Verification of distributed systems : an experiment

Didier Vergamini

INRIA

UNITE DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tél. (1) 39635511

Rapports de Recherche

N° 934

Programme 1

VERIFICATION OF DISTRIBUTED SYSTEMS : AN EXPERIMENT.

Didier VERGAMINI

Décembre 1988

RR_0934

Verification of Distributed Systems: an Experiment. Vérification de Systèmes Distribués: une Expérience.

Didier Vergamini
CERICS
Sophia Antipolis B.P. 48
06561 VALBONNE Cedex
France

October 12, 1988

Abstract

This paper describes the use of two software tools in verification of distributed systems. Our first tool AUTOGRAPH provides us with a graphic editor, dedicated to hierarchically building networks of automata. The semantics of such a network is itself a finite transition system, usually called the global system. Then the verification method, implemented in the second software tool called AUTO, consists in computing small-scale models of finite transition systems. These reduced systems are quotients of the one under study, up to generalized bisimulation ([Par81,Mil80]). The parameter of the reduction is a user-defined abstraction criterion ([Bou85b]), which embodies a particular view-point on a system. So one is able to build a variety of quotients of a same system, which are small enough to verify particular properties. To demonstrate the use of AUTOGRAPH and AUTO we deal with the well-known problem of philosophers eating spaghetti.

Résumé

Ce rapport décrit l'utilisation de deux logiciels pour la vérification des systèmes distribués. Le premier outil AUTOGRAPH est un éditeur graphique qui permet de construire des réseaux hierarchiques d'automates. La sémantique d'un tel réseau est elle-même un système de transitions fini, habituellement appelé le système global. La méthode de vérification, réalisée dans le second logiciel AUTO, consiste à calculer des modèles réduits d'automate. Ces systèmes réduits sont des quotients de celui étudié, par rapport à une bisimulation généralisée ([Par81,Mil80]). Le paramètre de la réduction est un critère d'abstraction défini par l'utilisateur ([Bou85b], qui comprend un point de vue particulier d'un système. Pour montrer l'utilisation de AUTOGRAPH et de AUTO, nous traitons le problème bien connu des philosophes mangeurs de spaghetti.

1 Introduction.

In this paper we aim to present two software tools for verification of distributed systems. Let us briefly discuss the problem, distinguishing as in [BS80] two classes of verification methods:

- temporal logic and verification of properties on global systems (see [C*85,Hai82] for instance),
- algebraic theories [BK84,HM85,Mil86,Mil84] and reductions [Lip75,BT82,Sif84].

The main advantage of the first approach is that it allows to check for various "exact" properties, looking at the same system from various view points. Its main drawback is that it deals with a global system: then it cannot cope with the well-known size explosion phenomenon. Compositional proof systems try to remedy this deficiency ([Pnu85]).

On the other hand, algebraic theories provide a natural framework for achieving modular verifications, since they provide a syntax to describe how a system is made out of subsystems. However, they do not allow one to check for a variety of properties. For instance, in CCS [Mil80], provision is made only for verifying that a system is observationally equivalent to its specification. But this specification is, in fact, rarely known, especially during the development phase of a program.

The verification method presented in the paper combines modularity in the description of the systems and flexibility in the verified properties. We shall graphically represent distributed systems as networks of communicating automata. Our first tool AUTOGRAPH provides us with a graphic editor, dedicated to hierarchically building networks of automata.

The semantics of such a network is itself a finite transition system, usually called the global system. Then the verification method, implemented in the second software tool called AUTO, consists in computing small-scale models of finite transition systems. These reduced systems are quotients of the one under study, up to generalized bisimulation ([Par81,Mil80]). The parameter of the reduction is a user-defined abstraction criterion ([Bou85b]), which embodies a particular view-point on a system. So one is able to build a variety of quotients of a same system, which are small enough to verify particular properties. Our approach is modular, since in many cases the equivalence we use is a congruence: a subsystem may be replaced by a reduced equivalent one.

To demonstrate the use of AUTOGRAPH and AUTO we deal with the well-known problem of philosophers eating spaghetti (see for instance [Hoa78]).

