Automatic Verification of Parameterized Networks of Processes [1]

David Lesens, Nicolas Halbwachs, and Pascal Raymond

VERIMAG, Centre Equation, 2, avenue de Vignate, F-38610 Gières, France.
{lesens,halbwach,raymond}@imag.fr [2]

Abstract

This paper describes a method to verify safety properties of parameterized networks of processes defined by network grammars. The method is based on the construction of a network invariant, defined as a fixpoint. We propose heuristics, based on Cousot's extrapolation techniques (widening), which often allow suitable invariants to be automatically constructed. We successively consider linear and binary tree networks. These techniques have been implemented in a verification tool, and several non-trivial examples are presented.

Key words: model-checking, parameterized networks, synchronous observers, widening.

1 Introduction

1.1 Parameterized Networks

Parameterized networks are infinite families of processes with regular structure, finitely generated from a finite number of basic processes. For instance, a family $\mathcal{F}$ of *linear* networks is generated from a multiset $\{P_1, \ldots, P_n\}$ of processes in one-one correspondence with a multiset $\{\times_1, \ldots, \times_n\}$ of binary

[1] This work has been partially supported by a grant from the CNET (French Telecommunications).
[2] Verimag is a joint laboratory of CNRS, Université Joseph Fourier and Institut National Polytechnique de Grenoble, associated with IMAG. http://www.imag.fr/VERIMAG.

composition operators over processes, in the following way:

$$\forall i = 1 \ldots n, \quad P_i \in \mathcal{F} \text{ and } P \in \mathcal{F} \Rightarrow P \times_i P_i \in \mathcal{F}$$

In [MG91,SG89], *context-free network grammars* are used to define more general networks. Such a grammar is a tuple $\Gamma =< T, N, \mathcal{R}, S >$ where:

- $T = \{P_1, \ldots, P_n\}$ is a finite set of basic processes.
- N is a set of non-terminals. Each non-terminal defines a sub-network.
- $\mathcal{R}$ is a finite set of production rules of the form $\rho : A \rightarrow B \times_\rho C$, where $A \in N$, and $B, C \in T \cup N$, and $\times_\rho$ is a binary composition operator (depending on the rule ρ).
- $S \in N$ is the start symbol that represents the network generated by the grammar.

The set $\mathcal{F}$ of processes generated by the grammar is the set of processes generated by the rules from the start symbol.

1.2 Network Invariants

A parameterized network $\mathcal{F}$ satisfies a property φ, if φ is fulfilled by *any* process in $\mathcal{F}$:

$$\mathcal{F} \models \varphi \iff \forall P \in \mathcal{F},\ P \models \varphi$$

[AK86] established the following negative result about the verification of parameterized networks:

> $\mathcal{F} \models \varphi$ is undecidable, even in the case where each basic process is finite state, i.e., where $P \models \varphi$ is decidable for each $P \in \mathcal{F}$.

Decidable subcases have been identified [EN95,EN96], but they are quite restrictive. Several attempts [KM89,WL89,HLR92] were made to extend model-checking techniques [QS82,CES86] to verify general networks generated from finite-state basic processes. These approaches use an induction principle, which can be expressed as follows in the case of linear networks:

- Let $\preceq$ be a preorder relation over processes, such that

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

- Define a *network invariant* to be a process I satisfying

$$\forall i = 1 \ldots n,\ I \times_i P_i \preceq I$$

- Find a network invariant I, such that $\forall i = 1 \ldots n, P_i \preceq I$. Then

$$I \models \varphi \;\Rightarrow\; \forall P \in \mathcal{F}, P \models \varphi$$

In the general case of networks generated by grammars, an invariant I_A has to be associated with each non-terminal A, in such a way that, for each production rule $A \to B \times_\rho C$, one has

$$I_B \times_\rho I_C \;\preceq\; I_A$$

(where $I_P = P$ when P is a basic process). Then,

$$I_S \models \varphi \implies \forall P \in \mathcal{F}, P \models \varphi$$

1.3 Automatic Verification

Practically, the verification of a parameterized network raises two problems:

(1) How to express the desired property φ independently of the number of component processes?
(2) How to find suitable network invariants $(I_P)_{P \in N}$, if such invariants exist ?

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 multiset of basic process states (this idea is also used in [KMM+97,FO97]); they propose to consider such a state as a word on the alphabet of basic 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 preorder. 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 network invariants $(I_P)_{P \in N}$.

Let us comment about this proposal: the language-based specification technique is surely well-suited to linear networks of processes, where a state of a compound process is naturally handled as a tuple of basic process states. It may be less easy to specify in this way more complex structures, where a 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). In this paper, we propose another specification method, based on *synchronous observers* [HLR93]. A synchronous observer is a process that is able to observe the behavior of another process without changing this behavior. In our approach, a state property is expressed by providing each basic process with an observer, taking as input the input/output behavior of its associated basic process, together with observations provided by the observers of its "neighbor processes" in the network. For the time being, we restrict ourselves to safety properties, and we use trace inclusion preorder.

Concerning the construction of the network invariant, the method proposed in [CGJ95] can raise the following problem: if the synthesized invariants $(I_A)_{A \in N}$ do not satisfy the desired property φ, the method does not provide any way to look for better invariants. 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.

1.4 Summary of the Paper

The paper is organized as follows. In section 2, we define the basic notions, including network observers. Section 3 states the problem of finding suitable invariants as the resolution of least fixpoint equations. Since the computation of these least fixpoints is generally untractable, a greatest fixpoint characterization of linear network invariants is introduced in section 4. In section 5, an extrapolation technique is presented to approximate this greatest fixpoint. Section 6 and 7 extends the computation of greatest fixpoints to tree networks.

Preliminary versions of this work have been published in [LHR96,LHR97,Les97].

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*[3]. 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.

[3] 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.

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

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

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

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

(where $|\tau|$ denotes the length of the trace τ). 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 (i.e., *any* subset of X' can be added to any term of any trace of T)..

Let X and X' be two sets of signals of the same cardinality related to each other by a one-one mapping $\phi = \lambda x.x'$. 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

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

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

$$\forall Y, T = \Theta_X \setminus (\exists Y, \Theta_X \setminus T)$$

Example 1 *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*

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

The computation of $\exists Y, T_P$ and $\forall Y, T$ is detailed in appendix A.

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

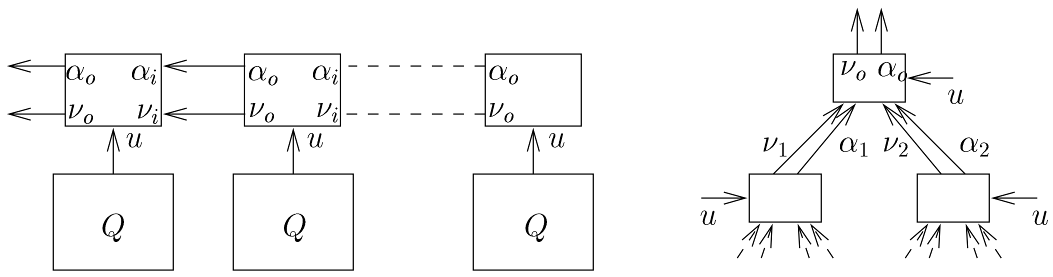
$$\forall T \subseteq \Theta_X, \quad T \subseteq \varphi \;\Leftrightarrow\; T \otimes T_\varphi \subseteq \Theta_X$$

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

Throughout the paper, we restrict ourselves to *regular* observers (i.e., regular languages T_φ).

2.3 Network observers

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



(a) Linear structure

(b) Tree structure

Fig. 1. Network observers

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:

$$\alpha_o = \alpha_i \vee (\nu_i \wedge u) \quad \text{and} \quad \nu_o = \nu_i \vee u$$

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" (see Fig 1.b):

$$\alpha_o = \alpha_1 \vee \alpha_2 \vee (\nu_1 \wedge \nu_2) \vee ((\nu_1 \vee \nu_2) \wedge u) \quad \text{and} \quad \nu_o = \nu_1 \vee \nu_2 \vee u$$

A network observer is said to be *regular*, if each observer of individual processes in the network is regular.

