Avoiding State Explosion for Distributed Systems with Timestamps

Fabrice Derepas† Paul Gastin* David Plainfossé †

fderepas@nortelnetworks.com Paul.Gastin@liafa.jussieu.fr plaindav@nortelnetworks.com

Abstract

This paper describes a reduction technique which is very useful against the state explosion problem which occurs when model checking many distributed systems. Timestamps are often used to keep track of the relative order of events. They are usually implemented with very large counters and therefore they generate state explosion. The aim of this paper is to present a very efficient reduction of the state space generated by a model checker when using timestamps. The basic idea is to map the timestamps values to the smallest possible range. This is done dynamically and on the fly by adding to the model checker a call to a reduction function after each newly generated state. Our reduction works for model checkers using explicit state enumeration and does not require any change in the model. Our method has been applied to an industrial example and the reduction obtained was spectacular.

Keywords

Formal verification, State explosion problem, Timestamp, Distributed systems.

I. Introduction

The current trend in software design is to use off the shelf products and reusable components. This is often the case in complex distributed systems or middleware platforms where several different softwares interact. The behaviour of each actor is known, but there is a need to validate the overall system behaviour. We propose a useful reduction for these systems.

The need for this reduction was raised by the verification of an industrial protocol. Using a natural modelisation of our system, we were not able to use a model checker due to the well-known state explosion problem. Then, we have implemented the reduction presented in this paper and thanks to the huge reduction of the state space, we were able to verify our system using the model checker Spin.

A. Related work

Distributed systems are difficult to verify using model checkers due to the well known state explosion problem. For n independent components, with S states and T transitions the number of states and transitions of the system may be as large as $S_n = S^n$ and $T_n = nTS^{n-1}$.

To reduce the state explosion, a number of formal methods and tools have been applied including SDL and Lotos [BB87]. However, many attempts end up compromising either

† Nortel Networks 1, Place des frères Montgolfier, BP 50 MS BA/53 78928 Yvelines Cedex 09, France.

* LIAFA, UMR 7089 case 7014 Université Paris 7 2, place Jussieu F-75251 Paris Cedex 05, France.

the level of abstraction [PR98], the number of processes in a network [Pec99], [GM97] or complete exploration of the state space [CM98].

In Spin [Hol97], partial oder reductions were implemented and used with a great success [God96], [Pel96]. In Murphi, symmetry was exploited thanks to a special data structure named scalarset [ID96] resulting again in a substantial reduction in the sate space generated. But these techniques have no effect on the state explosion generated by the very large range required by timestamps in a lot of protocols. This is why we have developped a new kind of reduction. Note that timestamps cannot be implemented using a scalarset since we need to compare their values.

Another way to validate a protocol is to use a proof checker [PPG+96], [KFL98]. Systems can be modeled using I/O automata [LT89], and then a manual proof is performed for these automata. This proof can then be checked using a theorem prover like Larch [GG91], but this method is highly time-consuming for both manual proof and verification with the theorem prover. In [PPG+96] a protocol using timestamps is verified this way.

B. Contributions of this paper

This paper is based on a case study of a real time distributed algorithm used in a telecommunication platform. Our protocol uses timestamps to keep track of the relative order of messages. The easiest way to implement timestamp variables is to use unbounded counters, or in practice, 32 bits or 64 bits counters. Indeed, this results in the state explosion problem when trying to verify our protocol with a model checker using explicit state enumeration. Actually, there are only a small number of different values of timestamps simultaneously stored in the variables of the system. Using the fact that we are only interested in the relative order of the timestamps, we have implemented a function which changes the timestamps contained in each newly generated state so that the smallest possible range for timestamps is used.

Our technique does not require any change of the model. Instead, it adds a call to our function in the model checker itself. The result is a huge reduction of the state space generated by the model checker. To implement this reduction, a library of approximately a hundred lines of C code has been linked with the existing model checker source code.

This work can be useful to anyone willing to fine tune model checkers targeted at validating distributed systems when a state explosion problem occur.

Some general issues regarding distributed systems are presented in section II. Section III details our reduction on timestamps.

II. Distributed systems

This section details issues in which we are interested regarding distributed systems.

A. Three-tier like model

In a distributed system we often have several actors which communicate only by exchanging messages. Here is a classification of their roles. There are three types of actors:

- allocator or server (which grants the right to use a resource),
- user or client (which takes decisions to perform operations on resources),

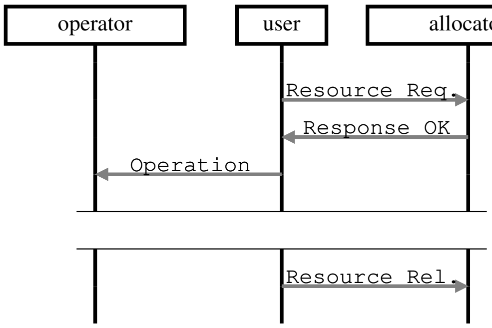

Fig. 1. Simple three tier model

- and operator (which performs operations).

