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

Gwen Salaün[1], Xavier Etchevers[2], Noel De Palma[3], Fabienne Boyer[3], and Thierry Coupaye[2]

[1] Grenoble INP, Inria, France
Gwen.Salaun@inria.fr
[2] Orange Labs, France
{xavier.etchevers,thierry.coupaye}@orange.com
[3] UJF-Grenoble 1, Inria, France
{Noel.Depalma,Fabienne.Boyer}@inria.fr

Abstract. Distributed applications in the cloud are composed of a set of virtual machines running a set of interconnected software components. In this context, setting up, (re)configuring, and monitoring these applications is a real burden since a software application may depend on several remote software and virtual machine configurations. These management tasks involve many complex protocols, which fully automate these tasks while preserving application consistency. In this paper, we focus on 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: Google App Engine for instance only deploys Web applications whose code conforms to very specific APIs (*e.g.*, no Java threads), Microsoft Azure only supports applications based on Microsoft technologies, Salesforce only focuses on customer relationship management, etc.

In this paper, we present an abstract model for describing component-based applications and an innovative self-configuration protocol which automates the deployment of these 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 formally specify and verify this protocol in order to find possible bugs using state-of-the-art model checking techniques. The self-configuration protocol was specified using the specification language LOTOS NT [9] (LNT for short) and verified using CADP verification tools [15]. LNT is a simplified variant of the E-LOTOS standard [19] that combines the best features of imperative programming languages and value-passing process algebras. LNT has a user-friendly syntax, and supports the description of complex data types written using a functional specification language. Since LNT relies on classic programming paradigms, this greatly simplifies the design and analysis process, and reduces the gap between the specification and the real implementation of the system. In this work, these formal techniques and tools helped to detect a major bug in the protocol, which was corrected in the Java reference implementation. The LNT specification also served as a workbench to experiment with several possible communication models, and these experiments helped us to avoid an erroneous design.

It is worth emphasizing that the self-configuration protocol is one of the base components of a French project called OpenCloudware[4], aiming at building an open software engineering platform, for the collaborative development of distributed applications to be deployed on multiple Cloud infrastructures. OpenCloudware is a funded project that started in 2012 for three years and involves many companies and research centers in France.

The rest of this paper is organized as follows. Section 2 introduces the distributed application model and the self-configuration protocol. We present the LNT 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.

[4] See `http://opencloudware.org` for more details.

2 Self-configuration Protocol

2.1 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 incoming 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 (left), where `c` stands for client and `s` for server.

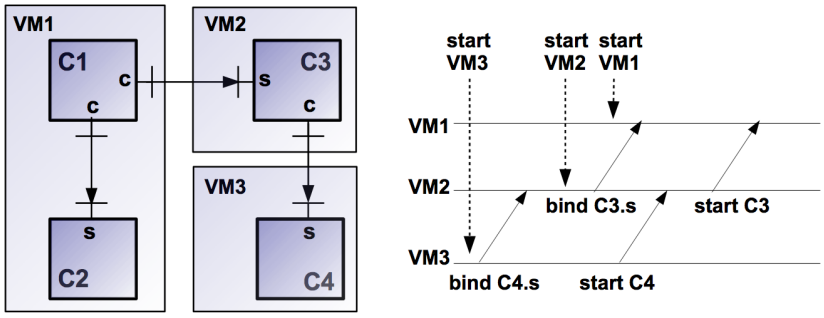

Fig. 1. Example of application configuration (left) and self-configuration protocol execution (right)

2.2 Self-configuration Principles

The configuration starts when the deployment manager instantiates all VMs. Each VM embeds a configurator which drives and encodes most of the self-configuration behaviour. A virtual machine is also equipped with two buffers

(one input buffer and one output buffer) for communicating with the other VMs. All communications transit through a MOM.

Each VM embeds the application model and a configurator. It is worth observing that embedding the whole application model on each VM is not an optimal solution, particularly if the system is planned to be reconfigured. Here, we made this simplification because we focused only on deployment. An alternative solution could be to embed only on each virtual machine the information necessary for its (re)configuration, that is information about the (remote) components connected to local components.

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 sends the server interface to the local configurator of the corresponding remote VM).

