HAL
open science

An Experience Report on the Verification of Autonomic Protocols in the Cloud

Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noël de Palma, Xavier Etchevers, Olivier Gruber

To cite this version:

Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noël de Palma, Xavier Etchevers, et al.. An Experience Report on the Verification of Autonomic Protocols in the Cloud. Innovations in Systems and Software Engineering, 2013, 10.1007/s11334-013-0204-0 . hal-00808565

HAL Id: hal-00808565

https://inria.hal.science/hal-00808565

Submitted on 5 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

An Experience Report on the Verification of Autonomic Protocols in the Cloud

Gwen Salaün · Fabienne Boyer · Thierry Coupaye · Noel De Palma · Xavier Etchevers · Olivier Gruber

the date of receipt and acceptance should be inserted later

Abstract Cloud applications are often complex distributed applications composed of multiple software components running on separate virtual machines. Setting up, (re)configuring, and monitoring these applications are complicated tasks because 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 as well as some key properties. In this article, we present two experiences we had in formally specifying and verifying such protocols. The first one aims at designing a reconfiguration protocol of a component-based platform, intended as the foundation for building robust dynamic systems. The second aims at automating the configuration task of a set of virtual machines running a set of interconnected software components. Both applications are specified using the LNT process algebra and verified using the CADP verifica-

G. Salaün
Grenoble INP, Inria, France
E-mail: Gwen.Salaun@inria.fr

F. Boyer
UJF, France
E-mail: Fabienne.Boyer@imag.fr

T. Coupaye
Orange Labs, France
E-mail: Thierry.Coupaye@orange.com

N. De Palma
UJF, France
E-mail: Noel.Depalma@imag.fr

X. Etchevers
Orange Labs, France
E-mail: Xavier.Etchevers@orange.com

O. Gruber
UJF, France
E-mail: Olivier.Gruber@imag.fr

tion toolbox. The use of formal specification languages and tools was a success. We conclude with a number of lessons we have learned while working on this topic in the last three years.

1 Introduction

Cloud computing [16] 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.

Cloud applications are often complex distributed applications composed of multiple software components running on separate virtual machines (VMs). Such applications benefit from several services provided in the cloud such as database storage, load balancing, and so on. However, setting up, (re)configuring, and monitoring distributed applications in the cloud is a real burden since software may depend on several remote software and virtual machine configurations. These management tasks involve many complex protocols, which fully automate them while preserving application consistency. In addition, some of these tasks are executed in parallel and tolerate faults/failures. These characteristics of the management tasks (full automation, robustness, parallel execution, fault tolerance) complicate their development compared to classical software. This explains why the use of formal techniques and tools turned out to be necessary for them.

In this article, we present our experiences in designing two protocols, one for dynamically reconfiguring component-based applications [8], and another for self-configuring distributed systems consisting of interacting virtual machines [15,31][1]. The first protocol is a robust reconfiguration protocol which is part of the virtual machine. This protocol applies a number of architectural changes to an assembly of components to reach a target assembly. This protocol preserves over its application some structural invariants and also tolerates faults that may occur during the reconfiguration process. There is a clear need in the cloud for dynamic reconfiguration of applications, and the first protocol falls into this category. The second protocol is a solution for deploying a set of components distributed over several virtual machines. This self-configuration protocol fully automates this task in a decentralized and loosely-coupled way. This means that this protocol does not require any centralized server and each virtual machine starts itself without needing any information about the

[1] This work results from a collaboration between experts in autonomic protocols and cloud computing on the one hand, and an expert in formal techniques and tools on the other. More precisely, the repartition of the work was as follows: The reconfiguration protocol (Section 2.1) was designed by F. Boyer and O. Gruber; The self-configuration protocol (Section 2.2) was designed by X. Etchevers, N. De Palma, F. Boyer, and T. Coupaye; Specification and verification tasks (Section 3) were carried out by G. Salaün.

current state of the other virtual machines. It is worth emphasizing that the self-configuration protocol is one of the base components of a French project[2] aiming at building an open software engineering platform for the collaborative development of distributed applications to be deployed on multiple Cloud infrastructures.

