ldots k$.

Decidable subcases have been identified [EN95, EN96], but they are quite restrictive. Several attempts [KM89, WL89, HLR92] were made to solve the general case, using the following induction principle: Let $\preceq$ be a pre-order relation over processes, such that

$$P \preceq Q \ \wedge \ Q \models \varphi \quad \Longrightarrow \quad P \models \varphi$$

Find a process I, called a *network invariant*, such that

$$\forall i = 1 \ldots k, \quad P_i \preceq I \quad \text{and} \quad I \times_i P_i \preceq I$$

Then, $I \models \varphi \implies \forall P \in \mathcal{F}, P \models \varphi$.

Practically, the verification of an infinite family of processes raises two problems:

1. How to express the desired property φ independently of the number of component processes?
2. How to find a suitable network invariant I, if such an invariant exists?

In [CGJ95], nice solutions were proposed to both of these problems: first, they solve problem (1) by noticing that a state of a process in $\mathcal{F}$ is a multi-set of elementary process states; they propose to consider such a state as a word on the alphabet of elementary process states, and to specify a set of states as a language on this alphabet. Then, they consider the temporal logic $\forall$CTL*, where such *regular* languages are basic propositions. For this logic, a suitable choice for $\preceq$ is the simulation pre-order. For solving problem (2), they propose a very clever method, based on the construction of the syntactic monoid [Eil74] of a regular language, to build a network invariant I.

Let us comment about this proposal: the language-based specification technique is surely well-suited to linear networks of processes (as in the restricted problem setting we propose in this introduction), where a state of a compound process is naturally handled as a tuple of elementary process states. It may be less easy to specify in this way more complex structures, where a

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

compound state could be, for instance, a tree (as it is generally the case when the family $\mathcal{F}$ is generated by a network grammar [SG89]). In this paper, we propose an other specification method, based on *synchronous observers* [HLR93]. A synchronous observer is a process that is able to observe the behavior of a process without changing this behavior. In our approach, a state property is expressed by providing each elementary process with an observer, taking as input the input/output behavior of its associated elementary process, together with observations provided by the observers of the "neighbor processes" in the network. For the time being, we restrict ourselves to safety properties, and we use trace inclusion pre-order.

Concerning the construction of the network invariant, the method proposed in [CGJ95] can raise the following problem: if the synthesized invariant I does not satisfy the desired property φ, the method does not provide any way to look for a better invariant. In this paper, we first state the problem of invariant synthesis as the resolution of a fixpoint equation. Then we propose a set of heuristics, based on Cousot's *widening* techniques [CC77, CC92], to compute such fixpoints. The point is that the heuristic can be arbitrarily refined to get better invariants.

2 Basic Definitions

2.1 Traces, and Processes

The model of process we have in mind is that of synchronous languages [Hal93], like Esterel [BG92], Argos [Mar92], Statecharts [Har87], or Lustre [HCRP91]. A behavior of a process is a sequence of steps, each step resulting in an *event*, i.e., a set of present *signals*[1]. So, if X is a set of signals, we define a *trace* on X to be a (finite or infinite) sequence $\tau = (\tau_0, \ldots, \tau_n, \ldots)$ of subsets of X. Let Θ_X denote the set of traces on X.

We will not define a very precise notion of process. We just need to define the semantics of a process P to be the set T_P of its traces. Since we are only interested in safety properties, we will assume T_P to be prefix-closed. P is *regular* if T_P is a regular language.

Let X and X' be two disjoint sets of signals, and $\tau \in \Theta_X$ and $\tau' \in \Theta_{X'}$ be two traces of the same length. Then, $\tau \odot \tau'$ is a trace on $X \cup X'$, defined by

$$\tau \odot \tau' = (\tau_0 \cup \tau'_0, \ldots, \tau_n \cup \tau'_n, \ldots)$$

[1] In practice, these signals are partitioned into *input* signals (emitted by the environment) and *output* signals, emitted by the process, but, in general, we will not need to make this distinction.

This operation is extended to sets of traces: let $T \subseteq \Theta_X$ and $T' \subseteq \Theta_{X'}$ be two sets of traces, then

$$T \odot T' = \{\tau \odot \tau' \mid \tau \in T, \tau' \in T', |\tau| = |\tau'|\}$$

For instance, $T_P \odot T_{P'}$ will be the set of traces of the synchronous composition of two independent (i.e., not sharing signals) processes P and P'. We will often write $T \odot \Theta_{X'}$ to consider T as a subset of $\Theta_{X \cup X'}$, where the signals of X' are left unconstrained.

Let X and X' be two sets of signals of the same cardinality related to each other by a one-one mapping ϕ. Then, for each trace $\tau = (\tau_0, \ldots, \tau_n, \ldots)$ on X, $\tau[X/X']$ is the trace $(\tau'_0, \ldots, \tau'_n, \ldots)$ on X' defined by $\tau'_i = \{\phi(x) \mid x \in \tau_i\}$. This operation is also extended to sets of traces.

Let X and X' be two sets of signals, $T \subseteq \Theta_X$, $T' \subseteq \Theta_{X'}$ be two sets of traces. Then

$$T \otimes T' = (T \odot \Theta_{(X' \setminus X)}) \cap (T' \odot \Theta_{(X \setminus X')})$$

i.e., $T \otimes T'$ is the set of traces that agree on signals in $X \cap X'$. For instance, $T_P \otimes T_{P'}$ represents the traces of the synchronous product of two processes P and P', possibly communicating (by means of shared signals). We define also

$$T \oplus T' = (T \odot \Theta_{(X' \setminus X)}) \cup (T' \odot \Theta_{(X \setminus X')})$$

i.e., the union of T and T' as subsets of $\Theta_{(X \cup X')}$.

