Avoiding State Explosion for Distributed Systems with Timestamps

Fabrice Derepas[1], Paul Gastin[2], and David Plainfossé[1]

[1] Nortel Networks, 1, Place des frères Montgolfier,
78928 Yvelines Cedex 09, France.
fderepas|plaindav@nortelnetworks.com
[2] LIAFA, UMR 7089 Université Paris 7, 2 place Jussieu,
F-75251 Paris Cedex 05, France.
Paul.Gastin@liafa.jussieu.fr

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.

1 Introduction

Now days there are more and mode critical systems which need to be verified. In this paper we focus on the use of timestamps to ensure causality between events. This kind of mechanism can be found in many distributed systems. We had to face this problem during the verification of an industrial protocol designed to resynchronize a telecommunication system after some fault.

Distributed systems are difficult to verify using model checkers due to the 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 [2]. However, many attempts end up compromising either the level of abstraction [3], the number of processes in a network [4, 5] or complete exploration of the state space [6].

In Spin, partial oder reductions were implemented and used with a great success [7, 8]. In Murφ, symmetry was exploited thanks to a special data structure named scalarset [9] 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. Note that timestamps cannot usually be implemented using a scalarset since we need to compare their values.

Another way to validate a protocol is to use a proof checker [10, 11]. Systems can be modeled using I/O automata [12], and then a manual proof is performed for these automata. This proof can then be checked using a theorem prover like Larch [13], but this method is highly time-consuming for both manual proof and verification with the theorem prover. In [10] a protocol using timestamps is verified this way.

Using a natural model of our system, we were not able to use a model checker due to the 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 [1].

More precisely, this paper is based on a case study of a real time fault tolerant distributed algorithm used in a telecommunication platform. Our protocol uses timestamps to keep track of the relative order of messages and to be able to get back to a coherent state after some fault. 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 with some causal dependencies.

In section 2 we detail the kind of system we are verifying, as well as the way we use timestamps. This enable to understand the reusability of our work. Section 3 details our reduction on timestamps.

2 High Availability and Distributed Systems

This section presents the application we are verifying. It is not our purpose to describe in details this application. We describe the general context, stressing those points which make the application complex. Then we describe in more details the part of the application we want to verify.

2.1 Managing Resources in a Distributed System

We are interested in the software managing a cellular phone network. When a call is established several links are reserved for the communication. These resources will be released afterwards. During these phases, several "users" ask some "allocators" for resources. Once they are granted the right to use them, they can perform operations, such as connection for a new incoming call for instance. We are in a distributed system and there is no global clock.

Automata for a user and an allocator are represented in figure 1. A question mark '?' denotes the fact that a message is received, and '!' denotes the sending of a message.

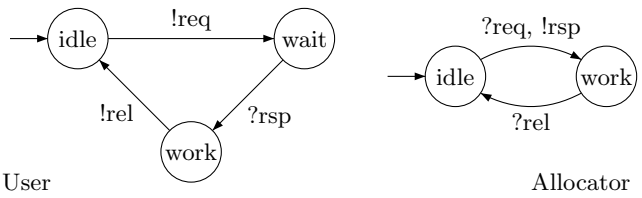


Fig. 1. Automata for user and allocator

The type of system previously described is named three tier systems. A simple execution is shown in figure 2. The user sends a request for a resource. The resource is then given to the user which can afterwards request the operator to perform operations on the resource. After some time the user releases the resource and informs the allocator.

This kind of model is very common. For instance on the world wide web, a browser is the user, the allocator is the web server, and transaction can lead to modification of a data base, which is the operator (in this case the `operation` request of figure 2 can be sent from the allocator instead of the user).

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 another, 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. This gives an idea of the description of our system.

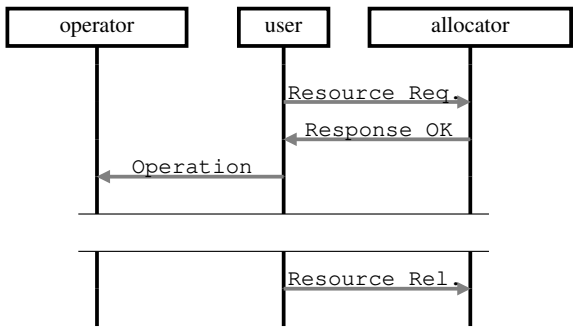


Fig. 2. Simple three tier model

2.2 Quality of Service and Fault Tolerance

There are some constraints regarding quality of service in our system. For instance a fault must be detected within three seconds. Recovery must then take at most two seconds. The total capability down time (average time during which the network does not work) is 0.4 minutes per year. There must not be more than 10^4 loss of calls for 10^9 hours of communication. These strong requirements of the system implies that some fault tolerance mechanisms have to be implemented. Redundancy is used for hardware components. We will focus on software aspects.

An active/passive replication scheme has been chosen [14]. In order to cope with real time constraints no group membership techniques have been implemented [15]. This means that every software application is duplicated. It has an active form which performs the regular work and a passive form which asynchronously receives messages to update its state. The passive form consumes much less resource than the active one. This allows to save hardware resources when we have a lot of applications running.

Fig. 3. A fault tolerant application

After a fault, the passive switches to active and starts from the latest state for which an update was received. This is called a swact, which stands for switch of activity. In this active/passive scheme, the passive process only knows the state of its active when the last update message was sent. Therefore a swact may cause some loss of information, typically the passive is not aware of the latest transactions. Before resuming its nominal activities, it must be resynchronized with other active entities. Figure 3 shows the high level states of each application.

In order to allow the resynchronization, timestamps are used to order the causality of events in our system. This is shown in figure 4. In this figure unspecified reception are dropped. And for both entities the passive form is:

Passive for $Alloc(a)$ or $User(u)$
when receiving $update(m)$ store m,
    goto Passive
when receiving *swact* from environment
    goto Resync.

The $newValue$ function in figure 4 denotes the creation of a new timestamps more recent than all existing ones. The response to the swact of an allocator can be in any of the following state for users $Idle$, $Wait$ or $Work$. The response to the swact of a user can be in the *Active* state of an allocator.

There are two kinds of properties we want to check:

- safety: current context of applications should not be corrupted after the resynchronization.
- liveness: unused resources should be released eventually.

3 Abstraction on Timestamps

Clocks and timestamps are often used in distributed systems. They provide a way to record causal relationships between events [16, 17]. Systems which use timestamps [10] 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. [18, 19]). 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 [20].

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.

Alloc(a)

Active
when receiving $req(u)$
  if $\exists r$ such that $busy[r] = False$
    $date[r] \leftarrow newValue$
    $user[u]!rsp(r, date[r])$
    $busy[r] \leftarrow True$
    update passive
  else
    $user[u]!rsp_nack$
  goto Active
when receiving $rel(r, d)$
  if $date[r] \leq d$
    $busy[r] \leftarrow False$
    update passive
  goto Active

Allocator resynchro.
For all r $busy[r] \leftarrow False$
For all $u \in [0, m-1]$
  $user[u]!resync_req(a)$
For all $u \in [0, m-1]$ receive
  $resync_rsp(r, d)$ or $resync_req(r, d)$
    if $\neg busy[r] \vee date[r] \leq d$
      $date[r] \leftarrow d$
      $busy[r] \leftarrow True$
  $resync_rsp_empty$ and do nothing.
update passive
goto Active

Response to user resynchro.
when receiving $resync_req(r, d)$ from u
  if $\neg busy[r]$
    $user[u]!resync_rsp_nack$
  else
    if $date[r] = d$
      $user[u]!resync_rsp_ack$
    else $user[u]!resync_rsp_nack$
      $busy[r] \leftarrow False$
      update passive

User (u)

Idle ($state \leftarrow idle$)
Environment send $a \in [0, n-1]$
$alloc[a]!req(u)$ goto Wait
Wait ($state \leftarrow wait$)
when receiving rsp_nack from a
  goto Idle
when receiving $rsp(r, d)$
  update passive
  goto Work
Work ($state \leftarrow work$)
Environment decides to finish the call.
send $alloc[a]!rel(r, d)$
update passive
goto Idle

Response to allocator resynchro.
when receiving $resync_req(a')$
  if $a = a' \wedge state = work$
    $alloc[a']!resync_rsp(r, d)$
    goto work
  else
    if $a \neq a$
      $alloc[a']!resync_rsp_empty$
      goto $state$
    else $alloc[a']!resync_rsp_empty$
      goto $idle$

User resynchronization
if $state = work$
  $alloc[a]!resync_req(r, d)$
  if receive $resync_rsp_ack$
    goto work
  if receive $resync_rsp_nack$
    update passive with an idle state
    goto idle
else
  goto idle

Fig. 4. Resynchronization for allocators and users

3.1 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 5 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.

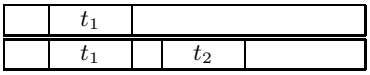


Fig. 5. A timestamp appearing

Actually in our system, each allocator has its own clock, and we have several families of timestamps as shown on figure 6. 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.

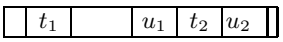


Fig. 6. Several families of timestamps

3.2 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 $0, \ldots, n-1$. Timestamps may also disappear leaving holes in the integer range $0, \ldots, n-1$. 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.

3.3 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 $t_i :=$ newValue (get a new value for this timestamp, the new value is positive and strictly 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 of atomic actions such as $x := 0$, $x := y$ or $x :=$ newValue.*

Examples previously shown in figure 1 and 4 can be encoded this way.

Definition 2. *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 $\rho : \mathbb{N}^V \to \{0, ..., |V|\}^V$. This corresponds to the remapping of timestamps. The implementation of function ρ will be described in figure 7. It maps 0 to 0, positive integers to positive integers and it is 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.

For instance the following sequence of timestamps $(1, 1, 3, 13, 13)$ will be mapped to $(1, 1, 2, 3, 3)$ by ρ. If 2 is removed, we have the sequence $(1, 1, 3, 3)$ which is mapped by ρ to $(1, 1, 2, 2)$.

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. Since ρ preserves the ordering we have:

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

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.

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.

$$\begin{array}{ccc} \sigma & \xrightarrow{\quad a \quad} & \sigma' = a(\sigma) \\ \Big\downarrow \rho & & \Big\downarrow \rho \\ \sigma_1 = \rho \circ \sigma & \xrightarrow{\quad a; \rho \quad} & \rho \circ \sigma'_1 = \rho \circ \sigma' \end{array}$$

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 $x :=$ 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 $\sigma(z) > \sigma(x) > 0$ then also $\sigma_1(z) > \sigma_1(x) > 0$ and we have

$$\rho \circ \sigma'(z) = \rho \circ \sigma(z) - 1 = \rho \circ \rho \circ \sigma(z) - 1 = \rho \circ \sigma_1(z) - 1 = \rho \circ \sigma'_1(z).$$

- If $\sigma(x) = 0$ then for all $z \neq x$, $\sigma(z) > 0$ and $\sigma_1(z) > 0$. It follows $\rho \circ \sigma'(z) = \rho \circ \sigma(z) = \rho \circ \rho \circ \sigma(z) = \rho \circ \sigma_1(z) = \rho \circ \sigma'_1(z)$.

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

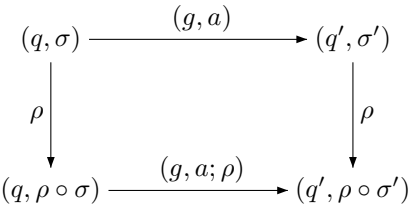
$$\rho \circ \sigma'(x) = \rho \circ \sigma'(y) = \rho \circ \sigma'_1(y) = \rho \circ \sigma'_1(x).$$

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

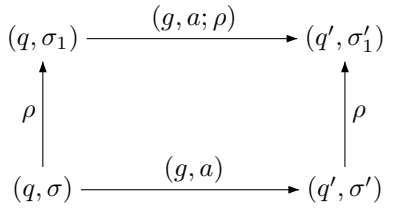
$$\rho \circ \sigma'(x) = 1 + \max_{z \neq x} \rho \circ \sigma'(z) = 1 + \max_{z \neq x} \rho \circ \sigma'_1(z) = \rho \circ \sigma'_1(x).$$

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 $\mathcal{A}'$.*



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 $\sigma'_1 = \rho \circ \sigma'$.*



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 2 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 [21]) or computation tree logic (CTL or CTL* [22]).

Proposition 3. *Let* Prop *be a set of atomic propositions and let $\ell : Q \to 2^{\mathrm{Prop}}$ be a mapping which associates with each state q the set of atomic propositions which are satisfied in state q. If φ is 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.

3.4 Implementation

We are using the model checker Spin 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 3.3. 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 7.

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. 7. 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 7.

3.5 Experimental Results

We performed the test with a system consisting of two allocators and two users as described in figure 4. 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 3.2.

The first column *timestamp range* shows how many different values are used for timestamps. As explained above, in order to avoid overlaps between timestamps, *timestamp 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

timestamp range	without reduction	with reduction	
5	3.3×10^6 4.9×10^6 438	7.5×10^4 1.1×10^5 375	states transitions Mb. of memory
6	5.9×10^6 8.7×10^6 488	*as above*	st. tr. Mb.
7	9.8×10^6 1.1×10^7 563	*as above*	st. tr. Mb.
128	N/A	*as above*	st. tr. Mb.

Fig. 8. Results for timestamps mapping

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 with the abstraction of section 3.2, whereas the basic implementation will require a timestamp range which will be much too 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 [23, 24]. Reduction in the number of states can be less significant for protocols which store a lot of causal information like the Raynal-Schiper-Toueg protocol [16].

3.6 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 from active to passive 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 has been 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 `Resource Rsp`.

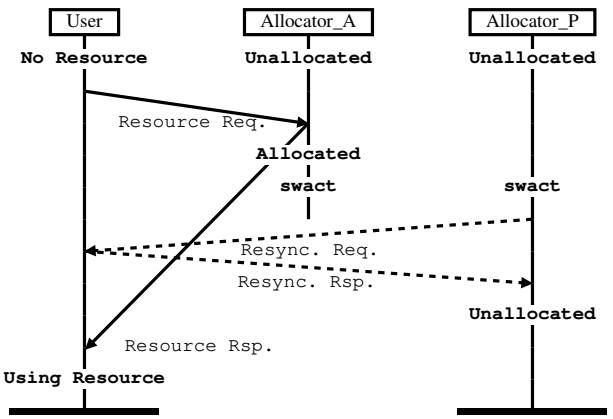


Fig. 9. Stream scheduling

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*. This enables the system to handle user mobility.

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

4 Conclusion and Future Work

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.

Two main axes can be considered for future work:

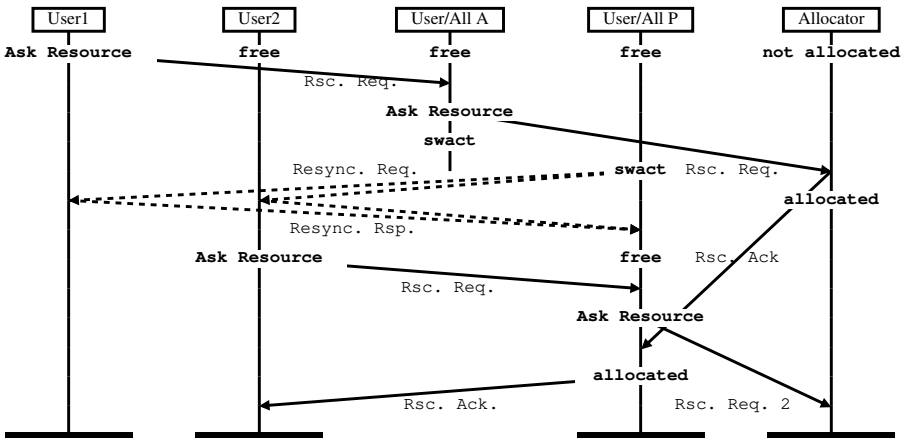


Fig. 10. Checking timestamps upon request response

- 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.

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

1. G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
2. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language lotos. *Computer Networks and ISDN Systems*, pages 25–59, 1987.
3. M. Plath and M. Ryan. Plug and Play Features. *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems*, 1998. IOS Press.
4. C. Pecheur. Advanced Modeling and Verification Techniques Applied to a Cluster File System. *Proc. of the 14th IEEE International Conference on Automated Software Engineering ASE-99*, October 1999.
5. 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.
6. M. Calder and A. Miller. Analyzing a Basic Call Protocol Using Promela/Xspin. *Fourth International Spin Workshop*, Nov. 1998.

7. 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.
8. 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.
9. C. Norris Ip and D. L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.
10. 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.
11. 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.
12. N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI-Quaterly*, 2(3):219–246, September 1989.
13. S. Garland and J. Guttag. A Guide to lp, the Larch Prover. Technical Report 82, DEC Systems Research Center, 1991.
14. 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.
15. D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. *Proc. of the 14th Symposium on Operating Systems Principles*, December 1993.
16. M. Raynal, A. Schiper, and S. Toueg. The Causal Ordering Abstraction and a Simple Way to Implement It. *IPL*, 39(6):343–350, 1991.
17. K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, pages 272–314, August 1991.
18. A. Israëli and M. Li. Bounded Time-Stamps. *Proc 28th IEE Symposium on Foundations of Computer Science*, pages 371–382, 1987.
19. D. Dolev and N. Shavit. Bounded Concurrent Time-Stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.
20. R. Cori and E. Sopena. Some Combinatorial Aspects of Time-Stamp Systems. *European Journal of Combinatorics*, pages 95–102, 1993.
21. A. Pnueli. The Temporal Logic of Programs. *Proc 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.
22. 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.
23. C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.
24. 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.