A Formal approach for the correct deployment of cloud applications

Amel Mammar, Meriem Belguidoum, Saddam Hocine

A Formal Approach for the Correct Deployment of Cloud Applications

Amel Mammar[a,*], Meriem Belguidoum[b], Hocine Hiba[b]

[a] *SAMOVAR, Institut Polytechnique de Paris, Télécom SudParis, France*
[b] *Constantine 2 University, Algeria*

Abstract

Deployment of cloud applications is a complex task. It refers to the enablement of SaaS, PaaS or IaaS solutions that may be accessed on demand by end users. It encompasses all the activities from installation to uninstallation, including reconfiguration, etc. To facilitate the deployment of cloud applications, it is essential to design them as component-based applications in order to favor the design by reuse and reduce the development cost. However, assembling components can be a tedious and error-prone task if sufficient precautions are not taken regarding different constraints, dependencies, and conflicts between components. In this paper, we introduce a formal Event-B-based approach for the modelling and the verification of component-based applications deployment. Our goal is to build correct by-construction systems that fulfill the different constraints regarding the components, the cloud infrastructure, and the deployment process. Basically, our approach starts with an abstract model describing the main concepts of the system. Then different details are gradually introduced by refinement. For each refinement step, proof obligations are produced to ensure the model's correctness. The obtained formal model consists of a precise specification on which mathematical reasoning can be carried out to prove the correctness of our component-based application model and validate its deployment in a cloud environment by using ProB. The presented approach is illustrated through a case study.

Keywords: Cloud computing, Software deployment, Component-based application, Elasticity management, Event-B, Formal verification, Refinement.

*Corresponding author
Email addresses: amel.mammar@telecom-sudparis.eu (Amel Mammar), meriem.belguidoum@univ-constantine2.dz (Meriem Belguidoum), hiba@univ-constantine2.dz (Hocine Hiba)

1. Introduction

1.1. Context and motivation

Over the years, cloud computing [15, 29] has been a strongly emerging IT for data storage, virtualization, and computing power. Its main growing appeal is reducing IT costs and resource availability at any time from any location via the internet. Cloud applications represent a set of interconnected components distributed over virtual machines and executed on servers. Deploying such applications, that is installation/uninstallation of components, activation/deactivation of services [6], etc., is still a challenging problem [13] as it is a tedious and error-prone task if sufficient precautions are not taken regarding different constraints, dependencies, and conflicts between components. On the one hand, it requires ensuring that all intra-dependencies and inter-dependencies of components are satisfied, i.e. requirements are still available in virtual machines, and conflicts between components/services or services/services are always avoided.

On the other hand, the deployment in the Cloud should take into account the most appropriate elasticity management that adjusts dynamically the number of allocated resources to meet changes in workload demands [5, 16, 19]. The elasticity mechanism aims at minimizing the resources of allocated virtual machines (i.e. add/remove VM or increase/decrease the amount of VM resource), reducing the cost of operation, including new requirements, fulfilling the users' expectations in terms of QoS, and performing failure recovery.

Existing deployment solutions are ad hoc and require human intervention to carry out a deployment. The internal application architecture dependencies are not described, and the elasticity is rarely handled. Moreover, they are customized and work for a specific application and a particular platform. For instance, Microsoft Azure only supports Microsoft-based applications. Furthermore, none of these solutions formally specifies or verifies the internal application architecture; including dependencies, conflicting constraints, and deployment phases (installation, uninstallation, and reconfiguration with cloud elasticity management). A more detailed description of these approaches is presented in Section 2.

To ensure the correct deployment of cloud applications, the use of formal methods is highly required. First, it is necessary to build a formal model for cloud application architecture, enabling a precise description regarding different constraints, dependencies, and conflicts between components/services. Indeed, for each service provided by a component we know exactly its software and hardware requirements, dependencies, conflicts, etc. Then, the deployment phases: installation, uninstallation, and reconfiguration must be formally specified. This formalization allows us to ensure that the deployment of component-based applications respects the desired properties like: (i) two conflict components are not deployed on the same VM, (ii) all the services required for the activation of a given service are also activated, (iii) all the resources required for the execution of a service are available, etc.

