Verification of a Self-configuration Protocol for Distributed Applications in the Cloud

Gwen Salaün
Grenoble INP, INRIA, France
gwen.salaun@inria.fr

Xavier Etchevers
Orange Labs, France
xavier.etchevers@orange.com

Noel De Palma
UJF-Grenoble 1, INRIA, France
Noel.Depalma@inria.fr

Fabienne Boyer
UJF-Grenoble 1, INRIA, France
Fabienne.Boyer@inria.fr

Thierry Coupaye
Orange Labs, France
thierry.coupaye@orange.com

ABSTRACT

Distributed applications in the cloud are composed of a set of virtual machines running a set of interconnected software components. In this context, the task of automatically configuring distributed applications is a very difficult issue. In this paper, we focus on such a self-configuration protocol, which is able to configure a whole distributed application without requiring any centralized server. The high degree of parallelism involved in this protocol makes its design complicated and error-prone. In order to check that this protocol works as expected, we specify it in LOTOS NT and verify it using the CADP toolbox. The use of these formal techniques and tools helped to detect a bug in the protocol, and served as a workbench to experiment with several possible communication models.

1. INTRODUCTION

Cloud computing emerged a few years ago as a major topic in modern programming. It leverages hosting platforms based on virtualization, and promises to deliver resources and applications that are faster and cheaper with a new software licensing and billing model based on the *pay-per-use* concept. For service providers, this means the opportunity to develop, deploy and sell cloud applications worldwide without having to invest upfront in expensive IT infrastructure.

Distributed applications in the cloud are composed of a set of virtual machines (VMs) running a set of interconnected software components. However, the task of configuring distributed applications is a real burden. Indeed, each VM includes many software configuration parameters. Some of them refer to local configuration aspects (*e.g.*, pool size, authentication data) whereas others contribute to the definition of the interconnections between the remote elements (*e.g.*, IP address and port to access a server). Therefore, once it has been instantiated, each VM has to apply a set of dynamic settings in order to properly configure the distributed application. On the whole, existing deployment solutions rarely take into account these different configuration parameters, which are mostly managed by dedicated scripts that do not work completly automatically (human intervention is needed). Moreover, these solutions are application-dependent and only work for specific distributed applications to be deployed. For instance, Google App Engine only deals with Web services that respect a restricted programming model (*e.g.*, no Java threads).

In this paper, we present the verification of an innovative self-configuration protocol which automates the configuration of distributed applications in the cloud. Once the VMs are instantiated, the self-configuration protocol is able to configure the whole application without requiring any centralized server and does not require a complex scripting effort. The high degree of parallelism involved in this protocol makes its design complicated and error-prone. Consequently, we decided to specify the self-configuration protocol using LOTOS NT [6] in order to verify it with the CADP toolbox [9]. In this work, these formal techniques and tools helped to detect a major bug in the protocol, which was corrected in the reference implementation in Java. The LOTOS NT specification also served as a workbench to experiment with several possible communication models, and these experiments helped us to avoid an erroneous design.

The rest of this paper is organized as follows. Section 2 introduces the distributed application model and the self-configuration protocol. We present the LOTOS NT specification of the protocol in Section 3 and its verification in Section 4. After comparing our experience with related work in Section 5, we conclude this paper in Section 6.

2. SELF-CONFIGURATION PROTOCOL

Application model. The configuration of a cloud application is specified using a global model composed of a set of interconnected software components running on different VMs. A component is a runtime entity that has some configuration parameters and one or more interfaces. An interface is an access point to a component that supports a finite set of methods. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

coming method calls, and client interfaces, which correspond to access points supporting outgoing method calls. Bindings make explicit connections between components' client interfaces and server interfaces. A binding is local if the components involved in the binding are running on the same VM. A remote binding is a binding between a client interface of a local component and a server interface provided by a component located in another VM. A client interface is also characterized by a property named *contingency*, which indicates whether this interface is optional or mandatory. By extension, the contingency of a binding corresponds to the contingency of its client side. A component has also a lifecycle that represents its state (started or stopped). Finally, an application model identifies each VM belonging to the application, the set of components running on each VM, and their local/remote bindings. A simple example of application model is given in Figure 1 (top), where `c` stands for client and `s` for server.