In both cases, we specified the protocol using LOTOS NT (LNT for short) [11] and verified it with the CADP verification toolbox [17]. LNT is a value passing process algebra that takes inspiration in imperative programming languages, and supports the description of complex data types written using a functional specification language. We chose LNT as our specification language because (i) it provides expressive enough operators, in particular rich datatype descriptions, for modelling the protocols, (ii) its user-friendly notation simplifies the specification writing, and (iii) it is equipped with state-of-the-art model checking tools in order to check that the protocols respect some key properties. Since LNT relies on classical programming paradigms, this simplifies the design and analysis process, and reduces the gap between the specification and the real implementation of the system. These formal techniques and tools helped in both cases to detect several issues and bugs in the protocols, which were corrected in the corresponding Java implementations.

Our first goal in this article is to give an overview of our work in the last three years[3]. We have chosen to introduce the specification and verification tasks for the self-configuration protocol only because it would be too long to describe them for both. We also present some lessons we have learned during this work. In particular, we discuss some possible improvements of the formal specification languages and tools we used. Some of these shortcomings are real challenges, and pave the way for the development of formal techniques that should be simple yet powerful enough to make verification mainstream in the cloud computing area.

The rest of this article is organized as follows. Section 2 introduces the two protocols analyzed in this experience report. We present in Section 3 specification and verification tasks for the self-configuration protocol. Section 4 reviews related work. Section 5 summarizes lessons we learned during these experiences. We conclude this article in Section 6.

2 Protocols

In this section, we present informally the two protocols on which we worked. The first one aims at dynamically reconfiguring an assembly of components. The second one aims at deploying a system composed of interconnected software components running on several virtual machines.

[2] OpenCloudware is a French project that started in 2012, involving many companies and research centers in France, see `http://opencloudware.org` for more details.

[3] This work started in September 2009 and the first version of this article was submitted in July 2012.

2.1 Reconfiguration Protocol

Component Assembly. In the component paradigm, complex systems are designed and built as a component assembly, depicted in Figure 1. Components are independent fragments of software, assembled together by wiring imports to exports. For each component, its exports (or server interfaces) describe services that the component is willing to provide and imports (or client interfaces) describe service requirements, that is, services that the component needs to function properly. A wire (or binding) from a client interface to a server interface indicates that the service requirement described by the client interface is to be satisfied by the provided service described by the server interface.

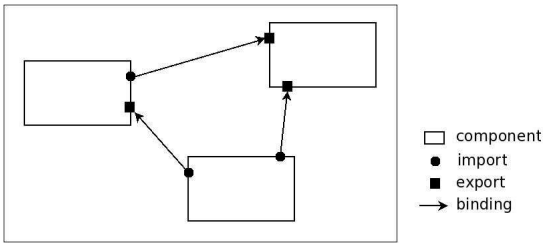

Fig. 1 A Component Assembly

To be correct, a component assembly must respect certain invariants that correlate the lifecycle of components, the different semantics of client interfaces, and the wiring of client interfaces to server interfaces. There are three semantics for a client interface: vital, mandatory, and optional. Vital imports represent services that are needed to construct and initialize a component. Mandatory imports represent references to services that are needed by a component to be functional. Finally, optional imports express that the component may function without the corresponding services. There is no cycle of bindings through vital (mandatory, resp.) imports (see [8] for more details). There are four states to the component lifecycle: *registered*, *constructed*, *resolved*, and *failed*. A client interface is said to be *satisfied* if it is wired to a server interface and the component of that server interface is resolved. We give below the four main invariants:

INV.1 A component is constructed if all its vital imports are satisfied.
INV.2 A component is resolved if all its mandatory and vital imports are satisfied.
INV.3 There can be no binding from a resolved component to either a constructed, registered, or failed component.
INV.4 If a component is failed or registered, none of its server interfaces are wired.

A component starts its life when it is registered in the assembly. It is constructed when its vital imports are satisfied. When constructed, a component has created the services it exports, but they are not yet available to use by other components. When a component is resolved, all its mandatory requirements are satisfied; it is therefore fully functional and the services it exports are available to use.

The Reconfiguration Protocol. The role of the reconfiguration protocol is to reconfigure the running system, called the *concrete assembly*. As depicted in Figure 2, the reconfiguration to apply to the concrete assembly is given to the protocol as two abstract assemblies: the *current assembly* and the *target assembly*. The *current assembly* is an abstract description of the current state of the running system. The *target assembly* is an abstract description of the desired assembly for the running system. Comparing the current and target assemblies, the protocol computes the ordered set of reconfiguration operations that must be invoked on the concrete assembly in order to reconfigure it to conform to the target assembly definition.