2.4 Comparison with existing specification languages

In [CGJ95,KMM$^+$97,FO97], properties on networks are specified by *regular specification languages*, defined as follows: they consider a network state as a word on the alphabet of basic process states, and they specify a set of states as a regular language on this alphabet. We have the following proposition:

Proposition 2 *The expression power of regular network observers is strictly greater than the one of regular specification languages.*

PROOF. We show that any property that can be expressed by means of a regular specification language, can also be expressed by means of a regular network observer, and we give an example where the converse is not true.

(1) We show that any regular specification language can be described by a regular network observer: the proof is similar to showing that the inclusion of a context-free language in a regular language is decidable. We consider a network described by a grammar Γ, and a regular specification language given by a regular expression e. The observer of a basic process reports the current state of the process. With each subnetwork, generated by a non-terminal A, will be associated an observer, telling whether the state of the subnetwork belongs to some regular expressions: Let α_e^A be the "alarm" signal sent whenever a subnetwork generated by A is not in a state satisfying e. It is easy, but technically tedious, to define these alarm signals. We just give some cases:

- Let $e = e_1 + e_2$, then $\alpha_e^A = \alpha_{e_1}^A \wedge \alpha_{e_2}^A$
- We only sketch the complex case of concatenation: let $e = e_1.e_2$, where neither e_1 nor e_2 can generate the empty string. For a rule $A \to P$, we have $\alpha_e^A = true$, since the state of P is a singleton which cannot satisfy e. For a rule $A \to B \times_\rho C$, we have to consider all the ways e can be restructured into $e_1'.e_2'$ (this set if finite, since it is isomorphic to the set of states of the automaton recognizing e). Then

$$\alpha_e^A = \bigwedge_{e_1.e_2 = e_1'.e_2'} (\alpha_{e_1'}^B \vee \alpha_{e_2'}^C)$$

(2) We give an example of property that can be expressed by a regular observer, but not by a regular specification language: The state language $\{\ a^n.b^n \mid \text{for all } n\ \}$ is not regular, but can be expressed by observers in constructing the network in the following way:

$$S \ \to\ P\|S\|P, \qquad\qquad S \ \to\ P\|P$$

With each S network is associated an observer that checks that the left P son is in state a and that the right one is in state b.

3 Network invariants as least fixpoints

3.1 Computation of a least fixpoint

Thanks to the preceding section, we can assume that each subnetwork 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 binary operator $\times_\rho$, let us define $C_\rho \subseteq \Theta_{X \cup X' \cup X''}$ to be a set of traces such that

$$T_{P' \times_\rho P''} = \exists X', \exists X'', C_\rho \otimes T_{P'}[X/X'] \otimes T_{P''}[X/X'']$$

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

Example 3 *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 terms of the sets T_P and T_{P_1} as*

$$\begin{aligned}&\exists\{\alpha'_i, \nu'_i, \alpha'_o, \nu'_o\}, \exists\{\alpha''_o, \nu''_o\},\\ &\qquad C \otimes T_{P_1}[\alpha_i/\alpha'_i, \nu_i/\nu'_i, \alpha_o/\alpha'_o, \nu_o/\nu'_o] \otimes T_P[\alpha_o/\alpha''_o, \nu_o/\nu''_o]\end{aligned}$$

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

$$C \;=\; (\alpha'_i \equiv \alpha''_o \;\wedge\; \nu'_i \equiv \nu''_o \;\wedge\; \alpha_o \equiv \alpha'_o \;\wedge\; \nu_o \equiv \nu'_o)^*$$

Let $\Sigma = \Theta_{X \setminus \{\alpha\}}$ be the set of traces which never emit the "alarm" signal α. Our parameterized verification problem consists in showing that for each process P generated by the network grammar, $T_P \subseteq \Sigma$. Following [KM89,WL89,HLR92], we can look for processes $(I_A)_{A\in N}$, called *network invariants*, satisfying

[SAT] $T_{I_S} \subseteq \Sigma$

[INIT] For each rule $\rho : A \to P$, $T_P \subseteq T_{I_A}$

[INDUC] For each rule $\rho : A \to B \times_\rho C$, $T_{I_B \times_\rho I_C} \subseteq T_{I_A}$

or, equivalently, $(\exists X', \exists X'',\ C_\rho \otimes T_{I_B}[X/X'] \otimes T_{I_C}[X/X'']) \subseteq T_{I_A}$

Let us note $V = (T_{I_A})_{A\in N}$ the vector of invariant trace sets. Such vectors are ordered by componentwise inclusion.

Proposition 4 *There is a least vector V^{min} of sets of traces satisfying* [INIT] *and* [INDUC]*. V^{min} is the least fixpoint of a monotone function F_1.*

PROOF. Rewriting*[INIT]* and *[INDUC]* as $F_1(V) \subseteq V$, we get that V is a post-fixpoint of F_1. Now, F_1 is monotone, since it only involves least upper bounds and the monotone operators $(T_{I_B}, T_{I_C}) \mapsto T_{I_B \times_\rho I_C}$. So, there is a least solution, V^{min}, which is the least fixpoint of F_1.

So, our verification problem is equivalent to showing $V_S^{min} \subseteq \Sigma$, where S is the start symbol of the grammar.

Of course, the undecidability of our verification problem results from the fact that V^{min} cannot be computed, in general (the iterations are infinite, and the limit is a vector of infinite state processes). Notice that V_S^{min} 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.

The method proposed in [CGJ95] is an automatic way of computing an upper approximation of V^{min}. The great advantage of this method is its generality. It can be applied to general network grammars and can deal with complex properties. However, in many cases, this method either leads to a state explosion or provides a too rough approximation, i.e., a result which does not fulfill *[SAT]*.

This is why we will investigate another approach, based on the computation of a greatest fixpoint. This approach will take into account the property to check. The computation will be more dependent on the property (which is usually quite small) than on the system size (which is almost always infinite).

Section 4 will state this problem as the resolution of a greatest fixpoint equation in the case of linear networks. Section 6 will extend this to binary tree networks.

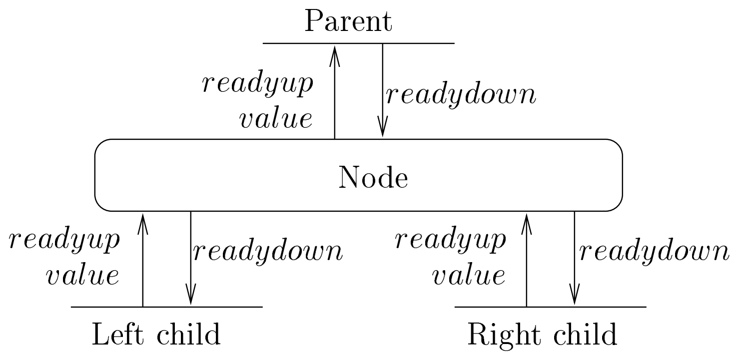


Fig. 2. Internal node of the parity tree

We first give an example (taken from [CGJ95]) for which a suitable approximation of the least fixpoint can be computed.

3.2 Example: a parity tree

Let us consider the network grammar $\Gamma =< \{L\}, \{S\}, \mathcal{R}, S >$, describing a binary tree network, where L is a leaf process, and $\mathcal{R}$ is defined by

$$\mathcal{R} \ = \ \{ \ S \rightarrow S \times S, \ \ S \rightarrow L \ \}$$

Each leaf process has an associated one-bit value. The algorithm computes the parity of the leaves values as follows [Ull84,CGJ95]. The root process initiates a wave by sending the *ready_down* signal to its children. Every internal node transmits this signal to its children. As soon as the *ready_down* signal reaches a leaf process, the leaf sends the *ready_up* signal and its value to its parent. When an internal node receives the *ready_up* signal from both its children, it sends the *ready_up* signal and the *xor* of the values of these children to its parent (see Fig 2). The root cannot send another wave before it receives the *ready_up* signal.