A simple sketch is shown on figure 1. The user sends a request for a resource. The resource is then given to the user which can ask the operator to perform operations on the resource. After some time the user releases the resource and informs the allocator.

Our protocol uses this notion at different level simultaneously. A single actor might have several different roles (server, client or operator) regarding the same resource. It might also have different roles for different resources. Since we use it in a mobile environment, resources are allowed to move from one actor to the other, e.g. when a mobile subscriber moves from one cell to another. In the protocol we want to verify, we have approximatively 100 actors in our network, running on 16 hosts.

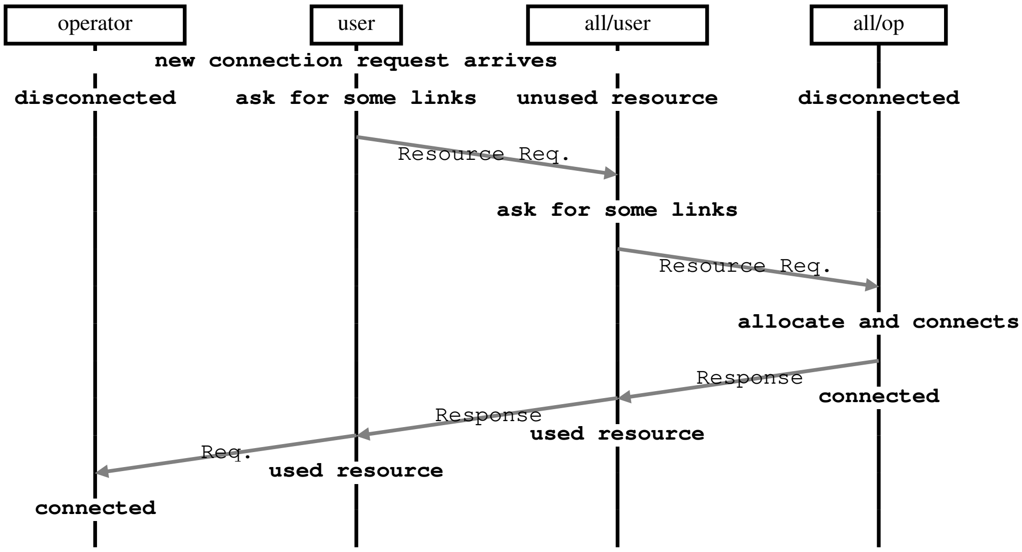

Fig. 2. Example of communication

Here is an example: when a call starts several links - resources - are allocated to some users. Some entities managing call processing request the use of some links. Once they are granted the right to use this link, they can send connection requests. This is shown on figure 2. The entity on the right is both an allocator and an operator because it both allocates and connects the communication link, the previous one is both a user (with regard to all/op) and an allocator (with regard to user). As a mobile subscriber moves from one cell to the other, connection requests must be sent accordingly.

Several systems can be described using these notions. In particular each network transaction such as object registration, location, and activation, which are part of CORBA (Common Object Request Broker Architecture) [OMG99].

B. Timestamps usage

We are in a system which is fault tolerant. Each application is replicated upon an active/passive scheme [GS96]. In nominal mode only the active works, and regularly sends messages to update its passive. After a fault, the passive switches to active. Before resuming its nominal activities, it must be resynchronized with other active entities, because some update messages might have been lost, or simply because they are almost never up to date with the rest of the system.

In order to keep track of the relative order of messages, our protocol uses timestamps. Each message is sent along with a timestamp. Timestamps allow to determine whether or not a request is valid. For instance if an operator O has a resource with timestamp t, and receives a release request with timestamp t' such that t is more recent than t' then the resource will not be released. Timestamps are also crucial for the resynchronization phase.

We have several clocks in the system, at least one per allocator (or server). When granting the use of a resource to a client, the allocator simultaneously sends the current value of its clock as a timestamp. In order to resolve potential conflicts which may occur after a fault in the system, the user must keep this timestamp until the resource is released. Therefore, for each clock in the system, several timestamps of past values of the clock are kept in various locations.

III. Abstraction on timestamps

Clocks and timestamps are often used in distributed systems. They provide a way to record causal relationships between events [RST91], [BSS91]. Systems which use timestamps [PPG$^+$96] often lead to state explosion. More precisely, the easiest way to implement a local clock is to use a counter which is large enough (e.g. 32 or 64 bits). This causes a major problem for automatic model checking using explicit state enumeration because the number of states of the system is then much too large. Since there are often a rather small number of values (timestamps) simultaneously present in the variables of the system, one would like to take advantage of this fact to reduce the number of states reached in the analysis of the model. This is precisely the aim of this section.

A classical abstraction to tackle the state explosion is to assume a relatively small bound on the counter, but this is not satisfactory in most cases. If the bound is small enough to avoid state explosion then it usually yields to counter overflows and the actual system is not faithfully described by the model.