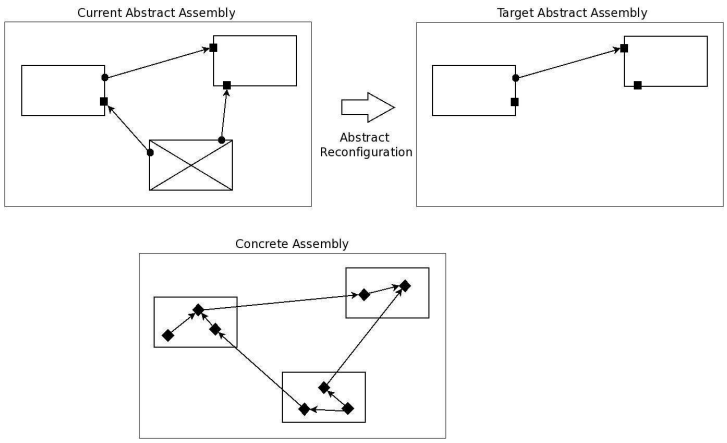

Fig. 2 Concrete and Abstract Assemblies

While computing the set of necessary operations is relatively straightforward, ordering these operations correctly is a real challenge. Correctness is defined here as (i) invariants must be respected before and after each operation, (ii) per component, the sequence of reconfiguration operations must respect the grammar corresponding to the automaton depicted in Figure 3. These requirements can be checked using model and equivalence checking as we will see in Section 3.3. This correctness is crucial because it is the cornerstone of the programming model exposed to component developers. Firstly, invariants control the lifecycle of components that governs when a component is operational and when wired services may be used. Secondly, the grammar is the behavioral contract given to component developers regarding reconfigurations.

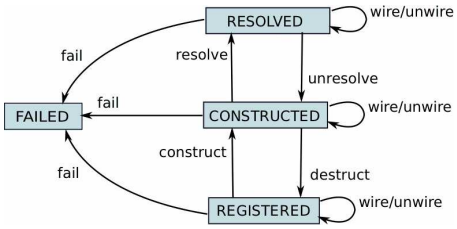

Fig. 3 Component Lifecycle / Reconfiguration Grammar

We present in Figure 4 the V-shape order in which the assembly must be reconfigured in order to preserve the correctness constraints presented above. During the *down phase*, it starts with *down operations* (unresolve, unwire, and destruct) applied to all components in the depicted order. During the *up phase*, it finishes with *up operations* (construct, wire, and resolve) in the depicted order. This precise order ensures that all our invariants are never violated. It is worth observing that parts of the protocol could be executed in parallel, but this would have complicated the protocol implementation and optimizing performance was not an objective in this work.

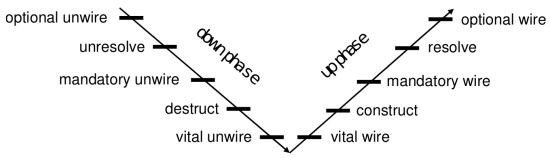

Fig. 4 Our V-shaped Protocol

Failures may happen during a reconfiguration and must be handled in a way that continuously respects both the invariants and the reconfiguration grammar. Any reconfiguration operation invoked on the concrete assembly by the reconfiguration protocol may fail. Modeling such failures is important because they happen in running systems, either because of exceptional situations or bugs. It is important to insist that these failures are not failures of our protocol but the failure of individual concrete components. Our protocol tolerates such faults, assists the running system to recover from them, and then continues to make progress towards the target assembly.

More details on how the different client interface semantics (optional, mandatory, vital) and the failure propagation are handled in the protocol can be found in [8].

2.2 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 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 (no vital or failed here). By extension, the contingency of a binding corresponds to the contingency of its client side. A component also has 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 an application model is given in Figure 5 (left), where `c` stands for client and `s` for server. An explanation of the application of the self-configuration protocol on this example (Fig. 5, right) will be given later on in this section.