Let $T \subseteq \Theta_X$ be a set of traces, and Y be a subset of X. Then $\exists Y, T$ and $\forall Y, T$ are sets of traces on $X \setminus Y$ defined by

$$\begin{aligned}
\exists Y, T &= \{\tau \in \Theta_{X \setminus Y} \mid \exists \tau' \in \Theta_Y \text{ such that } \tau \odot \tau' \in T\} \\
\forall Y, T &= \{\tau \in \Theta_{X \setminus Y} \mid \forall \tau' \in \Theta_Y, \\
&\qquad (|\tau| = |\tau'|) \Rightarrow (\tau \odot \tau' \in T)\}
\end{aligned}$$

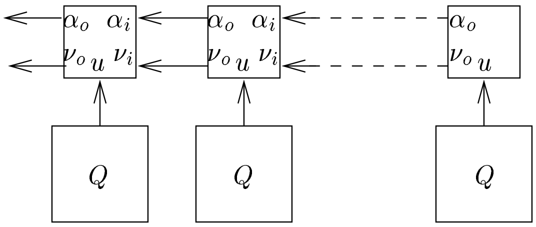
For instance, $\exists Y, T_P$ is the sets of traces of a process P where all signals in Y are considered internal (hidding). $\forall Y, T$ will be considered for duality, as

$$\forall Y, T = \neg \exists Y, \neg T$$

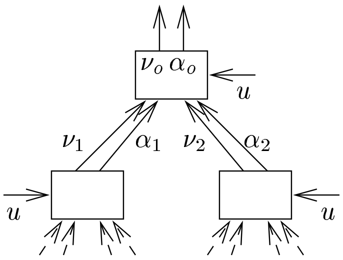
Example: Let $X = \{a, b\}$. Let us use boolean notations to write sets of subsets of X — e.g., writing $\overline{a}$ for $\{\ \{\}, \{b\}\ \}$ — and the standard notations of regular expressions to denote sets of traces on X. Let $T = (\overline{a})^* + (\overline{a}\overline{b}.ab)^*$. Then

$$\exists b, T = (\overline{a})^* + (\overline{a}.a)^* \qquad \forall b, T = (\overline{a})^*$$

[End of example]



(a) Linear structure



(b) Tree structure

Figure 1: Network observers

2.2 Properties and Observers

A *safety property* φ on the set of signals X is also a prefix-closed subset of Θ_X. With such a property φ, we associate another set of traces T_φ, called the traces of an *observer* [HLR93] of φ. Intuitively, an observer of φ is a process with input signals in X, which emits an "alarm signal" $\alpha \notin X$ whenever the input trace received so far does not belong to φ. So, $T_\varphi \subseteq \Theta_{X\cup\{\alpha\}}$, where α is a new signal, and

$$\forall\tau = (\tau_0, \ldots, \tau_n, \ldots) \in \Theta_X, \begin{cases} \tau \in T_\varphi & \text{if } \tau \in \varphi \\ \alpha(\tau) \in T_\varphi & \text{otherwise} \end{cases}$$

where $\alpha(\tau) = (\tau_0, \ldots, \tau_{n-1}, \tau_n \cup \{\alpha\}, \tau_{n+1} \cup \{\alpha\}, \ldots)$ where n is the least index such that $(\tau_0, \ldots, \tau_n) \notin \varphi$. T_φ is obviously prefix-closed, and,

$$\forall T \subseteq \Theta_X, \quad T \subseteq \varphi \Leftrightarrow T \otimes T_\varphi \subseteq \Theta_X$$

i.e., a process P satisfies the property φ if and only if its synchronous product with an observer of φ never emits α.

2.3 Network observers

Without telling more about the syntax of processes, let us show that the notion of synchronous observer readily provides a way of expressing properties of parameterized networks: with each process in the network one can associate an observer, reading the input/output behavior of the process together with observations provided by other observers. For instance, let us consider a linear network $Q\|Q\|\ldots\|Q$ of identical processes, each of which emitting some signal u when it uses some resource. Assume we want to express the mutual exclusion property, that at most one process uses the resource at a given instant.

Each process is given an observer, receiving the u signal of the process and the signals emitted by its right-neighbor observer in the network (see Fig. 1.a). Each observer emits two signals: α is emitted whenever a violation of the mutual exclusion is detected, and ν is emitted whenever the resource is used by either the process or one of its right-successors in the network. Such an observer can be described by the following system of Boolean equations:

$$\alpha_o = \alpha_i \vee (\nu_i \wedge u) \qquad \nu_o = \nu_i \vee u$$

Now, a network satisfies the mutual exclusion property if and only if the left-most observer never emits α. Notice that this technique naturally extends to more complex network structures: for instance, if the network has a binary tree structure, one can design a suitable observer, receiving the signals emitted by its "sons" (cf. Fig. 1.b):

$$\alpha_o = \alpha_1 \vee \alpha_2 \vee (\nu_1 \wedge \nu_2) \vee ((\nu_1 \vee \nu_2) \wedge u) \quad \nu_o = \nu_1 \vee \nu_2 \vee u$$

3 Network Invariants as Fixpoints

In the rest of the paper we will restrict ourselves to linear networks of regular processes: a family $\mathcal{F}$ of such linear networks is generated by a finite multi-set of regular processes $\{P_i, i = 1...k\}$ in correspondence with a multi-set of composition operators $\{\times_i, i = 1...k\}$:

$$\begin{array}{rl} \forall i = 1\ldots k, & (P_i \in \mathcal{F}) \\ \text{and} & (P \in \mathcal{F} \Longrightarrow P \times_i P_i \in \mathcal{F}) \end{array}$$

Thanks to the preceding section, we can assume that each P_i contains its local observer, and that all the networks in the family have the same set of external signals, say, X (with $\alpha \in X$).

For each operator $\times_i$, let $R_i \subseteq \Theta_{X \cup X' \cup X''}$ be a set of traces such that

$$T_{P \times_i P_i} = \exists X', \exists X'', R_i \otimes T_P[X/X'] \otimes T_{P_i}[X/X'']$$

where X' and X'' are two sets of signals in one-one correspondence with X, and X, X', X'' are pairwise disjoint. Intuitively, R_i expresses the relation between the external signals of P (renamed as X'), the external signals of P_i (renamed as X'') and the external signals X of $P_i \times_i P$.