1.2. Our proposal

In order to address the goals outlined above, we propose a formal approach for modelling and verifying cloud applications architecture and their deployment using the Event-B formal method [3]. Event-B is a refinement-based theorem proving method, which uses set theories and first-order logic to specify the behavior of software and/or hardware elements. The significant advantage aspect of Event-B is that it offers a set of tools, like the Rodin platform [4], for specification and animation that allows building a rigorous development with interactive proof support. We aim at building correct by-construction component-based applications that fulfill the different constraints regarding the component's dependencies, the cloud infrastructure, and the deployment process. Basically, our approach starts with an abstract model describing the main constraints of cloud applications and their deployment phases. Then, different details are gradually introduced by refinement. For each refinement step, proof obligations are produced to ensure the model's correctness. In this study, we used the ProB [26] model checker to validate the development by animation. The proposed approach makes it possible to model, verify and formally validate any application and its deployment. Indeed, before the real deployment, we can detect and prevent deadlock and conflict situations which could be really problematic.

1.3. Paper structure

The remainder of this paper is organized as follows. In Section 2, we present an overview of the literature related to our contribution with a comparative study. Sections 3 and 4 introduce the basic concepts of the Event-B method and the main characteristics of component-based applications. A motivating case study used throughout the paper for illustrating our proposal is described in Section 5. An Event-B model for the deployment of the component-based application is introduced in Section 6. Section 7 describes the verification of the proposed Event-B model using animation and proofs. Finally, we conclude the paper and present insights for future work in Section 8.

2. Related work

Several literature reviews have been conducted in the field of formal verification for cloud systems. For instance, the authors of [14] conducted a systematic literature review focusing on the use of formal methods for verifying the correctness of Cloud and Fog systems. They classified the reviewed studies based on the verification method, modeling language, verification tool, supported properties, and application domains. Similarly, in [38], a detailed survey on formal verification mechanisms and standards in cloud computing for proving the correctness of a system's behavior in cloud computing is provided.

There are also some studies that have investigated the verification of various aspects of cloud computing such as security, resource allocation, SLA (Service

Level Agreement), and performance analysis. Some notable examples include [40, 22, 20, 23, 32, 31]. In [40], a framework is presented for verifying SLAs in a semi-trusted cloud using a testing algorithm to detect violations related to VM memory size and defending against attempts to hide SLA violations, ensuring transparency and accountability in cloud computing. In a similar context. a formal approach based on BRSs and model-checking techniques to model and verify interaction behaviors of SLA-based cloud computing, providing a comprehensive formal description of cloud entities and their properties, is introduced [22]. A formal framework called cloud calculus is proposed in [20] to address security verification challenges in elastic cloud computing platforms. The framework allows for the specification and verification of virtual machine migration and security policy updates, ensuring consistent preservation of global security policies during dynamic changes. Karam et al. [23] introduce a secured objective-driven programming model for cloud-based applications, which is automatically created at runtime by the PAA Cloud Engine. The model incorporates the XACML security annotation representation, providing a separate abstraction layer for enforcing security policies and protecting user information in the cloud. In [32], a risk-based approach is proposed for analyzing security risks in elastic cloud applications. Markov decision process model and probabilistic model checking are used to perform online analysis and decision-making regarding security and elasticity trade-offs. Muniasamy et al., in [31], focus on ensuring the security of cloud-based manufacturing cyber-physical systems (C2PS). They use concepts from Communicating Sequential Processes (CSP) and formal methods to model and verify security properties in C2PS. The research emphasizes the importance of composability and scalability in enhancing system security, providing formalisms and authentication conditions for verifying C2PS.

While the above researches address more security concerns in cloud computing,, our current study serves a distinct purpose that aims at modeling and verifying of the architecture, the deployment, and the elasticity of cloud applications. It concerns several aspects in terms of dependency management and parameterized deployment and elasticity (e.g., conflicting services), optional services (personalized configuration), etc.). Hereafter, we report on the most relevant approaches dealing with these aspects and categorized according to the type of formal method used: *Proof-based* and *Model-checking-based* methods.

2.1. Proof-based methods

In [18], an Event-B-based approach is proposed for formal verification of elastic SCA-based application. The authors formally model the SCA artifacts and define the Event-B events to represent the elasticity mechanisms (duplication and consolidation). The verification is driven using the proof obligations and the ProB animator [26].

Similarly, to this work, the approach described in [1] is also based on Event-B. It starts by specifying the main concepts of PFOS (*Package-based Free and Open Source*) software, and then refines them through multiple steps to ensure the correctness of the horizontal and the vertical elasticity properties for composite

PFOS. The authors perform the verification using the integrated provers of the Rodin platform and present an animation process using ProB to trace possible modelling errors. However, these approaches do not consider the component deployment and the conflicting constraints.

2.2. Model-checking-based methods

In [33, 41], a formal framework based on the SMT solver Z3 [30] is proposed for modelling and verification of cloud applications. The authors start by specifying basic cloud services that are used then in a compositional approach building more elaborated cloud services. The proposed framework focuses on data and time-related properties of cloud services. Contrary to our approach, service dependencies, like conflict avoidance and optional services, are not considered.