The forward computation of invariant does not converge, but in this case, the limit can easily be extrapolated, using a technique similar to the one presented in section 5. The invariant has 23 states and 90 transitions.

4 Linear network invariants as greatest fixpoints

In this section, 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 $\{\times_i, i = 1...k\}$:

$$\forall i = 1 \ldots k,\ (P_i \in \mathcal{F}) \quad \text{and} \quad (P \in \mathcal{F} \Longrightarrow P \times_i P_i \in \mathcal{F})$$

4.1 Computation of a greatest fixpoint

Thanks to section 2, 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 $\alpha \in X$).

Recall that $\Sigma = \Theta_{X\setminus\{\alpha\}}$ is the set of traces which never emit the "alarm" signal α. A network invariant is a process I, satisfying

$$\begin{array}{ll} \textit{[SAT]} & T_I \subseteq \Sigma \\ \textit{[INIT]} & \forall i = 1...k, T_{P_i} \subseteq T_I \\ \textit{[INDUC]} & \forall i = 1...k,\ T_{I \times_i P_i} \subseteq T_I \quad \text{or equivalently} \\ & \exists X', \exists X'',\ C_i \otimes T_I[X/X'] \otimes T_{P_i}[X/X''] \ \subseteq\ T_I \end{array}$$

Proposition 5 *There is a greatest set of traces T_I^M satisfying both* [SAT] *and* [INDUC]*. T_I^M is the greatest fixpoint of a monotone function F_2.*

PROOF. *[INDUC]* can be easily transformed into: $\forall i = 1...k$,

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

or $\forall i = 1...k,\ T_I \subseteq F_i(T_I)$. 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

$$F_2 \ =\ \lambda T.\Sigma \cap \bigcap_{i=1}^{k} F_i(T)$$

So, our verification problem is equivalent to showing either $T_I^m \subseteq \Sigma$ (see previous section) or $\forall i = 1...k, T_{P_i} \subseteq T_I^M$, where

$$T_I^m = \bigcup_{n \geq 0} F_1^{(n)}(\emptyset) \quad \text{and} \quad T_I^M = \bigcap_{n \geq 0} F_2^{(n)}(\Theta_X)$$

$$F_1 = \lambda T. \bigcup_{i=1}^{k} T_{P_i} \cup \Big(\exists X', \exists X'',\ C_i \otimes T[X/X'] \otimes T_{P_i}[X/X''] \Big)$$

$$F_2 = \lambda T. \Sigma \cap \bigcap_{i=1}^{k} \Big(\forall X, \forall X'',\ (\Theta_{X \cup X' \cup X''} \setminus C_i) \oplus (\Theta_X \setminus T_{P_i})[X/X''] \ \oplus T \Big) [X'/X]$$

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!). The following example supports the choice of computing T_I^M, since it is a case where T_I^M is regular, and can be computed in a few iterations. This example is also interesting for several reasons:

- 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 next section, which is an attempt to extrapolate the limit of such a sequence.

4.2 Example: a simple token ring

We consider a very simple token ring: Let n units $U_1, U_2, \ldots, U_n$ 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:

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

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

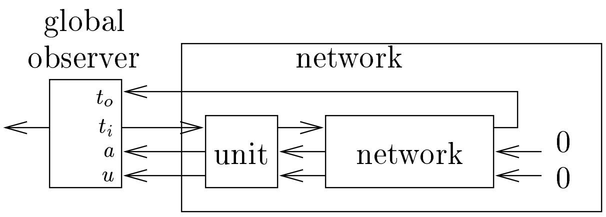


Fig. 3. Global network of the simple token ring

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:

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

We can connect such extended units in a linear network (see Fig. 3). Each such network will have a fixed interface, namely the input signal tk_{in} and the output 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:

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

where the initial value of $assumption$ is assumed to be $(tk_{in} = 1)$.

Fig. 3 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. 4, 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

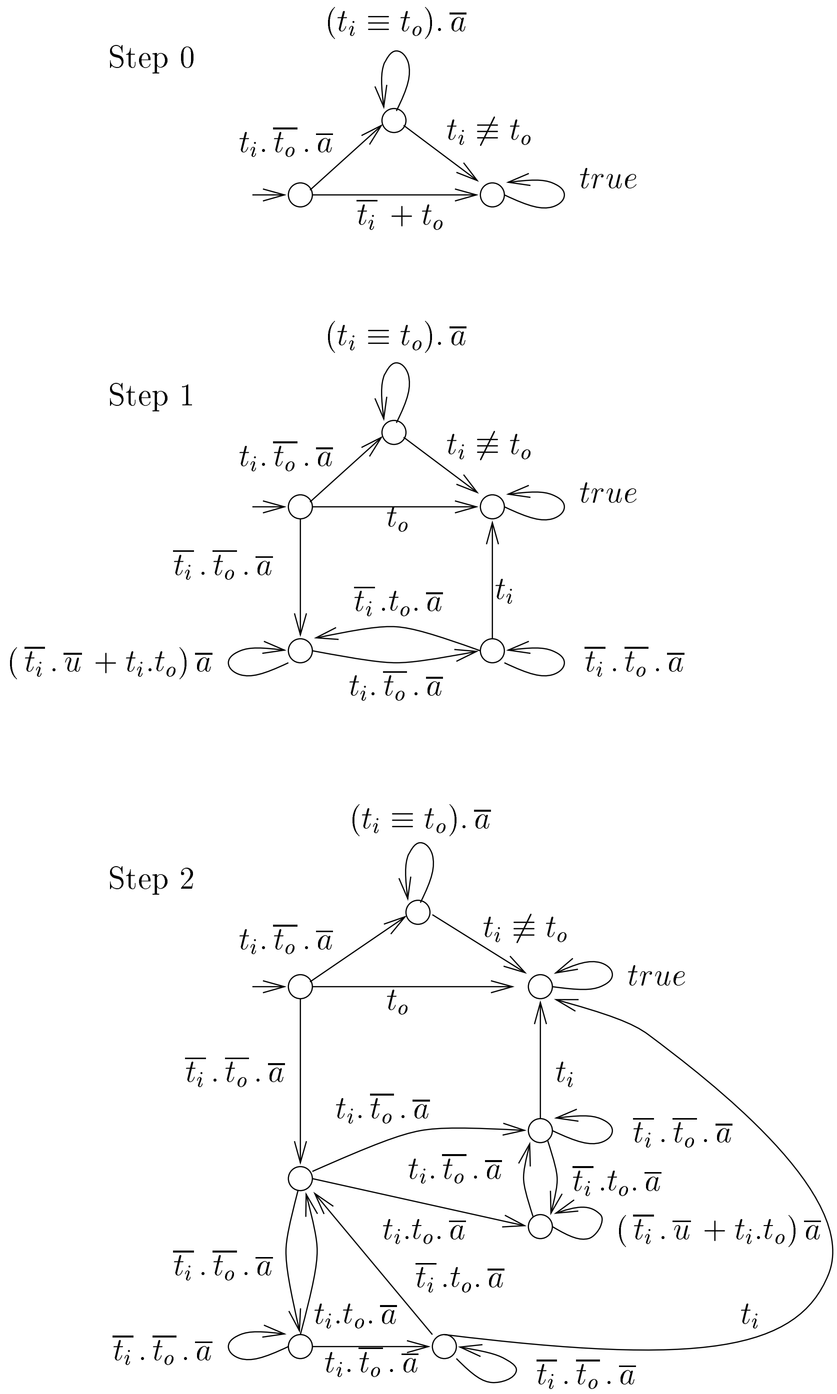


Fig. 4. Computation of the simple token ring

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 show how 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 $\forall i = 1...k, T_{P_i} \subseteq T$.

