TESTING MOBILE AGENTS

SAM: a tool based on a simulation approach

Mikael Marche, Yves-Marie Quemener, Roland Groz
France Télécom R&D/DTL/MSV,
2 Avenue Pierre Marzin, F-22307 Lannion CEDEX, France,
Tel +33 2 96 05 20 12
mikael.marche, yvesmarie.quemener, roland.groz @rd.francetelecom.com

Abstract For years to come, we can predict an important increase in the use of software systems using mobile code. Those systems give birth to new development problems compared to current distributed systems. In particular, testing for such systems must take into account the mobility of tested processes, which makes it difficult to maintain contact points between testing and tested processes. We propose a solution based on the simulation of the environment of mobile processes. For developing such a simulator, we need to abstract the notions found in different platforms enabling the execution of mobile processes, by giving a common API between those platforms and our simulator.

Keywords: Mobile code, mobile agents, testing, simulation, MASIF

1. INTRODUCTION

The changing nature of distributed computing, the availability of common middleware, the adaptability of software code to various environments due to the variety of terminals and the interlinking of servers on the Internet have raised the interest in the mobility of code over networks.

Just as the need for interoperability in distributed networks raised interest for testing protocols in the 1980's, the need for solutions in testing new types of distributed communicating software based on mobile code should come up rapidly. We found that software developers in our environment using e.g. agent technology were lacking test methods and tools, because very little research has been done on the specific problems implied by code mobility.

Various types of mobility [9] may be considered, from applets and code downloading, to full mobility of the code along with its execution

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35497-2_31
I. Schieferdecker et al. (eds.), *Testing of Communicating Systems XIV*

context such as in the agent paradigm and script languages in the sense initiated by Telescript [18]. All notions of mobility rely on the fact that a given software code may be executed on different hardware execution environments over time. In the most static form, the code may simply be moved along with the support on which it is stored, such as a smart card, for instance the SIM card of a GSM telephone handset. It can also be stored on a laptop or a handheld which may move to different locations on a network and interact with other software processes depending on the location from where it accesses the network. In a more dynamic form, no physical support moves, but the code is transfered over a network from one location to another. Of course, both types of mobility can be combined. And then, we have to take into account the nature of the code we are talking about: is it source code, bytecode to be interpreted or a compiled binary? Does it include the execution contexts, or at least dynamic bindings? Depending on which type of mobility we are considering, the test of software applications based on mobile code may entail different problems.

In this paper, we address the problem of testing an application based on a rich notion of mobility, where

- the execution context is moved along with the code;
- moves can be triggered by the code itself.

The first clause specifies that we are moving not just code, but processes, and the second clause further specifies that we are in fact dealing with autonomous processes. These characteristics correspond to what can be found in the mobile agent paradigm [5, 9]. Typically, an agent A would transfer its own execution from a given location S_1 to move to location S_2 where it would resume processing. Moving to S_2 might typically be decided by agent A to access specific resources available on S_2. In order to work on a relevant set of concepts with some potential impact on the development of telecommunication services, we decided to start from existing standards defining an infrastructure for mobile agents. Although FIPA [8] proposes a standard interface for interactions with agents, it does not really address mobility issues. Therefore we started mainly from the OMG MASIF specifications [15], although we consider also the primitives offered by various platforms offering an agent-based infrastructure which may be more or less compliant to MASIF.

Of course, applications based on mobile code are distributed software applications, and much of the theory and practice on software testing, as well as the theory of testing in distributed contexts (such as embodied in the ITU Z.500 standard for instance) is relevant. But there are

specific problems in testing mobile applications, and we are interested in addressing those problems.

Although the previous research orientation in our group was mainly towards test generation [11], in the case of mobile code we consider that the problems related to test execution should be solved first, before we move to higher degrees of test design. It is as yet unclear in our context of telecommunication services whether there would be a strong case for testing conformity of mobile agents to a given specification, or whether testing is well suited to address security issues such as the use of local resources by an incoming agent, or the protection of the integrity agents from their execution environment.