If one knows a bound on the number of values simultaneously present for a clock, one may use sophisticated algorithms to keep the number of timestamps bounded (see e.g. [IL87], [DS97]). But again this is not a solution to our problem. First because it is not used in practice and therefore one should not use this in the model, and second because the bound needed by these algorithms is still much too large [CS93].

Our technique is very simple but proved to be highly efficient. It is designed to work with a model checker using explicit state vectors. The idea is to link a small library with the model checker whose aim is to reduce the timestamps contained in each newly generated state vector to a smaller range. This allows to dynamically use the smallest possible range for the timestamps. Also, our method does not require any change in the model, which is a major advantage.

A. Timestamps in state vector

The state vector of the system is the vector composed of all the parameters which describe the global state of the system: automaton states, variable values, message queue contents. Figure 3 shows a state vector with a single timestamp t_1 associated with some clock c. Then another timestamp t_2 for the same clock c appears in the system, it can be in some message queue or some local variable. These two timestamps can be different since they reflect the value of the clock c at different moments.

	t_1			
	t_1		t_2	

Fig. 3. A timestamp appearing

Actually in our system, each actor has its own clock, and we have several families of timestamps as shown on figure 4. For instance two users have timestamps t_1 and t_2 from a clock c in an allocator, and two other have u_1 and u_2 from another allocator. And for instance an operator has a timestamp v_1 from a user.

	t_1	v_1		u_1	t_2	u_2	

Fig. 4. Several families of timestamps

B. Principle

As explained above, implementing timestamps as a counter over bytes results in a huge number of global states for the system. We propose to map these timestamps to a smaller domain. This is done directly at the level of the model checker. More precisely, since we are only interested in the relative values of timestamps attached to some clock x, we can map n timestamps $t_1, ..., t_n$ to the interval $0, \dots, n-1$. Timestamps may also disappear leaving holes in the integer range $0, \dots, n-1$. We then have to map these values so that they again form an integer interval. For instance if we have timestamps 0, 1, 2, 2, 3, 4 and during the next step the values 1 and 3 disappear and 8 is inserted then we get 0, 1, 1, 2, 3.

Using suitable data structures, we will show how to implement these operations with a complexity of $n \log n$, n being the number of timestamps in the system. The result is that for each clock, all of its timestamps are consecutive integers. This dramatically reduces the state space.

C. Formal description

The way we use timestamps is very important. For timestamps t_i and t_j, we do not need to check properties such as $t_i < 4$ or perform operation such as $t_i := t_j^2$, or verify a property like "sometimes in the future t_i will be greater than 5".

We only perform the following operations: $t_i := t_j$ (set to an old value), $t_i := 0$ (reset to the original value) or $t_i := \text{newValue}$ (get a new value for this timestamp, the new value is positive and stricly greater than all other existing values). And we can only perform the following tests $t_i \sharp t_j$ or $t_i \sharp 0$ where $\sharp$ is one of $<, >, \leq, \geq, =$.

In order to give formal results, we assume that all timestamps are kept into variables and we denote by V the set of those variables. All other variables that the system may use are abstracted in the state of the automaton. This leads to the following definition.

Definition 1. *An automaton $\mathcal{A}$ is a tuple (Q, S, V, T), where:*
- *Q is the set of states,*
- *$S \subseteq Q$ is the set of initial states,*
- *V is the set of integer valued variables holding timestamps, they are initialized to 0,*
- *T is the set of transitions of our system. A transition is a quadruple (q, g, a, q') where p and q are states, g is a guard which is a boolean combination of atomic conditions such as $x \sharp 0$ or $x \sharp y$ with $x, y \in V$ and $\sharp \in \{=, <, >, \leq, \geq\}$, and a is an action which is a sequence or atomic actions such as $x := 0$, $x := y$ or $x := \text{newValue}$.*

A concrete state of the automaton $\mathcal{A}$ is a pair $(q, \sigma) \in Q \times \mathbb{N}^V$, where σ gives the value of each variable in V. The concrete initial states are $S \times \{0\}^V$. A transition (q, g, a, q') is enabled in state (q, σ) if $\sigma \models g$ (the guard is true) and when fired the resulting state is (q', σ') where σ' is obtained by applying the action a to σ, denoted by $\sigma' = a(\sigma)$.

An execution of $\mathcal{A}$ is a sequence $(q_0, \sigma_0), (g_1, a_1), (q_1, \sigma_1), \ldots$ where for all i, $\sigma_i \models g_{i+1}$, $\sigma_{i+1} = a_{i+1}(\sigma_i)$ and $(q_i, g_{i+1}, a_{i+1}, q_{i+1})$ is a transition of $\mathcal{A}$.

In order to define the reduced automaton, we introduce the reduction $\rho : \mathbb{N}^V \to \mathbb{N}^V$ mapping 0 to 0, positive integers to positive integers and such that for all $\sigma \in \mathbb{N}^V$ we have
- $\rho \circ \sigma(V) \cup \{0\}$ is an initial interval of $\mathbb{N}$,
- for all $x, y \in V$, $\sigma(x) \leq \sigma(y)$ if and only if $\rho \circ \sigma(x) \leq \rho \circ \sigma(y)$.