2 The example.

We assume that some philosophers meet together to eat succulent spaghetti. They may use as many forks as they are. To eat spaghetti, one needs two forks. Each philosopher can take a fork if there is one let free on the table but he will only drop it after eating. In this paper, we shall deal with four philosophers. After showing the modelization, we verify that this system may deadlock (when each philosopher takes one fork). Then we give a solution that prevents the deadlock and we show that it is not fair (a philosopher may starve while the others are always eating). We then provide a last solution without starvation.

3 Basic structure: finite automata.

The behaviour of a "communicating process" representing its possible communications at each step of its life induces a natural operational model, namely that of transition systems. A transition system is a structure $\langle Q, s, T \rangle$ where Q is the set of states, $s \in Q$ is the initial state and $T \subset Q \times \mathcal{A} \times Q$ is the transition relation where $\mathcal{A}$ is the set of atomic actions. Figure 1 shows the graphic representation of two such entities.
The first one represents the behaviour of a *fork*:

- a fork can only be "free" or "in use" so it has two states, one for each possibility,
- the transition from the state `free` to the state `in use` is commanded by the reception of the signal `take`, the transition from `in use` to `free` is commanded by the reception of the signal `drop`,
- the initial state is `free` (double circled).

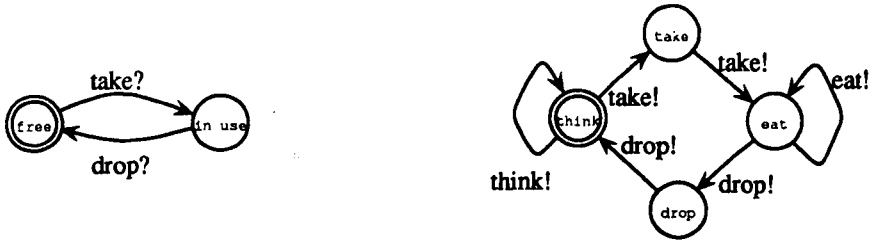


Figure 1: A fork and a philosopher.

At this point, we can be more precise about the actions we consider. We use the "rendez-vous" model for communication, so we have to deal with the reception and emission of a signal. We follow Milner's formalization of this model ([Mil80]). Moreover, as in Meije ([Bou85a]), we want to allow several actions to be done at the same instant; if a signal is both emitted and received in the same instant, the resulting action will be a special action noted **eps**, which marks that a communication has occurred without any indication about what signal is used for it. So the natural structure of our set of actions is the free abelian group generated by the set of names of signals, with **eps** as unit. As a convention, the emission of the signal **s** is noted **s!** and its reception **s?**.

For instance, we see in the second picture of the figure 1, showing the philosopher, its potential communications with the forks. A philosopher may "think" itover before he successfully grabs his two forks, after which he may "eat",then drop the forks.

4 Building networks of automata.

We have seen so far how a single system may be modelled by an automaton. The next step is to model parallel and communicating systems, building networks of automata. These networks, which are Milner's flowgraphs ([Mil79]), are made out of boxes with ports joined by wires – see for instance figure 2. Boxes may contain other boxes or automata. The system AUTOGRAPH provides a graphic editor for these networks as well as for finite automata, and generates from them terms of the Meije process algebra ([Bou85a,Bou85b], which is similar to SCCS ([Mil83]). We shall not describe this system in the present paper, but the reader can surely guess its functionalities by looking at figures: they were all produced using AUTOGRAPH through a PostScript output.

4.1 The ports.

A port on a box is associated with a signal name. The intuitive meaning is that if a port is on the edge of a box, this box is intended to communicate via the corresponding signal. On the other hand, the fact that a signal does not appear on the edge of a box but appears on inner boxes corresponds to CCS restriction.

4.2 Parallelism.

To model the parallel combination of networks, we just place the components aside in a same box. If one wants some signals used by a component to be also used at the outer level, one must connect the corresponding port to a port of the including box. This connection is either explicit, by connecting the two ports with a wire, or implicit, by giving the same name to inner and outer ports – see figure 2. One can also instantiate a signal by changing its "external name" (this corresponds to CCS renaming). Let us return to our example for illustration. We want to place four philosophers in a common network while being able to distinguish which philosopher is thinking or eating: we connect each port **eat** and **think** to a numbered port **eat-i** and **think-i** respectively. Figure 2 shows this first network.