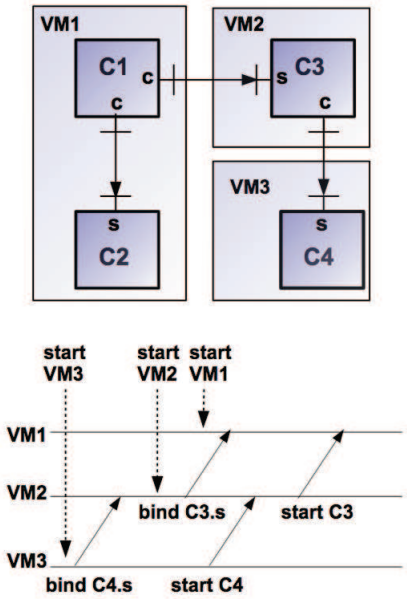

Figure 1: Example of application configuration (top) and self-configuration protocol execution (bottom)

Self-configuration design principles. For scalability purposes, the protocol used to configure distributed applications is *decentralized*. Once the VMs are instantiated, the self-configuration protocol is able to configure the whole application without requiring any centralized server. Therefore, each VM embeds the application model and a configurator. The configurator manages the configuration of the components inside the VM, and participates in the binding configuration between components and in the application start-up. To this end, each configurator has the ability to create and configure components, send server interfaces (for binding purposes), bind component client interfaces to server ones, start components, and send messages to other VMs indicating that a local component has been started. To bind a client interface, the local configurator in charge of the component on the client side needs the corresponding server interface, that is, the required information to access to this interface (IP, port, etc.). This server interface can be local (in this case the local configurator can manage this by itself), or it can be remote (in this case the remote configurator must send the server interface to the local configurator of the corresponding remote VM).

The self-configuration protocol is *loosely-coupled*. Each VM starts the self-configuration protocol just after the boot sequence without needing to know about the state of other VMs. The configuration of the distributed application will progress each time a VM belonging to the application becomes available. This avoids the need for global synchronization between VMs during the configuration protocol. To ensure this property, the configurators send their server interfaces and start messages, according to the application model, through a Message Oriented Middleware [3] (MOM). MOMs implement a message buffering system that enables configurators to exchange messages in a reliable and asynchronous way.

Protocol description. Self-configuration is driven by the configurators within each VM. All configurators evolve in parallel, and each of them carries out various tasks following a precise workflow that is summarized in Fig. 2 where boxes identified using natural numbers (❶, ❷, etc.) correspond to specific actions (`CREATEVM`, `CREATECOMPO`, etc.). Diamonds stand for choices, and each choice is accompanied by a list of box identifiers that can be reached from this point.

Based on the application model, the configurator first starts (❶), successively creates all the components described in the model for this VM (❷), and binds local components (❸). Note that diamonds in the workflow propose different choices, because a VM may not have local bindings for instance, and in such a case, the configurator jumps to the next step. In order to set up remote bindings, both VMs need to interact by exchanging messages through the MOM (❹). For each binding associated to two components C_1 and C_2 (involved respectively in the binding between a server interface and a client interface), the configurator K_1 (responsible for C_1) sends the server interface to configurator K_2 (responsible for C_2). This server interface includes all information required by C_2 to interact with C_1, that is, when K_2 receives a message containing such an interface, it proceeds with the binding of C_2 to C_1.