Example: Let us come back to the example of Fig. 1.a. Each network P has the same interface, i.e., the signals α_o, ν_o. A new network is built by connecting these signals to the inputs α_i, ν_i of a basic process, say P_1 (made of Q and its observer), and considering the outputs α_o, ν_o of P_1 as the ones of the new network. The traces of this new network can be expressed in functions of the sets T_P and T_{P_1} as

$$\begin{array}{l}\exists\{\alpha_i', \nu_i', \alpha_o', \nu_o'\}, \exists\{\alpha_o'', \nu_o''\},\\ \quad R \otimes T_{P_1}[\alpha_i/\alpha_i', \nu_i/\nu_i', \alpha_o/\alpha_o', \nu_o/\nu_o'] \otimes\\ \quad\quad T_P[\alpha_o/\alpha_o'', \nu_o/\nu_o'']\end{array}$$

where the composition operator R specifies that the outputs of P are connected to the inputs of P_1, and that the global outputs are those of P_1:

$$R = (\alpha_i' \equiv \alpha_o'' \ \wedge\ \nu_i' \equiv \nu_o'' \ \wedge\ \alpha_o \equiv \alpha_o' \ \wedge\ \nu_o \equiv \nu_o')^*$$

[End of example]

Let $\mathcal{S} = \Theta_{X \setminus \{\alpha\}}$ be the set of traces which never emit the "alarm" signal α. Our parameterized verification problem consists in showing: $\forall P \in \mathcal{F},\ T_P \subseteq \mathcal{S}$. Following [WL89], we can look for a process I, called a *network invariant*, satisfying

[SAT] $T_I \subseteq \mathcal{S}$

[INIT] $\forall i = 1...k, T_{P_i} \subseteq T_I$

[INDUC] $\forall i = 1...k,\ T_{I \times_i P_i} \subseteq T_I$,
or, equivalently,

$$\exists X', \exists X'',\ R_i \otimes T_I[X/X'] \otimes T_{P_i}[X/X''] \ \subseteq\ T_I$$

Proposition 1 *There is a least set of traces T_I^m satisfying both [INIT] and [INDUC], and a greatest one T_I^M satisfying both [SAT] and [INDUC]. T_I^m (resp., T_I^M) is the least (resp., greatest) fixpoint of a monotone function f (resp. F).*

Proof: Let us rewrite *[INDUC]* as: $\forall i = 1...k, f_i(T_I) \subseteq T_I$. The conjunction with *[INIT]* gives

$$f(T_I) \ \triangleq\ \bigcup_{i=1}^{k} (f_i(T_I) \cup T_{P_i}) \ \subseteq\ T_I$$

This means that T_I is a post-fixpoint of the (monotone) function f. There is a least solution, T_I^m, which is the least fixpoint of f.

On the other hand, *[INDUC]* can be easily transformed into: $\forall i = 1...k$,

$$T_I[X/X'] \subseteq \left(\forall X, \forall X'',\ (\Theta_{X \cup X' \cup X''} \setminus R_i) \oplus (\Theta_X \setminus T_{P_i})[X/X''] \ \oplus\ T_I \right)$$

or $\forall i = 1...k,\ T_I \subseteq F_i(T_I)$. Similarly, this shows that there is a greatest set of traces T_I^M satisfying both *[SAT]* and *[INDUC]*, which is the greatest fixpoint of the monotone function

$$F \ =\ \lambda T.\mathcal{S} \cap \bigcap_{i=1}^{k} F_i(T)$$

□

So, our verification problem is equivalent to showing either $T_I^m \subseteq \mathcal{S}$ or $\forall i = 1...k, T_{P_i} \subseteq T_I^M$, where

$$T_I^m = \bigcup_{n \geq 0} f^{(n)}(\emptyset) \quad \text{and} \quad T_I^M = \bigcap_{n \geq 0} F^{(n)}(\Theta_X)$$

$$f = \lambda T. \bigcup_{i=1}^{k} T_{P_i} \cup \left(\exists X', \exists X'',\ R_i \otimes T[X/X'] \otimes T_{P_i}[X/X''] \right)$$

$$F = \lambda T.\mathcal{S} \cap \bigcap_{i=1}^{k} \left(\forall X, \forall X'',\ (\Theta_{X \cup X' \cup X''} \setminus R_i) \oplus (\Theta_X \setminus T_{P_i})[X/X'']) \oplus T \right) [X'/X]$$

Of course, the undecidability of our verification problem results from the fact that T_I^m and T_I^M cannot be computed, in general (the iterations are infinite, and the limits are infinite state processes). Notice that T_I^m is the set of all possible traces of all the networks in $\mathcal{F}$; intuitively, it is very unlikely to be generated by a finite state automaton. On the contrary, it happens quite often that the iterative computation of T_I^M converges after a finite number of steps (in particular, when the property φ is already an invariant!). This is why we are more interested in computing T_I^M.

The following example supports this choice, since it is a case where T_I^M is regular, and can be computed in few iterations. This example is also interesting as

- it illustrates the modeling of a linear network by means of observers.
- it shows how the technique can be extended to cope with rings of processes.
- it shows how the iterative computation of T_I^M results in a sequence of automata, and prepares the rest of the paper, which is an attempt to extrapolate the limit of such a sequence.

4 First Example

We consider a very simple token ring: Let n units U_1, U_2,...,U_n share a resource in mutual exclusion. They are connected in a ring, along which a token travels. When a unit receives the token, either it does not request the resource and transmits the token, or it keeps the token as long as it uses the resource. In the following description, both signals and states are represented by boolean variables. If x is a variable, **next** x represents its value in the next state. All variables are supposed to be initially false. With these notations, the behavior of a unit can be represented by the following system of Boolean equations:

$$\begin{array}{lcl} use & = & has\_tk \wedge req \\ tk_{out} & = & has\_tk \wedge \neg req \\ \textbf{next } has\_tk & = & tk_{in} \vee (has\_tk \wedge \neg tk_{out}) \end{array}$$

Intuitively, the first equation tells that the unit uses the resource whenever it has the token and requests the resource. The second equation tells that the unit transmits the token if it has it and does not request it. The last equation states that the unit will have the token at the next step if either it receives it now, or it already has it and does not transmit it. The internal signal req is left unspecified.