Note that the reduction ρ is idempotent.

The reduced automaton $\mathcal{A}' = \rho(\mathcal{A})$ is then defined as the quadruple (Q, S, V, T') where $T' = \{(q, g, a; \rho, q') \mid (q, g, a, q') \in T\}$. As above, the transition $(q, g, a; \rho, q')$ is enabled in state (q, σ) if $\sigma \models g$ and applying the transition results in the state $(q', \rho(a(\sigma)))$.

The basic result is that ρ defines a bisimulation between the automata $\mathcal{A}$ and $\mathcal{A}'$. In order to prove this, we need the following two lemmas.

Lemma 1. *Let g be a guard and $\sigma \in \mathbb{N}^V$ a valuation of variables. Then, $\sigma \models g$ if and only if $\rho \circ \sigma \models g$.*

Proof. Let $\sigma \in \mathbb{N}^V$ a valuation of variables and $x \sharp y$ be an atomic guard. Then,

$$\sigma \models x \sharp y \quad \longleftrightarrow \quad \sigma(x) \sharp \sigma(y) \quad \longleftrightarrow \quad \rho \circ \sigma(x) \sharp \rho \circ \sigma(y) \quad \longleftrightarrow \quad \rho \circ \sigma \models x \sharp y.$$

Similarly, $\sigma \models x \sharp 0$ if and only if $\rho \circ \sigma \models x \sharp 0$ since the reduction ρ maps 0 to 0, and positive integers to positive integers.

Now we can easily conclude using the fact that an arbitrary guard is a boolean combination of atomic guards. $\square$

Lemma 2. *Let a be an action and $\sigma \in \mathbb{N}^V$ a valuation of variables. Then,*

$$(a;\rho)(\rho \circ \sigma) = \rho(a(\rho \circ \sigma))) = \rho(a(\sigma)).$$

Proof. We first introduce some notations. Let $\sigma_1 = \rho \circ \sigma$, $\sigma' = a(\sigma)$ and $\sigma_1' = a(\sigma_1)$. We have to show that $\rho \circ \sigma' = \rho \circ \sigma_1'$, that is that the following diagram commutes.

$$\begin{tikzcd}
\sigma \arrow[r, "a"] \arrow[d, "\rho"'] & \sigma' = a(\sigma) \arrow[d, "\rho"] \\
\sigma_1 = \rho \circ \sigma \arrow[r, "a;\rho"] & \rho \circ \sigma_1' = \rho \circ \sigma'
\end{tikzcd}$$

Since an arbitrary action is just a sequence of atomic actions, we can easily conclude by induction once we have shown the lemma for atomic actions.

Assume that a is one of the atomic action $x := 0$, $x := y$ or $x := \mathrm{newValue}$. We first show that $\rho \circ \sigma'(z) = \rho \circ \sigma_1'(z)$ for all $z \neq x$. There are two cases.

• If $\sigma(x') = \sigma(x)$ for some $x' \neq x$ (including $x' = y$ when a is the atomic action $x := y$). Then, we also have $\sigma_1(x') = \sigma_1(x)$ for the same x'. We deduce that for all $z \neq x$ it holds

$$\rho \circ \sigma'(z) = \rho \circ \sigma(z) = \rho \circ \rho \circ \sigma(z) = \rho \circ \sigma_1(z) = \rho \circ \sigma_1'(z).$$

• If $\sigma(x') \neq \sigma(x)$ for all $x' \neq x$. Then, $\sigma_1(x') \neq \sigma_1(x)$ for all $x' \neq x$ and we deduce that for all $z \neq x$ it holds

– If $\sigma(z) < \sigma(x)$ then also $\sigma_1(z) < \sigma_1(x)$ and we have

$$\rho \circ \sigma'(z) = \rho \circ \sigma(z) = \rho \circ \rho \circ \sigma(z) = \rho \circ \sigma_1(z) = \rho \circ \sigma_1'(z).$$

– If $\sigma(z) > \sigma(x)$ then also $\sigma_1(z) > \sigma_1(x)$ and we have

$$\rho \circ \sigma'(z) = \rho \circ \sigma(z) - 1 = \rho \circ \rho \circ \sigma(z) - 1 = \rho \circ \sigma_1(z) - 1 = \rho \circ \sigma_1'(z).$$

It remains to show that $\rho \circ \sigma'(x) = \rho \circ \sigma_1'(x)$. The proof depends on the specific atomic action.

• Assume that a is the atomic action $x := 0$. Then, $\sigma'(x) = 0$ and $\sigma_1'(x) = 0$. The result follows since ρ maps 0 to 0.