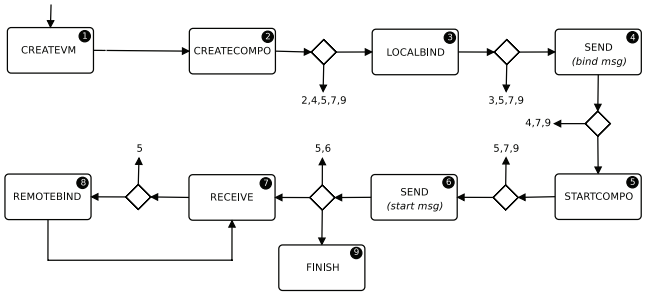

Figure 2: Configurator workflow

Once the configurator has sent all its server interfaces, it can launch the process for starting the applicative components. The configurator first launches the local components that can be started (❺). At that moment in the protocol execution, the only components that can be started are components without mandatory client interfaces or components whose mandatory client interfaces are all connected to local

components. For each component C_{server} then started, the configurator sends to every remote component connected to it through an application binding, a start message (❻) indicating to the remote component that this C_{server} component is started. When the configurator has started all the local components that can be launched, it starts reading from its input communication buffer (❼). Two kinds of message can be received: (i) upon receiving a binding request message, the configurator binds the local component to the remote one (❽), (ii) upon receiving a message indicating that a remote component has been started, the configurator keeps track of this information and goes back to ❺ in order to check whether other local components can be started (those with all mandatory client interfaces connected and corresponding server components started).

Fig. 1 provides an application example (top) and the corresponding self-configuration protocol execution (bottom). This execution scenario shows the communications exchanged between the VM configurators.

3. SPECIFICATION

LOTOS NT is an improved version of LOTOS and combines the best features of imperative programming languages and value-passing process algebras. It also supports the description of complex data types written using a functional specification language. LOTOS NT has a user-friendly syntax and a formal operational semantics defined in terms of Labeled Transition Systems (LTSs). LOTOS NT is supported by the LNT.OPEN tool of CADP, which enables the on-the-fly exploration of the LTSs corresponding to LOTOS NT specifications. In this section, we will present a few LOTOS NT specification excerpts of the self-configuration protocol.

Data types. They are used to describe the distributed application model, that is, VMs, components, interfaces (client and server), bindings between components, messages, buffers, etc. We show below a few examples of data types. An application (`TApplication`) consists of a set of VMs and a set of bindings. A VM (`TVM`) consists of an identifier and a set of components. A component (`TComponent`) is characterized by an identifier, a set of client interfaces, and a set of server interfaces. A client interface (`TClient`) is a couple *(identifier, contingency)*, the contingency (`TClientType`) being either mandatory or optional.

```
type TApplication is
  tapplication (vms:  TVMSet, bindings:  TBindingSet)
end type

type TVMSet is set of TVM end type

type TVM is
  tvm (id:  TID, cs:  TComponentSet)
end type

type TComponent is
  tcompo (id:  TID, cs:  TClientSet, ss:  TServerSet)
end type

type TClient is
  tclient (id:  TID, contingency:  TClientType)
end type

type TClientType is mandatory, optional end type
```

Functions. They apply on data expressions which describe the distributed application. These functions are necessary for three kinds of computation: (i) extracting information from the application, (ii) describing buffers and basic operations on them, (iii) keeping track of the started components to know when another component can be started, *i.e.*, when all its mandatory client interfaces are connected to started components. Functions are also defined to check that there is no cycle of mandatory client interfaces through bindings in the initial application, and that all the mandatory client interfaces are bound. Let us show, for illustration purposes, the function `add`, which adds a message `m` to a buffer[1] `q` storing messages in a list with respect to a FIFO strategy (we add messages at the end of the buffer and read from the beginning). It is worth observing in this example that LOTOS NT uses the classic ingredients of the functional programming style, namely pattern matching and recursion.

```
function add (m:  TMessage, q:  TBuffer):  TBuffer is
  case q in
  var hd:  TMessage, tl:  TBuffer in
       nil -> return cons(m,nil)
    | cons(hd,tl) -> return cons(hd,add(m,tl))
  end case
end function
```

