Model Checking Systems of Replicated Processes with Spin

Fabrice Derepas[1] and Paul Gastin[2]

[1] Nortel Networks, 1, Place des frères Montgolfier,
78928 Yvelines Cedex 09, France.
fderepas@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 distributed systems with several instances of the same process. Our technique uses symmetry which appears in the system. Exchanging those instances is not as simple as it seems, because there can be a lot of references to process locations in the system. We implemented a solution using the Spin model checker, and added two keywords to the Promela language to handle these new concepts.

1 Introduction

When a new protocol for a distributed system is designed, the behavior of each actor is known, but there is a need to validate the overall system behavior. 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 and nTS^{n-1}.

Very often in a distributed system several actors have similar roles, such as several clients regarding a server for instance. This is implemented as several processes running the same code. We say these actors are replicated processes. In this case, the system presents some symmetry that should be exploited during its verification. This has already been studied and implemented, e.g. in the Murφ model checker, where a special data structure with a restricted usage is introduced and used to describe the symmetric part of the system [6, 7, 5]. An implementation of scalarsets has been done under the Spin model Checker [4], in the Symmetric Spin package [1].

But in a distributed system with communicating processes, a process often keeps addresses of other processes in some variables, e.g. in order to send (or receive from) them messages. For instance we have several copies of a variable representing the process identifier in our system. This kind of variables does not fulfill the above requirements, hence, in this case, one cannot use the aforementioned techniques and tools.

The aim of this paper is to present an abstraction through symmetry which works also when using variables holding references to other processes. We give the theoretical background of our method and we describe an implementation for the Spin model Checker. We have introduced two new constructs to the Promela input language of Spin in order to allow the automatic reduction through symmetry. Our implementation translates the extended Promela description into a classical one and stores some extra information that is used later by the model checker. We have changed the next-state function of Spin so that it calls a reduction function after each newly generated state. The reduction function basically maps each state to a state which is equivalent up to the symmetry of the system. This reduction uses the extra information which was stored above.

We give some experimental results which show that our abstraction may induce a huge reduction of the number of states visited during the model checking.

A more theoretic approach to reduction using symmetry has been given in [3]. There, it is shown that one can reduce a system $\mathcal{M}$ to its quotient by a subgroup of $\mathrm{auto}(\mathcal{M}) \cap \mathrm{auto}(\varphi)$ where $\mathrm{auto}(\mathcal{M})$ is the group of automorphisms of $\mathcal{M}$ and $\mathrm{auto}(\varphi)$ is a similar group for the specification formula φ. For an automatic implementation, the difficulty is to compute a suitable subgroup. Our paper may be seen as a particular heuristic for this problem. By introducing the new constructs to the Promela Language and reducing the kind of formula one may write, we are able to compute a suitable subgroup that is contained in $\mathrm{auto}(\mathcal{M}) \cap \mathrm{auto}(\varphi)$. Actually, we do not use the formalism introduced in [3] and we give a direct proof of our result.

Part 2 presents the generic notions we are adding to Promela [4] to implement our abstractions. Soundness of the abstraction requires some constraints, verified by the model checker. These constraints are described.

Part 3 presents a formal approach to justify our abstraction. This is similar to proofs in [3], but it also allows to understand the former constraints.

Then part 4 gives some experimental results, and some heuristics for other systems.

2 Permutation of processes

This section details the need we have to permute processes. Notions required for this operation: references and public variables, are presented. Modifications to the Promela language follow.

2.1 Our goal

We are interested in systems where there are several processes running the same source code. This corresponds to several instances of the same procedure. Two instances of the same procedure are considered to be relatively equivalent, that is, for two such processes A_1 and A_2, we consider that the system state where A_1 is in state s_1 and A_2 in state s_2, is equivalent to the system state where A_1 is in state s_2 and A_2 in state s_1.

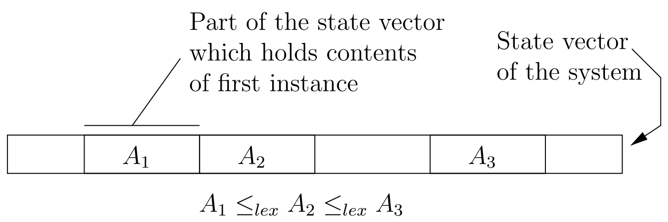

Fig. 1. Ordered instances in state vector

A simple idea, regarding all instances $A_1, \ldots, A_n$ of the same procedure, is to sort them in - for instance - lexicographic order in the state vector. This is shown in Figure 1. We have to apply this sorting procedure after each transition of the system in order to have only states where all instances of a same procedure are ordered. There can be two problems with this method:

- some global variables of the system are sometimes used as public variables attached to the processes. For instance, if there are N instances of process User one can imagine that each process has a boolean flag which may be used by other processes and hence must be declared as a global array. When exchanging two instances i and j of type User one should swap the global variables $flag[i]$ and $flag[j]$. Instead of defining such a flag as a global variable, a much better solution would be to declare it inside each process as a public variable.
- some variables in the system explicitly refer to the position of the process in the state vector. This is the case for the *_pid* variable. We should pay attention to the fact that if two processes are exchanged, then those variables should be updated since the positions in the state vector have changed. This is the notion of reference. If a byte variable n is a reference to the position of some process, then the value of n must be updated when two processes are exchanged.