• Assume that a is the atomic action $x := y$. If $x = y$ then the action a has no effect and we can conclude since ρ is idempotent. Hence we can assume that $y \neq x$ and use the result above. We have

$$\rho \circ \sigma'(x) = \rho \circ \sigma'(y) = \rho \circ \sigma_1'(y) = \rho \circ \sigma_1'(x).$$

- Finally, assume that a is the atomic action $x :=$ newValue. We have

$$\rho \circ \sigma'(x) = 1 + \max_{z \neq x} \rho \circ \sigma'(z) = 1 + \max_{z \neq x} \rho \circ \sigma_1'(z) = \rho \circ \sigma_1'(x).$$

□

Proposition 1. *The reduction ρ defines a bisimulation between the automata $\mathcal{A}$ and $\mathcal{A}'$. More precisely,*
1. If $(q,\sigma) \xrightarrow{g,a} (q',\sigma')$ is a transition in $\mathcal{A}$, then, $(q,\rho\circ\sigma) \xrightarrow{g,a;\rho} (q',\rho\circ\sigma')$ is a transition in $\mathcal{A}'$.

$$\begin{tikzcd}
(q,\sigma) \arrow[r, "{(g,a)}"] \arrow[d, "\rho"'] & (q',\sigma') \arrow[d, "\rho"] \\
(q,\rho\circ\sigma) \arrow[r, "{(g,a;\rho)}"] & (q',\rho\circ\sigma')
\end{tikzcd}$$

2. If $(q,\sigma_1) \xrightarrow{g,a;\rho} (q',\sigma_1')$ is a transition in $\mathcal{A}'$, then $(q,\sigma) \xrightarrow{g,a} (q',\sigma')$ is a transition in $\mathcal{A}$ for some σ,σ' such that $\sigma_1 = \rho\circ\sigma$ and $\sigma_1' = \rho\circ\sigma'$.

$$\begin{tikzcd}
(q,\sigma_1) \arrow[r, "{(g,a;\rho)}"] & (q',\sigma_1') \\
(q,\sigma) \arrow[r, "{(g,a)}"] \arrow[u, "\rho"] & (q',\sigma') \arrow[u, "\rho"']
\end{tikzcd}$$

Proof. First, if $(q,\sigma) \xrightarrow{g,a} (q',\sigma')$ is a transition in $\mathcal{A}$ then $\sigma \models g$ and $\sigma' = a(\sigma)$. Using Lemmas 1 and 1 we deduce that $\rho\circ\sigma \models g$ and that $(a;\rho)(\rho\circ\sigma) = \rho(a(\sigma))$. Therefore, $(q,\rho\circ\sigma) \xrightarrow{g,a;\rho} (q',\rho\circ\sigma')$ is a transition in $\mathcal{A}'$.

Second, if $(q,\sigma_1) \xrightarrow{g,a;\rho} (q',\sigma_1')$ is a transition in $\mathcal{A}'$ then $\sigma_1 \models g$ and $\sigma_1' = (a;\rho)(\sigma_1)$. If in addition we have $\sigma_1 = \rho\circ\sigma$ then using Lemma 1 we deduce that $\sigma \models g$. Therefore, $(q,\sigma) \xrightarrow{g,a} (q',a(\sigma))$ is a transition in $\mathcal{A}$. We can conclude using Lemma 2 that $(a;\rho)(\rho\circ\sigma) = \rho(a(\sigma))$. □

From Proposition 1 we deduce immediately that

Proposition 2.
1. If the state (q,σ) is reachable in $\mathcal{A}$ then the state $(q,\rho\circ\sigma)$ is reachable in $\mathcal{A}'$.
2. If the state (q,σ') is reachable in $\mathcal{A}'$ then there exists a valuation $\sigma \in \mathbb{N}^V$ such that (q,σ) is reachable in $\mathcal{A}$ and $\sigma' = \rho\circ\sigma$.
3. If $(q_0,\sigma_0),(g_1,a_1),(q_1,\sigma_1),\ldots$ is an execution of $\mathcal{A}$ then $(q_0,\rho\circ\sigma_0),(g_1,a_1;\rho),(q_1,\rho\circ\sigma_1),\ldots$ is an execution of $\mathcal{A}'$.
4. If $(q_0,\sigma_0'),(g_1,a_1;\rho),(q_1,\sigma_1'),\ldots$ is an execution of $\mathcal{A}'$ and $\rho\circ\sigma_0' = \sigma_0'$ then, for all i there exists σ_i such that $\sigma_i' = \rho\circ\sigma_i$ and $(q_0,\sigma_0),(g_1,a_1),(q_1,\sigma_1),\ldots$ is an execution of $\mathcal{A}$.

Let us now pay attention to model checking for linear temporal logic (LTL [Pnu77]) or computation tree logic (CTL or CTL* [Eme90]).