Processes. They are used to specify VMs (configurator, input and output buffer), the communication layer (MOM), and the whole system consisting of VMs interacting through the MOM. Each VM consists of a configurator and two buffers, namely `bufferIn` and `bufferOut`, which store input and output messages, respectively. The configurator drives the behaviour of each VM, and encodes most of the protocol functionality. The MOM process reproduces the communication media behaviour used to make VMs interact together. The MOM is equipped with a set of FIFO buffers in order to store messages being exchanged. There is a buffer for each VM, and messages transiting by the MOM are temporarily stored in the buffer corresponding to the VM to which the message is destinated.

For illustration purposes, we present the LOTOS NT process (named `SELFCONFIG`) encoding the behaviour of the whole protocol. We give in Figure 3 an architectural view of this process with the MOM and as many instances of the configurator and buffer processes as there are VMs.

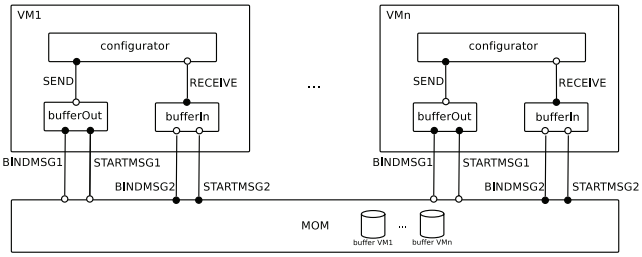

Figure 3: Architectural view of the whole protocol

The `SELFCONFIG` process applies on an input application defined in function `appli()`. A pair of actions (`CHECKCYCLE` and `CHECKMANDATORY`) are introduced at the beginning of the process body for verification purposes. These actions have Boolean parameters (returned values of called functions, *e.g.*, `check_cycle_mandatory`), which indicate whether the input application respects some structural constraints (*e.g.*, absence of cycle through mandatory client interfaces).

[1] `TBuffer` is specified as a list of messages of type `TMessage`, equipped with classic constructors **cons** and **nil**.

The LOTOS NT parallel composition is expressed with the `par` construct followed by the list of actions that must synchronize (nothing for pure interleaving). The first process called in the `SELFCONFIG` process is the MOM, which is composed in parallel with the rest of the system, and synchronizes with the other processes on `BINDMSG`i and `STARTMSG`i messages (i=1,2). More precisely, the MOM has five possible behaviours, it can receive a binding (`BINDMSG1`) or a start message (`STARTMSG1`), send a binding (`BINDMSG2`) or a start message (`STARTMSG2`) if one of its buffers is not empty, or terminate (`FINISH`). Messages suffixed with `1` correspond to emissions from a VM to the MOM, and messages suffixed with `2` correspond to emissions from the MOM to a VM.

After the MOM, a piece of specification (deployer) is in charge of instantiating the set of VMs (`CREATEVM`). Finally, as many VMs as are present in the input application (two machines `VM1` and `VM2` in the specification below) are generated[2]. Each machine consists of a configurator, which synchronizes with two local buffers (`bufferIn` and `bufferOut`) on messages `SEND` and `RECEIVE`. The two buffers as well as the MOM are initialised empty.