Now, this unit is provided with an observer: it has two additional inputs, telling if the resource is used and if the mutual exclusion is violated, farther in the network. It transmits the same information as outputs:

$$\begin{array}{lcl} otheruse_{out} & = & otheruse_{in} \vee use \\ alarm_{out} & = & alarm_{in} \vee (otheruse_{in} \wedge use) \end{array}$$

We can connect such extended units in a linear network (see Fig. 2). Each such network will have a fixed interface, namely the input signal tk_{in} and the output

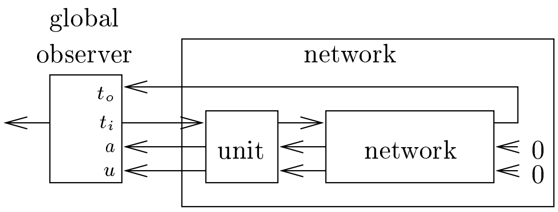


Figure 2: Global network of the simple token ring

signals tk_{out}, $otheruse_{out}$ and $alarm_{out}$. Global (rightmost) inputs $otheruse_{in}$ and $alarm_{in}$ are set to false. Adding a new unit can be done simply by a suitable renaming and hiding of communication signals. Now, we are faced with a last problem, which concerns the closure of the network as a ring. For that, we use again an observer: we will show that *if* the input tk_{in} is initially true (first insertion of the token) and then always equal to the output tk_{out}, *then* the network never emits an alarm. This global observer of the network interface can be specified by the following equation:

$$\begin{array}{lcl} alarm & = & alarm_{out} \wedge assumption \\ \textbf{next } assumption & = & assumption \wedge (tk_{in} = tk_{out}) \end{array}$$

where the initial value of $assumption$ is assumed to be tk_{in}.

Fig. 2 shows the general structure of the network, and how it can be extended with one unit.

The computation of the invariant converges in two steps. This shows that, in this case, T_I^M is a regular language, while T_I^m is obviously not. In Fig. 3, the sets of traces considered at each step are represented by their minimal deterministic acceptors. On these automata, t_i, t_o, a and u respectively stand for tk_{in}, tk_{out}, $alarm_{out}$, and $otheruse_{out}$. For simplicity, forbidden transitions have been removed, so the actual $alarm$ signal does not appear. For instance, the first automaton describes the set of traces that satisfy the initial property: it accepts all traces that either never emit a, or violate the closure assumption.

5 Computation of Network Invariants

In this section, we will try to compute under-approximations of T_I^M, using Cousot's extrapolation technique [CC77, CC92]. Notice that a solution $T \subseteq T_I^M$ can be sufficient to achieve the verification, if it happens that $\forall i = 1...k, T_{P_i} \subseteq T$.

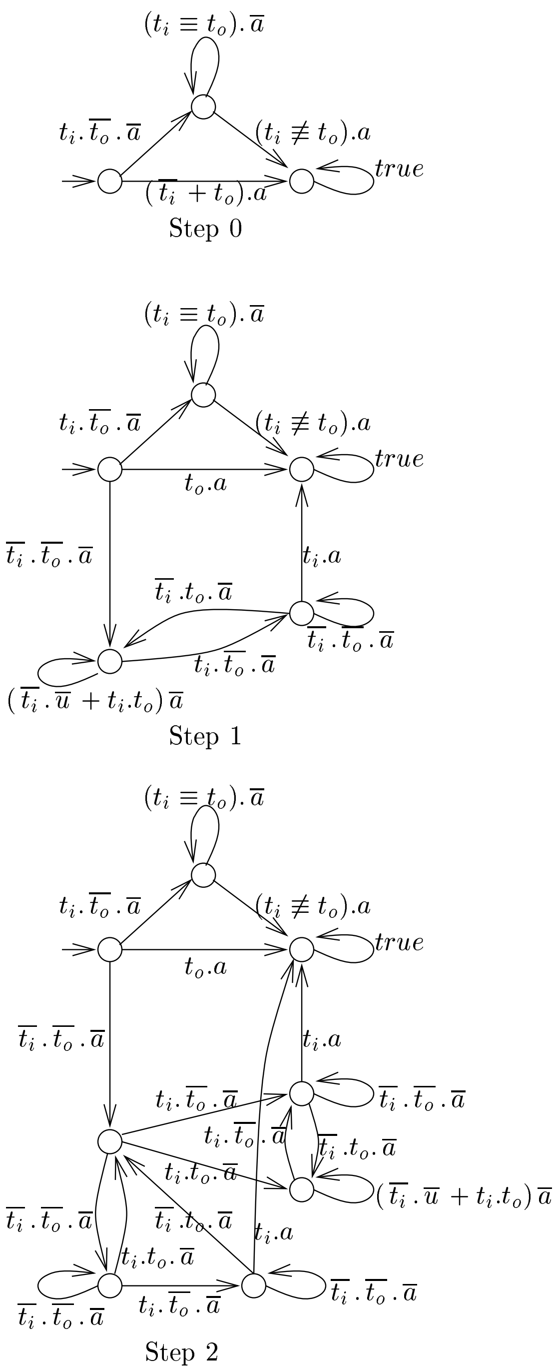


Figure 3: Computation of the simple token ring

5.1 Principle of Extrapolation

In order to under-approximate greatest fixpoints in a complete lattice L, the extrapolation method proposed by [CC77, CC92] consists in defining a binary operator ∇ on L, called "widening", satisfying the following two properties:

> [INCL]: $\forall x, y \in L,\ x \nabla y \subseteq x \cap y$
>
> [CHAIN]: for any decreasing chain $x_0 \supseteq x_1 \supseteq \ldots \supseteq x_n \supseteq \ldots$ in L, the sequence $(y_0, y_1, \ldots, y_n, \ldots)$, defined by $y_0 = x_0$, $y_{n+1} = y_n \nabla x_{n+1}$, is not strictly decreasing (i.e., becomes constant after a finite number of terms).

Then, for each monotone function $F : L \mapsto L$, the sequence $y_0 = \top$ (the supremum of the lattice), $y_{n+1} = y_n \nabla F(y_n)$ converges, after a finite number of steps, towards a limit y, which is smaller than the greatest fixpoint of F.

Following this approach, we have to define an extrapolation operator on sets of traces. The design of such an operator is an experimental task, searching for a compromise between the efficiency of the computation and the precision of the result: depending on the operator used, one can obtain either a very long sequence converging towards a solution very close to the fixpoint, or conversely, a fast convergence towards a rough solution.