The discussion above explains the reduction that can be obtained by sorting the state vector. Actually, we are considering an equivalence relation on the system states and we are taking a representative for each equivalence class. Sorting is just one possible algorithm to get such a representative but the technique extends indeed to other algorithms giving a representative.

2.2 New keywords for Promela

In order to use the two notions previously described we add two keywords to the Promela language, `ref` to declare reference variables, and `public` to declare local variables with a public scope. A reference variable of proctype P is used to access the public variables of a process of proctype P. A complete example is given in Figure 9.

For instance let us imagine a server with its own channel of messages. A regular declaration in Promela would be:

```
chan q[2] = [4] of {mtype};

active [2] proctype server () {
  q[_pid]?req;
  ...
}

active [3] proctype client () {
  if
  :: q[0]!req;
  :: q[1]!req;
  fi
}
```

The channel array `q` has to be a global variable since any process should be able to send messages to these channels. When two processes are exchanged, mailbox contents should also be exchanged. We propose to declare such a mailbox variable as a public element of the process:

```
active [2] proctype server () {
  public chan q = [4] of {mtype};
  q?req;
  ...
}

active [3] proctype client () {
  server[anyRef(server)].q!req
}
```

Actually the second declaration does not change the way the array is implemented in the model checker. The keyword public implicitly declares a global array. The size of the array is the number of instances of the proctype. We use a short cut `anyRef` which allows to choose a random reference in the servers.

Now if we want the client to send its identifier in the request. A local variable named `_lid` - for local identifier - is also defined. Its value is the number of the instance in the type, starting at 0. This enables to easily index array using the `_lid` variable. We get the following code:

```
active [2] proctype server () {
  public chan q = [4] of {mtype, ref client};
  ref client cli;
  q?req,cli;
  ...
}
```

```
active [3] proctype client () {
  server[anyRef(server)].q!req,_lid
}
```

2.3 Requirements

In order to use this abstraction some precise requirements must be fulfilled.

- One can only assign to a reference variable r of proctype P the values `_undef_ref`, or `anyRef`(P), or `_lid` if we are inside the proctype P, or s if s is also a reference variable of proctype P. In particular, it is not allowed to assign a constant such as 1 to a reference variable.
- One can only test whether a reference variable r of proctype P is equal to `_undef_ref`, or to `_lid` if we are inside the proctype P, or to s if s is also a reference variable of proctype P. It is neither allowed to use another comparison such as $<$ or $\leq$, nor to test whether r equals some constant such as 1. Some examples are presented below.

```
ref client cli;
int n;
chan q = [2] of {ref client};
ref server ser;
```

Allowed in client	Not allowed in client
`cli==_lid`	`cli==n`
`cli=_lid`	`cli=1`
`cli=_undef_ref`	`cli<_lid`
`q!_lid`	`q!3`
`q!anyRef(client)`	`q!ser`
`ser=anyRef(server)`	`ser==cli`

The `anyRef` function used in previous listings returns a reference of the specified type. From the Promela writer point of view this can be considered as a function which returns a random reference. From the model checker writer this is expanded as n transitions where n is the number of instances of the considered proctype. For instance the following code from a previous example:

```
q[anyRef(server)]!req
```

is expanded as:

```
if
:: q[0]!req
:: q[1]!req
fi
```

3 Abstraction based on process permutations

This section gives a formal framework for the abstraction we are performing.

3.1 Syntactic definition

Our aim is to study a system consisting of an asynchronous product of n automata $\mathcal{A}_i = (Q, V \cup W, P \cup R \cup S, I_i, T_i)$ where Q is a finite set of states. The sets V and W contain the public and the global (non-reference) variables of $\mathcal{A}_i$. The private (non-reference) variables of $\mathcal{A}_i$ can be encoded into its states, hence we do not need to keep them explicitly. Our reduction will permute the automata $\mathcal{A}_i$ and since the values of the reference variables must be updated, we need to keep them explicitly even if they are private variables. The sets P, R, S contains respectively the private, the public and the global reference variables of the system. Without loss of generality, we have assumed the same sets of states and of variables for all automata.

The set of initial states is $I_i \subseteq Q$. The set T_i consists of transitions which are tuples of the form (q, g, a, q') where $q, q' \in Q$ are the source and target states of the transition, g is the guard which must evaluate to true in order to enable the transition, and a is the action which modifies the variables of the system.

We define the set of visible references without indirection by $X_0 = P \cup R \cup S$. We define the set of visible references with k indirection by $X_k = R \times X_{k-1}$.

For instance, in the following code the level of indirection is $k = 2$.

```
ref server servid;
clientRef[serverRef[servid]]=lid;
```

We define the set X of visible references by

$$X = \bigcup_{0 \leq k \leq k_{max}} X_k$$

where k_{max} is the maximum level of indirection which can be found in the code. We also define $Y = W \cup V \cup (V \times X)$ to be the set of visible (non-reference) variables.