We give in figure 3 the second network of our example: the heap of four forks. In this network, we do not care about which fork is "free" or "in use", so all internal ports are (implicitly) connected to a same external port: this naming saves wires, and so avoids cramping the drawing.

4.3 Communication.

We can draw a communication wire connecting two ports of different components of a parallel system. The meaning of these wires is quite intuitive: this is a rendez-vous between the signals occurring at each end. We see an example

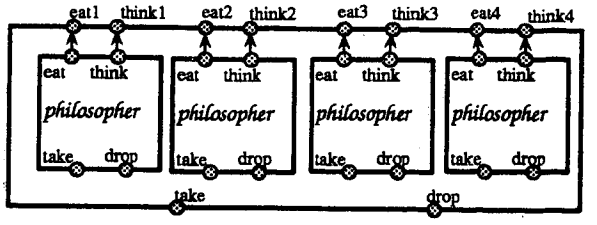


Figure 2: The academy.

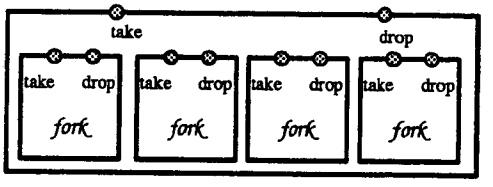


Figure 3: The heap.

of such communications in figure 4, which gives the network modelling the whole problem.

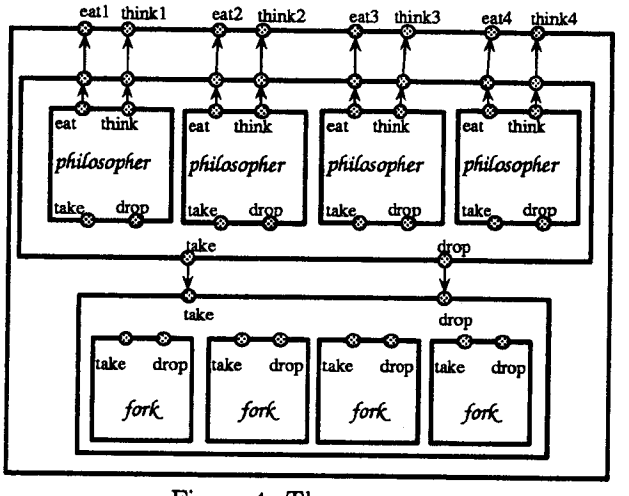


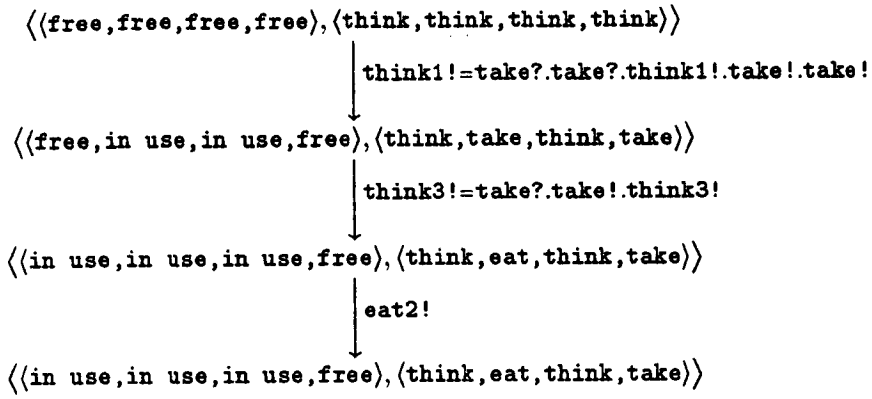
Figure 4: The orgy.

5 The semantics of networks.

The semantics of a network is again an automaton. We call it the global system determined by the network. It is defined structurally as follows:

- the semantics of an automaton (atomic network) is obviously itself,
- for a non atomic network, the set of states of the global system is a part of the cartesian product of the set of states of its components. The transitions of each state are computed using the following laws:
  - every component *may* participate in a transition,
  - the resulting action is the product of all the actions (ports connected together are renamed to a same new name),
  - the global actions must be composed of signals visible from the outer level,
  - the resulting state is the n-uple of new states of the components (some of them may be unchanged if the corresponding components did not take part in the transition).