5.2 Extrapolation Operators

We propose a parameterized extrapolation operator based on automata: if $T \subseteq \Theta_X$ is a prefix-closed set of traces, let $\mathcal{A}_T$ denote the (unique, up to isomorphism) minimal deterministic observer of T, i.e., a deterministic Mealy machine with 2^X as input alphabet, $\{\emptyset, \alpha\}$ as output alphabet, and returning α if and only if the trace read so far does not belong to T.

Now, let T and T' be two sets of traces, with $T' \subseteq T \subseteq \Theta_X$. We have in mind that T and T' are two consecutive steps of the previous computation (i.e., $T' = F(T)$). The principle of our extrapolation is to compare the structure of both $\mathcal{A}_T$ and $\mathcal{A}_{T'}$, so that the one of the next computation step can be guessed.

Let $\mathcal{A}_\times$ be the synchronous product of $\mathcal{A}_T$ and $\mathcal{A}_{T'}$. Notice that if we consider the signal α alone (resp. α'), $\mathcal{A}_\times$ recognizes the language T (resp. T'). Let $\mathcal{D}$ be the set of states (q, q') in $\mathcal{A}_\times$, from which $\mathcal{A}_{T'}$ can complain (i.e., emit α') while $\mathcal{A}_T$ cannot. Notice that if we consider the signal α alone (resp. α'), $\mathcal{A}_\times$ recognizes the language T (resp. T').

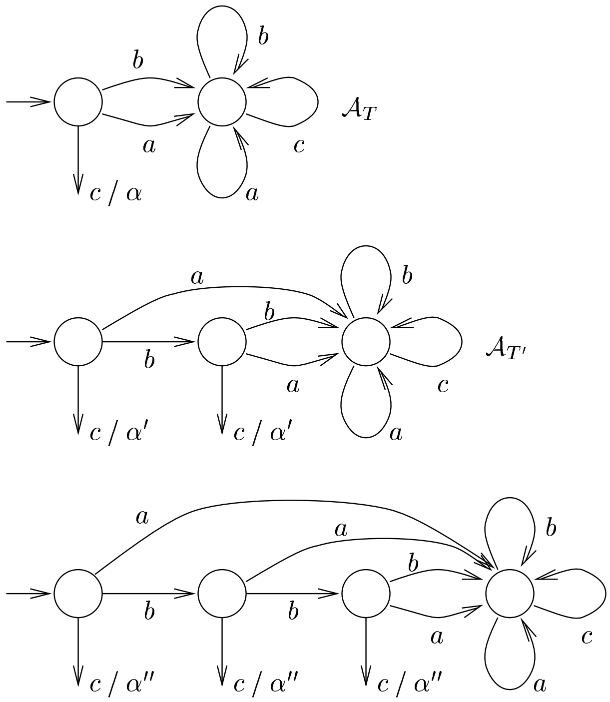
Since some behaviors from the states in $\mathcal{D}$ has been excluded when changing T into T', our first idea is to extrapolate the next computation step by forbidding all these states. More precisely, a possible choice would be to define $T \nabla T'$ as the language accepted by the automaton obtained as follows: remove from $\mathcal{A}_T$ all the states q such that there exists q' such that $(q, q') \in \mathcal{D}$, i.e., forbid all the transitions leading into such states. Since the new automaton has strictly less states than $\mathcal{A}_T$, this operator satisfies the property [CHAIN]. Unfortunately, experimentation shows that this operator is much too rough to provide interesting results: on most examples, it provides the empty language as a limit.

The point is that we have to forbid some behaviors passing through the states in $\mathcal{D}$, but not *all* these behaviors. Further, experience shows that infinite computations often result from the fact that "regular" patterns are repeated more and more times, which finally produce infinite loops in the limit language. For instance, the sequence $(T_k)_{k \geq 0}$ whose general term is $T_k = \{a^n.(a+b)^* \mid 0 \leq n \leq k\}$ is infinite, but converges towards $(a+b)^*$. So, the next idea is to create such loops by rerouting non-deterministically some transitions $(q_0, q'_0) \xrightarrow[o]{i} (q_1, q'_1)$ reaching $\mathcal{D}$ to other states $(q_2, q'_2) \notin \mathcal{D}$.

To ensure the trace inclusion [INCL], the language recognized from (q_2, q'_2) must be included in the one recognized from (q_1, q'_1). To create loops, the new target states (q_2, q'_2) are chosen among (q_0, q'_0) and its predecessors that satisfy this inclusion. They are searched up to a depth d which is a parameter of the operator.

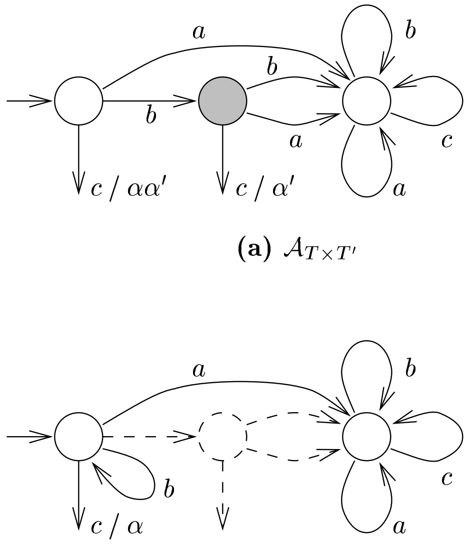
Unfortunately, such an operator no longer satisfies the [CHAIN] property: the number of states of the new automaton decreases, but, since this automaton is non deterministic, the number of states of its deterministic version can become larger. In fact, we were not able to define an operator satisfying both [INCL] and [CHAIN], and providing interesting (i.e., non empty) approximations. So, we decided to release the property [CHAIN], which ensures termination: our operator "speeds up" the convergence, but does not ensure its termination. As a consequence, the computation may not terminate, but if it does, the solution is a correct under-approximation of the fixpoint. This possibility of non-termination is discussed in §5.4.