We identified three main problems raising specific issues in test execution when the Implementation Under Test (IUT) is a mobile process.

First, it is clear that what can be tested, in any setting, depends on what can be observed. In a mobile setting, the IUT interacts with an execution platform which provides mobility and other services. Those actions are the ones which have to be tested, but they are typically not observable in common platforms such as Grasshopper [1] and Jade [2]. *Our first problem will be to make the interactions of the IUT with its execution platform observable.*

Second, the interaction between a tester or test system and the IUT goes through so-called Points of Control and Observation (PCOs). When the IUT is a mobile process, PCOs are attached to a mobile entity, i.e. they have to be dynamically modified after each migration of the IUT. It should be noted that PCOs are defined through primitives offered by execution platforms, which implies that the first problem of non-observability of those platforms have also to be taken into account here. *So the first problem, observability, has to be solved by adding PCOs to the system, and the second problem is that those PCOs are dynamic.*

Last, it is clear that a mobile process will be executed in several locations, existing in various configurations. Hence, the test environment should offer a way to define easily various configurations, in a tightly controlled way. In that way, tests would be able to validate an IUT sufficiently enough by defining the different configurations with which the IUT is bound to interact. Contrary to traditional network testing, where the number of configurations to address in a test is limited and static, so that it can be described enumeratively, *mobile code should be tested against a rich set of dynamic configurations.*

To solve those problems, we decided to go for an approach based on simulation of the environment representing the locations and services typically offered by a mobile agent platform. This is a natural choice as it provides the needed level of observability, control of the configurations

and some control over the execution. We have implemented those ideas in a tool called SAM.

The rest of the paper is organized as follows. In section 2, we present the notion of code mobility we retain, and the MASIF framework. In section 3, we describe with more details the problems raised by testing mobile code and the specific issues we address. Section 4 describes our simulation tool, and how it helps in solving the problems of testing mobile processes. We conclude with a few results about our implementation.

2. CODE MOBILITY FRAMEWORK CONSIDERED

Simple mobile code techniques are based on code on demand and remote evaluation (e.g. applets, RMI). This results in moving static code and/or data. As mentioned in the introduction, we consider in this paper a richer form of mobility corresponding to mobile agent techniques which bring everything together and enable the autonomous migration of active processes under execution. This concept has been introduced by Telescript [18] to reduce network load and latency and to suit temporary network connectivity.

The technology supporting mobile agent takes a perspective which differs from classical distributed systems. In a mobile paradigm, as shown in Figure 1, the structure of the underlying computer network consists of execution sites layered upon the operating system. The purpose of these execution sites is to provide an execution context to agent applications which are able to dynamically relocate their execution. The concept of mobile agent raises two significant problems, related for the first one to the heterogeneity of the host systems (operating system and hardware) and for the second one to the management of the execution environment. More precisely, an environment for mobile agents must bring a solution to the following points: portability and code migration; mobility management; communication management; resources management; security management.

Portability and code migration problems are solved by the use of *MCLs(mobile code languages)* [6, 16], particularly adapted to the mechanisms of mobility. Indeed, these languages such as Java, Obliq, Objective Caml, Tcl, etc, are interpreted and offer a *runtime* allowing code insulation and migration, although some languages, for example Java, propose only a serialization mechanism of the code. In this case, the agent's state is not completely restored after its migration.

The other problems which are linked to the agent's execution management (migration, communication, authorization, ...) are solved in an

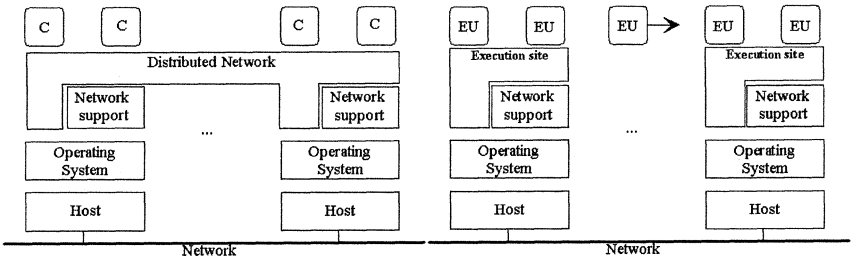

Figure 1. Classical distributed structures vs Mobile distributed structures

applicative way. A uniform solution is to consider the execution sites as a service-providing environment. The agent in this environment reaches the various services it needs by calling methods of an application which plays the role of the execution site.