These rules are those of the underlying Meije calculus. We shall not go in all the technical details, since this semantics is rather intuitive. Let us just see an example showing some transitions of the global system of **orgy** whose basic components are the four forks and the four philosophers:



The first task of the system AUTO is to compute the global system determined by a network. Then the first command we introduce is the one achieving this task. Dealing with our philosophers problem, we shall set an identifier `O` to the automaton denoted by the network **orgy** by:

```
@ set O=tta orgy
```

where `@` is the prompt of AUTO and **tta** is the name of the function transforming terms to automata. In fact, to limit the size explosion due to the parallel operation, one better uses another function **exclusion**, whereby some signals are declared incompatible:

```
@ set Orgy=exclusion(orgy,{{eat1,eat2,eat3,eat4
@                          think1,think2,think3,think4}});
```

`Orgy` is then a restriction of the global system built under the constraint that several philosophers cannot eat or think simultaneously. The second argument of this function is a list of lists of signal names, where two signals in a same list are incompatible (i.e they cannot appear simultaneously in a global action).

AUTO provides some commands to get basic informations about automata. The command **display** prints the size of an automaton (number of states, number of transitions and number of actions) and its transition table. With the optional keyword **short**, only the size is printed.

```
@ display Orgy short;
size = 162 states, 3021 transitions, 9 actions.
```

The command **explore** prints the transition table starting from the initial state and then, interactively, for any requested state. Two more commands deal with actions and signals:

- **sort** computes the list of signals by means of which the automaton communicates,
- **action** gives the list of all possible actions of the automaton.

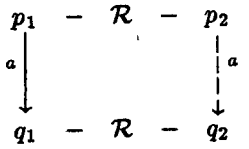
In our example, since the signals of the global system are incompatible, the sort and the actions differ only by the special action **eps** which means internal communication.

6 Reductions.

Of course, the preceding tools are not sufficient to give a true analysis of a network. The main method we want to advocate in this paper is based on reductions of a system. These collapse states of the automata to reach sizes reasonable enough to be outprinted and well understood. Let us consider an example. The network of the heap of four forks denotes an automaton of 16 states (2^4) and 240 transitions. However, the only relevant information should be the number of forks currently in use: this system has only five meaningful states. As a matter of fact, this is the result we get after reduction by strong equivalence. We now explain this first reduction method.

6.1 Strong equivalence.

The notion of bisimulation due to Park and Milner [Par81] expresses that two states are equivalent when, if one is able to perform a transition, the other one has a transition labelled with the same action, reaching equivalent states. It is usual to picture this property as follows:



The coarsest bisimulation is an equivalence, called strong equivalence, inducing a natural notion of quotient of the automaton. The quotient of an automaton by this relation is minimal among the equivalent automata, considering the number of states. It can be computed (for finite transition systems) using a fix-point method. The corresponding function in AUTO is called `mini`. For instance:

```
@ set Heap=mini heap;
@ display Heap short;
size = 5 states, 30 transitions, 14 actions
```

An interesting feature of strong equivalence is that it is a congruence. This means that we can reduce a component of a network before computing the global system. Indeed this method has been used to compute the previous system `Orgy`, replacing the heap of forks by its reduction. This system is thus equivalent to the global one, which needs not be computed.

6.2 Weaker equivalences.

The reduction by strong equivalence cannot be regarded as a sufficient verification method, since it preserves the full semantics. To perform further reductions, we have to parameterize the bisimulation with an *abstraction criterion*, thus obtaining weaker equivalences, leading to partial properties. A well-known example is Milner's observational equivalence [Mil80], where, roughly speaking, we ignore the internal communications. In other words, for this abstraction, performing a visible action is the same as performing it amid whatever sequence of `eps`. The observational equivalence induces a new quotient, which is computed in AUTO using the function `obs`. For instance:

```
@ set ORGY=obs Orgy;
```

reduces `Orgy` to the two-state automaton shown in figure 5. Its analysis is now quite clear: in our network, each philosopher can eat but there is a possible deadlock.