The configurators send their server interfaces and start messages, according to the application model, through a Message Oriented Middleware [4] (MOM). MOMs implement a message buffering system that enables configurators to exchange messages in a reliable and asynchronous way. From a local point of view, each VM is equipped with two buffers, one output buffer storing messages destinated to other VMS and one input buffer storing messages coming from other VMs.

It is worth observing that, for scalability purposes, the self-configuration 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. The self-configuration protocol is also *loosely-coupled*. Each VM starts the self-configuration protocol just after the boot sequence (instantiation of VMs by the deployment manager) 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.

2.3 Protocol Description

The protocol execution is driven by the configurators embedded on each VM. All configurators evolve in parallel, and each of them carries out various tasks following a precise workflow that is summarized in Figure 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 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 several options, 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.

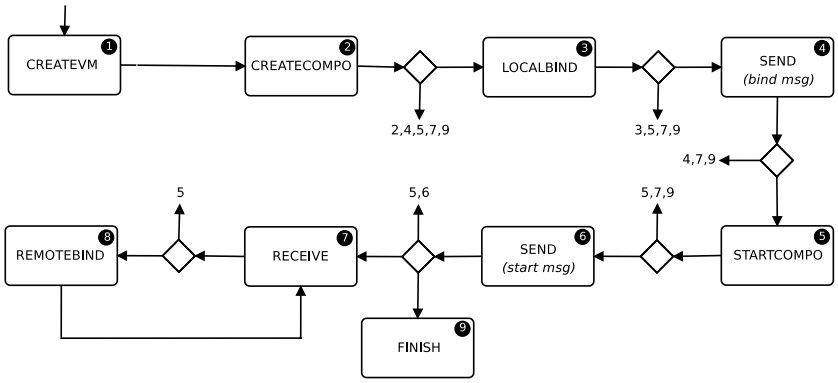

Fig. 2. Configurator workflow

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).

Figure 1 provides an application example (left) and the corresponding self-configuration protocol execution (right). This execution scenario shows the communications exchanged between the VM configurators to start the application. We can see that first the VM3 configurator (in charge of C4) sends a binding message with C4 server interface to VM2 configurator (in charge of C3). VM2 sends C3 server interface to VM1 configurator. Upon receptions both configurators can make these bindings effective. When VM3 starts C4, a message is sent to VM2. Upon reception, VM2 can start C3, and sends a message to VM1 indicating that C3 has been started.

2.4 Implementation

From an implementation point of view, we rely on the Open Virtualization Format [1] (OVF) in order to describe an application, that is a set of interconnected components hosted on various virtual machines. OVF is an open and extensible standard for packaging and distributing virtual appliances or software to be run in virtual machines. However, OVF is not designed for describing architectural aspects (components, interfaces, bindings). Therefore, we have proposed an extension of OVF, which aims at offering an architectural view of the distributed application which is embedded within the VMs of an OVF package. Using OVF offers higher level control abstractions, compared to specific configuration scripts existing in current industrial solutions.

Our extension of OVF enables the description of a distributed application through an XML-based description encompassing not only the notions of hardware requirements, disk images, etc., but also components, interfaces, and bindings. Adding the notion of VM to each component description (using the *virtualnode* tag) also enables the description of the distribution constraints of components within virtual machines.

The deployment engine is implemented in Java and builds upon this ADL to configure automatically an application described with this formalism. We have already used this process for deploying real applications such as Springoo, CLIF, or Tune. Springoo is a Java EE multitiered application enabling the management of markets, offers, and services in a company. CLIF [12] is a load injection framework, which provides a Java-based, open source, generic infrastructure to generate load on any kind of systems, and gather performance measurements (requests response times, computing resources usage, etc.). Tune [8] is a global autonomic management system in Java. Evaluation results show that the self-configuration protocol allows a human administrator to reduce significantly the duration for deploying a large number of applications. A more detailed description of the self-configuration protocol (description, technical details, evaluation, etc.) can be found in [13, 14].

3 Specification

We specified the protocol in LNT [9], one of the input languages of the CADP verification toolbox [15]. We chose LNT as our specification language because (i) it provides expressive enough operators, in particular rich datatype descriptions, for modelling the self-configuration protocol, (ii) its user-friendly notation simplifies the specification writing, and (iii) it is equipped with state-of-the-art verification tools in order to check that the protocol respects some key-properties.