It is worth noting that we use two kinds of action in our specification: actions which corresponds to communications between two processes (`SEND` and `RECEIVE` for synchronizations within a VM, `BINDMSG` and `STARTMSG` for synchronizations between VMs), and actions tagging specific moments of the execution that will be useful in the next section to analyse the protocol (`CHECKCYCLE`, `CHECKMANDATORY`, `CREATEVM`, `CREATECOMPO`, `LOCALBIND`, `REMOTEBIND`, `STARTCOMPO`, and `FINISH`). Here is the `SELFCONFIG` process:

```
process SELFCONFIG [CREATEVM:any, SEND:any, ..]  is
   var appli:  TApplication in
     appli:=appli();
     CHECKCYCLE (!check_cycle_mandatory(appli));
     CHECKMANDATORY(!check_mandatory_connected(..));
     par BINDMSG1, BINDMSG2, STARTMSG1, ..  in
       MOM[..](vmbuffer(VM1,nil),vmbuffer(VM2,nil))
     ||
       par CREATEVM, FINISH in
         par FINISH in (* virtual machine deployer *)
           CREATEVM (!VM1) ; FINISH
         ||
           CREATEVM (!VM2) ; FINISH
         end par
       ||
         par FINISH in
           (* first machine, VM1 *)
           par SEND, RECEIVE, FINISH in
             configurator [..]  (VM1,appli)
           ||
             par FINISH in
               bufferOut[SEND,BINDMSG1,..](nil)
             ||
               bufferIn[RECEIVE,BINDMSG2,..](VM1,nil)
             end par
           end par
         ||
           ...  (* second virtual machine, VM2 *)
  end par end par end par end var
end process
```

4. VERIFICATION

To verify the protocol, we apply the LOTOS NT specification of the protocol to a set of distributed applications to be configured. From the specification and the target application, CADP exploration tools generate an LTS describing all the possible executions of the protocol. In this LTS, transitions are labelled with the actions introduced previously, and we use these actions to check that the protocol works as expected.

First of all, we verify that each input application respects a few structural properties, such as *"there is no cycle in the application through mandatory client interfaces"* or *"all mandatory client interfaces are connected"*. This is checked at the beginning of the protocol using functions which extract this information from the application model given as input. These functions return Boolean values which are then passed as parameters to specific actions (`CHECKCYCLE` and `CHECKMANDATORY`). Then, we use a safety property to check that these actions do not appear in the LTS with the wrong Boolean parameter. For instance, we never want the `CHECKCYCLE` action to have a `TRUE` parameter value indicating that there is a cycle of mandatory client interfaces. This is written as follows in μ-calculus, the temporal logic used in CADP, and such properties are verified automatically using the EVALUATOR model-checker [14]:

```
[ true* .  "CHECKCYCLE !TRUE" ] false
```

Secondly, we use model-checking techniques to verify that the application order of actions is respected during the protocol execution. To do so, we formalise in μ-calculus (and check) 14 safety and liveness properties that must be preserved by the configuration protocol. Here are a few examples of these properties:

- `FINISH` is eventually reached in all paths

  ```
  mu X . (< true > true and [ not 'FINISH' ] X)
  ```

- A `STARTMSG2` message cannot appear before a `STARTMSG1` message with the same parameters

  ```
  [ true*.STARTMSG2 ?vm:String ?cx:String ?cy:String.
          true*.STARTMSG1 !vm !cx !cy ] false
  ```

  Note that we use the latest version of EVALUATOR (4.0) which enables us to formulate properties on actions and data terms. Here for example, we relate parameters in both messages saying that the VM (`vm`) and components (`cx` and `cy`) concerned by this message must be the same.

- A component cannot be started before the components it depends on

  ```
  [ true* .  'STARTCOMPO !.* !C1' .  true* .
        'STARTCOMPO !.* !C2' ] false
  ```

  This property is automatically generated from the application because it depends on the bindings for each component. As an example, if a component `C1` is connected through a mandatory client interface to a component `C2`, we generate the property above meaning that we will never find a sequence where `C1` is started before `C2`.

- All components are eventually started

[2]Since the number of VMs depends on the application, this LOTOS NT process is generated automatically by a Python program we wrote, for each new application.

```
( mu X . ( <true> true and [ not 'STARTCOMPO !.*
                  !C1' ] X ) )
                      and
( mu X . ( < true > true and [ not 'STARTCOMPO !.*
                  !C2' ] X ) )
                      and ...
```

This property is also generated because we do not know the number of components and their identifiers *a priori*.