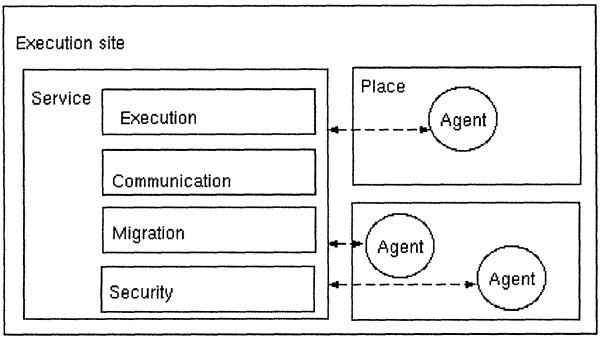

Figure 2. Hierarchical structure of the components (MASIF)

MASIF [15] clearly defines this hierarchy between agent and execution sites, as shown in Figure 2. However, MASIF is limited to interoperability between execution sites. Hence, it addresses the site interface and concepts, but not the problems of language interoperability and the standardization of agents' actions, such as migration and communication. So, it can be said that MASIF defines the interfaces at the execution site level rather than at agent level. MASIF standardizes agent management primitives (creation, execution, suspension, termination..), agent transfert primitives (sending and reception) and syntaxes and terms defining for example agent and execution site names, their characteristics and their properties.

In the rest of this paper, our study of test for mobile process is based on the MASIF framework. In this framework, agents encapsulated by an execution unit act on behalf of a person and are executed in logical places hosted by an execution site. Agents have the ability to move from place to place and to communicate with other agents to perform their computation task.

3. TESTING ISSUES ADDRESSED

3.1 Testing problems addressed

Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Testing a software system built on agents can be addressed from different points of views. Our choices were motivated by the current concerns of developers of telecommunication services based on this technology. They can be summed up as follows.

We address black-box testing of agents The test system interacts with agents only through their interfaces, either with the "upper" environment (consisting of other agents, user requests...) or with the underlying platform.

We are dealing with systems of multiple interacting agents In a sense, at that level, we are doing grey-box testing as we have access to interactions between agents and platforms. We are not testing agent platforms.

We concentrate on *functional* testing of those systems. We are not addressing the specific problems of other types of tests, such as load and stress tests. Also, our framework is not that of conformance testing as we are in a situation where there would be well defined reference specifications to which we would have to compare several implementations.

We address functional *interoperability* testing beetwen agents. One key point is that we consider software built on multiple agents running on different types of platforms, languages and paradigms. Therefore we tackle specifically the problems raised by *interoperability* (functional) testing of multiple platforms, interconnected through a MASIF framework.

We do not address the problem of test design, test selection. On the other hand, we provide a basis for expressing tests to be run on our testing environment. We provide a test language, which is abstract in the sense that it provides means of expressing tests which are independent from the platforms, and can adapt to various types of platforms.

Finally, in our research, we concentrate on features specific to mobile code. Although multi-agent systems are distributed systems, we

do not deal with problems already studied in the considerable amount of research done for distributed systems, such as synchronization of testers, timing of events.

Capitalizing on the advances done in the theory of network testing e.g. [17], we first tried [13] to give a semantic basis to the test of mobile processes by extending IOLTS with a notion of location and atomic mobility actions. This amounted to suppose that the mobility of processes is part of their observable behaviour. We then showed it was possible to automatically generate sound tests which take into account mobility. However we did not continue this line of work, because this preliminary study showed that migration actions must be observable if we want testing of mobile processes to be significant. Unfortunately, this is not true for common execution platforms. This is what sparked our interest in the problems of test execution for mobile processes.