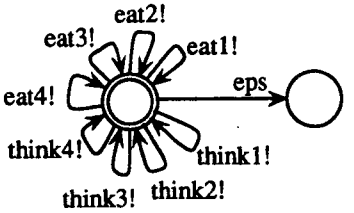


Figure 5: `ORGY`.

Using the same method, we could check that with one more fork there is no deadlock.

It could be remarked that the observational equivalence is a congruence with respect to the network constructors. Then we can use the previous strategy of reducing the components before computing the global system. This is usually much more efficient than working directly on the global system, which in some cases is too large even to be stored.

A first obvious generalization of the observational criterion is to specify that some signals are regarded as invisible too, i.e. are regarded as `eps`. The corresponding reduction function in AUTO is `visible`, which takes as a parameter the list of visible signals. For instance,

```
@ set V=visible(Orgy, {think1,eat1});
```

computes the automaton pictured in figure 6, where we only regard the actions of the philosopher number 1 in the whole system.

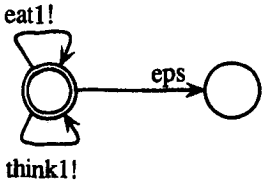


Figure 6: V.

Recall another criterion we have seen before, **exclusion**, which does not transform actions but only considers some of them, those which satisfy the required exclusion between certain signals.

6.3 Avoiding deadlock: the seats.

Having one more fork than people gives a deadlock-free system. To achieve this, we can add three seats "around the table", so that a philosopher can only take a fork when he is seated. The heap of seats is equivalent to the sequential butler usually introduced ([Hoa78]). Figure 7 shows us the behaviour of a new philosopher: when a philosopher sits down, he simultaneously takes his first fork. When he stands up, he drops his second fork. Moreover a philosopher must now eat once he grabs his two forks. This will allow us to test for possible starvation. A seat has the same behaviour as a fork, up to a renaming.

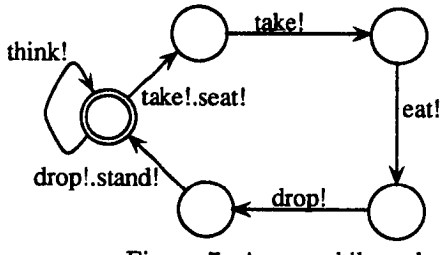


Figure 7: A new philosopher.

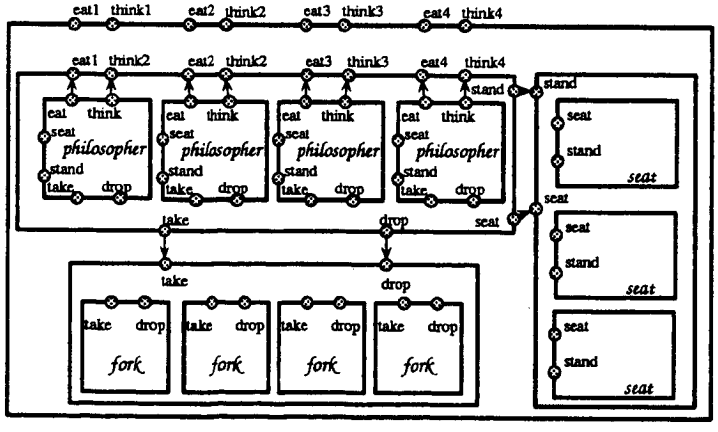


Figure 8: The assembly.

Figure 8 shows the network built from a bench (juxtaposed seats), the heap of forks and the philosophers. Figure 9 displays the result of the following commands:

```
@ set assembly=exclusion(assembly, {{think1,think2,think3,think4,
```

```
                                    eat1,eat2,eat3,eat4}});
@ set Assembly=obs assembly;
```

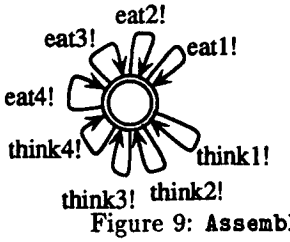


Figure 9: **Assembly**.