Example: Let $T = (a+b)(a+b+c)^*$ and $T' = (a+b(a+b))(a+b+c)^*$. Intuitively, one can expect that the next computation step will compute the language $T'' = (a + b(a + (b(a+b))))(a+b+c)^*$. Automata



expected $\mathcal{A}_{F(T')}$

Figure 4: An infinite sequence of languages



(a) $\mathcal{A}_{T \times T'}$

(b) $\mathcal{A}_{T \nabla T'}$

Figure 5: Widening on automata

recognizing T, T' and T'' are shown in Fig. 4. Fig. 5.a shows the automaton $\mathcal{A}_\times$ (where the grey state is the only one in $\mathcal{D}$). Fig. 5.b shows the rerouting performed, with $d = 1$ (i.e., the new target state can only be the source); so $T \nabla T' = b^* a(a+b+c)^*$. Notice that if the rerouting is not performed (i.e., with $d = 0$), we obtain $T \nabla T' = a(a+b+c)^*$: this extrapolation is probably too rough because it does not express the fact that an arbitrary number of events b can occur before an event a. *[End of example]*

5.3 The actual algorithm

In practice, we do not simply compute the limit T of the sequence $T_0 = \top$, $T_{n+1} = T_n \nabla F(T_n)$, as it is generally too rough. Instead, we can arbitrarily improve the solution by delaying the application of the extrapolation: For each $k \in \mathbb{N}$, let us define $T^{(k)}$ to be the limit of the sequence

$$T_0 = \top, \quad T_{n+1} = \begin{cases} F(T_n) & \text{if } n < k \\ T_n \nabla F(T_n) & \text{if } n \geq k \end{cases}$$

All the $T^{(k)}$ are under-approximations of the fixpoint (the standard approximation T is $T^{(0)}$), and the greater is k, the more precise is $T^{(k)}$. So, the method consists in computing $T^{(k)}$, letting the parameter k increase as long as the invariant $T^{(k)}$ is too strong (i.e., does not satisfy $[INIT]$). These iterations on k may not terminate, and for each k, the computation of $T^{(k)}$ may not terminate (since our extrapolation operator does not satisfy the property [CHAIN]). In principle, it could happen[2] that the computation of $T^{(k)}$ be infinite, while the one of $T^{(k+1)}$ converges to a suitable approximation.

From a theoretical point of view, if we want to get a semi-decision procedure — in the following sense: If a suitable approximation $T^{(k)}$ is finitely computable, it will be eventually reached by the algorithm — the algorithm has to perform a breadth-first exploration of the graph of approximations, i.e., letting both n and k grow in turn.

From a practical point of view, as the size of the considered automata grow rapidly, all computations either converge rapidly, or saturate the memory!.

5.4 Computation of Abstractions

The complexity of the invariant computation is mainly due to the computation of abstractions $\exists Y, T$ and $\forall Y, T$.

[2]although we never encountered such a situation during our experimentations.

Let us detail these computations.

Let $T \subseteq \Theta_X$ be a regular, prefix-closed, set of traces on X and $\mathcal{A}_T = (Q_T, q0_T, X, \{\alpha_T\}, \delta_T, \delta_T^\alpha)$ be the minimal deterministic observer of T, where

- Q_T is the (finite) set of states.
- $q0_T \in Q_T$ is the initial state.
- X is the set of input symbols.
- α_T is the alarm signal (the only output symbol).
- $\delta_T : Q_T \times 2^X \to Q_T$ is the *total* transition function.
- $\delta_T^\alpha : Q_T \times 2^X \to \{\emptyset, \alpha_T\}$ is the *total* output function.

Let $Y \subseteq X$. Let us suppose that there exists a sink state, noted q_T^s such that the alarm signal α_T is emitted only by transitions reaching this state. That's

$$\delta_T^\alpha(q, x) = \alpha_T \Leftrightarrow \delta_T(q, x) = q_T^s$$

To compute the observer of $\exists Y, T$, first remove from $\mathcal{A}_T$ labels all signals which are in Y. We obtain a non-deterministic automaton which can be determinized by the following classical algorithm

$$\mathcal{A}_{\exists Y, T} = (2^{Q_T}, \{q0_T\}, X \setminus Y, \{\alpha_T\}, \delta_{\exists Y, T}, \delta_{\exists Y, T}^\alpha)$$

where $\delta_{\exists Y, T} : 2^{Q_T} \times 2^{(X \setminus Y)} \to 2^{Q_T}$ and $\delta_{\exists Y, T}^\alpha : 2^{Q_T} \times 2^{(X \setminus Y)} \to \{\emptyset, \{\alpha_T\}\}$

$$\delta_{\exists Y, T}(\tilde{q}, x) = \{q' | \exists q \in \tilde{q}, \exists y \in Y,\ q' = \delta_T(q, x \cup y)\}$$

$$\delta_{\exists Y, T}^\alpha(\tilde{q}, x) = \begin{cases} \{\alpha_T\} & \text{if } \delta_{\exists Y, T}(\tilde{q}, x) = \{q_T^s\} \\ \emptyset & \text{otherwise} \end{cases}$$

i.e., the alarm signal α_T is only emitted when reaching the state $\{q_T^s\}$.

The observer of $\forall Y, T$ accepts a trace $\tau_x \in \Theta_{X \setminus Y}$ if and only if, for all traces $\tau_y \in \Theta_Y$, $\tau_x \odot \tau_y \in T$. Thus, it complains if and only if there exists a trace τ_y such that $\mathcal{A}_T$ complains on $\tau_x \odot \tau_y$, i.e., if $\mathcal{A}_{\exists Y, T}$ reaches a state which contains the sink state q_T^s. Then

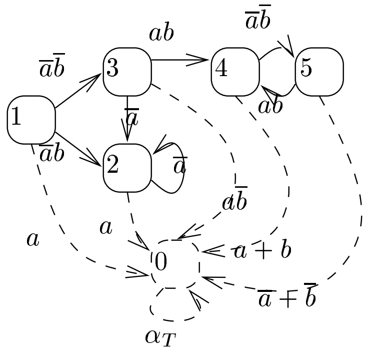
$$\mathcal{A}_{\forall Y, T} = (2^{Q_T}, \{q0_T\}, X \setminus Y, \{\alpha_T\}, \delta_{\exists Y, T}, \delta_{\forall Y, T}^\alpha)$$

where

$$\delta_{\forall Y, T}^\alpha(\tilde{q}, x) = \begin{cases} \{\alpha_T\} & \text{if } q_T^s \in \delta_{\exists Y, T}(\tilde{q}, x) \\ \emptyset & \text{otherwise} \end{cases}$$

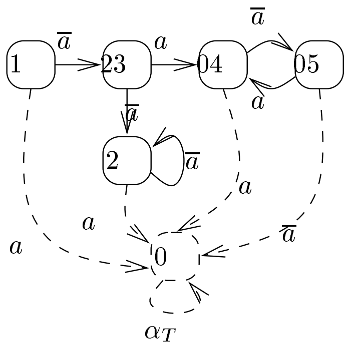
i.e., the alarm signal α_T is emitted when reaching any state containing the sink state q_T^s.