Finally, we check that each VM behaviour isolated from the whole LTS respects the correct ordering of actions. To do so, on the one hand, we have specified an LTS corresponding to the configurator lifecycle. On the other hand, we apply successively hiding and reduction techniques on the whole state space to keep configurator actions corresponding to a specific VM. Then, we check that the resulting LTS is included (branching pre-order) into the first one (configurator lifecycle) using the Bisimulator equivalence checker [4]. For each application, we also extract the MOM behaviour and check that it is included in the LTS given in Figure 4.

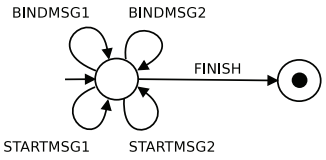

Figure 4: LTS representing the MOM lifecycle

Experiments. They were conducted on more than 100 applications. Experiments have been carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux, and it takes about 15 hours to make all these computations (checking invariants, properties, and equivalences) on all the examples of our database. For instance, an application consisting of 3 VMs, 5 components, and 4 (remote) bindings results in an LTS with about 30,000 states and 110,000 transitions; it takes about 10 minutes to generate this LTS and check all the features presented above. Computation times and LTS sizes grow exponentially as the number of remote bindings, but above all VMs, increase. Our goal here was not to fight the state explosion problem, but to find possible bugs in the protocol. Most bugs do not come from the system's size, but from boundary cases where enumerative tools are very efficient by exploring all the possible execution scenarios.

Issues identified. The specification and verification helped us to detect a major bug in the protocol and to experiment on the communication model. Firstly, there was a problem in the way local components are started during the protocol execution. After reading a message from the input buffer, the configurator must check all its local components, and start those with mandatory client interfaces bound to started components. However, one traversal of the local components is not enough. Indeed, launching a local component can make other local components startable. Consequently, starting local components must be done in successive iterations, the algorithm stops when no more components can be started. If this is not implemented as a fix point, the protocol does not ensure that all components involved in the architectue are eventually started. This bug was detected thanks to the last property presented above (all components are eventually started), and was corrected in both the specification and the Java implementation.

Secondly, there are many ways to implement the MOM. We used our specification, modifying the MOM process, to carry out experiments on how communication among VMs could be implemented (no MOM, MOM with one buffer, two buffers, MOM with n buffers, $2n$ buffers, etc.). We found out that using a single buffer in the MOM is erroneous because the protocol can get momentarily stuck if a VM is not yet started, and the first message in the buffer has to be sent out to that VM. One buffer per machine is necessary to avoid these blocking issues, and this MOM structure was chosen after having carried out these experiments.

5. RELATED WORK

The formalisms and mechanisms offered by the industrial solutions for configuring applications in the cloud are generally basic, proprietary, not exhaustive and not extensible: they permit neither a fine-grained description of the distributed application nor the management of its deployment process. Moreover, such solutions have often important restrictions concerning:

- the programming models like Google App Engine that only deploys Web applications whose code must conform to very specific APIs (*e.g.*, no Java threads)
- the underlying technologies like Microsoft Azure that is confined to the applications based on Microsoft technologies
- the business domains they address like Salesforce.com that focuses on customer relationship management

A few recent results [10, 7, 15] proposed languages and configuration protocols for distributed applications in the cloud. However, these protocols do not work in a decentralized fashion, and this harms scalability of applications that can be deployed with these platforms. In addition, these works do not consider the reliability of the proposed protocols, whereas we focused here on the self-configuration verification and showed its necessity to detect subtle bugs.

There exist many approaches which aim at specifying and verifying distributed components and component-based architectures. In the 90s, several works [11, 12, 1, 17] focused on dynamic reconfiguration of component-based systems, and proposed various formal notations (Darwin, Wright, etc.) to specify component-based systems whose architectures can evolve at run-time (addition/removal of components/bindings). Here, our goal was rather to verify the protocol at hand, to be sure that the corresponding Java implementation worked as expected. In [12, 13], the authors show how to formally analyse behavioural models of components using Ltsa. Another related work is [8], where the authors verify some temporal properties using model-checking techniques on a dynamic reconfiguration protocol used in agent-based applications. Lotos NT enables us to describe not only behaviours but also data types (*e.g.*, FIFO buffers). Moreover, Cadp is richer than Ltsa, which does not propose any tool for equivalence checking for instance. Other toolboxes might have been used, such as Mcrl2, but Lotos NT

is more intuitive than the MCRL2 input language, and CADP also provides efficient model checking tools.