In [13], a decentralized-based self-deployment protocol is introduced to automatically configure a set of software elements deployed on virtual machines. The proposed protocol is specified and verified using the LOTOS NT (LNT) language [11] and the CADP model checker [17]. This approach only considers a restrictive set of requirements, that are, the mandatory services. Inter-dependencies requirements, like conflicts between components and services, are not considered. Moreover, model-checking suffers from the well-known state explosion problem. In other words, this technique may fail when verifying large systems. Madeus [12] is a component-based deployment model for complex distributed software. It uses Petri nets to describe the life cycle of components and the dependencies between them in the form of a dependency graph. This dependency graph is then used to reduce deployment time by parallelizing deployment actions. However, this approach does not address the verification of the proposed model.

In [10], the planning problem of the deployment and redeployment of microservice architectures is considered. Real-world microservice architecture is modelled using the abstract behavioral specification language (ABS) [21] to allow for proving formal properties and realizing a set of deployment plans. Although the approach covers the required/provided/conflicting interfaces and bind/unbind between them, it does not take into account optional and vertical elasticity requirements. Moreover, as it uses a simulation technique to validate the deployment, the correctness of the deployment is not ensured.

In [24], Bigraphical Reactive Systems and Bigraphs are used to model the cloud system's structure and elastic behaviors, with Maude encoding for autonomic executability and verification. Future work includes enhancing resource management through load balancing and vertical scaling strategies.

In [39], Bigraphical Reactive Systems and Bigraphs are employed to model the cloud's elasticity structure and dynamic behavior, focusing on cross-layer elasticity and incorporating horizontal and vertical scaling strategies. The correctness of the approach is verified using the BigMC model checker tool.

Authors in [9] propose a model-based approach for formal verification and performance analysis of dynamic load-balancing protocols in cloud environ-

ments. A formal modeling language called BIP (Behavior, Interaction, Priority) is used to model the cloud architecture and load-balancing protocols, and then use a stochastic extension of BIP to analyze the performance of these protocols. In addition, the authors use the PRISM model checker to perform probabilistic model checking of the BIP model.

2.3. Comparative Analysis Criteria

In this section, we outline the criteria for conducting a comparative analysis of the synthesis-related aspects discussed in the related work. The chosen criteria aim to provide a comprehensive evaluation of the papers based on their application modeling, modeled and verified properties, formalism used, and the mechanisms or tools employed for formal verification. By considering these criteria, we can gain insights into the strengths and limitations of different approaches in addressing the challenges and requirements of synthesis in cloud computing. Table 1 presents a comparative analysis based on the following criteria:

- **Application Modelling (App. Model.)**: states whether the optional requirements (OR) and the conflict requirements (CR) are covered.
- **Modelled and Verified properties (Mod. and Verif. Props.)**: defines the set of deployment and elasticity properties modelled and verified by the approach.
- **Formalism**: presents the formal language/method used for the modelling and the verification.
- **Formal Verification (For. Ver)**: specifies which mechanisms/tools are used in the verification phase.

Work	**App. Model.**		**Mod. and Verif. Props.**		**Formalism**	**For. Ver.**
	OR	CR	Deployment	Elasticity		
[18]	-	-	-	Duplication/ Consolidation	Event-B	Proofs/ProB
[1]	-	-	Installation Uninstallation	Vertical/ Horizontal	Event-B	Proofs/ ProB
[33, 41]	-	-	Data and time-related properties	-	Z3	SMT solver
[13]	+	-	Configuration / Activation	-	LNT	Model checking
[12]	-	-	Operational semantics	-	Madeus	-
[10]	-	+	Reconfiguration	Horizontal	ABS	-
[24]	-	-	Reconfiguration	Vertical/ Horizontal	BRS, bigraph	Maude LTL model checker
[39]	-	-	N/P	Vertical/ Horizontal	BRS, Bigraphs	BigMC

[22]	-	-	N/P only interaction behaviors of SLA	-	BRS	BigMC and NuSMV symbolic model checker
[9]	-	-	N/P	N/P only load balancing	BIP	stochastic extension of BIP, PRISM model checker
Our approach	+	+	Installation/ Uninstallation/ Reconfiguration	Replication /Resizing	Event-B	Proofs/ProB

Table 1: Comparative study of formal approaches for cloud application deployment

In light of the aforementioned points, it is evident that no single approach fulfills all the criteria simultaneously: 1) modelling optional and conflict requirements, 2) modelling and verifying deployment and elasticity properties 3) Formal specification and verification, and 4) using verification and validation tools.