Fig. 6 illustrates these constructions.



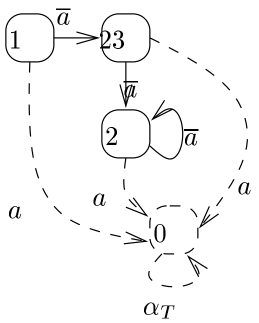
Automaton A_T

with $T = (\overline{a})^* + (\overline{a}\overline{b}.ab)^*$



Automaton $\mathcal{A}_{\exists b,T}$

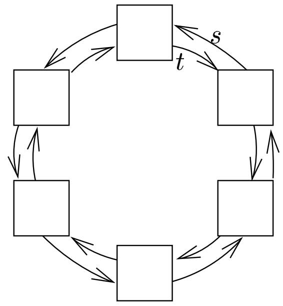
$\exists\{b\}, T = (\overline{a})^* + (\overline{a}.a)^*$



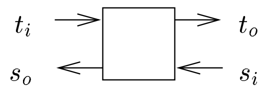
Automaton $\mathcal{A}_{\forall b,T}$

$\forall\{b\}, T = (\overline{a})^*$

Figure 6: Abstraction algorithm



The network



A unit

Figure 7: Dijkstra's token ring

6 Examples

Our method has been implemented in a tool, taking Lustre [HCRP91] programs as input, and computing invariants. We present here two problems of mutual exclusion, which have been automatically verified on a **HP Apollo 9000 series 735**.

6.1 Dijkstra's token ring

This algorithm is adapted from the one used in [CGJ95]. Let n units $U_1, U_2, \ldots, U_n$ share a resource in mutual exclusion. The units are connected in a ring (see Fig.7), along which a token can travel in the clockwise direction. To avoid useless token passing, a request signal can travel in the counter-clockwise direction. A unit can use the resource only when it has the token. Whenever a unit requires the token, it sends the request signal to its left. When the unit which has the token receives a request signal, it transmits the token to its right.

Each unit has 2 input signals: t_i (token arrival) and s_i (request signal arrival), 2 output signals: t_o (token passing) and s_o (request signal passing), and two internal moves *req* and *rel*, corresponding to the resource request and release. With the notations of Section 4, the following system of equations defines this behavior:

$$
\begin{array}{lcl}
\textbf{next } wait & = & (req \vee wait) \wedge \neg t_i \wedge \neg has\_tk \\
\textbf{next } right\_req & = & (right\_req \vee s_i) \wedge \neg t_o \\
\textbf{next } has\_tk & = & (has\_tk \vee t_i) \wedge \neg t_o \\
t_o & = & (right\_req \vee s_i) \wedge (t_i \vee has\_tk) \wedge \\
 & & \neg wait \wedge \neg(req \vee rel) \\
s_o & = & \neg right\_req \wedge \neg(has\_tk \vee t_i) \wedge \\
 & & \neg wait \wedge (s_i \vee req)
\end{array}
$$

Provided with an observer of the mutual exclusion as in Section 4, it has two more input signals: u_i (resource used on the right) and α_i (mutual exclusion violated on the right) and two more output signals u_o (resource used) and α_o (mutual exclusion violated). The ring is closed by means of an observer as in Section 4.

This example shows that proving a strong property is often easier than a weak one. For instance, to show that there is always *one and only one* token in the network, a suitable invariant is computed after 3 steps, in 20 seconds, using 1 extrapolation. The automaton of the computed invariant has 30 states and 1355 transitions. But, to show that there is always *at least one* token in the network (a weaker property than above), the invariant computation takes 40 seconds, again with 3 steps and 1 extrapolation. The resulting automaton has 39 states and 1849 transitions.

6.2 A Hardware Arbiter

Our third example comes from [HLR92]: as before, n units U_1, U_2,...,U_n share a resource in mutual exclusion. Units are served according to a fixed priority policy: whenever the resource is free, and a unit requires it, a token is emitted (as the rising edge of a condition), which will travel from unit to unit through the network, until being caught by the first unit requiring the resource (see Fig. 8).

Fig. 9 shows the circuit corresponding to an arbitration element: it samples the requesting status of the unit on the rising edge of the incoming token, by means of an edge-triggered flip-flop. According to the output of the flip-flop, the token either raises the grant or is passed to the next unit. The whole unit is described by the following system of equations:

$$\begin{array}{lcl}
\textbf{next } \mathit{flop} & = & (\mathit{edge} \wedge \mathit{wait}) \vee (\neg \mathit{edge} \wedge \mathit{flop}) \\
\mathit{edge} & = & \mathit{tk}_i \wedge \mathit{was_low} \\
\textbf{next } \mathit{was_low} & = & \neg \mathit{tk}_i \\
\textbf{next } \mathit{tk}_o & = & \mathit{tk}_i \wedge \neg \textbf{next } \mathit{flop} \\
\textbf{next } \mathit{grant} & = & \mathit{tk}_i \wedge \textbf{next } \mathit{flop} \\
\textbf{next } \mathit{use} & = & \textbf{next } \mathit{grant} \vee (\mathit{use} \wedge \neg \textbf{next } \mathit{release}) \\
\textbf{next } \mathit{wait} & = & \neg \textbf{next } \mathit{use} \wedge (\textbf{next } \mathit{req} \vee \mathit{wait})
\end{array}$$

The leftmost token wire is raised whenever the resource is requested and not used. To describe these computations, each unit is paired with a process representing the "or" gates on top of the global network: it receives four wires: *right_requested* and *right_used* form the right part of the network (which are always false for the rightmost unit), and *use* and *wait* from the associated unit:

$$\begin{array}{lcl}
\mathit{requested} & = & \mathit{right_requested} \vee \mathit{wait} \\
\mathit{used} & = & \mathit{right_used} \vee \mathit{use}
\end{array}$$