3.1 LNT in a Nutshell

LNT is a simplified variant of the E-LOTOS standard [19] that combines the best features of imperative programming languages and value-passing process algebras. LNT supports both the description of complex data types and of concurrent processes using the same user-friendly syntax. LNT formal operational semantics is defined in terms of LTSs (Labelled Transition Systems).

LNT processes are built from actions, sequential compositions (;), conditions (**if .. then .. else .. end if**), assignments (:=), looping behaviours (**loop .. end loop**), choices (**select .. [] .. end select**), and parallel compositions (**par ..** || **.. end par**). Communication is carried out by rendezvous on a set of synchronization actions (multiway synchronization points) with bidirectional transmission of multiple values. Synchronizations may also contain optional guards (**where**) expressing Boolean conditions on received values. Processes are parameterized by sets of actions (alphabets) and input/output data variables.

LNT specifications can be analysed using CADP, a verification toolbox that has been in continuous development since the late 80s. CADP is dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing. The toolbox contains about 70 tools and libraries that can be used to make different analyses such as simulation, model checking, equivalence checking, compositional verification, test case generation, or performance evaluation. CADP was successfully applied to real-world and industrial case studies in many different fields such as telecommunication protocols, hardware design, embedded systems, or avionics.

In the rest of this section, we will present a few excerpts of the self-configuration protocol LNT specification.

3.2 Data Types and Functions

Data types 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 apply on data expressions which describe the distributed application. These functions are necessary for three kinds of computation: (i) extracting information from the application model, (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 application model, 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 `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). `TBuffer` is specified as a list of messages of type `TMessage`, equipped with classic constructors **cons** and **nil**. It is worth observing in this example that LNT uses the classic ingredients of functional programming, 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
```

3.3 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 two excerpts of LNT processes. The first one is the `SELFCONFIG` process, which encodes 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.

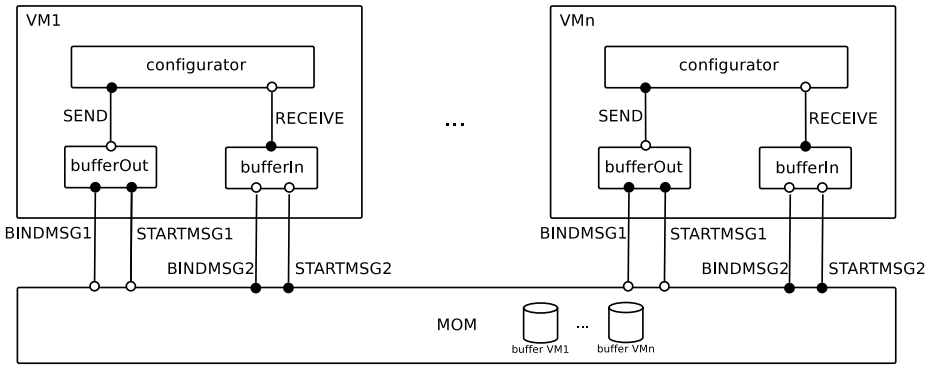

Fig. 3. Architectural view of the whole protocol

The `SELFCONFIG` process defines first the list of actions used in its behaviour (`CREATEVM`, `SEND`, etc.). Actions can be typed (with the types of their parameters), but this is optional and we use the keyword **any** in that case. This 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 as parameters Boolean values computed by calling 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.

The LNT 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 (deployment manager) 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. Since the number of VMs depends on the application, this LNT process is generated automatically for each new application, by a Python program we wrote. 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). For instance, termination of the protocol is made explicit by a synchronization involving all processes on FINISH. Here is an example of SELFCONFIG process (for two VMs identified by VM1 and VM2):