3.2 Test environment: our choice of simulation

Mobile agents depend upon their execution platforms for their execution, and specially for executing migration and communication actions. Those execution platforms are typically neither controllable nor observable: for example, migration is defined as a primitive service of the platforms, and are executed through system calls to the platform. So, the question for testing mobile agents is how to establish so-called Points of Control and Observation (PCOs)?

Testing mobile agents makes it necessary to define a testing context which will be able to fulfill two different goals. First, it will be used for the execution of mobile agents; second, it will enable the observation of those agents and their controllability.

A first solution could be to *instrument* existing execution platforms. Execution of mobile agents implies calls to platform primitives, which could be made observable and controllable by modifying the platform itself. This raises a few problems. First, such an instrumentation would probably require to have access to the source code of the execution platform, which is generally not available. Second, a different instrumentation work should be made for each different platform. And last, tests written for one mobile agent executing on a given platform could not be directly reused if this agent was rewritten for executing on another platform, because the tests would be written for observing (and controlling) actions which are executed as primitives specific to each platform.

Similar problems of control and observation exist in the field of embedded software. Indeed, as soon as software is placed inside an embedded environment, it is very difficult to observe it thoroughly. Hence, prior

testing is made using a simulator which will emulate the embedding context of such software. A simulator makes it easy to test the software in various different contexts which correspond to the different possible embedding contexts. Likewise, in the context of testing mobile agents, we also propose to use a simulation tool.

This tool should incorporate what are the common points between different execution platforms, in that way offering a generic context for testing mobile agents. A standard such as MASIF [15] can be used as a starting point to identify common concepts between different platforms. Mappings between specific execution platforms and this generic simulation tool will have to be implemented. In that way, mobile agents designed for a specific platform could be executed without modification. Their calls to primitives of the platform would in fact be directed to the simulation tool through the mappings defined for each platform. Such calls would of course be made controllable and observable, enabling the execution of tests on the mobile agent coupled with the simulator. This also would enable to reuse tests when rewriting mobile agents for different platforms, since tests will not be attached to a given platform but could be defined directly in terms of the simulator's primitives.

We implemented those principles in a tool called SAM (*Simulation d'Agents Mobiles*), presented in detail in the following section.

4. A TOOL FOR TESTING MOBILE AGENT SYSTEMS

4.1 Goals and overview

Our goal is to offer an observable and controllable environment for executing systems made of mobile agents. Those multi-agent systems could then be tested, once they are made observable and controllable.

We want to be able to test systems targetting various execution platforms. We use MASIF and FIPA standards for defining a common model, generic enough for encompassing the maximum number of existing execution platforms. We describe in 4.2 the general architecture of our execution platform, and how it enables to execute agents defined for other platforms, and we see in 4.3 the different services offered to mobile agent systems by SAM.

Tests will have to be executed in this multi-agent setting, and for preserving the unity of this framework, we decide to consider test programs themselves as multi-agent systems. We defined our own test language, based on ideas originating in the notion of observers [10], using as starting point the reactive script language [4]. We call such test programs observation agents. We describe these notions with more detail in 4.4.