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 Λ_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 iterative computation of the greatest fixpoint (i.e., $T' = F(T)$). The principle of our extrapolation is to compare the structure of both Λ_T and Λ'_T, so that the one of the next computation step can be guessed.

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

Since some behaviors from the states in D have 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 Λ_T all the states q such that there exists q' such that $(q, q') \in D$, i.e., forbid all the transitions leading into such states. Since the new automaton has strictly less states than Λ_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 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 D to other states $(q_2, q'_2) \notin D$.

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 section 5.3.

Example 6 *Let* $T = (a+b).(a+b+c)^*$ *and* $T' = (a+b.(a+b)).(a+b+c)^*$.

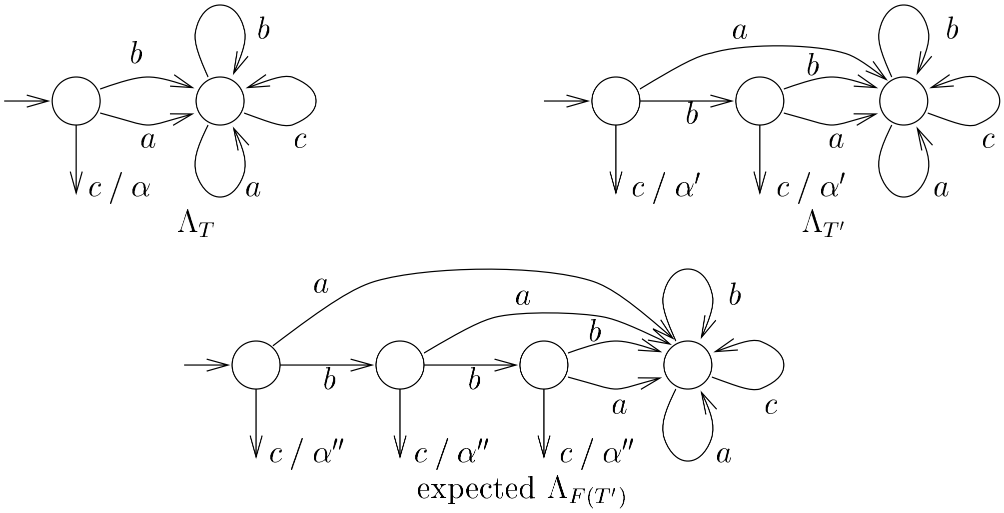
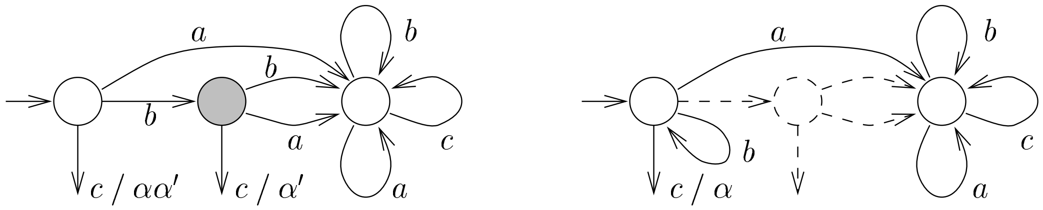


Fig. 5. Successive computation steps



(a) $\Lambda_{T\times T'}$ **(b)** $\Lambda_{T\nabla T'}$

Fig. 6. Widening on automata

Intuitively, one can expect that the next computation step will compute the language $T'' = (a+b.(a+(b.(a+b)))).(a+b+c)^*$*. Automata recognizing* T*,* T' *and* T'' *are shown in Fig 5. Fig 6.a shows the automaton* $\Lambda_\times$ *(where the grey state is the only one in* D*). Fig 6.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*.*

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 $k \geq 0$, 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 [4] 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 grows rapidly, all the computations either converge rapidly, or saturate the memory!

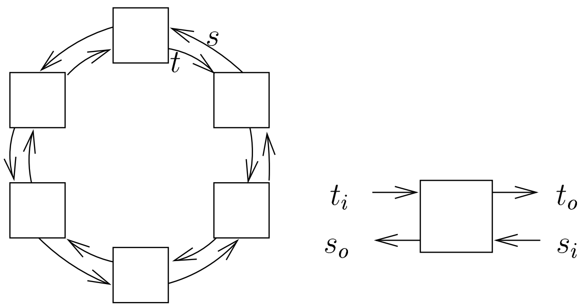
5.4 Examples

Dijkstra's token ring: This algorithm is adapted from the one used in [CGJ95] [5]. 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. A unit can use the resource only when it has the token. To avoid useless token passing, a request signal can travel in the counter-clockwise direction. 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 previous notations, the following system of equations defines this behavior:

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

[5] This algorithm has been presented for the first time in [Mar85], under the name of "reflecting privilege algorithm".



The network A unit

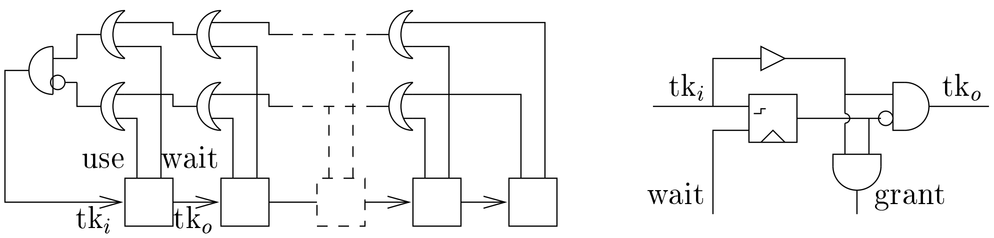
Fig. 7. Dijkstra's token ring

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

Provided with an observer of the mutual exclusion as in section 4.2, 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 4.2.

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 7 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 19 seconds, again with 3 steps and 1 extrapolation. The resulting automaton has 39 states and 1849 transitions.

A hardware arbiter: Our second example comes from [HLR92]: as before, n units $U_1, U_2, \ldots, U_n$ 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 (a)).



(a) network **(b)** arbitration element

Fig. 8. Hardware Arbiter

Fig. 8 (b) 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:

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

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:

$$\mathit{requested} = \mathit{right\_requested} \vee \mathit{wait} \qquad \mathit{used} = \mathit{right\_used} \vee \mathit{use}$$

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

$$grant_i \Longrightarrow \neg wait_j$$

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:

$$\begin{aligned}
&\alpha_o &&= \alpha_i \vee (prio_i \wedge grant)\\
&prio_o &&= prio_i \vee prio\\
&\textbf{next } prio &&= (arb_req \wedge wait) \vee (\neg arb_req \wedge prio)
\end{aligned}$$

We tried to verify each combination of these 3 properties. Results are shown in Table 1: 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 properties but one: when considering "no token lost" alone, the computation does not seem to converge (it runs out of memory after several hours).

6 Tree Network invariants as greatest fixpoints

Let us now consider the case of binary tree networks. Let $\{P_1, \ldots, P_k\}$ be a finite multi-set of processes on a common set X of signals, and $\times$ be a binary composition operator over processes, defined by a set C of traces on

Properties			Nb. extrapolations	Nb. steps	Nb. states	Nb. transitions	Time
E			2	4	31	556	35”
E	N		1	3	6	121	11”
E	N	P	1	3	9	313	9”
E		P	1	3	9	313	12”
	N		*	*	*	*	*
	N	P	1	3	11	224	15”
		P	2	4	16	312	49”

Table 1
Results of the Verification of the Hardware Arbiter

$X \cup X' \cup X''$. A simple binary tree network is a family $\mathcal{F}$ of processes generated by:

$$(\forall i = 1 \dots k,\ P_i \in \mathcal{F}) \quad \text{and} \quad (P', P'' \in \mathcal{F} \Longrightarrow P' \times P'' \in \mathcal{F})$$

In this framework, we have to search a network invariant I such that

$$\begin{array}{ll} \textit{[SAT]} & I \models \varphi \\ \textit{[INIT]} & \forall i = 1 \dots k,\ P_i \prec I \\ \textit{[INDUC]} & I \times I \prec I \quad \text{or equivalently} \\ & \exists X', \exists X'',\ C \otimes T_I[X/X'] \otimes T_I[X/X''] \ \subseteq\ T_I \end{array}$$

This case can easily be extended to more general networks generated by network grammars.