Proposition 3. *Let* Prop *be a set of atomic propositions and let* $\ell : Q \to 2^{\mathrm{Prop}}$*. If* φ *is an LTL or a CTL or a CTL* formula build over the propositions of* Prop *and the atomic predicates* $x \sharp 0$ *and* $x \sharp y$ *where* $x, y \in V$ *and* $\sharp \in \{=, <, >, \leq, \geq\}$*. Then* $\mathcal{A}, \ell \models \varphi$ *if and only if* $\mathcal{A}', \ell \models \varphi$*.*

This proposition in another consequence of Proposition 1. The key point is that the bisimulation between $\mathcal{A}$ and $\mathcal{A}'$ defined by ρ preserves the atomic formulas. Indeed, let (q, σ) and $(q, \rho \circ \sigma)$ be two bisimilar states. The atomic propositions from Prop satisfied by the two states are given by $\ell(q)$. Using Lemma 1, we also deduce that the two states satisfy the same atomic predicates. The proposition follows then easily using classical results concerning bisimilar systems.

We could also define an equivalence on the concrete states of $\mathcal{A}$ by setting $(q, \sigma) \equiv (q, \sigma')$ if and only if $\rho \circ \sigma = \rho \circ \sigma'$. We could then show this equivalence is a bisimulation. The automaton $\mathcal{A}'$ is then isomorphic to the quotient of $\mathcal{A}$ by this equivalence.

D. Implementation

We are using the model checker Spin [Hol97] and its associated modeling language Promela. Using a small library in C-code of approximately a hundred lines we managed to achieve the timestamps reduction described in section III-C. The library is linked with the C code (named pan) generated by the Spin model checker. The outline of the treatment which is done at each step is given in figure 5.

If m is the state vector size, and n_i the number of timestamps for clock i then the complexity at each step is $O(m + \sum_i n_i \log(n_i))$. Let us denote by n the maximum number of timestamps in the system, then $\sum_i n_i \leq n$ and the complexity for each step is smaller than $O(m + n \log n)$.

```
Empty all l_i lists.
Read state vector:
    when a timestamp from process i
       is encountered add a pointer to
       it in l_i.
for all processes i
    sort the list l_i according to the
       value pointed to.
    read the list l_i and change values
       to have a consecutive range
       of integers.
endfor
```

Fig. 5. Timestamps mapping

This implementation is done simply by adding one line of code to the Spin model checker: a call to our function which performs the reduction of timestamps described in figure 5.

E. Experimental results

We are in a system which is fault tolerant. Each application is replicated upon an active/passive scheme [GS96]. In nominal mode only the active works, and regularly sends messages to update its passive. After a fault, the passive switches to active. Before resuming its nominal activities, it must be resynchronized with other active entities, because some update messages might have been lost. Figure 6 show the high level states of each application.

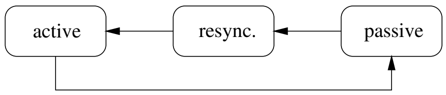

Fig. 6. A fault tolerant application

Figure 1 and 2 shows what happens when the application is in active or passive states. Figure 7 shows a little sketch of the resynchronization phase we are verifying. We performed the test with a system consisting of two allocators and two users after some faults which induce inconsistencies in the system. The current clock value in figure 7 is the $newValue$ function from III-C. Since each actor has its own family of timestamps (cf. figure 4), each actor has its own $newValue$ function. Only the code for an allocator is presented but it is the same for an operator.

Since an single entity can be both an allocator and a user for the same resource (e.g. the entity forwards the allocation of resource) both behaviour from figure 7 can take place in the same automaton.

In order to verify this protocol, we used Spin version 3.3.6 on a Sun Sparc Ultra 10 workstation running Solaris with 512Mb of RAM and a clock speed of 300 MHz.

Figure 8 presents some experimental results showing the difference between a naive implementation and the abstraction described in section III-B.

The first column *Ts Range* - timestamp range - shows how many different values are used for timestamps. As explained above, in order to avoid overlaps between timestamps, *Ts Range* should be large enough. Here we start at 5 because in our system we have at most 5 different timestamps which can be compared. Notice that this is not enough for the basic (without timestamp reduction) implementation because it would cause timestamp overlaps.

The second column shows the number of states, transitions and memory in megabytes used for the basic model. The third column shows results for the abstraction described in this section. N/A denotes the fact that the computation required more memory than was available. Using swap memory dramatically increases the time and has to be avoided in practice.

One can notice that timestamp reduction gives the same results regardless of the timestamp range. So if a timestamp stays a long time in the system there will be no problem

Resynchronization

allocator	***user***
for all users u send a *resync_req* to u with my identifier and current clock value. **endfor** **for** all *resync_rsp* received from u **if** timestamp in *resync_rsp* is more recent than $state[u]$. **then** update $state[u]$. **endfor**	**for** all allocators a send *resync_req* to a with my usage of resources owned by a and current clock value. **endfor**

Nominal state

allocator	***user***
When receiving *resync_req* from u **if** timestamp in *resync_rsp* is more recent than $state[u]$. **then** update $state[u]$.	When receiving *resync_req* from a Send *resync_rsp* with my usage of current resources owned by a and current clock value to a.