In [2], the authors present a formal framework for behavioural specification of distributed Fractal components. This specification relies on the *pNet* model that serves as a low-level semantic framework for expressing the behaviour of various classes of distributed languages. They also propose a connection to CADP tools in order to check properties on these specifications. A graphical toolset for verifying AADL models is presented in [5]. This platform integrates several existing tools such as the NuSMV symbolic model-checker or the MRMC probabilistic model-checker. As far as autonomic systems are concerned, a few recent solutions have been proposed to analyse such systems. For example, in [16], the authors present the application of ASSL (Autonomic System Specification Language) to the NASA Voyager mission. In their paper, they show how liveness properties can be checked on ASSL specifications, and also plan to consider safety properties. The verification toolbox we use here already provides model-checking techniques for liveness and safety properties, and many more analysis tools.

6. CONCLUDING REMARKS

We have presented in this paper the specification and verification of a cloud computing protocol self-configuring a set of components distributed over several VMs. This protocol is highly parallel, and we applied state-of-the-art verification tools to check that it ensured some key-properties. During this verification stage, we found a bug in the protocol, which was corrected in the Java implementation.

We would like to emphasize three interesting points we have noticed during this experience: (i) the formal languages and tools we used here turned out to be suitable for specifying and verifying such highly parallel protocols that exist in the lastest generation of component-based autonomic systems; (ii) LOTOS NT makes the formal specification accessible to non-experts and deeply simplifies the specification writing; (iii) formal techniques were used not only to chase bugs but also as a workbench for experimenting with different communication features (point-to-point, broadcast, different ways of implementing buffers, etc). This last point can particularly be of interest for optimizing an implementation (*e.g.*, the number of buffers) while preserving the same behaviour (*wrt.* a bisimulation notion for example).

Our main perspective is to extend the protocol to take component failures into account. When a component fails, it may impact the whole application, yet we want our protocol to keep on starting and configuring as many VMs and components as possible. The extended protocol will be extensively validated using analysis tools to check some new properties raised by the introduction of failure, *e.g.*, a component connected through a mandatory client interface to a failed component will never be started.

7. REFERENCES

[1] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.

[2] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural Models for Distributed Fractal Components. *Annales des Télécommunications*, 64(1-2):25–43, 2009.

[3] L. Bellissard, N. D. Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. In *Proc. of SRDS'99*, pages 294–295. IEEE Computer Society, 1999.

[4] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585. Springer, 2005.

[5] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, and R. Wimmer. A Model Checker for AADL. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pages 562–565. Springer, 2010.

[6] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 2011.

[7] C. Chapman, W. Emmerich, F. G. Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. In *Proc. of HPDC'10*, pages 61–72. ACM Press, 2010.

[8] M. A. Cornejo, H. Garavel, R. Mateescu, and N. D. Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proc. of DAIS'01*, volume 198 of *IFIP Conference Proceedings*, pages 229–244. Kluwer, 2001.

[9] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.

[10] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009.

[11] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE TSE*, 16(11):1293–1306, 1990.

[12] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[13] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proc. of WICSA'99*, volume 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.

[14] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.

[15] J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia. DADL: Distributed Application Description Language. USC/ISI Technical Report ISI-TR-664, 2010.

[16] E. Vassev, M. Hinchey, and A. Quigley. Model Checking for Autonomic Systems Specified with ASSL. In *Proc. of NFM'09*, pages 16–25, 2009.

[17] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE'01*, pages 21–32. ACM Press, 2001.