6.1 Geatest fixpoint doesn't exist

Let us rewrite *[INDUC]* as

$$T_I \ \subseteq\ \Big(\forall X, X',\ (\Theta_{X \cup X' \cup X''} \setminus C) \oplus (\Theta_X \setminus T_I)[X/X'] \oplus T_I \Big) [X''/X]$$

or

$$T_I \ \subseteq\ \Big(\forall X, X'',\ (\Theta_{X \cup X' \cup X''} \setminus C) \oplus (\Theta_X \setminus T_I)[X/X''] \oplus T_I \Big) [X'/X]$$

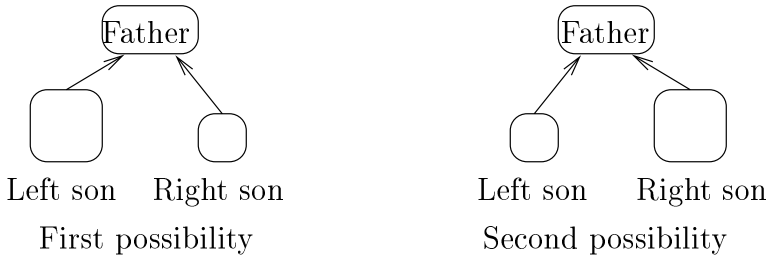


Fig. 9. Intuition about the absence of a greatest invariant

i.e., $T_I \subseteq F(T_I)$. Unfortunately, the function F is no longer monotone, because of the complement taken on T_I. Thus, one cannot conclude to the existence of a greatest fixpoint, as in the linear case.

An intuitive explanation is the following (see Fig 9): The induction consists in finding a condition on the children processes of a node implying a given property of their paent node. Now, it is possible to strongly constrain the left son, while letting the right son more loosely constrained, or conversely. The ideal solution would be to find a unique property for the two sons. In practice, this seems to be impossible, since the problem is generally not exactly symmetrical: the sons are not symmetrically connected to their father, or the father does not behave completely symmetrically with respects to its children (e.g., it transmits a token first to its left son, and then to its right son, ...).

In the next section, we will take into account the fact that properties of the left and right sons have to be distinguished.

6.2 Induction principle with two invariants

Let us consider the induction inequation $I \times I \preceq I$. This inequation means that if the left and the right sons of the node both satisfy the invariant I, then the whole subtree must satisfy the invariant I. Now, if we consider separately the left and right sons, it is enough to find two invariants L (for left child) and R (for right child) such that

$$\begin{array}{ll} \textit{[SAT]} & L \models \varphi \quad \text{and} \quad R \models \varphi \\ \textit{[INIT]} & \forall i = 1...k,\ P_i \preceq L \quad \text{and} \quad P_i \preceq R \\ \textit{[INDUC]} & L \times R \preceq L \quad \text{and} \quad L \times R \preceq R \end{array}$$

Recall $\Sigma = \Theta_{X\setminus\{\alpha\}}$ is the set of traces which never emit the "alarm" signal α..
In our traces semantics, these inequations can be written as

$$\begin{array}{ll} \textit{[SAT]} & T_L \subseteq \Sigma \quad \text{and} \quad T_R \subseteq \Sigma \\ \textit{[INIT]} & \forall i = 1...k,\ T_{P_i} \subseteq T_L \quad \text{and} \quad T_{P_i} \subseteq T_R \\ \textit{[INDUC]} & (\exists X', \exists X'', C \otimes T_L[X/X'] \otimes T_R[X/X'']) \subseteq T_L \quad \text{and} \\ & (\exists X', \exists X'', C \otimes T_L[X/X'] \otimes T_R[X/X'']) \subseteq T_R \end{array}$$

Let us note *[PROOF]=[SAT]∧[INIT]∧[INDUC]* the set of all these inequations.

6.3 Vector of invariants

The use of two invariants instead of one follows intrinsically from the problem of binary tree networks. However, a greatest fixpoint computation is only possible with only one invariant. In order to overcome this difficulty, let us assume that our problem is solved i.e., we know two invariants L and R satisfying the previous inequations, and let us define $V \subseteq \Theta_{X' \cup X''}$ as the following composition of $T_L[X/X']$ and $T_R[X/X'']$:

$$V = T_L[X/X'] \odot T_R[X/X'']$$

Notice that T_L and T_R can be easily retrieved from V by projection

$$T_L = (\exists X'', V)[X'/X] \quad \text{and} \quad T_R = (\exists X', V)[X''/X]$$

Let us now rewrite inequations *[PROOF]* using V.

Proposition 7 *If V can be written $V = T_L[X/X'] \odot T_R[X/X'']$, then* [PROOF]*is equivalent to* [PROOF']=[SAT']∧[INIT']∧[INDUC'] *where*

$$\begin{array}{ll} \text{[SAT']} & V \subseteq \Sigma[X/X'] \odot \Sigma[X/X''] \\ \text{[INIT']} & \forall i = 1 \ldots k,\ T_{P_i} \otimes V \subseteq V[X'/X] \otimes V[X''/X] \\ \text{[INDUC']} & C \otimes V \subseteq V[X'/X] \otimes V[X''/X] \end{array}$$

PROOF. We show that, under the assumption on V, inequations *[SAT]*, *[INIT]*, and *[INDUC]* are respetively equivalent to *[SAT']*, *[INIT']*, and *[INDUC']*.

[SAT]: The rewriting of *[SAT]*into *[SAT']* is straightforward.

[INIT]: First, rewrite the first inequation of *[INIT]* on $X \cup X' \cup X''$:

$$\forall i = 1 \dots k,\ T_{P_i} \odot \Theta_{X'} \odot \Theta_{X''}\ \subseteq\ T_L \odot \Theta_{X'} \odot \Theta_{X''}$$

Since $T_L[X/X'] \subseteq \Theta_{X'}$ and $T_R[X/X''] \subseteq \Theta_{X''}$, this is equivalent to

$$\forall i = 1 \dots k,\ T_{P_i} \odot T_L[X/X'] \odot T_R[X/X'']\ \subseteq\ T_L \odot \Theta_{X'} \odot \Theta_{X''}$$

The first inequation of *[INIT]* can be rewritten[6] into

$$\forall i = 1 \dots k,\ T_{P_i} \odot T_L[X/X'] \odot T_R[X/X'']\ \subseteq\ T_L \odot T_R[X/X''] \odot \Theta_{X'}$$
$$\text{i.e., } \forall i = 1 \dots k,\ T_{P_i} \odot V\ \subseteq\ V[X'/X] \odot \Theta_{X'}$$

In the same way, the conjunction of the two inequations of *[INIT]* can be rewritten into

$$\forall i = 1 \dots k,\ T_{P_i} \odot V\ \subseteq\ V[X'/X] \otimes V[X''/X]$$

[INDUC]: In a similar way, *[INDUC]* is rewritten into

$$C \otimes T_L[X/X'] \otimes T_R[X/X'']\ \subseteq\ T_L \odot T_R[X/X''] \odot \Theta_{X'} \text{ and}$$
$$C \otimes T_L[X/X'] \otimes T_R[X/X'']\ \subseteq\ T_L[X/X'] \odot T_R \odot \Theta_{X''}$$
$$\textit{i.e.,}\ C \otimes V\ \subseteq\ V[X'/X] \otimes V[X''/X]$$

Proposition 8 *There is a greatest set of traces V^{max} satisfying* [PROOF'].

PROOF. It is easy to show that *[INIT']* and *[INDUC']* can be rewritten as

$$\textit{[INIT']} \quad \forall i = 1 ... k,\ V \subseteq \forall X, \Big((\Theta_X \setminus T_{P_i}) \oplus (V[X'/X] \otimes V[X''/X]) \Big)$$
$$\textit{[INDUC']}\ V \subseteq \forall X, \Big((\Theta_{X \cup X' \cup X''} \setminus C) \oplus (V[X'/X] \otimes V[X''/X]) \Big)$$

This means that V is a pre-fixpoint of the (monotone) function

$$F\ =\ \lambda V.\ \forall X, \Big(\Sigma[X/X'] \odot \Sigma[X/X''] \Big) \otimes \Big((\Theta_{X \cup X' \cup X''} \setminus (C \otimes \bigcup_{i=1}^{k} T_{P_i})) \oplus (V[X'/X] \otimes V[X''/X]) \Big)$$

There is a greatest solution, V^{max}, which is the greatest fixpoint of F.

Proposition 9 *If V^{max} is empty, some processes generated by the binary tree network do not satisfy the property φ.*

[6] Using a trivial property on sets: $\forall A, B, C,\ (A \cap B \subseteq C)\ \Leftrightarrow\ (A \cap B \subseteq C \cap B)$

PROOF. Proposition 4 shows that there exists a minimal set of traces T_I^m satisfying both *[INIT]* and *[INDUC]*. Suppose that the tree network satisfies the property φ. Since T_I^m represents the set of traces of all possible tree networks, T_I^m must satisfy the property φ. Let $V = T_I^m[X/X'] \otimes T_I^m[X/X'']$. Then V is non empty and must satisfy the inequations *[PROOF']*, i.e., $V \subseteq V^{max}$. This contradicts the hypothesis.

As before, in general, V^{max} cannot be exactly computed. Heuristics proposed in section 5 can be used to get under-approximations V of V^{max} satisfying *[PROOF']*.

Now, does the existence of such a non-empty vector V imply that φ is satisfied by the binary tree network? It is only the case if V can be decomposed into two invariants L and R, since inequations *[PROOF]* are equivalent to *[PROOF']* only if V can be written $V = T_L[X/X'] \odot T_R[X/X'']$ (which is not generally the case). In section 7, we will propose heuristics to find two languages T_L and T_R such that the vector $T_L[X/X'] \odot T_R[X/X'']$ satisfies *[PROOF']*.

6.4 Examples

6.4.1 A token tree

Let n units P_1, P_2, ... ,P_n share a resource in mutual exclusion. They are connected in a binary tree, along which a token travels in depth. A process P_i is defined as in section 4.2. It can only use the resource when it has the token. It has one input signal tk_{in} (token in) and two output signals tk_{out} (token out) and *use* (resource used).

Each node has 4 input signals and 4 output signals, corresponding to the communication with its father, its left child, its right child and its associated unit. When a node receives the token from its father, it transmits it to its left child. When it receives the token from its left child, it transmits it to its right child. And finally, when it receives the token from its right child, it gives it back to its father. Each time the token reaches the node, it is transmitted also to the unit associated to the node, which can keep it for some time in order to use the resource (see Fig 10).

The mutual exclusion observer of a node has 5 input signals: *use* (the resource is used by the associated unit), ν_l (the resource is used in the left branch), α_l (mutual exclusion is violated in the left branch), ν_r (the resource is used in the right branch) and α_r (mutual exclusion is violated in the right branch). It emits the two signals α (the mutual exclusion property is violated) and ν (the resource is used by its unit or one of its children).

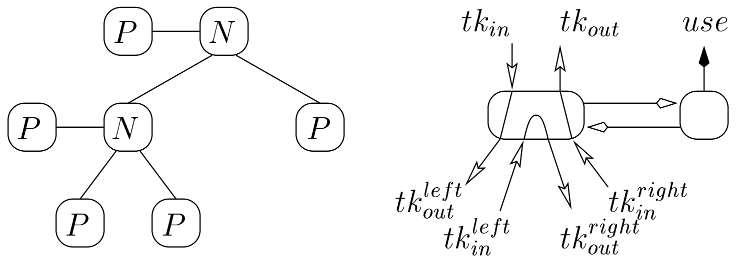


Fig. 10. Processes are connected to the nodes. P is a process, N is a node.

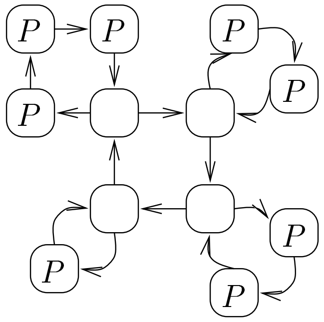


Fig. 11. Network in petals

The forward computation of invariant saturates the memory after 2 steps taking several hours. In contrast, the invariant V^{max} is exactly computed backward in 5 iterations in 19'15". It has 928 states and 72379 transitions.

6.4.2 A network in petals

This second example shows that the technique of invariant computation on binary tree networks can be applied to asymmetrical networks (where left and right children are defined differently).

Let us consider a main ring, composed of nodes (called main nodes) which are associated with secondary rings (see Fig 11). This kind of networks can be generated by the following grammar (see Fig 12):

$$S \to L \times_1 R \qquad L \to L \times_2 P \qquad R \to L \times_1 R$$
$$L \to P \qquad R \to P$$

As soon as a process of a secondary ring needs the resource, it sends a request signal to the corresponding main node. Then the token is received by a main node:

- Either a request signal has been received (i.e., a process of the ring asks for the resource), and the token is sent in this secondary ring.

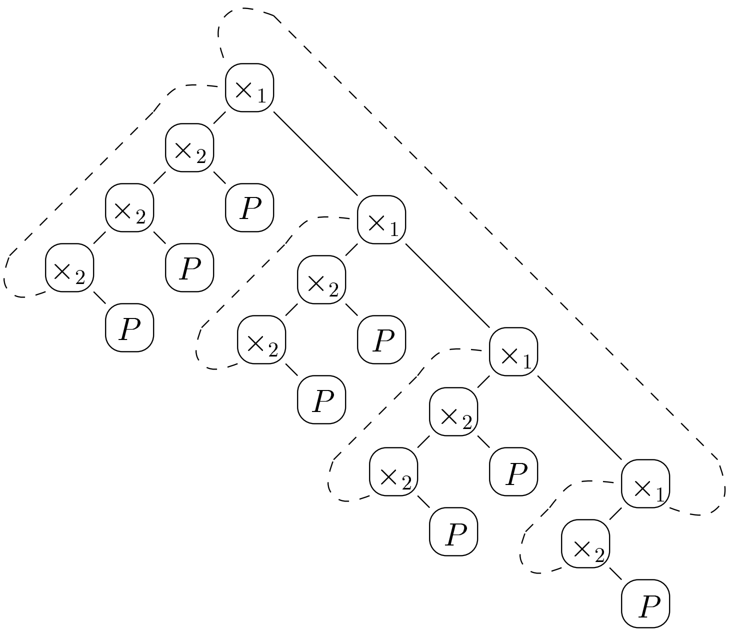


Fig. 12. Construction of a network in petal by a binary tree grammar

- Or no request signal has been received, and the token is transmitted to the following main node.

Each process has 1 input signal tk_{in}, for "token in" (the process receives a token), and 2 output signals tk_{out}, for "token out" (the process emits a token) and sg_{out}, for "signal out" (the process emits a request signal). It has 2 internal signals req and rel for the token request and the token release. The composition operator $\times_1$ is defined in such a way that the token is sent in a secondary ring only if a request signal has been received.

The forward computation of invariant saturates the memory after 2 steps taking several hours. In contrast, we were able to compute the greatest invariant V^{max} on an abstract network, where the request signal is abstracted. The computation takes 3h18'. V^{max} has 612 states and 50782 transitions.

