INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications

Manuel Aguilar Cornejo — Hubert Garavel — Radu Mateescu — Noël de Palma

N° 4222

Juillet 2001

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4222--FR+ENG

Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications

Manuel Aguilar Cornejo* , Hubert Garavel† , Radu Mateescu‡ , Noël de Palma§

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n° 4222 — Juillet 2001 — 32 pages

Abstract: Dynamic reconfiguration increases the availability of distributed applications by allowing them to evolve at run-time. This report deals with the formal specification and model-checking verification of a dynamic reconfiguration protocol used in industrial agent-based applications. Starting from a reference implementation in JAVA, we produced a specification of the protocol using the Formal Description Technique LOTOS. We also specified a set of temporal logic formulas characterizing the correct behaviour of each protocol primitive. Finally, we studied various finite state configurations of the protocol, on which we verified these requirements using the CADP protocol engineering tool set.

Key-words: compositional verification, distributed application, dynamic reconfiguration, LOTOS, mobile agent, model-checking, specification, temporal logic

A short version of this report is available as "*Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications*", in Aleksander Laurentowski, editor, *Proceedings of the Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001 (Krakow, Poland)*, September 17–19, 2001.

* Manuel.Aguilar@imag.fr
† Hubert.Garavel@inria.fr
‡ Radu.Mateescu@inria.fr
§ Noel.De-Palma@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Spécification et vérification d'un protocole de reconfiguration dynamique d'applications à base d'agents mobiles

Résumé : La reconfiguration dynamique augmente la disponibilité des applications réparties en leur permettant d'évoluer pendant l'exécution. Ce rapport concerne la spécification formelle et la vérification énumérative d'un protocole de reconfiguration dynamique utilisé dans des applications industrielles à base d'agents mobiles. Sur la base d'une implémentation de référence en Java, nous avons produit une spécification du protocole en utilisant la technique de description formelle Lotos. Nous avons également spécifié un ensemble de formules de logique temporelle caractérisant le comportement correct de chaque primitive du protocole. Finalement, nous avons étudié différentes configurations du protocole ayant un nombre fini d'états, sur lesquelles nous avons vérifié ces formules au moyen de la boîte à outils Cadp pour l'ingénierie des protocoles.

Mots-clés : agent mobile, application distribuée, logique temporelle, Lotos, reconfiguration dynamique, spécification, vérification énumérative, vérification compositionnelle

1 Introduction

As computing resources become decentralized, the development of distributed applications receives increasing attention from the software engineering community. These applications are often complex and must satisfy strong reliability and availability constraints. To avoid stopping an entire distributed application for maintenance operations (e.g., repair, upgrade, etc.), it is essential to provide mechanisms allowing distributed applications to be reconfigured at run-time. Such mechanisms should ensure a proper functioning of the application regardless of run-time changes (e.g., creation or deletion of agents, replacement of agents, migration of agents across execution sites, modification of communication routes, etc). Moreover, these mechanisms should not induce heavy penalties on applications during maintenance operations.

Dynamic reconfiguration has been studied and implemented in various middlewares, such as Conic [KM89], Argus [BD93], and Polylith [Pur94]. In some approaches, e.g., Polylith, dynamic reconfiguration is part of the applications developed on top of the middleware, thus transferring to application developers the responsibility to ensure consistency after reconfiguration. In other approaches, e.g., Conic and Argus, the middleware is extended with (application-independent) dynamic reconfiguration features.

This report studies the protocol for dynamic reconfiguration of agent-based applications defined in [PBR99], which follows the latter approach. This protocol has been implemented in the middleware platform Aaa (*Agents Anytime Anywhere*) [BPF+99, PBF+00], which allows a flexible, scalable, and reliable development of distributed applications. The protocol has been experimented on several industrial applications developed in cooperation with Bull, and especially on an application for managing a set of network firewalls [PBF+00]. In this application (included in Bull's NetWall security product), each firewall produces a log file of audit information; agents are used to manage logged information, to provide filtering functionalities that can be added and customized lately according to customer requirements, to correlate and coordinate multiple firewalls, and to deploy a set of log management applications over the firewalls.

As this dynamic reconfiguration protocol is non-trivial, it was suitable to ensure its correctness using formal methods, and especially to establish that reconfiguration preserves the consistency of the application. Starting from the informal description of the protocol given in [PBR99] and a Java implementation that was already in use, we produced a formal specification of the protocol using the Iso Formal Description Technique Lotos [ISO88]. We then identified a set of safety and liveness properties characterizing the desired behaviour of each reconfiguration primitive of the protocol. To verify whether these correctness properties hold for the Lotos specification, we used the *model-checking* approach [CGP00]; verification was carried out using Cadp [FGK+96], a protocol engineering tool set providing state-of-the-art compilation, simulation, and verification functionalities.

This report is organized as follows. Section 2 presents the Aaa agent-based middleware and its dynamic reconfiguration protocol. Section 3 describes the Lotos specification of the protocol. Section 4 reports about the verification process performed using Cadp. Section 5 discusses the results and gives directions for future work. The complete Lotos specification of the protocol is given in Annex A.

2 The dynamic reconfiguration protocol

In this section, we first introduce the Aaa distributed agent model. Then, we state the dynamic reconfiguration problem and present the principles of the reconfiguration protocol under study.

2.1 The AAA distributed agent model

In the Aaa model [BPF+99], the basic software elements are *agents* executing concurrently on several *sites*. Each agent has only one execution flow (single-thread). Agents are connected by *communication channels*, i.e., unidirectional point-to-point links. Agents can synchronize and communicate only by sending or receiving messages on communication channels, which play the role of references to other agents.

Agents behave according to an *event-reaction* scheme: when receiving an event on a communication channel, an agent executes the appropriate reaction, i.e., a piece of code that may update the agent state and/or send messages to other agents (including the agent itself).

The Aaa infrastructure ensures that agents and communications satisfy certain properties [BPF+99] listed in the table below. The dynamic reconfiguration protocol relies upon some of these properties, and especially the *causality* property (also called *causal ordering*) [RST91, LBBK01].

Agent properties	
Persistency	Agent lifetime is not bounded to the duration of execution (however, this does not ensure consistent state retrieval after failures).
Atomicity	Upon receipt of an event, the reaction of an agent is either fully executed or not executed at all.
Configurability	Agent attributes and references to other agents can be changed at run-time by a third party (e.g., an administrator).
Communication properties	
Asynchrony	No assumption is made on transmission speed, allowing applications to be designed and implemented in a time-independent manner.
Reliability	Message delivery is guaranteed in spite of network failures or system crashes and without any involvement from the application.
Causality	Messages are delivered in the same order as they are sent.

2.2 Dynamic reconfiguration

Dynamic reconfiguration of an agent-based application encompasses (at least) four possible changes in the structure of the application at run-time: *architectural changes* (creation or deletion of agents, modification of communication routes), *migration changes* (modification of the placement of agents on execution sites), *agent implementation changes*, and *agent interface changes*. The dynamic reconfiguration protocol under study takes into account only the first two aspects.

Figure 1 shows an example of application reconfiguration involving the migration of an agent across two sites. This example will be used throughout this section.

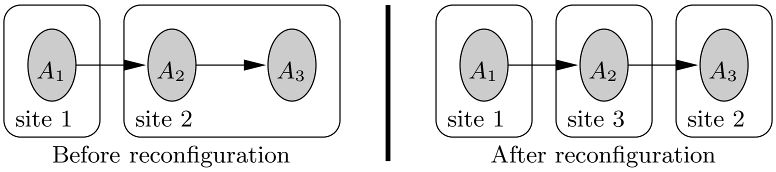

Figure 1: Migration of agent A_2 from site 2 to site 3

Dynamic reconfiguration must preserve *consistency* [KM90]: after reconfiguration, the application should be able to resume its execution from its global state prior to reconfiguration. Figure 2 shows an inconsistency that may occur during the reconfiguration depicted on Figure 1: message m_3 is lost because while it was in transit, its destination (agent A_2) has migrated from site 2 to site 3.

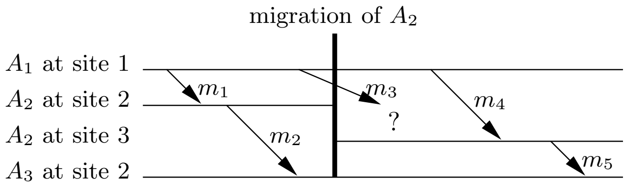

Figure 2: Inconsistency arising from migration of A_2 from site 2 to site 3

To avoid inconsistencies, three issues must be taken into account:

- *Agent naming:* references to migrating agents must be properly updated (e.g., assuming that agent names include site information, the reference to agent A_2 used by agent A_1 when sending message m_3 may become outdated after A_2 has moved from site 2 to site 3).

- *Agent states:* after an agent has been reconfigured, it must be able to resume its actual computation from its former state (e.g., agent A_2 must resume its computation on site 3 from its state on site 2 prior to migration).
- *Communication channels:* messages in transit during a reconfiguration must be preserved and properly redirected to their destination agents after reconfiguration (e.g., message m_3 should reach A_2 after A_2 has migrated to site 3).

2.3 Principles of the protocol

To ensure consistency in presence of agent migration, different approaches have been proposed, such as checkpointing [LS92] (which performs a rollback of the application to its last consistent state, on which reconfiguration is performed), forwarding techniques [PM83] (which temporarily replace a migrating agent by a *forwarder* responsible for redirecting incoming messages to the new location of the agent), and transparent protocols for location-independent communication [SWP98] (which avoid reference updates between agents by preserving agent names).

Checkpointing techniques require the additional cost of maintaining consistent distributed snapshots of the application (i.e., the agent states and the messages in transit) and of rollbacking. Forwarding techniques induce residual dependencies that may affect application reliability (e.g., in case of a forwarder failure). The Aaa agent-based middleware does not provide location-independent communications, but rather reliable communication and agent management primitives.

For these reasons, the dynamic reconfiguration protocol described in [PBR99] does not rely on these techniques. It is derived from the protocol used in Conic, but improved to take advantage of the properties (event-reaction model, asynchrony, persistency) guaranteed by the Aaa middleware. The protocol associates to each application a particular agent, named *configurator*, which is responsible for handling all reconfiguration commands. The configurator maintains a view of the application configuration (placement of agents on sites and communication routes between agents), determines if a reconfiguration command can be performed, executes the corresponding actions, and updates the configuration view accordingly. Unlike a forwarder, the configurator can handle more complex reconfiguration primitives, such as code replacement and agent deletion.

The communication infrastructure provided by the Aaa model can be seen as a logical bus that carries all messages between application agents and/or the configurator. Each agent is referenced by an address $\langle a, s \rangle$, where s is the identifier of the current site of the agent and a is the local identifier of the agent on site s. When an agent moves across different sites, its address must be updated appropriately (note that the local identifier may also change when the agent migrates to another site).

The following reconfiguration primitives are supported by the protocol: `ADD` (addition of a new agent to the application), `DELETE` (removal of an agent from the application), `MOVE` (migration of an agent to another site), `BIND` and `REBIND` (creation and modification of a communication channel between two agents). The implementation of the `REBIND`, `MOVE`, and `DELETE` primitives must avoid inconsistencies. Intuitively, when an agent is under reconfiguration, its execution must be suspended; in the event-reaction model, this can be obtained by ensuring that the agent receives no more events during its reconfiguration. The preconditions for a safe execution of the reconfiguration primitives can be summarized as follows: *all communication channels involved must be empty (i.e., must not contain any message in transit) before reconfiguration can occur.*

The dynamic reconfiguration protocol implementing these primitives can be defined using a notion of *abstract state* for application agents. At any time, an agent can be in one of the three abstract states listed in the table below.

State	Meaning
Active	The agent can execute normally and communicate with other agents according to the event-reaction model.
Passive	The agent can react to events but cannot send any event to other agents; all events that it must send are delayed until its reactivation.
Frozen	The agent does not receive any event anymore; all agents having a reference towards it are passive and the corresponding channels are empty.

During the execution of reconfiguration commands, the configurator forces certain agents into appropriate abstract states in order to preserve consistency. Roughly speaking, to reconfigure an agent A or one of its outgoing channels, the configurator implements the following protocol:

1. Compute the *Change Passive Set*, noted $cps(A)$, which contains all the agents having a communication channel directed to A: these agents must be made passive in order to freeze A. For the `REBIND` primitive, $cps(A)$ is empty, but A itself must be made passive.

2. Passivate all agents in $cps(A)$. So doing, all agents with references to A are becoming passive and all communication channels directed to A are progressively flushed. When this is complete, agent A is frozen (except in the case of `REBIND`, where A is made passive, but not frozen).

3. Send the reconfiguration command to A. The causal ordering property ensures that this command will only be received when A is frozen (although the configurator never knows exactly when A is frozen).

4. Activate all agents in $cps(A)$. Agents in $cps(A)$ that have received messages while they were passive must react to these messages as soon as they are reactivated. In the case of `REBIND`, agent A is reactivated when it receives the `REBIND` command.

3 Formal specification

In this section we give a brief overview of Lotos and then we detail the specification of the dynamic reconfiguration protocol.

3.1 Overview of LOTOS

LOTOS (*Language Of Temporal Ordering Specification*) [ISO88] is a Formal Description Technique standardized by ISO for specifying communication protocols and distributed systems. Its design was motivated by the need for a language with a high abstraction level and strong mathematical basis, which could be used for the description and analysis of complex systems. LOTOS consists of two "orthogonal" sub-languages:

The data part is based on the well-known theory of algebraic abstract data types, more specifically on the ACTONE specification language [dMRV92]. A data type is described by its sorts and operations, which are specified using algebraic equations.

The behaviour part is based on process algebras, combining the best features of CCS [Mil89] and CSP [Hoa85]. A concurrent system is usually described as a collection of parallel processes interacting by rendezvous. Each process behaviour is specified using an algebra of operators (see the table below). Processes can manipulate data values and exchange them at interaction points called *gates*.

BEHAVIOUR OPERATOR	INTUITIVE MEANING
`stop`	Do nothing.
G `!`V `?`X`:`S `;` B	Interact on gate G, sending value V and receiving in variable X a value of sort S, then execute B.
B_1 `[]` B_2	Execute either B_1 or B_2.
`[`E`] ->` B	If E is true then execute B, else do nothing.
B_1 `|[`$G_1, ..., G_n$`]|` B_2	Execute B_1 and B_2 in parallel with synchronization on gates $G_1, ..., G_n$.
B_1 `|||` B_2	Execute B_1 and B_2 in parallel without synchronization.
`exit`	Terminate successfully.
B_1 `>>` B_2	Execute B_1 followed by B_2 when B_1 terminates.
P `[`$G_1, ..., G_n$`]` `(`$V_1, ..., V_n$`)`	Call process P with gate parameters $G_1, ..., G_n$ and value parameters $V_1, ..., V_n$.

3.2 Architecture of the protocol

The architecture of the LOTOS specification (see Figure 3) consists of a configurator agent and n application agents. All agents are modelled as LOTOS processes, which execute concurrently and communicate through a software bus (an abstraction of the AAA infrastructure), which is also modeled by a LOTOS process. Agents can send and receive messages (events) via the gates `SEND` and `RECV`, respectively. The `Bus` process acts as an unbounded buffer (initially empty) accepting messages on gate `SEND` and delivering them on gate `RECV`.

Dynamic agent creation is modelled in a finite manner by considering a fixed set of `Agent` processes that initially are all "dead" (an auxiliary abstract state, noted `DEAD`, meaning that the agent is not part of the application) and will be progressively added to the application.

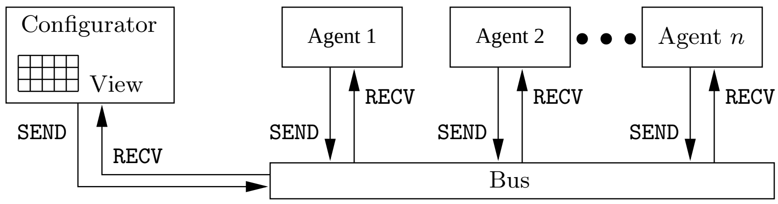

Figure 3: Architecture of the dynamic reconfiguration protocol

3.3 Configurator agent

The configurator agent is responsible for keeping track of the application configuration and for executing the reconfiguration commands coming from some external user. Since we seek to study a general behaviour of the protocol, we do not specify a particular user, letting the configurator behave as if it would receive an infinite sequence of arbitrary reconfiguration commands.

The `Configurator` process has two parameters: the application configuration `C` (initially empty) and the address set `R` of agents currently in the `DEAD` state. The configuration `C` is modelled as a list of tuples $\langle\langle a, s\rangle, A\rangle$, where $\langle a, s\rangle$ is the address of an agent present in the application and A is the set of agent addresses towards which the agent has a reference (output channels). The `Configurator` process has a cyclic behaviour: it chooses a reconfiguration command non-deterministically, executes the appropriate operations, and calls itself recursively with an updated configuration. In the following example, we only detail the `MOVE` primitive, the other reconfiguration primitives being specified similarly.

```
process Configurator [SEND, RECV] (C:Config, R:AddrSet) : noexit :=
   (* ... other reconfiguration primitives *)
   (choice A:Addr, S:SiteId []
      [(A isin C) and (getsite (A) ne S)] ->
      (let A2:Addr = newaddr (S, C) in
         Passivate [SEND, RECV] (cps (A, C)) >>
         SEND !A !confaddr !MOVE !A2 !dummy;
            RECV !confaddr !A2 !ACK !dummy !dummy;
               Activate [SEND, RECV] (A, A2, cps (A, C)) >>
               Configurator [SEND, RECV]
                  (setaddr (A, A2, setchan (cps (A, C), A, A2, C)), R)
      )
   )
endproc
```

The address `A` of the agent to be moved and its destination site identifier `S` are chosen non-deterministically. The agents in the set *cps*(`A`) are made passive by calling the auxiliary process `Passivate`. Then, a `MOVE` command is sent to agent `A`, which must respond with an acknowledgement upon completion of its migration to site `S`. The agents in *cps*(`A`) are then reactivated by calling the auxiliary process `Activate`, which also notifies them with the new address `A2` of agent `A`. Finally, the `Configurator` calls itself recursively with a modified configuration obtained from `C` by updating the address of agent `A` and the output channels of the agents in *cps*(`A`).

3.4 Application agents

Application agents execute the code of the application according to the event-reaction model and must also react to the reconfiguration commands sent by the configurator agent. Since we focus on the reconfiguration protocol itself rather than on the agent-based applications built upon it, we consider only one application-level message (called `SERVICE`) sent between agents.

The `Agent` process has four parameters: its current abstract state `S`, its current address `A`, the set `R` of agent addresses (output channels) towards which it has a reference and a boolean `B` indicating whether a message was received while it was passive (this may occur during the migration of another agent towards which the current agent has an output channel). The `Agent` process has a cyclic behaviour: it receives an event, executes the corresponding reaction according to its current abstract state `S`, and calls itself recursively with the parameters updated appropriately. In the following example, we only detail the reaction of an agent to the `MOVE` command, the other reconfiguration commands being specified similarly.

```
process Agent [SEND, RECV] (S:State, A:Addr, R:AddrSet, B:Bool):noexit:=
  (* ... other reconfiguration commands *)
  [S eq ACTIVE] ->
     RECV !A !confaddr !MOVE ?A2:Addr !dummy;
        SEND !confaddr !A2 !ACK !dummy !dummy;
           Agent [SEND, RECV] (S, A2, R, B)
  []
  [S eq PASSIVE] ->
     RECV !A !confaddr !MOVE ?A2:Addr !dummy;
     ([B] ->
       (choice A3:Addr [] [A3 isin replace (A, A2, R)] ->
        SEND !A3 !A !SERVICE !dummy !dummy;
           SEND !confaddr !A2 !ACK !dummy !dummy;
              Agent [SEND, RECV] (ACTIVE, A2, replace (A, A2, R), false)
       )
     []
     [not (B)] -> SEND !confaddr !A2 !ACK !dummy !dummy;
        Agent [SEND, RECV] (ACTIVE, A2, replace (A, A2, R), false)
     )
endproc
```

The migration is specified simply by changing the agent address. If the agent is active, it simply sends an acknowledgement with its new address `A2` back to the configurator, and then calls itself recursively with an updated address. If the agent is passive (this can happen only if it has an output channel directed to itself), it first reacts to the events received from other agents while it was passive, then sends an acknowledgement to the configurator, and finally becomes active, updating its address and its output channels.

4 Model-checking verification

To analyze the behaviour of the dynamic reconfiguration protocol, we used the CADP tool set, which we briefly present. We then express the correctness properties of the protocol and give experimental results regarding model-checking verification.

4.1 Overview of the CADP tool set

CADP (CÆSAR/ALDÉBARAN Development Package) [FGK+96] is a state-of-the-art tool set dedicated to the verification of communication protocols and distributed systems. CADP offers an integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification. In this case-study, we used the following tools of CADP:

CAESAR.ADT [Gar89] and **CAESAR** [GS90] are compilers for the data part and the control part of LOTOS specifications, respectively. They can be used to translate a LOTOS specification into a Labelled Transition System (LTS), i.e., a state-transition graph modelling exhaustively the behaviour of the specification. Each LTS transition is labelled with an action resulting from synchronization on a gate, possibly with communication of data values.

EVALUATOR 3.0 [MS00] is an on-the-fly model-checker for temporal logic formulas over LTSs. The logic considered is an extension of the alternation-free μ-calculus [EL86] with action predicates and regular expressions. The tool also provides diagnostics (examples and counterexamples) explaining the truth value of the formulas.

BCG_MIN is a tool for minimizing LTSs according to various equivalence relations, such as strong bisimulation, observational or branching equivalence, etc.

SVL 2.0 [GL01] is a tool for compositional and on-the-fly verification based on the approach proposed in [KM97]. Compositional verification is a mean to avoid state explosion in model-checking by dividing a concurrent system into its parallel components (e.g., the configurator agent, application agents, and the bus), generating (modulo some abstractions) the LTS corresponding to each component, minimizing each LTS and recombining the minimized LTSs to obtain the whole system.

4.2 Correctness properties

To express the correct behaviour of the dynamic reconfiguration protocol, we expressed a set of relevant properties about its behaviour. Two main classes of properties are usually considered for distributed systems: *safety properties*, stating that “something bad never happens", and *liveness properties*, stating that "something good eventually happens" during the execution of the system. For the dynamic reconfiguration protocol under study, we identified, together with the developers of the AAA middleware, 10 safety and liveness properties characterizing either the global behaviour of the protocol or the particular behaviour of each reconfiguration primitive. These properties are shown in the table below (the S and L superscripts indicate safety and liveness, respectively).

No.	Correctness Property
P_1^L	There is no deadlock in the specification.
P_2^L	Every reconfiguration command is eventually followed by an acknowledgement.
P_3^S	There is a strict alternation between commands and acknowledgements.
P_4^L	Every command sent to the bus is eventually delivered to its receiver.
P_5^S	Initially, no event can be sent before at least one agent has been created.
P_6^S	Initially, no application event can be sent before the underlying channel has been created.
P_7^L	Every event sent to a migrating agent will be delivered properly.
P_8^S	After a move command has been sent, the target agent cannot receive any event until it completes its migration.
P_9^S	Every event sent to a channel being rebound will be delivered before the rebind completes.
P_{10}^S	An agent that has been removed from the application cannot execute anymore.

Then, we expressed these properties in regular alternation-free μ-calculus, the temporal logic accepted by the EVALUATOR 3.0 model-checker. This logic allows to succinctly encode safety properties by using regular modalities of the form $[\mathsf{R}]\,\mathsf{F}$, which state the absence of "bad" execution sequences characterized by a regular expression R. For instance, property P_3 is encoded by the formula $[\mathsf{T}^*.\mathtt{SEND\_CMD1}.(\neg\mathtt{SEND\_ACK})^*.\mathtt{SEND\_CMD2}]\,\mathsf{F}$, where the action predicates SEND_CMD1, SEND_CMD2, and SEND_ACK denote the emission of two reconfiguration commands and of an acknowledgement, respectively.

4.3 Verification results

As model-checking verification is only applicable to finite-state models (of tractable size), we considered several instances of the protocol involving a finite number of agents, sites, and reconfiguration commands. The experimental results regarding LTS generation are shown in the table below. For each instance, the table gives the LTS size (number of states and transitions) and the time required for its generation using CADP. All experiments have been performed on a 500 MHz Pentium II machine with 768 Mbytes of memory.

As expected, the LTS size increases rapidly with the number of agents present in the instance, because the number of possible application configurations is exponential in the number of agents. Using the EVALUATOR 3.0 model-checker, we verified that all temporal properties given in Section 4.2 are valid on each instance considered. The average verification time of a property over an LTS was about one minute.

Agents	Sites	Commands	States	Trans.	Time
2	2	ADD, BIND, REBIND	77	84	9"
2	2	ADD, DELETE, BIND, REBIND	4 424	5 832	43"
2	2	ADD, BIND, REBIND, MOVE	599 474	832 864	4'25"
3	1	ADD, DELETE	493	639	15"
3	1	ADD, BIND	3 391	5 031	32"
3	1	ADD, BIND, REBIND	590 119	935 397	5'13"
3	1	ADD, BIND, MOVE	646 592	917 796	5'46"

5 Conclusion and future work

In this report, we used the Iso language Lotos [ISO88] and the Cadp verification tool set [FGK+96] to analyse a protocol for dynamic reconfiguration proposed in [PBR99] and used in the Aaa platform [BPF+99].

The Lotos specification developed (about 900 lines) provides a non-ambiguous description of the protocol and a basis for future development and experimentation of new reconfiguration primitives. Using model-checking and temporal logic, we were able to verify the correct functioning of the protocol on various configurations involving several agents, sites, and reconfiguration primitives. This experiment increased the confidence in the correctness of the protocol and demonstrated the usefulness of formal methods for agent-based applications.

Three future research directions are of interest. Firstly, to improve scalability, the protocol could be extended to use a distributed configurator instead of the current centralized solution. Secondly, the validation activity could be continued on larger configurations of the protocol and more detailed specification of the Aaa communication infrastructure. Thirdly, one could investigate the generation of test suites for the Java implementation of the protocol, by using the Tgv tool [FJJ+96] recently integrated in Cadp, which allows to automatically derive test suites from user-defined test purposes.

Acknowledgments

We are grateful to Frédéric Lang for his careful reading and valuable comments on this report, and for his assistance in using the Svl tool. The first author was partially supported by Conacyt-Sfere and Uam Iztapalapa, Mexico.

A The LOTOS specification of the protocol

This annex contains the complete Lotos specification of the dynamic reconfiguration protocol (data part and behaviour part).

A.1 Data part

```
library BOOLEAN, NATURAL endlib

(****************************************************************************
 * Agent identifier
 ****************************************************************************)

type AgentIdentifier is Boolean
   sorts
      AgentId

   opns
      agent1 (*! constructor *), agent2 (*! constructor *),
      agent3 (*! constructor *), aconf  (*! constructor *) :-> AgentId
      _eq_, _ne_, _lt_ : AgentId, AgentId -> Bool
      succ : AgentId -> AgentId
      dummyagent :-> AgentId

   eqns
      forall P1, P2:AgentId

   ofsort Bool
      P1 eq P1 = true;
      P1 eq P2 = false;

   ofsort Bool
      P1 ne P2 = not (P1 eq P2);

   ofsort Bool
      agent1 lt agent2 = true;
      agent1 lt agent3 = true;
      agent1 lt aconf = true;
      agent2 lt agent3 = true;
      agent2 lt aconf = true;
      agent3 lt aconf = true;
      P1 lt P2 = false;

   ofsort AgentId
      succ (agent1) = agent2;
      succ (agent2) = agent3;
      succ (agent3) = aconf;
      succ (aconf) = agent1;

   ofsort AgentId
      dummyagent = agent1;
endtype
```

```
(****************************************************************************
 * Site identifier
 ****************************************************************************)

type SiteIdentifier is Boolean
   sorts
      SiteId

   opns
      site1 (*! constructor *), site2 (*! constructor *) :-> SiteId
      _eq_, _ne_, _lt_ : SiteId, SiteId -> Bool
      dummysite :-> SiteId

   eqns
      forall S1, S2:SiteId

   ofsort Bool
      S1 eq S1 = true;
      S1 eq S2 = false;

   ofsort Bool
      S1 ne S2 = not (S1 eq S2);

   ofsort Bool
      site1 lt site2 = true;
      S1 lt S2 = false;

   ofsort SiteId
      dummysite = site1;
endtype

(****************************************************************************
 * Agent address
 ****************************************************************************)

type AgentAddress is AgentIdentifier, SiteIdentifier
   sorts
      Addr

   opns
      _@_ (*! constructor *) : AgentId, SiteId -> Addr
      getsite : Addr -> SiteId
      _eq_, _ne_, _lt_ : Addr, Addr -> Bool
      confaddr :-> Addr
      dummy :-> Addr
```

```
   eqns
      forall P, P1, P2:AgentId, S, S1, S2:SiteId, A1, A2:Addr

   ofsort SiteId
      getsite (P@S) = S;

   ofsort Bool
      (P1@S1) eq (P2@S2) = (P1 eq P2) and (S1 eq S2);

   ofsort Bool
      A1 ne A2 = not (A1 eq A2);

   ofsort Bool
      (P1@S1) lt (P2@S2) = (P1 lt P2) or ((P1 eq P2) and (S1 lt S2));

   ofsort Addr
      confaddr = aconf@dummysite;

   ofsort Addr
      dummy = dummyagent@dummysite;
endtype

(****************************************************************************
 * Set of agent addresses
 ****************************************************************************)

type AgentAddressSet is Natural, AgentAddress
   sorts
      AddrSet

   opns
      {}  (*! constructor *) :-> AddrSet
      _+_ (*! constructor *) : Addr, AddrSet -> AddrSet
      insert : Addr, AddrSet -> AddrSet
      remove : Addr, AddrSet -> AddrSet
      replace : Addr, Addr, AddrSet -> AddrSet
      empty : AddrSet -> Bool
      _isin_, _notin_ : Addr, AddrSet -> Bool
      _subset_ : AddrSet, AddrSet -> Bool
      _eq_, _ne_, _lt_ : AddrSet, AddrSet -> Bool
      card : AddrSet -> Nat
      pick : AddrSet -> Addr

   eqns
      forall A, A1, A2:Addr, R, R1, R2:AddrSet
```

```
ofsort AddrSet
   (* assert: elements are unique and sorted in increasing order *)
   insert (A, {}) = A + {};
   insert (A, A + R) = A + R;
   A lt A1 => insert (A, A1 + R1) = A + (A1 + R1);
   insert (A, A1 + R1) = A1 + insert (A, R1);

ofsort AddrSet
   remove (A, {}) = {};
   remove (A, A + R) = R;
   remove (A, A1 + R1) = A1 + remove (A, R1);

ofsort AddrSet
   replace (A1, A2, R) = insert (A2, remove (A1, R));

ofsort Bool
   empty ({}) = true;
   empty (A + R) = false;

ofsort Bool
   A isin {} = false;
   A isin (A1 + R1) = (A eq A1) or (A isin R1);

ofsort Bool
   A notin R = not (A isin R);

ofsort Bool
   {} subset R = true;
   (A1 + R1) subset R2 = (A1 isin R2) and (R1 subset R2);

ofsort Bool
   R1 eq R2 = (R1 subset R2) and (R2 subset R1);

ofsort Bool
   R1 ne R2 = not (R1 eq R2);

ofsort Bool
   (* assert: elements are unique and sorted in increasing order *)
   {} lt (A + R) = true;
   (A1 + R1) lt (A2 + R2) = (A1 lt A2) or ((A1 eq A2) and (R1 lt R2));
   R1 lt R2 = false;

ofsort Nat
   card ({}) = 0;
   card (A + R) = 1 + card (R);
```

```
      ofsort Addr
         (* assert: set not empty *)
         pick (A + R) = A;
endtype

(***************************************************************************
 * Reconfiguration and application commands
 ***************************************************************************)

type Command is Boolean
   sorts
      Cmd

   opns
      ACTIVATE  (*! constructor *), ACK       (*! constructor *),
      ADD       (*! constructor *), BIND      (*! constructor *),
      DELETE    (*! constructor *), FLUSH     (*! constructor *),
      MOVE      (*! constructor *), PASSIVATE (*! constructor *),
      REBIND    (*! constructor *), SERVICE   (*! constructor *) :-> Cmd
      _eq_, _ne_ : Cmd, Cmd -> Bool

   eqns
      forall D1, D2:Cmd

   ofsort Bool
      D1 eq D1 = true;
      D1 eq D2 = false;

   ofsort Bool
      D1 ne D2 = not (D1 eq D2);
endtype

(***************************************************************************
 * Abstract states of an agent
 ***************************************************************************)

type AbstractAgentState is Boolean
   sorts
      State

   opns
      ACTIVE  (*! constructor *),
      DEAD    (*! constructor *),
      PASSIVE (*! constructor *) :-> State
      _eq_, _ne_ : State, State -> Bool
```

```
   eqns
      forall T1, T2:State

   ofsort Bool
      T1 eq T1 = true;
      T1 eq T2 = false;

   ofsort Bool
      T1 ne T2 = not (T1 eq T2);
endtype

(****************************************************************************
 * Configuration of an agent (address, set of "output" agents)
 ****************************************************************************)

type AgentConfiguration is AgentAddressSet
   sorts
      AgentConfig

   opns
      _&_ (*! constructor *) : Addr, AddrSet -> AgentConfig
      _eq_, _ne_, _lt_ : AgentConfig, AgentConfig -> Bool

   eqns
      forall C1, C2:AgentConfig, R1, R2:AddrSet, A1, A2:Addr

   ofsort Bool
      (A1 & R1) eq (A2 & R2) = (A1 eq A2) and (R1 eq R2);

   ofsort Bool
      C1 ne C2 = not (C1 eq C2);

   ofsort Bool
      (A1 & R1) lt (A2 & R2) = (A1 lt A2) or ((A1 eq A2) and (R1 lt R2));
endtype

(****************************************************************************
 * Configuration of an application (list of agent configurations)
 ****************************************************************************)

type Configuration is AgentConfiguration
   sorts
      Config
```

```
opns
   nil (*! constructor *) :-> Config
   _._ (*! constructor *) : AgentConfig, Config -> Config
   insert : AgentConfig, Config -> Config
   delete : Addr, Config -> Config
   remove : Addr, Config -> Config
   getchan : Addr, Config -> AddrSet
   addchan : Addr, Addr, Config -> Config
   setchan : Addr, Addr, Addr, Config -> Config
   setchan : AddrSet, Addr, Addr, Config -> Config
   setaddr : Addr, Addr, Config -> Config
   cps : Addr, Config -> AddrSet
   _isin_, _notin_ : Addr, Config -> Bool
   newaddr : SiteId, Config -> Addr
   newaddr2 : AgentId, SiteId, Config -> Addr

eqns
   forall C, C1:AgentConfig, C1_Cn, C2_Cn:Config, A, A1, A2, A3:Addr,
          R, R1:AddrSet, P:AgentId, S:SiteId

ofsort Config
   (* assert: elements are unique and sorted in increasing order *)
   insert (C, nil) = C.nil;
   insert (C, C.C2_Cn) = C.C2_Cn;
   C lt C1 => insert (C, C1.C2_Cn) = C.(C1.C2_Cn);
   insert (C, C1.C2_Cn) = C1.insert (C, C2_Cn);

ofsort Config
   delete (A, nil) = nil;
   delete (A, (A & R).C2_Cn) = delete (A, C2_Cn);
   delete (A, (A1 & R1).C2_Cn) = (A1 & remove (A, R1)).delete (A, C2_Cn);

ofsort Config
   remove (A, nil) = nil;
   remove (A, (A & R).C2_Cn) = C2_Cn;
   remove (A, C1.C2_Cn) = C1.remove (A, C2_Cn);

ofsort AddrSet
   (* assert: A is in the configuration *)
   getchan (A, (A & R).C2_Cn) = R;
   getchan (A, C1.C2_Cn) = getchan (A, C2_Cn);

ofsort Config
   (* assert: A1 isin C1_Cn *)
   addchan (A1, A2, C1_Cn) =
      insert (A1 & insert (A2, getchan (A1, C1_Cn)), remove (A1, C1_Cn));
```

```
   ofsort Config
      (* assert: A1 isin C1_Cn *)
      setchan (A1, A2, A3, C1_Cn) =
         insert (A1 & insert (A3, remove (A2, getchan (A1, C1_Cn))),
                 remove (A1, C1_Cn));

   ofsort Config
      (* assert: each address in the set isin C1_Cn *)
      setchan ({}, A2, A3, C1_Cn) = C1_Cn;
      setchan (A + R, A2, A3, C1_Cn) =
         setchan (A, A2, A3, setchan (R, A2, A3, C1_Cn));

   ofsort Config
      (* assert: A1 isin C1_Cn *)
      setaddr (A1, A2, C1_Cn) =
         insert (A2 & getchan (A1, C1_Cn), remove (A1, C1_Cn));

   ofsort AddrSet
      cps (A, nil) = {};
      A isin R1 => cps (A, (A1 & R1).C2_Cn) = insert (A1, cps (A, C2_Cn));
      cps (A, C1.C2_Cn) = cps (A, C2_Cn);

   ofsort Bool
      A isin nil = false;
      A isin ((A1 & R1).C2_Cn) = (A eq A1) or (A isin C2_Cn);

   ofsort Bool
      A notin C1_Cn = not (A isin C1_Cn);

   ofsort Addr
      (* iterate on all agent identifiers until get a new address *)
      newaddr (S, C1_Cn) = newaddr2 (agent1, S, C1_Cn);

   ofsort Addr
      (P@S) isin C1_Cn =>
      newaddr2 (P, S, C1_Cn) = newaddr2 (succ (P), S, C1_Cn);
      newaddr2 (P, S, C1_Cn) = P@S;
endtype

(****************************************************************************
 * Messages between agents
 ****************************************************************************)

type Message is Command, AgentAddressSet
   sorts
      Msg
```

```
   opns
      message (*! constructor *) :
         Addr, (* address of the receiver agent *)
         Addr, (* address of the sender agent *)
         Cmd,  (* reconfiguration command *)
         Addr, (* first agent address sent *)
         Addr  (* second agent address sent *)
      -> Msg
      getrcv : Msg -> Addr
      getsnd : Msg -> Addr
      getcmd : Msg -> Cmd
      getad1 : Msg -> Addr
      getad2 : Msg -> Addr

   eqns
      forall A1, A2, A3, A4:Addr, D:Cmd

   ofsort Addr
      getrcv (message (A1, A2, D, A3, A4)) = A1;
      getsnd (message (A1, A2, D, A3, A4)) = A2;
      getad1 (message (A1, A2, D, A3, A4)) = A3;
      getad2 (message (A1, A2, D, A3, A4)) = A4;

   ofsort Cmd
      getcmd (message (A1, A2, D, A3, A4)) = D;
endtype

(*****************************************************************************
 * Buffer (FIFO) of messages
 *****************************************************************************)

type MessageBuffer is Message
   sorts
      Buffer

   opns
      <>  (*! constructor *) :-> Buffer
      _+_ (*! constructor *) : Buffer, Msg -> Buffer
      head : Buffer -> Msg
      tail : Buffer -> Buffer
      empty : Buffer -> Bool
      length : Buffer -> Nat

   eqns
      forall M:Msg, B:Buffer
```

```
   ofsort Msg
      (* assert: queue not empty *)
      head (<> + M) = M;
      head (B + M) = head (B);

   ofsort Buffer
      (* assert: queue not empty *)
      tail (<> + M) = <>;
      tail (B + M) = tail (B) + M;

   ofsort Bool
      empty (<>) = true;
      empty (B + M) = false;

   ofsort Nat
      length (<>) = 0;
      length (B + M) = 1 + length (B);
endtype
```

A.2 Behaviour part

```
specification RECONFIGURATION_PROTOCOL [SEND, RECV, INBUS, OUTBUS] : noexit

library DATA endlib

(******************************************************************************
 * Architecture of the protocol
 ******************************************************************************)

behaviour

   (
      Agent [INBUS, OUTBUS] (DEAD, agent1@site1, {}, false)
      |||
      Agent [INBUS, OUTBUS] (DEAD, agent2@site1, {}, false)
      |||
      Agent [INBUS, OUTBUS] (DEAD, agent3@site1, {}, false)
      |||
      Configurator [INBUS, OUTBUS] (nil, agent1@site1 + (agent2@site1 +
                                    (agent3@site1 + {})))
   )
   |[INBUS, OUTBUS]|
   Bus [INBUS, OUTBUS] (<>)

where

(******************************************************************************
 * Configurator agent
 ******************************************************************************)

process Configurator [SEND, RECV] (C:Config, R:AddrSet) : noexit :=
   (* ADD: add a new agent to the application *)
   (choice A:Addr [] [(A notin C) and (A isin R)] ->
       SEND !A !confaddr !ADD !dummy !dummy;
          RECV !confaddr !A !ACK !dummy !dummy;
             Configurator [SEND, RECV] (insert (A & {}, C), remove (A, R))
   )
   []
   (* DELETE: delete an agent from the application *)
   (choice A:Addr [] [A isin C] ->
       Passivate [SEND, RECV] (cps (A, C)) >>
          SEND !A !confaddr !DELETE !dummy !dummy;
             RECV !confaddr !A !ACK !dummy !dummy;
                Activate [SEND, RECV] (A, A, cps (A, C)) >>
                   Configurator [SEND, RECV] (delete (A, C), insert (A, R))
   )
```

```
   []
   (* BIND: add a new output channel to an agent *)
   (choice A2:Addr [] [A2 isin C] ->
       (choice A3:Addr []
        [(A3 isin C) and (A2 ne A3) and not (A3 isin getchan (A2, C))] ->
          SEND !A2 !confaddr !BIND !A3 !dummy;
             RECV !confaddr !A2 !ACK !dummy !dummy;
                SEND !A3 !A2 !SERVICE !dummy !dummy;
                   Configurator [SEND, RECV] (addchan (A2, A3, C), R)
       )
   )
   []
   (* REBIND: change an existing communication channel *)
   (choice A:Addr [] [A isin C] ->
       (choice A2:Addr []
        [(A2 isin C) and (A2 notin getchan (A, C)) and (A ne A2)] ->
           (choice A1:Addr []
            [A1 isin getchan (A, C)] ->
               SEND !A !confaddr !PASSIVATE !dummy !dummy;
                  RECV !confaddr !A !ACK !dummy !dummy;
                     SEND !A1 !confaddr !FLUSH !dummy !dummy;
                        RECV !confaddr !A1 !ACK !dummy !dummy;
                           SEND !A !confaddr !REBIND !A1 !A2;
                              RECV !confaddr !A !ACK !dummy !dummy;
                                 Configurator [SEND, RECV]
                                              (setchan (A, A1, A2, C), R)
           )
       )
   )
   []
   (* MOVE: move an agent to another site *)
   (choice A:Addr [] [A isin C] ->
       (choice S:SiteId []
          (let A2:Addr = newaddr (S, C) in
             [A2 ne confaddr] ->
             (* new valid address *)
                Passivate [SEND, RECV] (cps (A, C)) >>
                   SEND !A !confaddr !MOVE !A2 !dummy;
                      RECV !confaddr !A2 !ACK !dummy !dummy;
                         Activate [SEND, RECV] (A, A2, cps (A, C)) >>
                            Configurator [SEND, RECV] (setaddr (A, A2,
                                         setchan (cps (A, C), A, A2, C)), R)
          )
       )
   )
endproc
```

```
(*************************************************************************
 * Auxiliary process for making passive a set of agents
 *************************************************************************)

process Passivate [SEND, RECV] (AS:AddrSet) : exit :=
   [card (AS) > 0] ->
   (let A:Addr = pick (AS) in
      SEND !A !confaddr !PASSIVATE !dummy !dummy;
         RECV !confaddr !A !ACK !dummy !dummy;
            Passivate [SEND, RECV] (remove (A, AS))
   )
   []
   [card (AS) = 0] ->
      exit
endproc

(*************************************************************************
 * Auxiliary process for making active a set of agents
 *************************************************************************)

process Activate [SEND, RECV] (A1, A2:Addr, AS:AddrSet) : exit :=
   [card (AS) > 0] ->
   (let A:Addr = pick (AS) in
      SEND !A !confaddr !ACTIVATE !A1 !A2;
         RECV !confaddr !A !ACK !dummy !dummy;
            Activate [SEND, RECV] (A1, A2, remove (A, AS))
   )
   []
   [card (AS) = 0] ->
      exit
endproc

(*************************************************************************
 * Application agent
 *************************************************************************)

process Agent [SEND, RECV] (S:State, A:Addr, R:AddrSet, B:Bool) : noexit :=
   [S eq DEAD] ->
      RECV !A !confaddr !ADD !dummy !dummy;
         SEND !confaddr !A !ACK !dummy !dummy;
            Agent [SEND, RECV] (ACTIVE, A, {}, false)
   []
   [S eq ACTIVE] ->
```

```
(
   (* receive an application event *)
   RECV !A ?A1:Addr !SERVICE !dummy !dummy [A ne A1];
   (
      [not (empty (R))] ->
         (choice A2:Addr []
          [A2 isin R] ->
             (* react to the event *)
             SEND !A2 !A !SERVICE !dummy !dummy;
                Agent [SEND, RECV] (S, A, R, B)
         )
      []
      (* silently ignore the event *)
      Agent [SEND, RECV] (S, A, R, B)
   )
   []
   RECV !A !confaddr !BIND ?A2:Addr !dummy [(A ne A2) and (R eq {})];
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (S, A, insert (A2, R), B)
   []
   RECV !A !confaddr !PASSIVATE !dummy !dummy;
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (PASSIVE, A, R, B)
   []
   RECV !A !confaddr !MOVE ?A2:Addr !dummy;
      SEND !confaddr !A2 !ACK !dummy !dummy;
         Agent [SEND, RECV] (S, A2, R, B)
   []
   RECV !A !confaddr !FLUSH !dummy !dummy;
      SEND !confaddr !A !ACK ! dummy !dummy;
         Agent [SEND, RECV] (S, A, R, B)
   []
   RECV !A !confaddr !DELETE !dummy !dummy;
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (DEAD, A, R, B)
)
[]
[S eq PASSIVE] ->
(
   (* receive and store an application event *)
   RECV !A ?A1:Addr !SERVICE !dummy !dummy [A ne A1];
      Agent [SEND, RECV] (S, A, R, true)
   []
   RECV !A !confaddr !ACTIVATE ?A1:Addr ?A2:Addr [(A ne A2) and (A1 eq A2)];
   (* agent A1 has been deleted *)
   (
```

```
   [B] ->
   (
      (choice A3:Addr []
       [A3 isin remove (A1, R)] ->
          (* react to the event received when the agent was passive *)
          SEND !A3 !A !SERVICE !dummy !dummy;
             SEND !confaddr !A !ACK !dummy !dummy;
                Agent [SEND, RECV] (ACTIVE, A, remove (A1, R), false)
      )
      []
      (* silently ignore the event *)
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (ACTIVE, A, remove (A1, R), false)
   )
   []
   [not (B)] ->
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (ACTIVE, A, remove (A1, R), false)
)
[]
RECV !A !confaddr !ACTIVATE ?A1:Addr ?A2:Addr [(A ne A2) and (A1 ne A2)];
(
   [B and not (empty (R))] ->
   (
      (choice A3:Addr []
       [A3 isin replace (A1, A2, R)] ->
          (* react to the event received when the agent was passive *)
          SEND !A3 !A !SERVICE !dummy !dummy;
             SEND !confaddr !A !ACK !dummy !dummy;
                Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
      )
      []
      (* silently ignore the event *)
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
   )
   []
   [not (B) or empty (R)] ->
      SEND !confaddr !A !ACK !dummy !dummy;
         Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
)
[]
```

```
      RECV !A !confaddr !REBIND ?A1:Addr ?A2:Addr [A ne A2];
      (
         [B and not (empty (R))] ->
         (
            (choice A3:Addr []
             [A3 isin replace (A1, A2, R)] ->
                (* react to the event received when the agent was passive *)
                SEND !A3 !A !SERVICE !dummy !dummy;
                   SEND !confaddr !A !ACK !dummy !dummy;
                      Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
            )
            []
            (* silently ignore the event *)
            SEND !confaddr !A !ACK !dummy !dummy;
               Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
         )
         []
         [not (B) or empty (R)] ->
            SEND !confaddr !A !ACK !dummy !dummy;
               Agent [SEND, RECV] (ACTIVE, A, replace (A1, A2, R), false)
      )
   )
endproc

(****************************************************************************
 * Software bus (communication medium)
 ****************************************************************************)

process Bus [INBUS, OUTBUS] (B:Buffer) : noexit :=
   INBUS ?R:Addr ?S:Addr ?D:Cmd ?A1:Addr ?A2:Addr;
      Bus [INBUS, OUTBUS] (B + Message (R, S, D, A1, A2))
   []
   [not (empty (B))] ->
   (let M:Msg = head (B) in
      OUTBUS !getrcv (M) !getsnd (M) !getcmd (M) !getad1 (M) !getad2 (M);
         Bus [INBUS, OUTBUS] (tail (B))
   )
endproc

endspec
```

References

[BD93] T. Bloom and M. Day. Reconfiguration and Module Replacement in Argus: theory and practice. *Software Engineering Journal*, pages 102–108, March 1993.

[BPF+99] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. In *Proceedings of the Symposium on Reliable Distributed Systems SRDS'99 (Lausanne, Suisse)*, October 1999.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[EL86] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 1st LICS*, pages 267–278, 1986.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FJJ+96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, and César Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Verlag, August 1996. Also available as INRIA Research Report RR-2987.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[GL01] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[KM89] J. Kramer and J. Magee. Constructing Distributed Systems in CONIC. *IEEE Transactions on Software Engineering*, SE–15(6):663–675, June 1989.

[KM90] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, pages 1293–1306, November 1990.

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[LBBK01] P. Laumay, E. Bruneton, L. Bellissard, and S. Krakowiak. Preserving Causality in a Scalable Message-Oriented Middleware. C3DS 3rd Year Report Deliverable, ESPRIT Long Term Research Project no. 24962, 2001.

[LS92] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proceedings of the USENIX Winter Conference (San Francisco, USA)*, pages 283–290, 1992.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MS00] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, April 2000. Also available as INRIA Research Report RR-3899.

[PBF+00] N. De Palma, L. Bellissard, D. Féliot, A. Freyssinet, M. Herrmann, and S. Lacourte. The AAA Agent-based Message Oriented Middleware. C3DS Public Technical Report Series 30, ESPRIT Long Term Research Project no. 24962, 2000.

[PBR99] N. De Palma, L. Bellissard, and M. Riveill. Dynamic Reconfiguration of Agent-Based Applications. In *Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems ERSADS'99 (Madeira Island, Portugal)*, April 1999.

[PM83] M. L. Powell and B. P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 6th ACM Symposium on Operating System Principles*, pages 110–119, 1983.

[Pur94] J. M. Purtilo. The POLYLITH Software Bus. *IEEE Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[RST91] M. Raynal, A. Schiper, and S. Toueg. The Causal Ordering Abstraction and a Simple Way to Implement It. *Information Processing Letters*, 39(6):343–350, 1991.

[SWP98] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-Independent Communication for Mobile Agents: A Two-Level Architecture. In *Proceedings of ICCL'98 (Chicago, USA)*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer Verlag, 1998.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399