It is clear that there is no deadlock in this reduced system. But this solution does not guarantee that each philosopher will eventually eat: it is not fair. We now present a method using a more discriminating observation criterion that may enable the user to verify such a property.

6.4 Abstraction criteria.

An *abstraction criterion* is a set of abstract actions. Each *abstract action* is a set of sequences of actions, that is a formal language. Then a state q of an automaton performs an abstract action a, in notation $q \stackrel{a}{\Longrightarrow} q'$, when q is able to perform a sequence of actions representing a (that is belonging to a), while reaching q'. The notion of bisimulation relative to an abstraction criterion is defined in the same manner as the strong equivalence (cf [Bou85b]), that is:

$$\begin{array}{ccccc} p_1 & - & \mathcal{R} & - & p_2 \\ {\scriptstyle a}\Big\| & & & & \Big\|{\scriptstyle a} \\ \Big\downarrow & & & & \Big\downarrow \\ q_1 & - & \mathcal{R} & - & q_2 \end{array}$$

The coarsest bisimulation relative to a criterion is again an equivalence, inducing a quotient transition system, where the actions are abstract ones. Since we want to compute such a quotient, we must have an effective notion of abstraction criterion. In AUTO, the criteria are finite lists of regular expressions on the set of actions. We can enter a criterion using the command **parse-criterion** and then reduce an automaton using the function **quotient**. For instance, we show that the previous system **Assembly** is not fair i.e. a philosopher may starve while the others are eating. Let us consider the following abstraction criterion **STARVATION4** containing only one abstract action **STARVE4**:

```
@ parse-criterion STARVATION4=
  STARVE4=eps * : (eat1! + eat2! + eat3!) : eps * ;
```

We ask AUTO to compute

```
@ set Starvation4=quotient(assembly, STARVATION4);
```

We want to show that an infinite sequence of **STARVE4** is possible. It suffices then to check this on the smallest automaton that recognizes the same language as **Starvation4**. The corresponding function of AUTO is **trace**. It computes the minimal deterministic automaton for the language of traces. This function has the same syntax as **visible**, the signals given as arguments are considered as non empty actions. Figure 10 shows the result of the following command:

```
@ set unfair-starve4=trace(Starvation4,{STARVE4});
```

This shows that there is a possible infinite loop while a philosopher is endlessly deprived of spaghetti.

Figure 10: **unfair-starve4**

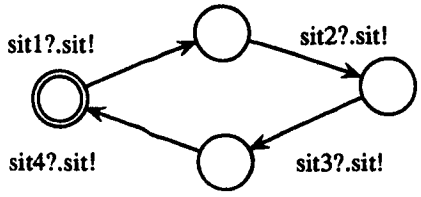


Figure 11: The scheduler.

6.5 Avoiding Starvation: the scheduler.

In order to give a solution avoiding starvation, we add another piece to our network, a scheduler compelling the philosophers to take seats in order. This scheduler is pictured in figure 11. The whole system is pictured in figure 12.

We can now require AUTO to perform some verifications:

- at first, let us compute the global system reduced by observational equivalence, under the constraint that the philosophers do not sit down while another is just standing up.

```
@ set global=obs exclusion(ACADEMY,
@                          {{sit1,sit2,sit3,sit4,stand_up}});
```

This system has 144 states and 828 transitions, so we can't conclude on it.

- let us verify there is no deadlock:

```
@ set res1=visible(global,{eat1});
```

The result is the same as that pictured in figure 9.

- let us verify the absence of starvation:

```
@ set fair-starve4=trace(quotient(global,STARVATION4),{STARVE4});
```

The result pictured in figure 13 shows us that one cannot deprive one philosopher of eating without creating a deadlock. More precisely, in this case, after the sixth deprivation, there is a deadlock.