4.2 General architecture

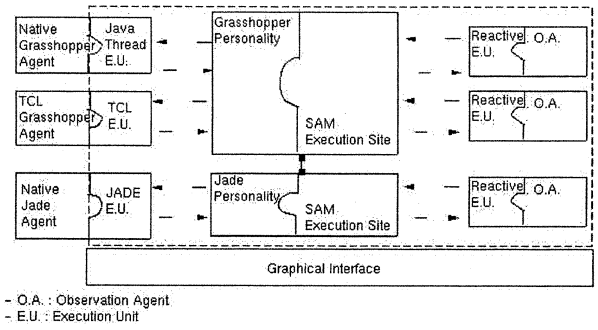

Figure 3. Tool architecture

The general architecture of our tool is described on Figure 3. This shows an example of what kind of system could be executed and tested using SAM. On the one hand, the Implementation Under Test is made of three agents executing and possibly migrating between two execution sites. One of the site is a simulated Grasshopper platform, with one native Grasshopper agent and a TCL one. The other site is a simulated Jade platform. On the other hand, the test program is made of three observation agents, written in our own test language. All those agents can migrate, and communicate with each other, and the observation agents can moreover react to specific actions of the IUT such as specific migrations or communications. For such an interaction to take place, without ever modifying the initial agents comprising the IUT, it is necessary to devise an architecture where different adapters can fit.

4.2.1 Generic execution sites. The main module defined by SAM are the execution sites. They are named environments that can create, interpret, execute, transfer and terminate agents. More generally, they offer all services (or primitives) defined by the MASIF and FIPA standards to all agents requesting them. As a starting point for our implementation, we used an user-custimizable MASIF implementation designed at France Telecom R&D, called MobiliTools [7].

SAM is designed to make the calls to those primitives observable by the observation agents. For this, SAM generates for each call an event in the execution runtimes of observation events. Those runtimes are synchronized virtual machines which enable to have events known simultaneously to all observation agents. See 4.4 for more details on this

mechanism. Execution sites also have to manage the presence of agents, and for example they give access to management data such as the number of agents executing at a given moment.

Those generic sites can be customized by what we call personalities. Personalities propose a set of entry points making it possible to define specific execution environments. In this way, generic sites can offer interfaces similar to those provided by existing platforms.

4.2.2 Personalities. Personalities are adapters which enable generic sites to be seen as already existing platforms such as Grasshopper. They realize a mapping between the primitives specific to a given platform and those offered by the generic sites. For developing those personalities, we are helped by the emergence of standards such as MASIF. The concepts behind different platforms are very close to those defined for our generic sites, and the mappings are often only syntactic.

More than a simple adapter, a personality can encode the specificities of the simulated platform by a call-back mechanism. A call-back defines which procedures are invoked on the personality when one of the primitives of the generic execution site is performed. This call-back can modify the generic execution.

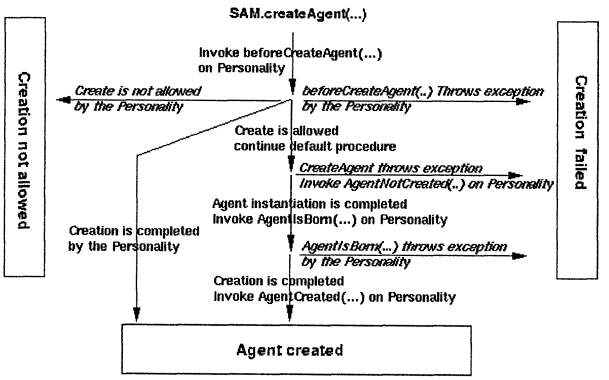

Figure 4. Call-back mechanism on personality: agent creation procedure

For example, when creating an agent (see Figure 4), the personality is notified by an agent creation request. The personality can then execute the request and return the reference of the created agent; refuse the action; or accept the default creation procedure initiated by SAM. In the last case, SAM carries out the creation action and notifies again the personality through the call-back *agentIsBorn*. The personality can

then modify the reference of the created agent and realize some initializations on the agent or throw an exception to stop the initiated creation procedure.

4.2.3 Abstract execution unit. SAM does not handle directly the executed agents but abstracts them to an entity which we name execution unit. Indeed, since an agent represents the sequential flow of computation of a process, according to its type the execution and the control of the process can be basically different. For example, agent run-time can be a Java or C++ thread, and the agent activity can be controlled by the agent (reactive control) or by the execution site (proactive control). Thus, to integrate those different models, SAM does not enforce any execution and activity model on the level of execution sites, and only gives entry points for implementing them. Section 4.3.1 gives an example of interaction between SAM and execution units.