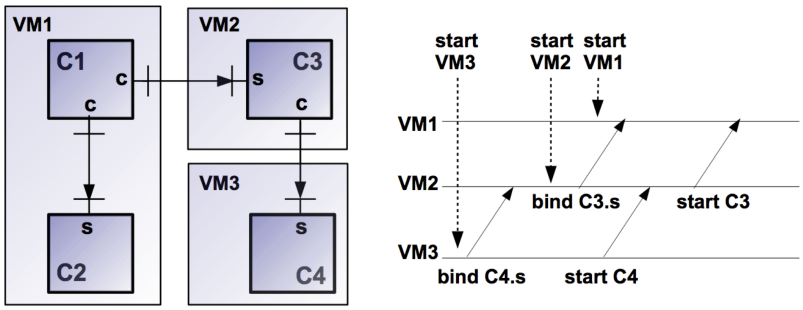

Fig. 5 Example of Application Configuration (left) and Self-configuration Protocol Execution (right)

Self-configuration Design Principles. The configuration starts when a deployment manager instantiates all VMs. Each VM embeds the application model and a configurator which drives and encodes most of the self-configuration behavior. A VM is also equipped with two buffers (one input buffer and one output buffer) for communicating with the other VMs. All

communications transit through a Message Oriented Middleware (MOM), see Fig. 6 for an architectural view.

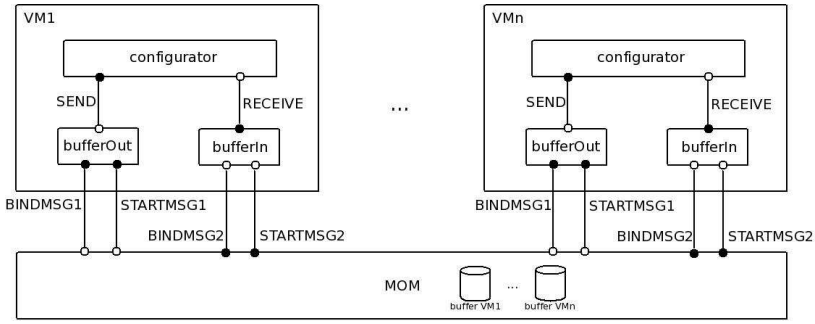

Fig. 6 Architectural View of the Protocol

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 MOM [5]. 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.

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 Figure 7 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.

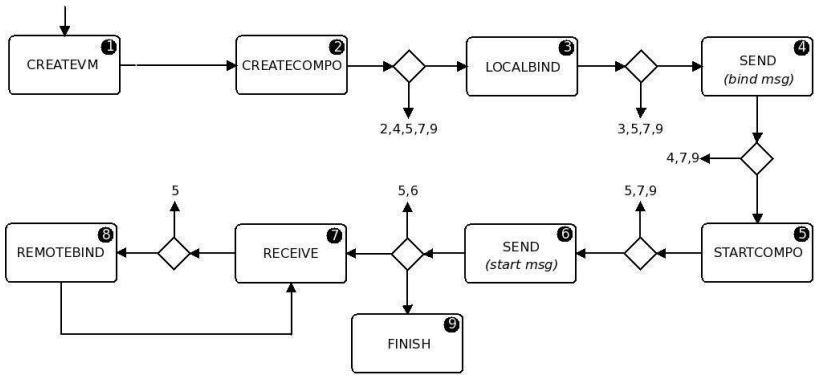

Fig. 7 Configurator Workflow

Once the configurator has sent all its server interfaces, it can launch the process for starting the application 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 5 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 the C4 server interface to the VM2 configurator (in charge of C3). VM2 sends the C3 server interface to the 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.

A more detailed description of this protocol can be found in [15].

3 Specification and Verification: Illustration with the Self-configuration Protocol

In this section, we present the specification of the self-configuration protocol in LNT [11] and its verification with the CADP verification toolbox [17]. We have chosen to present these specification and verification tasks for one protocol only because it would be too long to present them for both. In addition, the approach was very similar for both experiences. The reader interested in more details on the formal analysis of the reconfiguration protocol can refer to [8].

3.1 LNT and CADP

LNT is a simplified variant of the E-LOTOS standard [22] 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 (Labeled Transition Systems). An LTS consists of a set of states and transitions, which are labeled with labels chosen from an alphabet.

LNT processes are built from actions, sequential compositions (;), conditions (**if**), assignments (:=), looping behaviors (**loop**), choices (**select**), and parallel compositions (**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 analyzed using CADP, a verification toolbox dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing. LNT is supported by the LNT.OPEN tool of CADP, which enables the on-the-fly exploration of the LTSs corresponding to LNT specifications. 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.

3.2 Specification

The specification consists of four parts: data types (300 lines), functions (1200 lines), processes (at least 600 lines), and MCL properties (at least 300 lines). The number of lines for processes and MCL properties depends on the size of the input application model, as explained later on in this section. Therefore, the numbers given above are indicative and correspond to a very simple application model (2 VMs and 2 components). These numbers grow with the number of VMs and components involved in the application model.