Using **explore**, we can try to analyze this deadlock:

```
@ set fair4=quotient(global,STARVATION4});
fair4 : Automaton

@ explore fair4;
State 2
think-think-think-think-sit1-free-free-free-free-free-free-free
-- STARVE4 --> 1 think-take-take-drop-sit4-inuse-inuse-inuse-inuse-inuse-inuse-inuse
               3 think-take-eat-stand-sit4-inuse-inuse-inuse-inuse-inuse-inuse-inuse
               4 take-think-take-drop-sit4-inuse-inuse-inuse-inuse-inuse-inuse-inuse
               9 take-eat-think-take-sit1-inuse-inuse-inuse-inuse-inuse-inuse-inuse
               12 drop-think-think-take-sit4-inuse-inuse-free-inuse-inuse-free-inuse
# 9
State 9
```

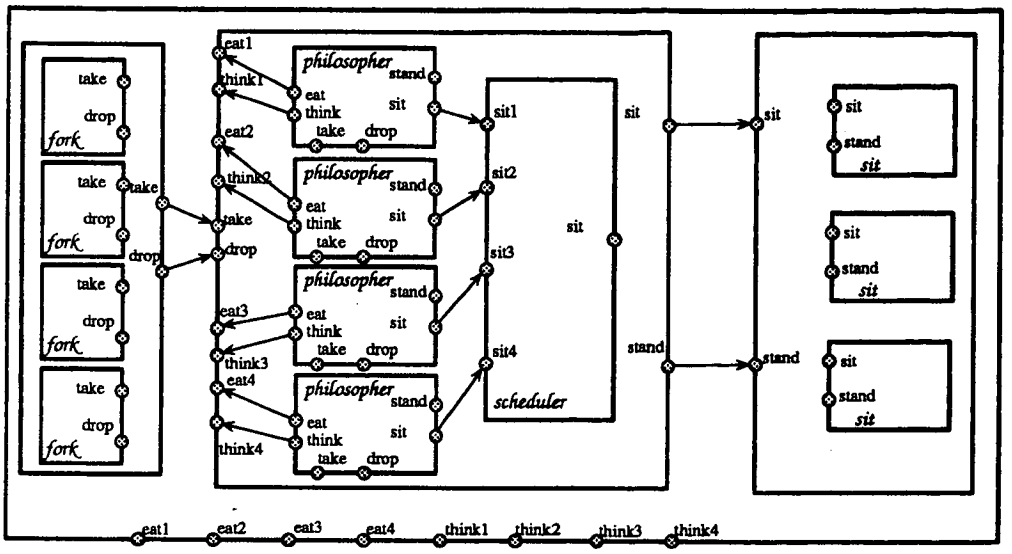


Figure 12: The fair academy.

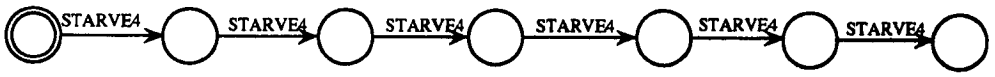


Figure 13: `fair-starve4`

```
take-eat-think-take-sit1-inuse-inuse-inuse-inuse-inuse-inuse-inuse
-- STARVE4 --> 5 take-think-stand-eat-sit1-inuse-inuse-inuse-inuse-inuse-inuse-inuse
# 5
State 5
take-think-stand-eat-sit1-inuse-inuse-inuse-inuse-inuse-inuse-inuse
-- STARVE4 --> 6 think-eat-think-eat-sit2-inuse-inuse-inuse-inuse-inuse-free-inuse
              12 drop-think-think-take-sit4-inuse-inuse-free-inuse-inuse-free-inuse
# 6
State 6
think-eat-think-eat-sit2-inuse-inuse-inuse-inuse-inuse-free-inuse
-- STARVE4 --> 7 take-think-think-take-sit3-free-inuse-free-inuse-inuse-free-inuse
              12 drop-think-think-take-sit4-inuse-inuse-free-inuse-inuse-free-inuse
# 7
State 7
take-think-think-take-sit3-free-inuse-free-inuse-inuse-free-inuse
-- STARVE4 --> 11 eat-think-think-eat-sit4-inuse-inuse-inuse-inuse-inuse-free-inuse
# 11
State 11
eat-think-think-eat-sit4-inuse-inuse-inuse-inuse-inuse-free-inuse
-- STARVE4 --> 12 drop-think-think-take-sit4-inuse-inuse-free-inuse-inuse-free-inuse
# 12
State 12
drop-think-think-take-sit4-inuse-inuse-free-inuse-inuse-free-inuse
     deadlock
# .
```

Analyzing the structure of each state name, we can see that the three first philosophers can eat. Then the fourth one sits down and takes a fork. The scheduler can authorize the three first philosophers to sit down before it blocks because the fourth one does not ask for a seat. So after each one has completed eating, the system is blocked

because the fourth philosopher is. As the reader can see, this analysis is rather difficult because the structure of the state names depends of the ordering of the automata of the network done by AUTOGRAPH. This is the main reason to study a graphical representation of the results obtained with AUTO, using AUTOGRAPH and the initial designs of a network.