7 Computation of invariants L and R

Let V be a vector satisfying *[PROOF']*. V expresses a property that the tuple (L, R) must satisfy to verify equations *[PROOF]*. Intuitively, the fact that it cannot be decomposed in the form $V = T_L[X/X'] \odot T_R[X/X'']$, means that L and R are *dependent* (some behavior of L implies a specific reaction of R). The goal of this section is to find independent L and R.

7.1 Approximations of L and R

7.1.1 Upper-bound

Let T_L^M and T_R^M be two sets of traces defined by

$$T_L^M \;=\; \Big(\exists X'', V\Big)[X'/X] \quad \text{and} \quad T_R^M \;=\; \Big(\exists X', V\Big)[X''/X]$$

All solutions (T_L, T_R) of inequations *[PROOF]* will be such that $T_L \subseteq T_L^M$ and $T_R \subseteq T_R^M$. (T_L^M, T_R^M) can be considered as a upper-bound.

7.1.2 Lower-bound

If we choose $T_L = T_L^M$, in order to satisfy the inequation $T_L[X/X'] \odot T_R[X/X''] \subseteq V$ one has to choose

$$T_R \;\supseteq\; \Big(\forall X', (\Theta_X \setminus T_L^M)[X/X'] \oplus V\Big)[X''/X]$$

Thus, let T_L^m and T_R^m be two sets of traces defined by

$$\begin{aligned} T_L^m &= \Big(\forall X'', (\Theta_X \setminus T_R^M)[X/X''] \oplus V\Big)[X'/X] \;\cup\; \bigcup_{i=1}^{k} T_{P_i} \\ T_R^m &= \Big(\forall X', (\Theta_X \setminus T_L^M)[X/X'] \oplus V\Big)[X''/X] \;\cup\; \bigcup_{i=1}^{k} T_{P_i} \end{aligned}$$

All solutions (T_L, T_R) of inequations *[PROOF]* will be such that $T_L^m \subseteq T_L$ and $T_R^m \subseteq T_R$. (T_L^m, T_R^m) can be considered as a lower-bound. Generally, T_L^m, T_L^M, T_R^m and T_R^M do not satisfy *[PROOF]*. The next section will propose an algorithm based on heuristics to compute suitable invariants.

7.2 Decomposition of V with respect to a lower-bound

Our goal is to compute two sets of traces T_L and T_R, satisfying

$$T_L[X/X'] \odot T_R[X/X''] \subseteq V \tag{1}$$

This problem has no maximal unique solution. Intuitively, in order that T_L and T_R satisfy *[PROOF]*, the product $T_L[X/X'] \odot T_R[X/X'']$ must be as close as possible to V. This section proposes heuristics allowing the computation of a solution of (1) which satisfies also *[PROOF]*.

7.2.1 Principle

We propose an algorithm based on automata: if Λ is a process, i.e., a deterministic Mealy machine with 2^X as input alphabet, let us note T_Λ the set of traces of Λ. Now, let Λ_L^0 and Λ_R^0 be two automata on X such that $T_{\Lambda_L^0}[X/X'] \odot T_{\Lambda_R^0}[X/X''] \not\subseteq V$. The principle of our algorithm is to remove some transitions of Λ_L^0 or of Λ_R^0 in such a way to obtain two new automata Λ_L and Λ_R such that $T_{\Lambda_L}[X/X'] \odot T_{\Lambda_R}[X/X''] \subseteq V$.

This way, if $T_{\Lambda_L^0}[X/X'] \odot T_{\Lambda_R^0}[X/X''] \not\subseteq V$, there exist two traces $\tau L, \tau R \in \Theta_X$ accepted respectively by Λ_L^0 and Λ_R^0, and such that $\tau L[X/X'] \odot \tau R[X/X'']$ is not element of V. We can then, either remove a transition of Λ_L^0 in order that τL is refused, or remove a transition of Λ_R^0 in order that τR is refused. One can remove any transitions, as long as inclusions $T_{P_i} \subseteq T_{\Lambda_L}$ and $T_{P_i} \subseteq T_{\Lambda_R}$ are satisfied. More generally, following inclusions must be preserved:

$$T_L^m \subseteq T_{\Lambda_L} \quad \text{and} \quad T_R^m \subseteq T_{\Lambda_R} \tag{2}$$

7.2.2 Choice of Λ_L^0 and Λ_R^0

Theoretically, any automata Λ_L^0 and Λ_R^0 verifying $T_{\Lambda_L^0}[X/X'] \odot T_{\Lambda_R^0}[X/X''] \not\subseteq V$ are suitable. In practice, these automata structures must be derived from the one of V. Automata Λ_L^M and Λ_R^M, which recognize respectively T_L^M and T_R^M, satisfy these properties.

In order to preserve inclusions (2), we propose moreover to mark transitions of Λ_L^M and of Λ_R^M which cannot be removed. Thus, one can choose Λ_L^0 as the automaton recognizing T_L^M such that any trace of T_L^m is recognized by marked transitions, and any trace of $(T_L^M \setminus T_L^m)$ is recognized by transitions at least one of which is not marked. Λ_R^0 will be chosen in the same way. Let Λ_L^m and Λ_R^m be automata recognizing respectively T_L^m and T_R^m. Let us assume that each of them has a sink state such that the alarm signal α is emitted only by transitions reaching this state, and let us mark all transitions of Λ_L^m and Λ_R^m. Thus, one states $\Lambda_L^0 = \Lambda_L^m \| \Lambda_L^M$ and $\Lambda_R^0 = \Lambda_R^m \| \Lambda_R^M$, where $\|$ denotes the synchronous product and where alarm signals of Λ_L^0 and Λ_R^0 are respectively the one of Λ_L^M and the one of Λ_R^M.

7.2.3 Heuristics

In this section, we propose heuristics to remove some transitions of Λ_L^0 or of Λ_R^0 in order to satisfy the inclusion (1). Let τL and τR be two traces on X

Example 10

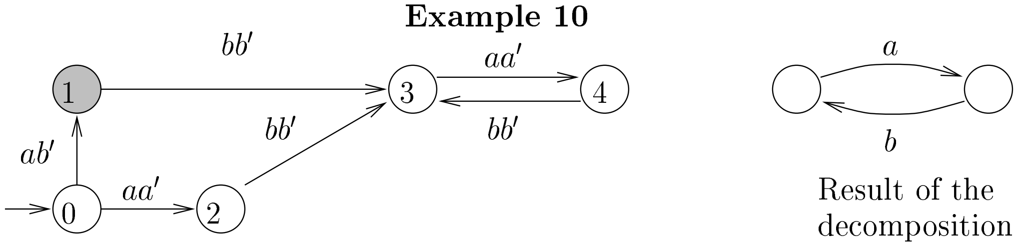


Fig. 13. Example of an automaton recognizing the language V. The grey state has to be removed to make this automaton symmetrical

such that

$$\tau L[X/X'] \odot \tau R[X/X''] \in (T_{\Lambda_L^0}[X/X'] \odot T_{\Lambda_R^0}[X/X'']) \setminus V$$

There are two paths π_L and π_R respectively in Λ_L^0 and Λ_R^0 corresponding to traces τL and τR. In order to satisfy the inclusion (1), one has to choose two indexes k_L and k_R such that, either the k_L^{th} transition on the path π_L or the k_R^{th} transition on the path π_R, is removed. One can choose either k_L or k_R *maximal* (intuitively, this comes to remove all traces with a particular suffix), or k_L or k_R *minimal* (intuitively, this comes to remove all traces with a particular prefix). Experiments showed that only the second choice gives good results. In order to formalize the algorithm, let us introduce the function f_m taking as argument a trace τ and returning the minimal index k such that the k^{th} transition of the path corresponding to τ is not marked. Thus, our decomposition algorithm is the following:

Algorithm 1

$\Lambda_L = \Lambda_L^m \| \Lambda_L^M$; $\Lambda_R = \Lambda_R^m \| \Lambda_R^M$;
While $T_{\Lambda_L}[X/X'] \odot T_{\Lambda_R}[X/X''] \not\subseteq V$
  Let $\tau L \odot \tau R \in (T_{\Lambda_L}[X/X'] \odot T_{\Lambda_R}[X/X'']) \setminus V$
  Let $k_L = f_m(\tau L)$ *and* $k_R = f_m(\tau R)$
  if $k_L \leq k_R$ *then one removes the* k_L^{th} *transition on the path of* Λ_L *corresponding to* τL
  else one removes the k_R^{th} *transition on the path of* Λ_R *corresponding to* τR
End of while

Let us consider the set of signals $\{a, b\}$. *The automaton of Fig 13 recognizes the language* V *defined by*

$$V \;=\; ab'.(bb'.aa')^* + (aa'.bb')^*$$

The word $(aa'.bb')^*$ *is symmetrical, in that if the prime and non prime variables are exchanged, the obtained words belong to the language* V. *In the opposite, if* $ab'.(bb'.aa')^*$ *belongs to* V, *the symmetrical word* $ba'.(bb'.aa')^*$ *does not. This* *word is "removed" in forbidding the transition* $0 \rightarrow 1$. *Thus, we obtain as invariant the language* $(a.b)^*$.

The previous algorithm is not exactly symmetrical with respect to L and R, since the tests $k_L \leq k_R$ and $k_R < k_L$ are computed. A dual algorithm can be defined, where the tests $k_L < k_R$ and $k_R \leq k_L$ would be computed. It is not possible *à priori* to find the best solution.

7.3 Examples

7.3.1 First example: the token tree

Let us come back to the example of section 6.4.1. In the first case where the processes are connected with the leaves, the greatest fixpoint V^{max} is decomposed in two invariants L and R which have respectively 32 states and 276 transitions, and 7 states and 57 transitions. In the second case where the processes are connected to the nodes, V^{max} is decomposed in two invariants L and R which have respectively 27 states and 266 transitions, and 15 states and 111 transitions. Thus, we can conclude that any process generated by the grammar satisfies the property.

7.3.2 Second example: the network in petals

Let us come back to the example of section 6.4.2. In the first case without arbitration device, the greatest fixpoint V^{max} is decomposed in two invariants L and R which have respectively 32 states and 284 transitions, and 16 states and 127 transitions. In the second case with an arbitration device, V^{max} is decomposed in two invariants It has 777 states and 51711 transitions. This invariant is then decomposed in two invariants L and R which have respectively 28 states and 219 transitions, and 22 states and 157 transitions. Thus, we can conclude that any process generated by the grammar satisfies the property.

8 Conclusion

We have proposed a way to specify safety properties of parameterized networks of processes and a method and a tool to verify such properties by synthesizing network invariants. To avoid the non convergence of the least fixpoint computation, a technique of computation of greatest fixpoint is proposed, which takes care of the two children of a node at the same time in the case of tree networks. Heuristics have been proposed to

(1) under-approximate the greatest fixpoint;
(2) decompose a vector of invariants.

All these techniques have been implemented in a tool. 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], we think that, in our 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. In fact, we believe that the least and the greatest fixpoint are complementary. For the parity-tree example of section 3 (which is the only one used in [CGJ95]), the computation of the greatest fixpoint is much longer than the one of the least fixpoint. In contrast, in all the examples of sections 5.4 and 6.4, the least fixpoint computation saturates the memory after only two steps, while the one of greatest fixpoint converges rapidly.

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.

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.

[FO97] L. Fribourg and H. Olsén. Reachability sets of parameterized rings as regular languages. In *International Workshop on Verification of Infinite State Systems (INFINITY)*, Bologna, July 1997.

[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.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *9th Conference on Computer Aided Verification, CAV'97*, July 1997.

[Les97] D. Lesens. Invariants of parameterized binary tree networks as greatest fixpoints. In *Sixth International Conference on Algebraic Methodology and Software Technology, AMAST'97*, Sydney, December 1997.

[LHR96] D. Lesens, N. Halbwachs, and P. Raymond. Automatic construction of network invariants. In *International Workshop on Verification of Infinite State Systems (INFINITY)*, Pisa, August 1996.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, January 1997.

[Mar85] A. J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

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

[MG91] R. Marelly and O. Grumberg. Gormel-grammar oriented model checker. Technical Report 697, The Technion, October 1991.

[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.

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[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.

A Computation of abstractions

The complexity of the algorithms presented in this paper is mainly due to the computation of abstractions $\exists Y, T$ and $\forall Y, T$ (see §3). Let us detail these computations.

Let $T \subseteq \Theta_X$ be a regular, prefix-closed, set of traces on X and $\Lambda_T = (Q_T, q0_T, X, \{\alpha_T\}, \delta_T, \delta_T^\alpha)$ 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).
- $\delta_T : Q_T \times 2^X \rightarrow Q_T$ is the *total* transition function.
- $\delta_T^\alpha : Q_T \times 2^X \rightarrow \{\emptyset, \alpha_T\}$ 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 Λ_T labels all signals which are in Y. We obtain a non-deterministic automaton which can be determinized by the following classical algorithm

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

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

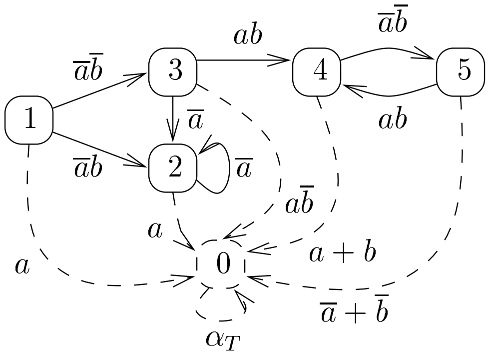
$$\delta_{\exists Y,T}(\tilde{q}, x) = \{q' | \exists q \in \tilde{q}, \exists y \in Y,\ q' = \delta_T(q, x \cup y)\}$$

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

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

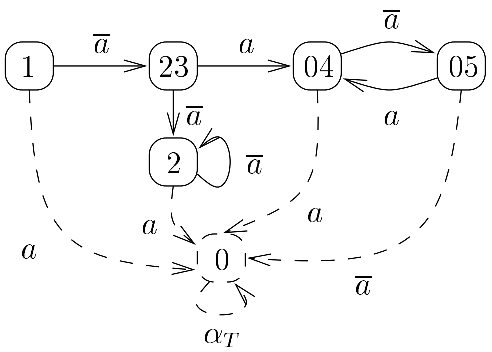
The observer of $\forall Y, T$ accepts a trace $\tau_x \in \Theta_{X \setminus Y}$ if and only if, for all traces $\tau_y \in \Theta_Y$, $\tau_x \odot \tau_y \in T$. Thus, it complains if and only if there exists a trace τ_y such that Λ_T complains on $\tau_x \odot \tau_y$, i.e., if $\Lambda_{\exists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

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



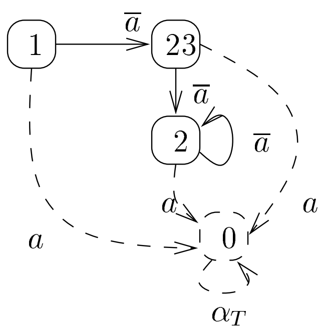
Automaton Λ_T

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



Automaton $\Lambda_{\exists b,\ T}$

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



Automaton $\Lambda_{\forall b,\ T}$

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

Fig. A.1. Abstraction algorithm

where

$$\delta^{\alpha}_{\forall Y,T}(\tilde{q}, x) = \begin{cases} \{\alpha_T\} \text{ if } q^s_T \in \delta_{\exists Y,T}(\tilde{q}, x) \\ \emptyset \qquad \text{otherwise} \end{cases}$$

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

Fig A.1 illustrates these constructions.