This paper contributes to the state of the art by:

1. introducing an Event-B-based approach that enables the modeling and verification of component-based application architecture taking into account the deployment phases (installation, uninstallation, and reconfiguration), as well as horizontal and vertical elasticity strategies.

2. specifying and verifying components' intra-dependencies [8] in terms of optional, mandatory, and conflicting requirements between components and services.

3. proposing a proof-based approach that scales up and can handle applications of any size.

However, our approach has certain limitations that we intend to address in future work:

- *Dynamic reconfiguration*: The current approach does not facilitate dynamic adaptation to new configurations of virtual machines (VMs), such as adding, removing, or updating resources, while the application is running. In our approach, reconfiguration is performed offline.

- *Considering other elasticity strategies*: Although this paper focuses on resource resizing and VM duplication, other elasticity strategies could be explored, such as duplication/consolidation of overloaded/under-provisioned services. Additionally, incorporating timed constraints on elasticity decisions could prevent repetitive duplication/consolidation operations. For example, a duplication operation would only be performed if a VM remains overloaded for a specified period of time.

- *Security issues*: To address security concerns, integrating an access control model, such as the one defined in [27, 34, 36], to restrict access to certain resources would be beneficial.

3. Component-based applications characteristics

In this section, we focus on the main characteristics of the component-based applications which we consider. A component-based application can be viewed as a set of components interacting together through their required and provided services. A component may provide one or several services while each provided service generally requires one or several services, which are provided by other components. In addition to the required services, different constraints can be expressed for each provided service as hardware resources or software dependencies. In a nutshell, a component-based application should respect the following requirements introduced in [7, 8]:

Req1 The domain we deal with is composed of a set of virtual machines (VMs).

Req2 We have to deal with a set of components: *Postfix* and *Sendmail* are examples of components.

Req3 A conflict constraint can be defined on two components to state that they cannot be deployed, at the same time on the same VM. For instance, the components *Postfix* and *Sendmail* are in conflict because the installation of both components will create a symlink called /usr/bin/mailq, pointing to the main executable. Also, Postfix does emulate Sendmail's implementation, they serve the same purpose but accomplish it by rather different means. Indeed, Postfix will actually install an executable called "Sendmail" for compatibility reasons. So both software has common files that make them conflicting components.

Req4 A component can be deployed on one or several VMs. Of course, at a given moment, a component may not be deployed on any VM. One or several components can be deployed on the same VM.

Req5 A component provides one or several services: at least one of them is mandatory while others could be optional. *Postfix*, for instance, provides *MTA* as a mandatory service, while the services *AmavisdMTA* and *AntivirusMTA* are optional and can be activated if needed.

Req6 Each service of a deployed component can be active or not.

Req7 After the deployment of a component, all its mandatory provided services are activated. These mandatory services remain activated while the component is not uninstalled. When deploying the service *Postfix*, for instance, the mandatory service *MTA* is directly activated and cannot be deactivated until uninstalling the component.

Req8 An optional service can be activated/deactivated during the deployment of the component. As the services *AmavisdMTA* and *Anti-virusMTA* are optional, they can be activated/deactivated during the deployment of *Postfix*.

Req9 On a service, the following constraints can be defined:

a. *Conflicting services*: it denotes a conflicting constraint between two different services regardless of the components that provide them. It means that two services cannot be active at the same time on the same VM. Let us note that at least one of these services is optional, otherwise, this comes to conflicting components, which are already covered by the requirement **Req3**. For instance, any *anti-virus* service is in conflict with the service *ftp*.

b. *Conflicting services of given components*: it means that two services provided by two specific components cannot be active on the same VM. For instance, the service *Anti-virusMTA*, of component *Postfix*, is in conflict with the service *Firewall* provided by the component *OS Firewal*.

c. *Needed resources*: to be executed, a service needs a specific amount of RAM and disk space. Let us note that the same RAM may be shared by several services whereas it should be enough disk space for each service. For instance, a RAM of 2 Gb can be shared by several services if the maximum capacity needed by each of them does not exceed 2 Gb. However, in terms of disk space, the size of the disk should be at least equal to the sum of the capacity needed by each of those services.

d. *Needed service of a specific component*: it means that a service requires another service from a specific component to be activated. For instance, the service *AntiVirus* of the component *ClamAV* is needed by the service *AntiVirusMTA* of the component *Postfix*.

e. *Needed services*: it means that the activation of a service requires some services regardless of the components that provide them. For instance, the service *Mailbox* needs the service *DNS* to work.

Req10 A component can be uninstalled only if none of its provided services is used by other components in a mandatory way (see Requiement **Req9.d**).