```
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 (* deployment manager *)
           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
```

Now we detail the bufferIn process which can synchronize with other processes (MOM and local configurator) on four actions, namely RECEIVE, BINDMSG, STARMSG, and FINISH. This process also has two data parameters corresponding

to the identifier of its VM, and to the buffer storing messages. Its behaviour is a choice (**select** in LNT) among different possibilities: `bufferIn` can either (i) store messages coming from the MOM and destinated to its VM, or (ii) interact with the local configurator when the configurator decides to read from the input buffer. In the first case, two kinds of messages can be received: a request for binding (`BINDMSG`), or a message announcing that a remote component has been started (`STARTMSG`). In both cases, a message (`TMessage`) is built from the information provided as parameter to the action (*i.e.*, `csvr`, `cclt`, etc.) and stored in the local buffer (`add(..)`). In order to ensure that the input buffer receives only messages destinated to its VM, we use a LNT feature which makes messages with sent parameters synchronize only if they share common parameters. In this case, we use the VM identifier (`!vmid`, first parameter of `BINDMSG` and `STARTMSG` messages). This means that the MOM process will also use such VM identifiers as first parameter of these messages. In the second case, if the buffer is not empty, a message may be retrieved and treated by the local configurator (synchronization on `RECEIVE`).

```
process bufferIn [RECEIVE:any,BINDMSG:any,STARTMSG:any,FINISH:any]
            (vmid: TID, q: TBuffer) is
   var recip, csvr, cclt, idclt, idsvr: TID, m: TMessage in
     select
       BINDMSG (!vmid, ?csvr, ?cclt, ?idclt, ?idsvr) ;
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH]
           (vmid,add(bindmsg(vmid,csvr,cclt,idclt,idsvr),q))
       []
       STARTMSG (!vmid, ?csvr, ?cclt) ;
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH]
           (vmid,add(startmsg(vmid,csvr,cclt),q))
       []
       if not(empty(q)) then
         m:=retrieve(q); RECEIVE (!vmid, !m);
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH] (vmid,remove(q))
       else
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH] (vmid,q)
       end if
       []
       FINISH
     end select
   end var
end process
```

4 Verification

To verify the protocol, we apply the LNT 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.

4.1 Verification Tasks

We identified three facets of the protocol that must be preserved by the protocol, that are structural invariants, temporal properties, and lifecyles.

Invariants. 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 [24]:

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

Properties. Secondly, we use model checking techniques to verify that some key-properties are 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 model 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

```
( 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 the number of components and their identifiers depend on the application model.

Lifecycles. 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. This LTS is obtained by flattening the workflow presented in Figure 2 and consists of 8 states and 26 transitions. 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 [5]. For each application, we also extract the MOM behaviour and check that it is included in the LTS given in Figure 4.

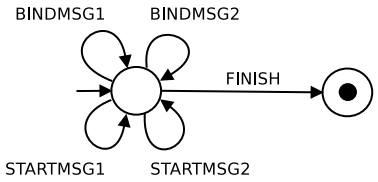

Fig. 4. LTS representing the MOM lifecycle

4.2 Experiments

They were conducted on about 150 applications, which are quite different and enabled us to check boundary cases. For instance, we used applications where components can be started in parallel (interleaving) and others where they can only be started in a very precise order. It is worth observing that, as model checking helps to fing bugs, the more examples we check, the more chances we have to find problems in the protocol.

Table 1 summarizes some of the results obtained on application examples of our dataset. Each example is characterized in terms of number of virtual machines, number of components, and number of local/remote bindings ("b." stand for bindings in the table). We give the size of the LTS generated using CADP by enumerating all the possible executions of the system, as well as the

time to obtain this LTS and verify all the features presented above (checking invariants, properties, and lifecycles). The resulting LTS has been minimized using strong reduction.

Experiments have been carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux, and it takes about 3 days to generate and check all the examples of our database. We can see first that systems involving only a couple of virtual machines and a few remote bindings are generated and checked in reasonable time (examples 0010, 0061, and 0090 in Table 1).

Computation times and LTS sizes grow exponentially as the number of remote bindings and VMs increase. As far as remote bindings are concerned, the more bindings, the more messages exchanged among VMs. This results in large LTSs (see, *e.g.*, example 0092) which are generated quite fastly because the number of processes in parallel is reasonable (3 VMs in example 0092 for instance). In this case, verification takes time because LTSs have to be traversed exhaustively. This is also interesting to note the size and time increase when looking at examples 0086, 0087, and 0088 where by adding one remote binding, the LTS size approximately doubles as well as verification time. In contrast, we can see that the number of local bindings can be quite high without really impacting size and time verification results. Similarly, the number of components does not really affect the results (see, *e.g.*, example 0010).

If we focus now on the number of VMs, we can see that when we have systems with four VMs, LTS generation time grow exponentially (examples 0136 and 0145). This is because the number of processes evolving in parallel increase with the number of VMs and this makes the exploration step very time consuming. The resulting LTS is quite small though and its verification pretty fast.