7 Conclusion.

The approach presented in this paper allows verification of distributed systems by computing reductions. These reductions are parameterized by criteria choosen by the user, reflecting the aspect he wants to put in evidence. Some classical examples of distributed algorithms verification (communication protocols, mutual exclusion, termination detection, election) are covered in [Ver87] using AUTO. The largest case treated so far is that of a protocol due to Stenning [Ste76] where we deal with automata which average 10 000 states and 100 000 transitions.

References

[BK84] J. A. Bergstra and J. W. Klop.
A complete inference system for regular processes with silent moves.
Technical Report Report CS-R8420, CWI AMSTERDAM, 1984.

[Bou85a] G. Boudol.
Algèbre de processus et vérification.
In *Actes du Premier Colloque C^3*, 1985.

[Bou85b] G. Boudol.
Notes on algebraic calculi of processes.
In K. Apt Editor, editor, *Logics and Models for Concurrent Systems*, Springer-Verlag, 1985.

[BS80] G. Bochmann and C. Sunshine.
Formal methods in communication protocol design.
IEEE TRANSACTIONS ON COMMUNICATIONS, 30(4), 1980.

[BT82] G. Berthelot and R. Terrat.
Petri nets theory for the correctness of protocols.
IEEE TRANSACTIONS ON COMMUNICATIONS, 30(12), 1982.

[C*85] E. Clarke et al.
Using temporal logic for automatic verification of finte state systems.
In K. Apt Editor, editor, *Logics and Models for Concurrent Systems*, Springer-Verlag, 1985.

[Hai82] B. Hailpern.
Verifying concurrent processes using temporal logic.
Lectures Notes in Computer Science, 129, 1982.

[HM85] M. Hennessy and R. Milner.
Algebraic laws for nondeterminism and concurrency.
Journal of the ACM, 32:137–161, 1985.

[Hoa78] C.A.R. Hoare.
Communicating sequential processes.
Communication of the ACM, 21:666–676, 1978.

[Lip75] R. Lipton.
Reduction: a method of proving properties on parallel programs.
Communication of the ACM, 18(12), 1975.

[Mil79] R. Milner.
Flowgraphs and flow algebras.
Journal of the ACM, 26:794–818, 1979.

[Mil80] R. Milner.
A Calculus of Communicating Systems.

Volume 92 of *Lectures Notes in Computer Science*, Springer-Verlag, 1980.

[Mil83] R. Milner.
Calculi for synchrony and asynchrony.
Theoretical Computer Science, 267–310, 1983.

[Mil84] R. Milner.
A complete inference system for a class of regular behaviour.
Journal of Computer and Systems Sciences, 28:439–466, 1984.

[Mil86] R. Milner.
A Complete Axiomatisation for Observational Congruence of Finite-state Behaviours.
Technical Report ECS-LFCS-86-8, LFCS University of Edinburgh, 1986.

[Par81] D. Park.
Concurrency and automata on infinite sequences.
Lectures Notes in Computer Science, 104:167–183, 1981.

[Pnu85] A. Pnueli.
In transition from global to modular temporal reasoning about programs.
In K. Apt Editor, editor, *Logics and Models for Concurrent Systems*, Springer-Verlag, 1985.

[Sif84] J. Sifakis.
Properties preserving homomorphisms of transition systems.
Lectures Notes in Computer Science, 164:458–473, 1984.

[Ste76] N. V. Stenning.
A data transfert protocol.
Computer Networks, 1:99–110, 1976.

[Ver87] D. Vergamini.
Vérification de réseaux d'automates finis par équivalences observationnelles: le systeme AUTO.
Thèse de doctorat Sciences, Université de Nice, 1987.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique