SPECIFICATION AND VERIFICATION OF A DYNAMIC RECONFIGURATION PROTOCOL FOR AGENT-BASED APPLICATIONS

Manuel Aguilar Cornejo[*], Hubert Garavel,
Radu Mateescu, and Noël de Palma
INRIA Rhône-Alpes, 655, avenue de l'Europe, F-38330 Montbonnot St. Martin, France
Manuel.Aguilar@imag.fr, {Hubert.Garavel, Radu.Maleescu, Noel.De-Palma}@inria.fr

Abstract Dynamic reconfiguration increases the availability of distributed applications by allowing them to evolve at run-time. This paper 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.

Keywords: compositional verification, distributed application, dynamic reconfiguration, LOTOS, mobile agent, model-checking, specification

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.

[*]This author was partially supported by CONACYT-SFERE and UAM Iztapalapa, Mexico.

Dynamic reconfiguration has been studied and implemented in various middlewares, such as CONIC [13], ARGUS [2], and POLYLITH [23]. 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 paper studies the protocol for dynamic reconfiguration of agent-based applications defined in [20], which follows the latter approach. This protocol has been implemented in the middleware platform AAA (*Agents Anytime Anywhere*) [1, 21], 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 [21]. 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 [20] and a JAVA implementation that was already in use, we produced a formal specification of the protocol using the ISO Formal Description Technique LOTOS [12]. 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 [3]; verification was carried out using CADP [6], a protocol engineering tool set providing state-of-the-art compilation, simulation, and verification functionalities.

This article 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. Finally, Section 5 discusses the results and gives directions for future work.

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 [1], 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 [1] listed in the table below. The dynamic reconfiguration protocol relies upon some of these properties, and especially the *causality* property (also called *causal ordering*) [24, 16].

	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.

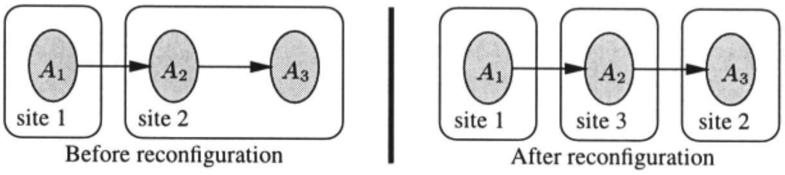

Figure 1. Migration of agent A_2 from site 2 to site 3

Dynamic reconfiguration must preserve *consistency* [14]: 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.

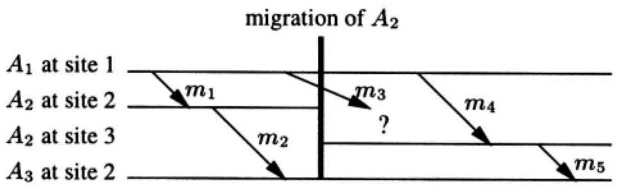

Figure 2. Inconsistency arising from migration of A_2 from site 2 to site 3

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 [17] (which performs a rollback of the application to its last consistent state, on which reconfiguration is performed), forwarding techniques [22] (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 [25] (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 [20] 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*) [12] 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 [4]. 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 [19] and CSP [11]. 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 $G_1, ..., G_n$ (`|||` means no 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.

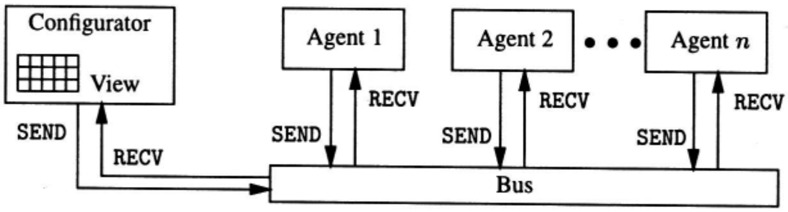

Figure 3. Architecture of the dynamic reconfiguration protocol

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.

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)
      )
   )
end proc
```

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) [6] 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 [8] and **CAESAR** [10] 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 [18] 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 [5] 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 [9] is a tool for compositional and on-the-fly verification based on the approach proposed in [15]. 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 `[R] 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 `[T*.SEND_CMD1.(-SEND_ACK)*.SEND_CMD2] 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.

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”

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.

5. CONCLUSION AND FUTURE WORK

In this paper, we used the ISO language LOTOS [12] and the CADP verification tool set [6] to analyse a protocol for dynamic reconfiguration proposed in [20] and used in the AAA platform [1].

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 [7] 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 paper, and for his assistance in using the SVL tool.

References

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

[2] T. Bloom and M. Day. Reconfiguration and Module Replacement in Argus: Theory and Practice. *Soft. Eng. Journal,* 8:102–108, 1993.

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

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

[5] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Prepositional Mu-Calculus. In *Proc. of LICS'86*, pp. 267–278.

[6] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDÉBARAN Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. of CAV'96 (New Brunswick, NJ, USA)*, LNCS vol. 1102, pp. 437–440.

[7] J-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In R. Alur and T. A. Henzinger, editors, *Proc. of CAV'96 (New Brunswick, NJ, USA)*, LNCS vol. 1102, pp. 348–359.

[8] H. Garavel. Compilation of LOTOS Abstract Data Types. In S. Vuong, editor, *Proc. of FORTE'89 (Vancouver, Canada),* pp. 147–162. North-Holland, 1989.

[9] H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proc. of FORTE'01 (Cheju Island, Korea)*, Kluwer Academic, 2001.

[10] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proc. of PSTV'90 (Ottawa, Canada)*, Kluwer Academic, pp. 379–394.

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

[12] ISO/IEC. LOTOS–A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Int. Std. 8807, Iso, Genève, 1988.

[13] J. Kramer and J. Magee. Constructing Distributed Systems in CONIC. *IEEE Trans. on Soft. Eng.,* 15(6):663–675, 1989.

[14] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. on Soft. Eng.,* pp. 1293–1306, 1990.

[15] J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proc. of TACAS'97 (Enschede, The Netherlands),* LNCS vol. 1217.

[16] 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 (http://www.newcastle.research.ec.org/c3ds), 2001.

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

[18] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proc. of FMICS'2000 (Berlin, Germany)*, GMD Report 91, pp. 65–86. Full version available as INRIA Research Report RR-3899.

[19] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[20] N. De Palma, L. Bellissard, and M. Riveill. Dynamic Reconfiguration of Agent-Based Applications. In *Proc. of ERSADS'99 (Madeira Island, Portugal)*, 1999.

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

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

[23] J. M. Purtilo. The POLYLITH Software Bus. ACM TOPLAS, 16(1):151–174, 1994.

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

[25] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-Independent Communication for Mobile Agents: A Two-Level Architecture. In *Proc. of ICCL'98 (Chicago, USA)*, LNCS vol. 1686, pp. 1–31.