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.

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.

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 behavior of each VM, and encodes most of the protocol functionality. The MOM process reproduces the communication media behavior 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 addressed.

For illustration purposes, we present the LNT process (named `SELFCONFIG`) encoding the behavior of the whole protocol. We give in Figure 6 an architectural view of this process with the MOM and as many instances of the configurator and buffer processes as there are VMs.

The SELFCONFIG process takes as input an application model, similar to the left-hand side of Figure 5. This application model is defined directly as an LNT expression, which is the body of a function apply(). This function does nothing but defines a concrete application model and returns it. 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).

The LNT parallel composition is expressed with the par construct followed by the list of actions that must synchronize together (nothing for pure interleaving). Two processes synchronize if they share the same action name (see [11] for more details on the LNT synchronization mechanisms). 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 BINDMSGi and STARTMSGi messages (i=1,2). More precisely, the MOM has five possible behaviors, 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[4]. 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 initially empty.

It is worth noting that we use two kinds of action in our specification: actions which correspond 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 analyze the protocol (CHECKCYCLE, CHECKMANDATORY, CREATEVM, CREATECOMPO, LOCALBIND, REMOTEBIND, STARTCOMPO, and FINISH). Here is the SELFCONFIG process generated for an application model involving two VMs:

```
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 *)
```

[4] 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.

```
            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
```

3.3 Verification

To verify the protocol, we use as benchmark a set of distributed applications to be configured. From the LNT specification of the protocol and a specific target application taken from that set, CADP exploration tools generate an LTS describing all the possible executions of the protocol. In this LTS, transitions are labeled with the actions introduced previously, and we use these actions to check that the protocol works as expected. We identified three aspects of the protocol that must be preserved by the protocol, namely temporal properties, structural invariants, and lifecyles.

Properties. We use model checking techniques to verify that some key properties are respected during the protocol execution. To do so, we formalise 14 safety and liveness properties in μ-calculus [29], the temporal logic used in CADP, and such properties are verified automatically using the EVALUATOR model checker. We give a couple of properties here for illustration purposes:

- 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. Variables `vm`, `cx`, and `cy` capture the name of the virtual machine, and the name of the two involved components (sender and receiver). Question marks are used for extracting variables from an action (action name + values as parameter) and exclamation marks for matching values against expressions.

Concretely, here for example, we mean that the virtual machine identifier used as first parameter to `STARTMSG2` must be the first parameter in `STARTMSG1` (same reasoning for component identifiers appearing as second and third parameters).

– 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 automatically generated from the application model because the number of components and their identifiers depend on the application model.

Invariants. 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.

Lifecycles. We check that each VM behavior 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 7 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 simulation pre-order) into the first one (configurator lifecycle) using the Bisimulator equivalence checker [6].

3.4 Experiments

Experiments were conducted on about 150 quite different applications, which 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.

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

	Size				LTS	Time (m:s)	
	VMs	compo.	loc. b.	rem. b.	(states/transitions)	LTS gen.	Verif.
0010	2	15	2	2	1,788/4,943	0:09	2:23
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
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

using strong reduction. It is worth observing that strong reduction does not importantly impact the resulting LTS. For example, the LTS obtained before minimization for example 0088 consists of 309,556 states and 1,405,249 transitions.

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 and 0090 in Table 1).

Computation times and LTS sizes grow exponentially as the number of VMs and remote bindings increase. More precisely, we can see that when we increase the number of VMs, the number of processes (configurator, local buffers, etc.) evolving in parallel increases and therefore this induces more parallelism in the system resulting in longer computation time for state space exploration (from a few minutes for systems with 2 or 3 VMs to a few hours for 4 VMs). The resulting LTSs are quite small though (see examples 0136 and 0145) and their verification pretty fast.

As far as remote bindings are concerned, the more bindings, the more messages exchanged among VMs. This results in larger LTSs (see, *e.g.*, example 0092) which are generated quite rapidly because the number of processes in parallel is reasonable (2 VMs in example 0092 for instance). However, verification takes some time because LTSs have to be traversed exhaustively. It is also interesting to note that the addition of a single remote binding in examples 0086, 0087, and 0088, approximately doubles the LTS size and 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).

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.