Fortunately, 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.

4.3 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 quite early during the verification process (after a few examples) thanks to one of the properties presented in Section 4.1 (all components are eventually started). It was corrected in both the specification and the Java implementation.

	Size				LTS	Time (m:s)	
	VMs	compo.	local b.	remote b.	(states/transitions)	LTS gen.	Verification
0010	2	15	2	2	1,788/4,943	0:09	2:23
0061	2	6	3	5	5,091/18,354	0:10	1:45
0090	2	6	3	8	33,486/137,401	0:50	6:44
0092	2	6	9	10	81,822/349,319	1:20	27:20
0122	3	6	6	0	514/1,346	0:14	00:26
0038	3	5	0	4	31,334/109,315	4:01	8:15
0086	3	6	34	4	60,851/226,217	8:14	19:30
0087	3	6	34	5	153,056/645,168	14:02	49:42
0088	3	6	34	6	306,136/1,392,439	25:53	98:42
0136	4	4	0	3	3,350/11,997	84:24	1:02
0145	4	7	4	2	18,314/78,206	191:20	6:02

Table 1. Experimental results

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 that focuses on customer relationship management

A few recent projects [16, 10, 25] proposed languages and configuration protocols for distributed applications in the cloud. [10] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax[5]

[5] This syntax has been defined by the *Model Driven Architecture* (*MDA*) initiative of the *Object Management Group* (*OMG*).

to describe a distributed application, its requirements towards the underlying execution platforms, and its architectural constraints (*e.g.*, concerning placement and collocation). Regarding the configuration protocol, particularly the distributed bindings configuration and the activation order of components that are the core of the present paper, [10] does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed. Moreover, this works does not consider the reliability of the proposed protocol, whereas we focused here on the self-configuration verification and showed its necessity to detect subtle bugs.

[25] suggests an extension of *SmartFrog* [16] that enables an automated and optimized allocation of cloud resources for application deployment. It is based on a declarative description of the available resources and of the components building up a distributed application. Descriptions of applicative architectures and resources are defined using the *Distributed Application Description Language* (DADL). This language describes, on the one hand, the applications constraints related to the resources in terms of *Service Level Agreements* (*SLAs*) and, on the other hand, elasticity constraints. Compared to the present paper, [25] focuses on the language aspects and intends to address the optimal resources allocation. It does not give any details concerning the deployment process itself, which was our focus here.

There exist many approaches which aim at specifying and verifying distributed components and component-based architectures. In the 90s, several works [21, 22, 2, 29] 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 runtime (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 [22, 23], the authors show how to formally analyse behavioural models of components using LTSA. Another related work is [11], where the authors verify some temporal properties using model checking techniques on a dynamic reconfiguration protocol used in agent-based applications. Recently, [6] reported on the co-design and specification of the reconfiguration protocol of a component-based platform, intended as the foundation for building robust dynamic systems. The formal analysis of this protocol helped to detect several issues which were corrected in the corresponding implementation. CADP is richer in terms of verification techniques than LTSA, which does not propose any tool for equivalence checking for instance. Moreover, LNT user-friendliness and expressiveness for specifying both behaviours and data types (*e.g.*, FIFO buffers) makes it very convenient compared to other specification languages. Other toolboxes might have been used, such as SPIN [18] or MCRL2 [17]. LNT is more intuitive than Promela or the MCRL2 input language, and CADP also provides efficient model checking tools.

In [20], the authors present the formal verification of an operating system microkernel. They proved the functional correctness of the microkernel using the Isabelle theorem prover. The formal specification was generated automatically

from an Haskell prototype, and the final implementation was manually encoded in C. This formal process helped to detect and correct many bugs in the system algorithms. Here, we focused on an alternative approach which requires much less effort in the verification process (automated versus semi-automated verification). Nevertheless, although model checking techniques are very suitable to detect bugs in any kind of application, they do not ensure correctness of the system as it may be achieved using theorem proving techniques.

In [3], 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 [7]. 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 [28], 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 other formal analysis tools.