A guard g of a transition is a boolean function with domain

$$\mathbb{N}^Y \times \{0,1\}^{X \times X} \times \{0,1\}^{X \times \{\mathtt{undef}\}} \times \{0,1\}^{X \times \{\mathtt{lid}\}}.$$

Intuitively, the truth of the guard g depends only on the values of the visible non-reference variables ($\mathbb{N}^Y$) and on whether a visible reference variable from X equals another visible reference variable ($\{0,1\}^{X \times X}$), or is undefined ($\{0,1\}^{X \times \{\mathtt{undef}\}}$), or is the local identifier ($\{0,1\}^{X \times \{\mathtt{lid}\}}$).

An action a is a (possibly empty) sequence of assignments to visible variables from $X \cup Y$. An assignment is either a pair (v, f) where $v \in Y$ and f is a mapping from $\mathbb{N}^Y$ to $\mathbb{N}$ (the new value of a visible non-reference variable depends only on the old values of the visible variables); or a pair (r, h) where $r \in X$ and $h \in X \cup \{\mathtt{undef}, \mathtt{lid}, \mathtt{anyRef}(A)\}$ for some $A \subseteq \{1, \ldots, n\}$. The intuition is that one can only assign to a reference variable the value of another reference variable or one of the values $\mathtt{undef}$, $\mathtt{lid}$, $\mathtt{anyRef}(A)$.

3.2 Semantic definition

In the asynchronous product $\mathcal{M} = \prod \mathcal{A}_i = (\mathcal{S}, \mathcal{I}, \mathcal{T})$, each automaton $\mathcal{A}_i$ must use a separate copy of the local variables V, P and R. Hence, the actual non-reference variables of the system are $\mathcal{V} = W \cup (V \times \{1, \ldots, n\})$ and the actual reference variables of the system are $\mathcal{R} = S \cup (P \times \{1, \ldots, n\}) \cup (R \times \{1, \ldots, n\})$.

The set of concrete states of $\mathcal{M}$ is $\mathcal{S} = Q^n \times \mathbb{N}^{\mathcal{V}} \times \{0, \ldots, n\}^{\mathcal{R}}$ and the set of concrete initial states of $\mathcal{M}$ is $\mathcal{I} = (\prod_{1 \leq i \leq n} I_i) \times \{0\}^{\mathcal{V}} \times \{0\}^{\mathcal{R}}$.

Since we are considering an asynchronous product, the set of transitions is defined by $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \subseteq \mathcal{S} \times \mathcal{S}$ where $(q, \nu, \rho, q', \nu', \rho') \in \mathcal{T}_i$ if there exists some transition $(p, g, a, p') \in T_i$ such that $q_i = p$ and the guard g evaluates to true in state (i, ν, ρ) and the new state q' is defined by $q'_i = p'$ and $q'_j = q_j$ for all $j \neq i$ and (ν', ρ') is obtained by executing the sequence of assignments defined by the action a.

In order to give a formal definition, we first define the actual variables associated with the visible variables from $X \cup Y$ in state (i, ρ).

- For $r \in X$, the actual variable $r_{i,\rho}$ associated with r in state (i, ρ) is defined by $r_{i,\rho} = r$ if $r \in S$, $r_{i,\rho} = (r, i)$ if $r \in P \cup R$, and $r_{i,\rho} = (s, \rho(t_{i,\rho}))$ if $r = (s, t) \in R \times X_k$ for some $k \geq 0$.
- For $v \in Y$, the actual variable $v_{i,\rho}$ associated with v in state (i, ρ) is defined by $v_{i,\rho} = v$ if $v \in W$, $v_{i,\rho} = (v, i)$ if $v \in V$, and $v_{i,\rho} = (w, \rho(r_{i,\rho}))$ if $w = (w, r) \in V \times X$.

Now the evaluation of the guard g in state (i, ν, ρ) is

$$g(\nu(v_{i,\rho})_{v \in Y}, (\rho(r_{i,\rho}) = \rho(s_{i,\rho}))_{r,s \in X}, (\rho(r_{i,\rho}) = 0)_{r \in X}, (\rho(r_{i,\rho}) = i)_{r \in X})$$

which must be true for the transition to be enabled.

An assignment (v, f) to a visible non-reference variable $v \in Y$ changes the current value of the variable $v_{i,\rho}$ to $f((\nu(u_{i,\rho}))_{u \in Y})$.

An assignment (r, h) to a visible reference variable $r \in X$ changes the current value of the variable $r_{i,\rho}$ to $\rho(h_{i,\rho})$ if $h \in X$, or to 0 if $h = \texttt{undef}$, or to i if $h = \texttt{lid}$, or to any value from A if $h = \texttt{anyRef}(A)$ (this $\texttt{anyRef}(A)$ assignment creates several transitions, to different states).

3.3 Permutations of states

Here is a definition which specifies what happens to a concrete state of the system $\mathcal{M}$ when we permute the processes.

Definition 1. *Let π be a permutation over $\{1, \ldots, n\}$ which is extended to $\{0, \ldots, n\}$ by setting $\pi(0) = 0$.*

We first extend π to $\mathcal{V} \cup \mathcal{R}$: for $v \in W \cup S$ we set $\pi(v) = v$, and for $v \in V \cup P \cup R$ and $1 \leq i \leq n$, we set $\pi((v, i)) = (v, \pi(i))$.

Next, for a concrete state $s = (q, \nu, \rho) \in \mathcal{S}$ of $\mathcal{M}$, we define

$$\pi(s) = (q \circ \pi, \nu \circ \pi, \pi^{-1} \circ \rho \circ \pi).$$

Intuitively, applying the permutation π to the state s results in a new state $\pi(s)$ where the process at position i in $\pi(s)$ is the process at position $\pi(i)$ in s.