3.5 Bug Detection

The specification and verification helped us to detect a major bug in the protocol. 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 architecture are eventually started. This bug was detected thanks to one of the properties presented in Section 3.3 (all components are eventually started). This property turns out to be false for specific applications and specific scenarios for these applications. Exhaustive exploration as used in model checking was therefore convenient for detecting this issue, but other techniques (*e.g.*, testing) could have worked too. This problem was corrected in both the specification and the Java implementation.

4 Related Work

A few recent projects [21,12,30] proposed languages and configuration protocols for distributed applications in the cloud. [12] adopts a model driven approach with extensions of the Essential Meta-Object Facility (EMOF) abstract syntax[5] 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 article (Section 3), [12] does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed. Moreover, that work 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.

[30] suggests an extension of SmartFrog [21] 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 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 article, [30] focuses on the language aspects and intends to address the optimal

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

resources allocation. It does not give any details concerning the deployment process itself.

There exist many approaches which aim at specifying and verifying distributed systems and component-based applications. Several works [24, 25, 1, 34, 10] 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 protocols at hand, to be sure that the corresponding Java implementation worked as expected. In [25, 28], the authors show how to formally analyze behavioral models of components using LTSA (Labeled Transition System Analyzer). Another related work is [13], where the authors verify some temporal properties using model checking techniques on a dynamic reconfiguration protocol used in agent-based applications. The main difference here is that we focus on autonomic protocols, which makes their development and specification/verification much more complicated. Handling failures of components, as it is the case for the reconfiguration protocol for instance, makes the design (hence the specification/verification) of such self-management protocols even trickier.

In [4], the authors present a formal framework for behavioral specification of distributed Fractal components. This specification relies on the *pNet* model that serves as a low-level semantic framework for expressing the behavior 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 [9]. This platform integrates several existing tools such as the NuSMV symbolic model checker and the MRMC probabilistic model checker. As far as autonomic systems are concerned, a few recent solutions have been proposed to analyze such systems. For example, in [33], 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 plan to consider also safety properties. The toolbox we use here for verification purposes already provides model checking techniques for any class of temporal property.

Graph grammars, in particular Reo [3], have been used in [26] for modeling dynamic reconfigurations of systems evolving in changing environments, and verifying properties (safety, consistency) on them. In [2], the authors rely on the Paradigm coordination language and propose an approach for dynamically adapting the coordination model. They also translate the migration model into ACP for analysis with the mCRL2 model checker. Our approach shares some similarities with these papers, but the area (cloud computing) and inherently the two protocols are quite different (*e.g.*, import semantics, fault tolerance, or component configuration).

In [23], 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 automat-

ically from a 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.

5 Lessons Learned

In this section, we would first like to emphasize some positive feedback we had during these experiences. In a second part, we will present some improvements and challenges that have to be investigated (in our opinion) if we want formal specification languages and model checking tools to become mainstream in the development of autonomic protocols and component-based systems in the cloud.

- Our experiences were successful due to the late introduction of specification and verification techniques in the design process (Java implementations were already available, but still under development). Therefore, we had several iterations between designing, specifying, and verifying the protocol on the one hand, and completing its implementation on the other hand. Through these iterations, the specification and verification refined our understanding of the finer points of the procotol, ultimately fixing bugs in the most pathological cases that would have been impossible to identify manually. The introduction of specification and verification tasks had to be adapted in terms of methodology to the specific context of cloud computing and protocol designers. It is worth observing that, to some extent, we relied on a methodology very close to agile techniques with short and successive iterations, but including formal methods in the software development process. We could have started from the formal specification as advocated by classical software development models, 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 (see the code generation item below in this section). Another alternative could have been to start the specification and implementation in parallel from the beginning, but this does not seem a good solution either, because it can generate an unnecessary extra cost if it turns out that the use of formal verification is finally not required for the protocol under development (*e.g.*, the protocol was much simpler than expected). More generally, we can see that classical software development models (such as the V-Model) are not adapted to recent computing areas, and new solutions deserve to be proposed to take flexibility, variation, and change into account.

- LNT, thanks to its user-friendly and programming-like notation, makes the formal specification accessible to non-experts and deeply simplifies the specification writing. For instance, we noticed that even people who were not familiar at all with LNT were able to understand LNT specifications and interact with specifiers in order to clarify and refine finer points of the protocols. Furthermore, the LNT expressive language enables the specification of concurrent behaviors but also of complex datatypes. We believe that LNT could become mainstream for specifying concurrent and distributed systems.
- In the self-configuration protocol design, 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 can especially be of interest for optimizing an implementation (*e.g.*, the number of buffers) while preserving the same behavior (*wrt.* a bisimulation notion for example).
- Finally, 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 reconfigurable and component-based autonomic systems.