Just as personalities, an execution unit is an adapter which enable generic sites to manage an agent. This adapter mainly consists of callbacks related to the management of agents. For example, when an agent should be suspended, the generic execution site calls the execution unit to suspend it, then the execution unit computes or not the action and returns the result. Thus, the interface representing the abstraction of the execution unit defines the runtime of the agent implementation.

We choose to move the execution environment of the agent on agent level, because it makes it possible to dissociate the agent behaviour linked to the functionalities of the simulated system of the mechanism which execute and manage it. For example, as Figure 3 shows, agents implemented in different languages can be executed on a same site because they offer the same interface through the execution unit.

4.2.4 Graphical interface. We developed a simple interface for accessing SAM. It enables to visualize the existing execution sites, and which agents execute on them. It also provides a little shell language for enabling a human tester to manage the system under test (e.g. launching and destroying agents as required by the tests).

4.3 Services

4.3.1 Activity management. Activity management in SAM is related to the agent life-cycle defined in MASIF. An agent can be activated, suspended, resumed and finished. As we saw before, activity management and run-time of the agents are moved on execution unit level. This one being informed by the execution site of a change of

activity can compute it. Let us note that the personality can also refuse the change of activity (see Figure 5).

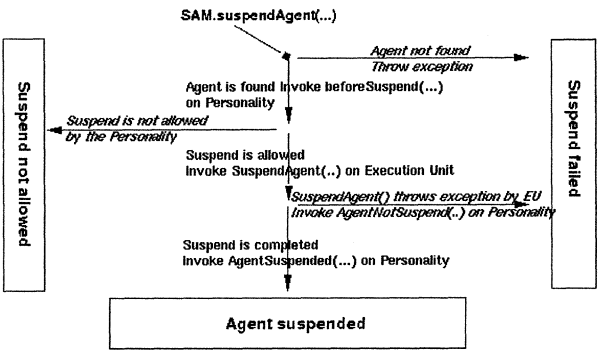

Figure 5. Call-back mechanism: suspend activity procedure

4.3.2 Communication. The communication mechanism implemented in SAM is independent from a specific platform, and conforms to the FIPA model. Agents can send messages to named mailboxes. Those (unique) names can be acquired during communication with other agents or by a naming service offered by generic execution sites. Also, agents can create mailboxes. Those mailboxes can be configured in one of two modes. In store mode, all incoming messages are stored, and the agent retrieve the stored messages by calling a designated primitive. In forward mode, the agent is notified of all incoming messages. A dedicated mechanism ensures this forward even if the agent has left the site. Mailboxes can migrate with their creating agents, following the policy defined by the personnality of the execution site (for example, this is not the case with Jade personalities). This general mechanism enables to implement a variety of synchronous or asynchronous communication methods, such as the method-call mechanism used in Grasshopper.

Finally, let us note that a particular communication mechanism (described in section 4.4) exists between observation agents.

4.3.3 Mobility. On the base of MASIF primitives, we have implemented two kinds of mobility: mobility by cloning, where a clone of the agent moves and simple agent migration.

Migration induces the backup and the restoration of the agent state, but this mechanism is closely related to the runtime of the agent. Thus, as for activity management, mechanisms of code insulation in an ap-

propriate SAM structure are moved to execution unit level, and SAM enables the migration of this structure. So, SAM does not enforce any mobility level and remains generic. Since most agent platforms are implemented in Java, SAM also proposes in a native way the Java object serialization mechanism.

4.3.4 Security. Security is a crucial and difficult problem with mobile code [14]. In SAM we just provide entry points which will enable to develop security policies adapted to the different personalities. Those security policies will be called on the personalities of execution sites when agents try to reach or use remote or local entities. They return booleans indicating whether the operation is authorized. If no policy is defined, the returned booleans are always true.

4.4 Observation agents and test language

The observation notion is linked to the main tester whose role is to stimulate the implementation under test and to observe its reactions for returning a verdict on its conformity. Our idea is to integrate into the tool such entities. We name these entities *observation agents*: those two nouns attest that they have both an observation and an active role.