Fig. 7. Resynchronization for different actors

with the abstraction of section III-B, whereas the basic implementation will require a timestamp range which will be much to large to allow automatic verification.

This idea of modifying the state vector generated by the model checker in order to get a bisimilar transition system can be used in other systems using vector of timestamps, like the Fidge-Mattern protocol [Fid88], [Mat89]. Reduction in the number of states can be less significant for protocol which store a lot of causal information like the Raynal-Schiper-Toueg protocol [RST91].

F. Some results

This section gives two examples of some errors which were found in the protocol. This was possible due to the previous abstraction.

Let us consider a dialog between two actors A and B. If B switches its activity then messages sent by the old active entity and the new one may not be in FIFO order. Such an error is shown in figure 9: user sends a requests to obtain a resource. This requests is successful, and a Resource Response is sent. Unfortunately the allocator switches its activity (swacts), before its passive instance being updated.

A resynchronization request issued by the newly active allocator is read by the user. Since this is a third actor it may be treated before the response from the old active allocator. This leads to a system state where the allocator has an unused resource but this resource is used by the user.

We found a solution to this problem, by adding a timestamp verification when receiving

Ts range	basic	III-B	
5	3.3×10^6	7.5×10^4	st.
	4.9×10^6	1.1×10^5	tr.
	438	375	Mb.
6	5.9×10^6		st.
	8.7×10^6	*as above*	tr.
	488		Mb.
7	9.8×10^6		st.
	1.1×10^7	*as above*	tr.
	563		Mb.
128			st.
	N/A	*as above*	tr.
			Mb.

Fig. 8. Results for timestamps mapping

`Resource Rsp.`

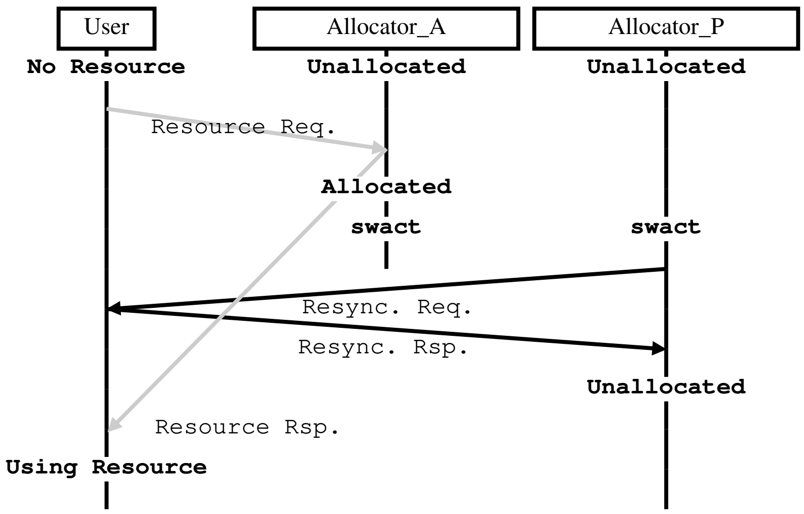

Fig. 9. Stream scheduling

Here is a more subtle error. A message sequence chart (MSC) is shown in figure 10. There are two levels of allocation: a pure allocator named *Allocator* on figure 10 allocates resources for another actor named *User/All*. Theses resources are then again allocated for actors *User1* or *User2* by *User/All*.

Let us have a closer look at the MSC: *User1* sends a message to *User/All* which is forwarded to *Allocator*. *User/All* swacts, forgets its state. Then *User/All* understands the Rsc. Ack. message from allocator as the response to the resource request message from *User2*.

In order to have a correct protocol we had to add an identifier (which was missing) to