The design of these protocols was also very helpful because it enabled us to identify a set of extensions and improvements of the formal methods and verification tools we used in our experiments:

- Specification languages: There is a clear need of simple and user-friendly, yet expressive, specification languages. LNT is a first step in that direction, but it could be improved in various ways. This work helped to identify about 30 possible improvements, either in the LNT language or in the LNT.OPEN tool, which will further simplify the specification and verification steps for future LNT users. An example is the generalized parallel composition operator originally proposed in [19], which makes the synchronization of n processes among m possible (with $n \leq m$). This feature would have been very handy in the self-configuration protocol specification. Indeed, the message bus can be implemented in a distributed fashion. In that case it means that all virtual machines can interact together on the same messages, but only two of them actually synchronize (binary communication). Unfortunately, this operator is not yet available in LNT and there is no other simple solution to emulate this behavior. Therefore, we were only able to specify the centralized version of the message bus.
- Model checking tools: In our experiments, we used a simulation tool (step-by-step animation) and a model checker. Simulation tools are simple to use but not powerful enough to find subtle bugs, particularly when state spaces are huge and exhaustive exploration is therefore required. In contrast, temporal logics and model checking techniques are powerful techniques but not

intuitive enough to be used by non-experts. There is a clear need of automated, yet simple tools to analyze LTSs. An idea would be to identify some common properties in autonomic systems in order to propose some patterns in temporal logics for formalising these properties.

- Debugging techniques: When a bug is found by model checking tools, it is identified with a counterexample (a sequence of actions violating the property). This diagnostic can be quite long (up to one hundred actions when verifying the reconfiguration protocol), and in this case it is rather complicated to extract the configuration of the system when the error occurs. Similarly to state-based techniques, a direct access to the data expressions state would help debugging. Another issue comes from the lack of insight concerning the parallelism of the system: the current state of each concurrent process would help to understand what each participant was doing when the error occurred.
- Coverage analysis: In the classical verification setting, we have an LNT specification of a system, a set of temporal properties to be verified on the LTS model corresponding to the LNT specification, and a dataset of examples (test cases) we use for validation purposes. At this stage, building the set of validation examples and debugging the system is a real burden, in particular for non-experts. There are a few limitations of the approach. First, we do not know whether the set of test cases covers all the possible execution scenarios described in the specification. Second, the LNT specification might be refactored and improved but this cannot be deduced by the counterexamples returned when applying model checking techniques. Third, the set of properties may miss some interesting verification scenarios.
  Coverage analysis aims at proposing and developing techniques for automatically detecting parts of an LNT specification not (yet) covered during verification. Such LNT coverage analysis techniques would be very helpful for (i) extending the set of test cases with new inputs covering parts of the LNT specification that have not been analyzed yet, (ii) eliminating dead code in the LNT specification, and (iii) extending the set of properties with new formulas. We also plan to develop tool support for automating the coverage analysis. To do so, we will rely on and extend some of the tools available in CADP for compiling LNT into LOTOS and LTS, and exploring the resulting LTS for verification purposes.
- Co-simulation and testing: In our experiences, specification/verification and implementation (in Java) have been achieved in parallel. When a bug was discovered on the specification, the reference implementation was systematically corrected, but we did not use formal techniques to check that both levels were consistent. Co-simulation techniques could help and existing works, such as [20, 27], could inspire further investigation for proposing a simple and reusable solution for co-simulating a formal specification with an implementation. Model-based testing [7] is another alternative that deserves to be studied.

- Code generation: The existence of code generation techniques in OO programming languages (which is the main paradigm used in this community) would help for rapid prototyping purposes, but also for implementing protocols starting from the formal specification and then generating code automatically (both the specification and the implementation were developed at the same time and manually in our experiences). This would be a good argument to convince users to start from the formal specification. An interesting perspective in that direction is to generate some parallel code to take advantage of the multiple servers available on cloud architectures.
- Scalability: The self-configuration protocol being highly-parallel, we had to face an exponential growth in the time necessary for generating LTSs for some examples from our database. A promising perspective is to apply recent techniques such as distributed model checking [18] or smart composition/reduction, *e.g.,* [14], to handle larger systems and generate corresponding LTSs in a shorter time.
- Agile Software Development: We found out when working on the reconfiguration protocol that we were following an approach very similar to what is advocated in software development with agile methods. Nowadays agile techniques are used daily in most software development companies. However, formal methods are not yet part of these software development processes. We would like to study more thoroughly how they could be introduced in this development process, for instance in the Scrum method [32], while preserving the basic principles of agile methods.