To that end, we develop a high-level language to describe observation agents. This is similar to the notion of observation language that was proposed for protocol verification in [10] and later implemented in the Veda verification-oriented simulator [12]. Initially, the objective was to replace the human reading of traces, but the concept was enriched following the idea that the observation language used should not be restrained and should allow any operations on the simulation. In the same spirit, we do not see observation agents only as testers. The goal is to make the test tool able to execute tests in a richer environment. For that, the notion of observation agent should exceed the notion of tester and be extended to concepts such as resources or other active entities useful in the development and the enrichment of the environment.

The programming language for observation agents must not restrict the expressivity to the tester description, but may include any other function that a test pilot might consider of interest. For example, we must be able to program entities of different natures, such as testers agents, to guide the test, to check the conformity of the implementation, and give a verdict; resource agents or stubs, to enrich the environment; control agents, to check on-line properties of the implementation; tools agents, programmed by the human tester to analyze the events on test execution. Lastly, the semantics of the language has to be well defined

and in direct relationship to the concept of abstract test, which can be considered as a finite state machine.

For conveniency reasons, we decided to extend an existing script language [4] with features of test and object language, such as verdict, method-call, thread, etc. In coherence with the exposed principles, we use the reactive objects studied in [3]. Initiated from the actors world, the reactive objects are autonomous passive or active entities, which are executed in parallel. Based on a semantics of kept events - the transition selected usually depends on a certain stimulus (event input) or boolean expression (guard) -, the reactive approach provides programmers with concurrency, broadcast events, alternative steps, and several primivitives for gaining fine control over the execution of reactive programs.

In this model, reactive agents are executed on a reactive synchronized machine which is an abstraction of an execution unit, controlled by the tool. All reactive machines are synchronized together, in such a way that all observation agents share the same notion of instant. Thus, there is a kind of global logical clock, used by parallel agents to synchronize together, and a new instant can only take place when all parallel agents have finished their execution for the current instant.

The purpose of the reactive machine is to execute the reactive instructions of the observation agents in an environment made of instantaneous broadcast events. Observation agents share the same environment, and communicate using such events. Moreover, new processes or events can be added dynamically in the reactive machine. The observable actions at the SAM level are added in that way, making them instantaneously broadcast to all observation agents. This makes the programming of those agents which have to react to those events quite easy, because the developer does not have to take into account how the events are made known to the testers. Note that synchronization of observation agents is non-trivial since SAM allows for distributed simulation.

Using so tightly synchronized machines would be inefficient in a widely distributed setting, with execution sites far distant from each other, that is in the context of deployment of real multi-agent systems. But this is not our goal here: we just want to test such systems in a controlled way. In that setting where we control the distance between execution sites, the simplicity given by the complete synchronisation between testers and observed events for programming testers seems to us more important that the loss of realism implied by the synchronization hypotheses.

Finally, the formal semantics of the language, based on transition system is very close to abstract test semantics. And the writing of tests is completely independent of the language of the tested system, hence tests can be reused on different implementations of the same specification.

5. RESULTS AND PERSPECTIVES

Development of applications based on mobile agents is still in its infancy, and the software development problems related to this approach still need a fair amount of research to reach the level of maturity achieved in other fields of software development, especially for critical systems. Although testing is often the last part of development to be studied, we consider that it is becoming quite urgent to develop suitable testing approaches to ensure that the applications developed are safe enough.

In this paper, we presented an approach based on a simulation of an underlying platform for mobile agents compliant with the MASIF an FIPA standard. This makes it possible to test in a well controlled environment the behaviour of an application based on a set of mobile processes. Of course, one of the main limits is that it does not test the behaviour of the application on a real environment. Although the tool may be tuned to represent specific characteristics of a real platform (e.g. performance, response time under various loads).

We have implemented those ideas in a first prototype. Our implementation language is Java. The tool enables to execute observation agents written using reactive objects, but also agents designed to execute upon the Grasshopper [1] and Jade [2] platforms. It also offers full support for the MASIF and FIPA standards, offering agent migration, communication, execution and management services.