Our system consists of instances of a few proctypes. For instance, we may have only two proctypes, e.g. client and server. Then we can write $\{1, \ldots, n\}$ as a disjoint union $A \dot\cup B$ where A and B correspond to the indices of the client and server instances. Since all clients (resp. servers) share the same code, they are modeled with the same transitions: $T_i = T_A$ for all $i \in A$ and similarly for the servers. Also, we restrict the `anyRef` assignment to use sets of indices corresponding to some proctype. In the example above, only `anyRef`(A) and `anyRef`(B) are allowed.

It is only meaningful to permute processes of the same type, hence we will only consider permutations that preserve proctypes. Formally, we say that a permutation π of $\{1, \ldots, n\}$ preserves proctypes if for any proctype, the set of indices corresponding to the instances of this proctype is invariant under the permutation π. In the example above, this means that $\pi(A) = A$ and $\pi(B) = B$. Note that, if the permutation π preserves proctypes then we have $T_{\pi(i)} = T_i$ for all $i \in \{1, \ldots, n\}$. We are going to show that such permutations do not have any effect on the transitions which can be triggered.

Lemma 1. *Let $s, s' \in \mathcal{S}$ be two concrete states of $\mathcal{M}$ and let π be a permutation of $\{1, \ldots, n\}$ which preserves proctypes. If $(s, s') \in \mathcal{T}$ then $(\pi(s), \pi(s')) \in \mathcal{T}$.*

Proof. Let $s = (q, \nu, \rho)$ and $s' = (q', \nu', \rho')$ be such that $(s, s') \in \mathcal{T}$. Let $i \in \{1, \ldots, n\}$ be such that $(s, s') \in \mathcal{T}_i$ and let $(p, g, a, p') \in T_i$ be the associated transition. Let π be a permutation of $\{1, \ldots, n\}$ preserving proctypes.

We want to prove that $(\pi(s), \pi(s')) \in \mathcal{T}_j$ with $j = \pi^{-1}(i)$ and the same associated transition $(p, g, a, p') \in T_j = T_{\pi(j)} = T_i$. We let $\pi(s) = (\tilde{q}, \tilde{\nu}, \tilde{\rho})$ and $\pi(s') = (\tilde{q}', \tilde{\nu}', \tilde{\rho}')$.

$$\begin{tikzcd}
s = (q, \nu, \rho) \arrow[r, "\mathcal{T}_i"] \arrow[d, "\pi"'] & s' = (q', \nu', \rho') \arrow[d, "\pi"] \\
\pi(s) = (\tilde{q}, \tilde{\nu}, \tilde{\rho}) \arrow[r, "\mathcal{T}_j"] & \pi(s') = (\tilde{q}', \tilde{\nu}', \tilde{\rho}')
\end{tikzcd}$$

Claim.
(1) For all $r \in X$, we have $\pi(r_{j,\tilde{\rho}}) = r_{i,\rho}$.
(2) For all $v \in Y$, we have $\pi(v_{j,\tilde{\rho}}) = v_{i,\rho}$.

Proof of (1). Let $r \in X = \bigcup_k X_k$. We proceed by induction on k.
– If $r \in S$, then $r_{j,\tilde{\rho}} = r = r_{i,\rho} = \pi(r)$.
– If $r \in P \cup R$, then $\pi(r_{j,\tilde{\rho}}) = \pi((r, j)) = (r, i) = r_{i,\rho}$.
– Assume the result holds for some $k \geq 0$ and let $r = (s, t) \in R \times X_k$. We have

$$\pi(r_{j,\tilde{\rho}}) = \pi((s, \tilde{\rho}(t_{j,\tilde{\rho}}))) = \pi((s, \pi^{-1} \circ \rho \circ \pi(t_{j,\tilde{\rho}}))) = (s, \rho(t_{i,\rho})) = r_{i,\rho}.$$

Proof of (2).
– If $v \in W$, then $v_{j,\tilde\rho} = v = v_{i,\rho} = \pi(v)$.
– If $v = (w,t) \in V \times X$. We have

$$\pi(v_{j,\tilde\rho}) = \pi((w, \tilde\rho(t_{j,\tilde\rho}))) = \pi((w, \pi^{-1} \circ \rho \circ \pi(t_{j,\tilde\rho}))) = (w, \rho(t_{i,\rho})) = v_{i,\rho}.$$

We deduce first that the evaluation of the guard g at $(j, \tilde\nu, \tilde\rho)$ equals the evaluation of the guard g at (i, ν, ρ). Indeed, the arguments are the same since for all $u \in Y$ we have $\tilde\nu(u_{j,\tilde\rho}) = \nu \circ \pi(u_{j,\tilde\rho}) = \nu(u_{i,\rho})$; and for all $r \in X$ we have $\tilde\rho(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho \circ \pi(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho(r_{i,\rho})$. Hence,

– $\tilde\rho(r_{j,\tilde\rho}) = \tilde\rho(s_{j,\tilde\rho})$ if and only if $\rho(r_{i,\rho}) = \rho(s_{i,\rho})$,
– $\tilde\rho(r_{j,\tilde\rho}) = 0$ if and only if $\rho(r_{i,\rho}) = 0$, and
– $\tilde\rho(r_{j,\tilde\rho}) = j$ if and only if $\rho(r_{i,\rho}) = i$.