A preliminary version of this work has been presented in [27]. It is extended here in several aspects:

- the presentation of the protocol was extended, particularly with details on its implementation;
- the specification of the self-configuration protocol is presented with more details. In particular, we show and comment on several excerpts of the LNT specification;
- the dataset of applications consists now of 150 applications (100 before);
- we have added a subsection in Section 4 dedicated to experimental results where we present LTS sizes and computation times for several representative examples of our dataset;
- the related work section was revised and enhanced;
- we present in the conclusion some lessons learned from our experience.

6 Concluding Remarks

We have presented in this paper a cloud computing protocol self-configuring a set of components distributed over several VMs. This protocol is highly parallel and loosely-coupled, and this makes its design error-prone. In order to check that some key-properties are ensured, we have specified and verified it using state-of-the-art specification languages and verification tools. During the verification phase, we found a bug in the protocol using model checking techniques, which was corrected in the Java implementation.

As a result, we would like to emphasize lessons we have learned during this experience:

- Specification and verification techniques were introduced lately in the design process; the Java implementation was already available, but still under development. The goal was to detect bugs in pathological cases as the one we found. We could have started from the formal specification as advocated by classic software development processes, but this does not seem a good option for protocol designers who are not experts in formal methods. Coming up with code generation techniques might be an argument for convincing them to do so in the future.
- LNT, thanks to its user-friendly and programming-like notation, makes the formal specification accessible to non-experts and deeply simplifies the specification writing. Its expressive language enables the specification of concurrent behaviours and complex data types. In particular, LNT turned out to be suitable for specifying self-management protocols that exist in the lastest generation of component-based autonomic systems.
- The use of formal verification tools was successful because it helped to debug the protocol. All verification steps are fully automated, but the writing of temporal properties. However, we had to face state space explosion and this obliges us to validate applications involving only a few VMs.
- 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 [26] for example).
- This work shows that formal techniques and tools are not only of interest for critical systems but are also necessary for the design and development of complex system protocols existing in dynamically (re)configurable component-based systems.

A short-term 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 verification 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.

A long-term perspective would be to propose code generation techniques in OO programming languages (which is the main paradigm used in this community) for rapid prototyping purposes. This could also be used for implementing protocols starting from the formal specification and then generating code automatically.

Acknowledgements. The authors would like to thank Frédéric Lang for his help on compositional verification techniques and Wendelin Serwe for his

comments on a former version of this paper. This work has been supported by the OpenCloudware French FSN project (2012-2014).

References

1. Open Virtualization Format Specification. Specification 1.1.0, Distributed Management Task Force DMTF Standard, 2010.
2. 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.
3. 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.
4. L. Bellissard, N. De 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.
5. 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.
6. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer, 2011.
7. 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.
8. L. Broto, D. Hagimont, P. Stolf, N. De Palma, and S. Temate. Autonomic Management Policy Specification in Tune. In *Proc. of SAC'08*, pages 1658–1663. ACM, 2008.
9. 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.
10. C. Chapman, W. Emmerich, F. Galán 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.
11. M. A. Cornejo, H. Garavel, R. Mateescu, and N. De 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.
12. B. Dillenseger. CLIF, a Framework based on Fractal for Flexible, Distributed Load Testing. *Annales des Télécommunications*, 64(1-2):101–120, 2009.
13. X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675. IEEE Computer Society, 2011.
14. X. Etchevers, T. Coupaye, F. Boyer, N. de Palma, and G. Salaün. Automated Configuration of Legacy Applications in the Cloud. In *Proc. of UCC'11*, pages 170–177. IEEE Computer Society, 2011.
15. 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.

16. 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.
17. J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. Van Weerdenburg. The Formal Specification Language mCRL2. In *Dagstuhl Seminars*, 2007.
18. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
19. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, 2001.
20. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of SOSP'09*, pages 207–220. ACM Press, 2009.
21. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE TSE*, 16(11):1293–1306, 1990.
22. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
23. 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.
24. 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.
25. J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia. DADL: Distributed Application Description Language. USC/ISI Technical Report ISI-TR-664, 2010.
26. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
27. G. Salaün, X. Etchevers, N. De Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proc. of SAC'12*, pages 1278–1283. ACM Press, 2012.
28. E. Vassev, M. Hinchey, and A. Quigley. Model Checking for Autonomic Systems Specified with ASSL. In *Proc. of NFM'09*, pages 16–25, 2009.
29. 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.