Mappings to execution platforms proved to be easily written, as soon as the platform is advertised as MASIF-compliant. For example, implementing the Jade mapping took only four days to one of us which had no prior experience with this platform. This mapping is made of 2500 lines of Java code, for 125 ko, whereas the Jade sources use 2 Mo.

For validating SAM, we used all the examples distributed with the Grasshopper and Jade platforms. We were able to execute directly those examples without any modification, even without recompiling. Nevertheless, all migration and communication actions initiated by the agents in those examples were now observable: we wrote observation agents interacting with those examples, and observing them, again without recompiling the examples. This demonstrates how our tool can be used for functional black-box testing of multi-agent mobile systems.

For the future, we plan to demonstrate our approach by collaborating with the developers of multi-agents systems inside FT R&D.[1] SAM will be evaluated as a development tool inside an internal research project.

[1]France Télécom Research and Development

References

[1] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper — A universal agent platform based on OMG MASIF and FIPA standards. In *First International Workshop MATA'99*, October 1999.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE — A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference PAAM-99*, London, UK, 1999.

[3] F. Boussinot, G. Doumenc, and J.B. Stefani. Reactive Objects. *Annals of Telecommunications*, 51:459–473, 1996.

[4] Frédéric Boussinot and Laurent Hazard. Reactive scripts. In *International Conference on Real-Time Computing Systems and Applications (RTCSA'96)* Seoul, October 1996. IEEE.

[5] A. Carzaniga, G.P Pico, and G. Vigna. Designing Distributed Application with Mobile Code Paradigms. In ACM Press, editor, *19th Conference on Software Engineering (ICSE'97)*, pages 22–32, 1997.

[6] G. Cugola, C. Ghezzi, G. Pico, and G. Vigna. Analyzing Mobile Code Languages. *Mobile Object Systems, Lecture Notes in Computer Science*, 1222:94–109, February 1997.

[7] Bruno Dillenseger. Mobilitools: A toolbox for agent mobility and interoperability based on omg standards. In *2nd International Symposium ASA/MA2000*, Zürich, September 2000.

[8] FIPA. FIPA97 Specification. In *Foundation for Intelligent Physical Agents* available from http://www.fipa.org.

[9] A. Fuggetta, G.P. Pico, and G. Vigna. Understanding Code Mobility. In *IEEE Transactions on Softwarw Engineering*, volume 24, pages 342–360, May 1998.

[10] R. Groz. Unrestricted verification of protocol properties in a simulation using an observer approach. In Bochmann & Sarikaya, editor, *In Proceedings of the IFIP WG 6.1 6th Workshop on Protocol Specification, Testing, and Verification*, pages 255–256, Gray Rocks - Montréal, Canada, June 1986.

[11] R. Groz and N. Risser. Eight years of experience in test generation from fdts using tveda. In *In Proceedings of FORTE/PSTV'97*, Osaka, November 1997.

[12] C. Jard, R. Groz, and J.F. Monin. Development of Veda, a prototyping tool for distributed algorithms. *IEEE Transactions on Software Engineering*, 14(3):339–352, March 1988.

[13] M. Marche and Y.M. Quemener. Une approche formelle pour le test de conformité de processus mobile. In *Proceedings of Renpar'12*, Juin 2000.

[14] Jonathan T. Moore. Mobile code security techniques. Technical Report MSCIS -98-28, University of Pennsylvania, Department of Computer and Information Science, May 1998.

[15] OMG. Mobile Agent Systems Interoperability Facilities Specification (MASIF). available from http://www.camb.opengroup.org/RI/MAF/.

[16] Tommy Thorn. Programming Languages for Mobile Code. Technical Report 3134, INRIA, March 1997.

[17] J. Tretmans. Conformance testing with labelled transitions systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, (29):49–79, 1996.

[18] James E. White. Telescript technology: Mobile agents. Available as General Magic White Paper.