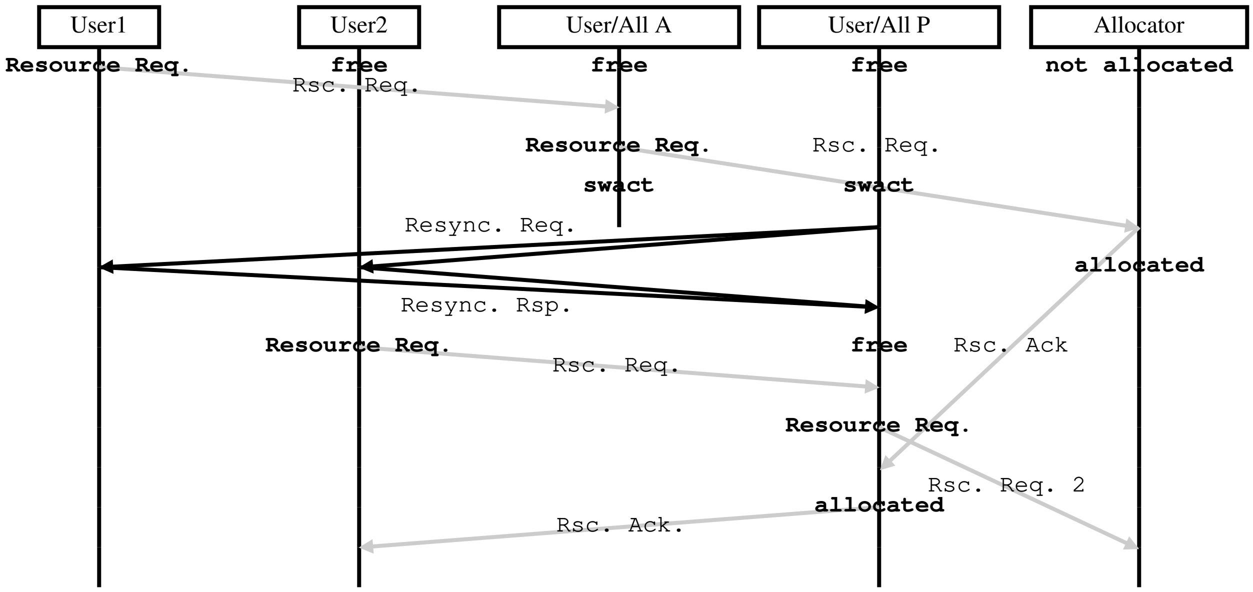

Fig. 10. Checking timestamps upon request response

the request response in order to check it corresponds to the desired request.

IV. Conclusion and future work

A. *What we have been doing*

We managed to reduce the combinatorial explosion by mapping timestamps onto a list of consecutive integers. This reduction enabled us to formally model check the untimed properties of an industrial protocol and to exhibit some unknown errors. Hence the work was very helpful for the designers of the protocol.

Our reduction technique can also be applied to other structures which memorize a small number of causal events.

Our method is efficient and simple. It does not require any change of the model. It can cope with some kind of infinite state systems. It is not tailored to any special kind of property to be checked but instead works for all CTL* formulae since it gives a reduced system which is bisimilar to the original one.

On the other hand the process is not automatic yet, and it seems difficult to use it with symbolic model checking.

B. *Future work*

Two main axes can be considered:

- The work described in this paper has been done manually. We think it would be interesting to add some extensions to the Promela language to describe distributed system, in a way that the reduction described in this paper could be performed automatically.
- Our method has been designed to work with model checker using an explicit state

enumeration. Though it does not seem clear how to extend this to symbolic model checking, this would be a very interesting track of research.

Also, we still have to model and check real time properties of our protocol. For instance: "how long will it take to switch activity between two applications", "how long will the first phase of the protocol last" and so on. We have not worked on this part yet.

V. Acknowledgments

We would like to thank Xavier Lasne designer of resynchronization protocol, for his continuous interests in our activities. We would like to thank Gawain Bolton for comments and suggestions on the paper.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language lotos. *Computer Networks and ISDN Systems*, pages 25–59, 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, pages 272–314, August 1991.

[CM98] M. Calder and A. Miller. Analysing a basic call protocol using Promela/Xspin. *Fourth International Spin Workshop*, Nov. 1998.

[CS93] R. Cori and E. Sopena. Some combinatorial aspects of time-stamp systems. *European Journal of Combinatorics*, pages 95–102, 1993.

[DS97] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, chapter 16*, pages 995–1072. Elsevier Publ. Co., Amsterdam, 1990.

[Fid88] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

[GG91] S. Garland and J. Guttag. A guide to lp, the larch prover. Technical Report 82, DEC Systems Research Center, 1991.

[GM97] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1-2):171–197, 1997.

[God96] P. Godefroid. *"Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996. ISBN 3-540-60761-7.

[GS96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. *Reliable Software Technologies - Ada Europe '96. Lecture Notes in Computer Science*, 1088, 1996.

[Hol97] G. J. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[ID96] C. Norris Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

[IL87] A. Israëli and M. Li. Bounded time-stamps. *Proc 28th IEE Symposium on Foundations of Computer Science*, pages 371–382, 1987.

[KFL98] R. Khazan, A. Fekete, and N. A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. *DISC '98(formerly WDAG) 12th International Symposium on DIStributed Computing*, 1998.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quaterly*, 2(3):219–246, september 1989.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[OMG99] OMG. Corba 2.3.1/IIOP specification, 1999. available from `www.omg.org`.

[Pec99] C. Pecheur. Advanced modelling and verification techniques applied to a cluster file system. *Proc. of the 14th IEEE International Conference on Automated Software Engineering ASE-99*, October 1999.

[Pel96] D. Peled. Partial-order reductions: model checking using representatives. In *Proc. of MFCS'96*, number 1113 in Lecture Notes in Computer Science, pages 93–112. Springer-Verlag, 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. *Proc 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[PPG+96] T. P. Petrov, A. Pogosyants, S. J. Garland, V. Luchangco, and N. A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools*, pages 29–44. Chapman & Hall, 1996.

[PR98] M. Plath and M. Ryan. Plug and play features. *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems*, 1998. IOS Press.

[RST91] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *IPL*, 39(6):343–350, 1991.