6 Concluding Remarks

We have overviewed in this article two protocols for applications deployed in cloud computing environments. The first one is a robust reconfiguration protocol which is part of the virtual machine. This protocol applies a number of architectural changes to a current assembly of components to reach a target assembly. This protocol preserves over its application some structural invariants and tolerates faults that may occur during the reconfiguration process. The second protocol is highly parallel and aims at self-configuring a set of components distributed over several virtual machines. Both protocols have been specified in LNT and verified with the CADP toolbox. These state-of-the-art verification tools enabled us to check that some key properties were ensured. More importantly, during this verification stage, we found several bugs. In the reconfiguration protocol, we detected several issues which enabled us to revise several parts of the protocol, for instance: introduction of two additional (un)wire phases (a single wire/unwire was originally present in the V-shaped protocol), several corrections of the failure propagation algorithm, and several corrections in the reconfiguration grammar and structural invariants. In the self-configuration protocol, the use of these formal techniques and tools helped to detect a bug in the protocol. All these issues were corrected in the corresponding Java implementations.

Our main perspective is to extend the self-configuration protocol to take component failures into account. When a VM fails, the deployer starts a new instance of it and warns the other VMs of this failure. In turn, they should start over parts of their configuration behavior to consider this failure. Multiple failures are possible and we want the protocol to be robust and fault tolerant. The extended protocol will be extensively validated using analysis tools to check for instance that in spite of a finite number of failures, all components are finally started. Another interesting perspective would be to use other languages and tools for specifying and verifying these protocols. This would enable us to compare them *wrt.* several criteria such as user-friendliness of the input specification languages, efficiency of the formal analysis tools, debugging techniques available, etc.

Acknowledgements. The authors would like to thank Frédéric Lang and Radu Mateescu for their very interesting comments on a former version of this paper. This work has been supported by the OpenCloudware French FSN project (2012-2015).

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. S. Andova, L. Groenewegen, J. Stafleu, and E. P. de Vink. Formalizing Adaptation On-the-Fly. *Electr. Notes Theor. Comput. Sci.*, 255:23–44, 2009.
3. F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
4. 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.
5. 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.
6. 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.
7. G. Bernot, M.-C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, 6(6):387–405, 1991.
8. 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.
9. 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.
10. A. Cansado, C. Canal, G. Salaün, and J. Cubo. A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. *Electr. Notes Theor. Comput. Sci.*, 263:95–110, 2010.
11. 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.
12. 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.

13. 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.
14. P. Crouzen and F. Lang. Smart Reduction. In *Proc. of FASE'11*, volume 6603 of *LNCS*, pages 111–126. Springer, 2011.
15. 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.
16. Borko Furht and Armando Escalante, editors. *Handbook of Cloud Computing*. Springer, 2010.
17. 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.
18. H. Garavel, R. Mateescu, and W. Serwe. Large-scale Distributed Verification using CADP: Beyond Clusters to Grids. In *Proc. of PDMC'12*, 2012.
19. H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proc. of FORTE'99*, volume 156 of *IFIP Conference Proceedings*, pages 185–202. Kluwer, 1999.
20. H. Garavel, C. Viho, and M. Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model Checking, Co-simulation, and Test Generation. *STTT*, 3(3):314–331, 2001.
21. 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.
22. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, 2001.
23. 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.
24. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE TSE*, 16(11):1293–1306, 1990.
25. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
26. C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo using High-level Replacement Systems. *Science of Computer Programming*, 76(1):23–36, 2011.
27. E. Lantreibecq and W. Serwe. Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit using CADP. In *Proc. of FMICS'11*, volume 6959 of *LNCS*, pages 180–195. Springer, 2011.
28. 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.
29. 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.
30. J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia. DADL: Distributed Application Description Language. USC/ISI Technical Report ISI-TR-664, 2010.
31. 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.
32. K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.
33. E. Vassev, M. Hinchey, and A. Quigley. Model Checking for Autonomic Systems Specified with ASSL. In *Proc. of NFM'09*, pages 16–25, 2009.
34. 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.