It remains to show that executing from $\pi(s)$ the transition associated with $(p, g, a, p') \in T_i = T_j$ in the j-th automaton yields the state $\pi(s')$. We give the proof when the action a consists of a single assignment. The generalization to a sequence of assignments is easy.

– $\tilde q'_j = q'_{\pi(j)} = q'_i = p'$ and for $k \neq j$, we have $\pi(k) \neq i$ and therefore $\tilde q'_k = q'_{\pi(k)} = q_{\pi(k)} = \tilde q_k$.
– If the assignment is (v, f) with $v \in Y$ then we have $\tilde\nu'(v_{j,\tilde\rho}) = \nu' \circ \pi(v_{j,\tilde\rho}) = \nu'(v_{i,\rho}) = f((\nu(u_{i,\rho}))_{u\in Y}) = f((\nu \circ \pi \circ \pi^{-1}(u_{i,\rho}))_{u\in Y}) = f((\tilde\nu(u_{j,\tilde\rho}))_{u\in Y})$.
– If the assignment is (r, h) with $r \in X$ then we have
  - If $h \in X$ then $\tilde\rho'(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho' \circ \pi(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1} \circ \rho(h_{i,\rho}) = \pi^{-1} \circ \rho \circ \pi(h_{j,\tilde\rho}) = \tilde\rho(h_{j,\tilde\rho})$.
  - If $h = \texttt{undef}$ then $\tilde\rho'(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1}(0) = 0$.
  - If $h = \texttt{lid}$ then $\tilde\rho'(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1}(i) = j$.
  - If $h = \texttt{anyRef}(A)$ then $\tilde\rho'(r_{j,\tilde\rho}) = \pi^{-1} \circ \rho'(r_{i,\rho}) \in A$ since $\rho'(r_{i,\rho}) \in A$ and π is a permutation preserving proctypes.

Therefore, we have shown that $(\pi(s), \pi(s')) \in \mathcal{T}_j$.

3.4 Abstraction

An abstraction consists of representing several states by a single state. We will be using an equivalence relation $\mathbf{R}$. Then an element will be chosen in each equivalence class to represent all the class.

Using the permutations on $\mathcal{M}$ we can define the equivalence relation $\mathbf{R}$ between concrete states of $\mathcal{M}$: we say that two concrete states s and s' of $\mathcal{M}$ are $\mathbf{R}$-equivalent if $\pi(s) = s'$ for some permutation π of $\{1, \dots, n\}$.

Here is now the main property which validates our abstraction.

Theorem 1. *The equivalence relation $\mathbf{R}$ is a bisimulation on $\mathcal{M}$.*

Proof. Let s, s', s_1 be three states of $\mathcal{M}$ such that $(s, s') \in \mathcal{T}$ and $s\,\mathbf{R}\,s_1$. We have to prove that there exists s'_1 such that $(s_1, s'_1) \in \mathcal{T}$ and $s'\,\mathbf{R}\,s'_1$. Note that since $\mathbf{R}$ is an equivalence relation, we do not have to prove the converse.

By definition of $\mathbf{R}$, there exists a permutation π of $\{1, \ldots, n\}$ such that $\pi(s) = s_1$. Let $s'_1 = \pi(s')$, hence we have $s'\,\mathbf{R}\,s'_1$. By Lemma 1, we know that $(s_1, s'_1) \in \mathcal{T}$, which proves the theorem.

The quotient of $\mathcal{M}$ by the equivalence $\mathbf{R}$ is the transition system $\bar{\mathcal{M}} = (\bar{\mathcal{S}}, \bar{\mathcal{T}}, \bar{\mathcal{I}})$ where: $\bar{\mathcal{S}} = \{\bar{s} \mid s \in \mathcal{S}\}$, $\bar{\mathcal{I}} = \{\bar{s} \mid s \in \mathcal{I}\}$ and $\bar{\mathcal{T}} = \{(\bar{s}, \bar{s'}) | (s, s') \in \mathcal{T}\}$. Now it is well known that when an equivalence relation $\mathbf{R}$ on a transition system $\mathcal{M}$ is a bisimulation, then the quotient $\bar{\mathcal{M}}$ is bisimilar to $\mathcal{M}$.

3.5 Pragmatic abstraction

In order to apply the above reduction, one has to compute the quotient $\bar{\mathcal{M}}$. A possibility is to choose a canonical representative in each class and to use a function $f : S \rightarrow S$ mapping each state to its canonical representative. Such a mapping f satisfies the two properties

1. for all $s \in S$, $f(s)\,\mathbf{R}\,s$,
2. for all $s, s' \in S$, $s\,\mathbf{R}\,s'$ implies $f(s) = f(s')$ (canonical representative).

Actually, our mapping f is given by the pseudo-sorting algorithm presented in Section 4.1. It maps each state s to a permuted state $f(s)$ hence it satisfies (1) but it does not necessarily satisfy (2). Therefore, our reduced system may not be isomorphic to the quotient $\bar{\mathcal{M}}$ and we need to prove that it is still bisimilar to the original system.

Proposition 1. *Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T})$ be a transition system and let $\mathbf{R}$ be an equivalence relation on $\mathcal{S}$ which is a bisimulation. Let $f : \mathcal{S} \rightarrow \mathcal{S}$ be a mapping satisfying $f(s)\mathbf{R}s$ for all $s \in \mathcal{S}$. We define the reduced system $\mathcal{M}' = (\mathcal{S}, f(\mathcal{I}), \mathcal{T}')$ by $\mathcal{T}' = \{(s, f(s')) \mid (s, s') \in \mathcal{T}\}$.*

$\mathcal{M}$ and $\mathcal{M}'$ are bisimilar.

Proof. let $s, s', s_1 \in \mathcal{S}$ be such that $(s, s') \in \mathcal{T}$ and $s\,\mathbf{R}\,s_1$. There exists $s'_1 \in \mathcal{S}$ such that $(s_1, s'_1) \in \mathcal{T}$ and $s'\,\mathbf{R}\,s'_1$. We deduce that $(s_1, f(s'_1)) \in \mathcal{T}'$ and $s'\,\mathbf{R}\,s'_1\,\mathbf{R}\,f(s'_1)$.

Let $s_1, s'_1 \in \mathcal{S}$ be such that $(s_1, s'_1) \in \mathcal{T}'$. There exists s''_1 such that $(s_1, s''_1) \in \mathcal{T}$ and $s'_1 = f(s''_1)$. Now let s be such that $s\,\mathbf{R}\,s_1$. There exists $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{T}$ and $s'\,\mathbf{R}\,s''_1$. We are done since $s'\,\mathbf{R}\,s''_1\,\mathbf{R}\,s'_1$.

3.6 Permutable specification

Now the question is to determine which properties can be verified on the reduced system $\mathcal{M}'$ (or $\bar{\mathcal{M}}$). Since the original system is bisimilar to the reduced one, we can verify any CTL* property provided the state formulas are invariant under permutation. This includes a lot of interesting properties. For instance, in a mutual exclusion algorithm we want to check whether there exists an accessible state such that two processes are in the critical section. Such a property is invariant under permutation.

On the contrary, consider the following response property: "whenever process 1 requests a resource, process 1 will eventually be granted the resource". Such a property cannot be checked directly on our reduced system. If we write literraly this property, then Spin may erroneously detect that it holds or that it does not hold. For instance, if the process at position 1 requests the resource and only gets the resource when being in some other position, Spin may erroneously say that the property does not hold. Also, if whenever some process at position 1 requests the resource then in the next step some other process that has the resource takes position 1, then Spin may erroneously say that the property holds.

Therefore, in order to use our reduction method, one has to be careful in writing the specification to be checked. Explicit references to process positions should be avoided. It would be very interesting to define a specification language which guarantees that the specifications are invariant under permutations. Another interesting direction is to permute the specification when permuting the state vector after each step of the computation.

4 Computation of the reduced system

The state vector consists of a tuple giving for each variable of the system its present value. This state vector has however some structure. The global variables occur only once and the local variables declared in each proctype are replicated for each instance of this proctype. Hence, each process running in the system has an associated tuple of variables in the state vector. Permuting the processes of some proctype A means permuting the associated tuples of variables in the state vector and modifying accordingly the values of the reference variables to this proctype as explained in Section 2.

4.1 The pseudo-sorting algorithm

The mapping f transforming each state into an equivalent one is implemented by a sorting algorithm. More precisely, we consider the lexicographic order on the tuples of variables associated with the processes of some proctype A and we sort these tuples according to this lexicographic order. Indeed, we perform this sorting operation for each proctype. We use a classical sorting algorithm, like quick sort [8] for instance. We consider the sort procedure as a procedure which takes three arguments: the list to sort, the procedure to exchange two arguments in the list, the function to compare two arguments. The algorithm is presented in Figure 2.

Actually this algorithm can lead to state vectors that are still not sorted. This is why we call it pseudo-sorting. The final vector might not be sorted because when we exchange two processes we change reference values afterwards, and this might change the relative ordering. For instance, consider a system with only 3 processes of some proctype A defined by

```
procedure Exch_i(k,l)
    exchanges data in state vector at the positions of
        k-th and l-th processes of type i.
    For all references in the state vector to processes of type i
        if the reference is k change it to l
        else if the reference is l change it to k
Function Compare_i(k,l)
    returns if k-th process of type i
        is smaller or equal to l-th process of type i.
procedure pseudoSortStateVector()
    For each type of process i
        Let l_i be the list of processes of type i.
        Sort(l_i,Exch_i,Compare_i)
```

Fig. 2. Pseudo sorting algorithm

```
active [3] proctype A ()
{ ref A r;
  public byte v;
  /* A body */
}
```

Note that there is always an implicit `_state` variable that is declared last. Hence the tuple of variables associated with a process is `(r,v,_state)`. A possible state vector is $3, 1, 1|0, 3, 4|1, 2, 1$. It is not sorted. Swapping the first two processes gives the state $0, 3, 4|3, 1, 1|2, 2, 1$. Note that the reference value of the third process has changed. This state vector is still not sorted. Now swapping the last two processes yields the state $0, 3, 4|3, 2, 1|2, 1, 1$ which again is not sorted. Therefore no permutations of the processes yield a sorted state vector.

4.2 Efficiency depends on the variable declaration order

In the tuple of variables associated with some proctype A, the order of the variables is the declaration order. If we change the declaration order then the lexicographic order is changed accordingly and the reduction that we get may be completely different.

Let us consider a very simple example where our system consists only of 3 clients and 3 servers where each server can be connected with a single client. We assume that each client (respectively server) keeps a reference to the server (respectively client) it is connected to. In addition, clients and servers have a public variable.

```
active [3] proctype Client ()
{ ref Server r;
  public byte v;
  /* Client body */
```

```
active [3] proctype Server ()
{ ref Client r;
  public byte v;
  /* Server body */
```

```
}                                            }
```

Note that there is always an implicit `_state` variable that is declared last. Hence the tuple of variables associated with a client or a server is `(r,v,_state)`. An example is shown on Figure 3 for the first value r of each instance in the state vector.

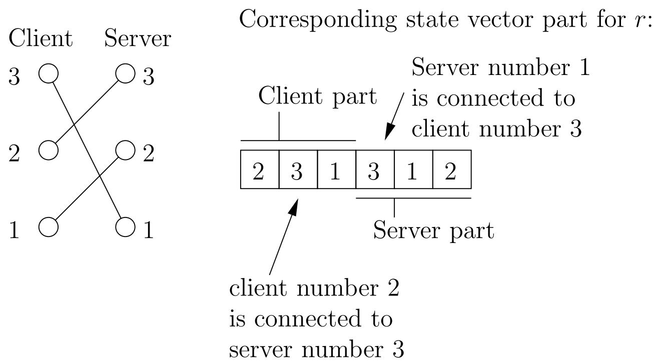

Fig. 3. An example of configuration

In this example, the state vector is $2,1,2|3,2,3|1,3,4||3,4,3|1,5,2|2,6,1$ where the vertical bars separate the process tuples and the double bar separates the clients from the servers. Applying the sorting algorithm on the clients gives the state vector $1,3,4|2,1,2|3,2,3||1,4,3|2,5,2|3,6,1$. Note that the client references in the server tuples have been changed accordingly. Now the state vector is also sorted according to the server tuples. As shown on Figure 4, from any initial connection graph we get the most simple configuration possible. Of course this dramatically reduces the state space.

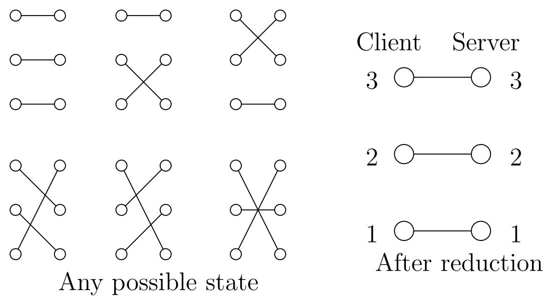

Fig. 4. Before and after a reduction

Now, assume that the declaration order between r and v is reversed as in

```
active [3] proctype Client ()
{ public byte v;
  ref Server r;
  /* Client body */
}
```

```
active [3] proctype Server ()
{ public byte v;
  ref Client r;
  /* Server body */
}
```

Then the state vector associated with the same configuration is $1, 2, 2|2, 3, 3|3, 1, 4 \,||4, 3, 3|5, 1, 2|6, 2, 1$. Since this vector is already sorted, no reduction will occur.

This very simple example illustrates why the order in which variables are declared has a great influence on the reduction obtained. As stated above, there is always an implicit variable `_state` which is declared last. If we want to change the position in the tuple of this variable we could declare it explicitly as in

```
active [3] proctype Client ()
{ byte _state;
  public byte v;
  ref Server r;
  /* Client body */
}
```

The results of a more realistic experiment is shown in Figure 5. Here we consider a Client-Server system whose complete description is given in the appendix (Figure 9). We denote by n the number of clients in the system, n is also the number of servers. A client can be idle or can try to connect to a server. The connection will be granted if the server is not busy. The first (respectively second) column of Figure 5 corresponds to the reduced system when the variable `serv_id` of proctype client is declared first (respectively last). The third column corresponds to the system without reduction. This experiment shows that our reduction is quite efficient and that it depends noticeably on the order in which the variables are declared. Hence, in order to get the best benefit from our method, one must have an intuition of which order will be good. This is difficult to guess, and one need to have a good understanding of the system being modeled.

n	ref. first	ref. last	no reduction
2	27	40	63
3	108	227	918
4	405	928	16 929
5	1 458	3 518	375 678

Fig. 5. Number of states for different orders and different sizes

4.3 Layered Model

The pseudo sorting method will work well on layered models. Here is an informal description of a layered model. We have ℓ type of processes (like client, server ...), named $t_1, \ldots, t_\ell$. Each process of type t_i has references to some other process of type t_{i+1} or has an undefined reference. We assume all defined references to process of type t_i are different. This is shown on Figure 6 We assume that the

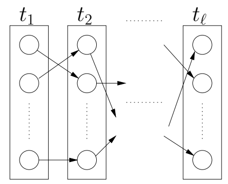

Fig. 6. A layered model

reference to layer $i+1$ comes first in the lexicographic order for an instance of type t_i. If we sort according to proctype $t_{\ell-1}$, then $t_{\ell-2}$ until t_1, then from any element of the equivalence class of **R** we will get the same element.

Figure 7 shows the result for a three tier model [2] (client/server/data base).

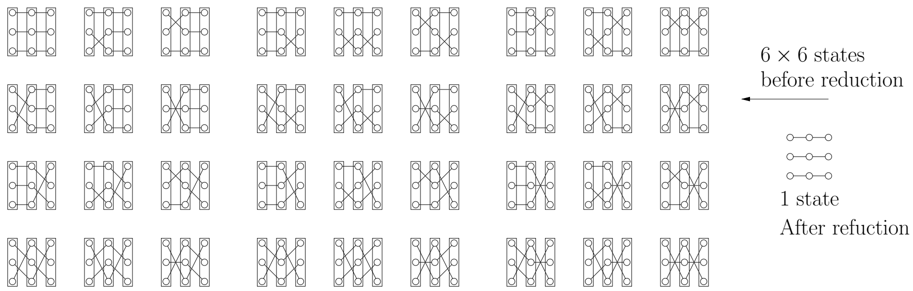

Fig. 7. Three tier model

5 Conclusions

We have introduced two notions which enables abstraction for distributed systems: references to other processes, and a "public" scope for local variables.

These notions need to be used carefully, in a way that all processes of the same type can be exchanged. We have clearly stated in Section 2.3 what are those constraints. They can be easily checked when the Promela model is compiled.

In the future we should specify how to easily write LTL formula which are unchanged by permutations. It would also be interesting to write pluggable modules, to implement other heuristics than the pseudo sorting algorithm to get canonical representatives of equivalence classes of **R**.

Appendix

This appendix shows a practical implementation on Figure 9. Figure 8 shows how our abstraction has been implemented. We start from the Promela code with our

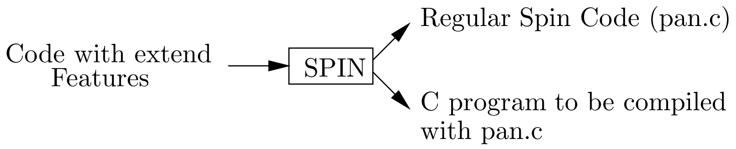

Fig. 8. Software architecture

extended features. Then our modified version of Spin generates a regular model checking code (`pan.c`), together with a file which will be compiled with `pan.c`.

References

1. D. Bošnački, D. Dams, and L. Holenderski. Symmetric spin. In *Proceedings of the 7th SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 2000.
2. N. G. Depledge, W. A. Turner, and A. Woog. An open, distributable, three-tier client-server architecture with transaction semantics. *Digital Technical Journal*, 7(1), 1995.
3. E. A. Emerson and A. P. Sistla. Symmetry and Modelchecking. *Formal Methods in System Design*, 9(1):105–130, 1996.
4. G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
5. C. Norris Ip and D. L. Dill. Verifying Systems with Replicated Components in Murφ. *Formal Methods in System Design*, 1997.
6. C. N. Ip and D. L. Dill. Better Verification through Symmetry. *International Conference on Computer Hardware Description Languages*, pages 87–100, April 1993.
7. C. N. Ip and D. L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234. IEEE Computer Society, 1993.
8. D. E. Knuth. *The Art of Computer Programming*, volume 3, chapter 5, pages 114–123. Addison Wesley, 1973.

Code with extented features

```
mtype = {ack,nack,rel,req};

active [3] proctype client () {
  ref server serv_id;
  public chan cliChan =
                 [4] of {mtype};
idle :
  serv_id=anyRef(server);
  server[serv_id].serChan!req(_lid);
  if
  :: cliChan?ack; goto work
  :: cliChan?nack; goto idle
  fi;
work:
  server[serv_id].serChan!rel;
  goto idle
}

active [2] proctype server () {
  public chan serChan =
          [4] of {mtype,ref client};
  byte inUse;
  ref client cli;

  do
  :: serChan?req(cli) ->
    if
    :: inUse==0 ->  inUse=1;
      client[cli].cliChan!ack,_lid
    :: inUse==1 ->
      client[cli].cliChan!nack
    fi;
  :: serChan?rel -> inUse=0
  od
}
```

Generated Promela code

```
mtype = {ack,nack,rel,req};

chan cliChan[3]=
  [4] of {mtype};
chan serChan[3]=
  [4] of {mtype,byte};

#define _lid _pid-0
active [3] proctype client () {
  byte serv_id;
idle :
  if
  :: serv_id=0
  :: serv_id=1
  fi;
  serChan[serv_id]!req(_lid);
  if
  :: cliChan[_lid]?ack; goto work
  :: cliChan[_lid]?nack; goto idle
  fi;
work:
  serChan[serv_id]!rel,0;
  goto idle
}
#undef _lid

#define _lid _pid-3
active [2] proctype server () {
  byte inUse;
  byte cli;
  do
  :: serChan[_lid]?req(cli) ->
    if
    :: inUse==0 -> inUse=1;
      cliChan[cli]!ack,_lid
    :: inUse==1 ->
      cliChan[cli]!nack,0
    fi
  :: serChan[_lid]?rel,_ ->
      inUse=0
  od
}
#undef _lid
```

Fig. 9. Example of new keywords