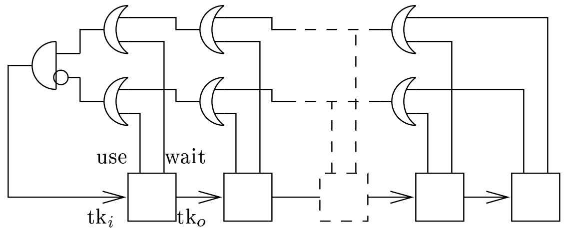


Figure 8: Arbiter

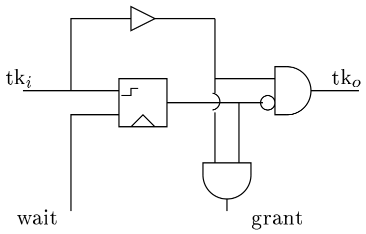


Figure 9: Arbitration element

Finally, each network has two outputs *requested* and *used* and one input *tk*. The network is closed by an observer which check properties under the assumption that

$$\mathit{tk} \equiv \mathit{requested} \wedge \neg \mathit{used}$$

The following properties have been verified:

Mutual exclusion, by providing each unit with an observer as in preceding examples.

No token lost, i.e., the rightmost token wire is always low. This is done by a slight change in the global observer.

Priority, which is an example of non trivial temporal property: ideally, the arbiter should satisfy a priority rule like

$$\mathit{grant}_i \Longrightarrow \neg \mathit{wait}_j$$

for each pair (i, j) such that $j < i$. But this rule cannot be satisfied by such a distributed device since it would involve an instantaneous knowledge of all requests to the resource. Instead, the arbiter ensures the following weaker priority rule: if the resource is granted to U_i at time t, no unit $U_j, (j < i)$ was waiting for the resource *at the last arbitration request preceding* t, where an arbitration request is a rising edge of the leftmost token wire. This property can be expressed by providing each unit observer with an instantaneous knowledge of the arbitration request (arb_req). Of course, this is for specification only, and does not change the circuit itself. The observer samples the waiting status of its associated unit on each arbitration request, and transmits it to the next observer in the network. So, each observer receives a wire telling if a more prioritary unit was waiting at the last arbitration request, and can evaluate the property:

$$\begin{array}{lcl} \alpha_o & = & \alpha_i \vee (prio_i \wedge grant) \\ prio_o & = & prio_i \vee prio \\ \textbf{next}\ prio & = & (arb\_req \wedge wait) \vee \\ & & (\neg arb\_req \wedge prio) \end{array}$$

We tried to verify each combination of these 3 properties. The following table show the results: for each combination of properties (where "E", "N", "P", respectively stand for "exclusivity", "no token lost", and "priority"), the table gives the number of applications of the extrapolation operator, the number of steps, the numbers of states and transitions of the final invariant, and the total computation time. We were able to verify all combinations of property but one: when considering "no token lost" alone, the computation does not seem to converge (it runs out of memory after 10').

Properties			Nb. extrap.	Nb. steps	Nb. states	Nb. trans.	Time
E			2	4	31	556	2'57"
E	N		1	3	6	121	1'12"
E	N	P	1	3	9	313	1'11"
E		P	1	3	9	313	1'26"
	N		*	*	*	*	*
	N	P	1	3	11	224	2'00"
		P	2	4	16	312	5'58"

7 Conclusion and Future Work

We have proposed a way to specify safety properties of parameterized networks of processes, and, in the case of linear or cyclic networks, a method and a tool to verify such properties by synthesizing network invariants. Since the synthesis method may not terminate, and often requires the user to adjust some extrapolation parameters, it should be viewed as a mechanical help for constructing invariants, rather than as an automatic model-checker. Compared with the approach proposed in [CGJ95], let us first note that our proposal is less general, both concerning the shape of networks, and the considered properties. However, we think that, in our restricted framework, our approach could be more practical: on one hand, the specification of properties by synchronous observers appears to be very flexible, and on the other hand, one can improve the precision of the result by playing with parameters. For the time being, we have only very few elements for comparing the precision of the generated invariants.

Notice that, in all our examples, the resulting automata are very small. This is due to three reasons:

- Of course, the extrapolation operator generally simplifies the computations and reduces the size of automata.
- We compute greatest fixpoints "backward", starting from the automaton of the property, which is generally simple. This should be compared with a "forward" method, computing least fixpoints from the basic processes: as a matter of fact, all our examples show that the backward computation is less "explosive" than the forward one. Typically, we are able to compute up to 20 exact steps in the backward sequence, while the forward method explodes after 3 or 4 steps.
- At each step, the automata are minimized. However, the abstraction operation performs a determinization followed by a minimization: the determinization often produces large automata, which are then highly reduced by the minimization. This explains the rather long execution time of our experiments.

This work should be extended in several directions:

- First, the performances of the tool should be improved. We are investigating BDD-based methods, to handle symbolic representations of automata. More important, the complexity of the method highly depend on automata determinization, which is exponential. So, we have to check whether this determinization could be avoided.
- Second, we want to see if our approach could be generalized to more general networks, like those described by network grammars. Our first steps in this direction raised the following problem: the

existence of a greatest invariant is no longer obvious, since the characterization of network invariants as pre-fixpoints (like in the definition of T_I^M) may involve a non-monotone function.

References

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

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

[CC77] 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, POPL'77*, Los Angeles, January 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, Leuven (Belgium), January 1992. LNCS 631, Springer Verlag.

[CES86] 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.

[CGJ95] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR'95*. LNCS 962, Springer Verlag, August 1995.

[Eil74] S. Eilenberg. *Automata, Languages, and Machines*. Academic Press, 1974.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, January 1995.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *8th International Conference on Computer Aided Verification, CAV'96*, Rutgers (N.J.), 1996.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 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):523–543, 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[KM89] R. P. Kurshan and K McMillan. A structural induction theorem for processes. In *8th ACM Symposium on Principles of Distributed Computing*, pages 239–247, Edmonton (Alberta), August 1989.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag.

[QS82] 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.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[WL89] 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.