Req11 Each VM provides different resources like RAM, Memory disk space, etc.

Req12 A VM can be, among others, in one of the two following states: *overloaded*, and *unused*. An *overloaded* VM means that at least one of its resources is fully used, whereas an *unused* VM denotes a VM on which no component is deployed.

Req13 When a VM becomes unused (no component is installed on it), this latter can be removed.

Req14 When the existing VMs become overloaded, new ones can be added.

Req15 For a given VM, we can decide to increase (resp. decrease) the amount of a given resource when the VM is over-used (resp. under-used).

Req16 To keep the costs at an acceptable level, we put a constraint on the maximum number of installed VMs.

In Section 6, we present a formalization of these requirements using EVENT-B. The goal of this formal model is to obtain precise and unambiguous specifications on which mathematical reasoning can be carried out, in order to prove the correctness of our component-based application model and to verify elasticity management during its deployment in a cloud environment. In this paper, a correct elasticity management means the possibility to execute elasticity actions (adding/removing VMs, increasing/decreasing resources) when conditions specified in requirements **Req13-Req16** are fulfilled. Optimization aspects related to elasticity, that is the more appropriate strategy to execute an elasticity operation, are not considered.

4. EVENT-B method

Event-B is the successor of the B method [2], it permits to model discrete systems using mathematical notations. The complexity of a system is mastered thanks to the refinement concept that introduce gradually the different parts that constitute the system starting from an abstract model to a more concrete one. An Event-B specification is made of two elements: *context* and *machine*. A context describes the static part of an Event-B specification; it consists of constants C and sets S (user-defined types) together with axioms Ax that specify their properties. The dynamic part of an Event-B specification is included in a machine that defines variables V and a set of events. The possible values that the variables hold are restricted using an invariant, denoted Inv, written using a first-order predicate on the state variables. Each event is of the form $(G \mid Act)$; it can be executed if it is enabled, i.e. all the conditions G, named guards, hold. In this case, the substitutions Act, called actions, are applied over variables. For each event, the following proof obligation is generated to ensure that its execution maintains the invariant:

$$\forall\ (S,\ C,\ V).\ (Ax \wedge G \wedge Inv \Rightarrow [Act]Inv)$$

Refinement is a process of enriching a model in order to augment the functionality being modelled, or/and explain how some purposes are achieved. Both EVENT-B *context* and *machine* can be refined. A context can be extended by defining new sets S_r and/or constants C_r together with new axioms Ax_r. A machine is refined by adding new variables and/or replacing existing variables with new ones V_r that are typed with an additional invariant Inv_r. New events can also be introduced to implicitly refine a **skip** event. In this paper, we are interested in safety properties that mean that no bad situation can happen.

Liveness properties that ensure that a desired situation will eventually happen are not considered. Therefore, we accept scenarios in which the triggering of abstract events are prevented by concrete events that are triggered infinitely many times. In other words, our refinement may be subject to livelock without affecting the safety properties. Therefore, the correctness of a refinement we consider in this paper comes down to establishing that the effect of the refined event is included in that of the abstract one. For each event $(G \mid Act)$ refined by the event $(G_r \mid Act_r)$, we have to establish the following two proof obligations:

- *guard refinement*: the guard of the refined event should be stronger than the guard of the abstract one:

  $\forall(S, C, S_r, C_r, V, V_r)\ .\ (Ax \wedge Ax_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$

- *Simulation*: the effect of the refined action should be stronger than the effect of the abstract one:

  $\forall\ (S,\ C,\ S_r,\ C_r,\ V,\ V_r,\ X,\ X_r).\ (A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow [Act_r]\neg[Act]\neg Inv_r)$

To discharge the different proof obligations, the Rodin[1] platform offers an automatic prover, but also the possibility to plug additional external provers like the SMT and Atelier B provers that we use in this work. Both provers offer automatic and interactive options to discharge the proof obligations. Table A.5 in Appendix A gives the semantics of the different mathematical symbols used in the rest of the paper.

5. A motivating case study: Zimbra software

We illustrate our approach through the example of Zimbra Collaboration Suite (ZCS)[2]. It is a collaborative software and a complete messaging solution. The choice of such a case study is motivated by its entities (components, services) and characteristics that permit to show different aspects and constraints, constituting a typical could application. The installation and utilization requirements (software and/or hardware) of Zimbra provide us the possibility to better explore, test, verify and validate all the aspects and properties of our approach regarding different constraints, dependencies, optional requirements, and conflicts between components and services. ZCS includes an email and a calendar server, document sharing and storing, instant messaging, and simplified administrative controls using a web interface. Figure 1 depicts the Zimbra component-based architecture, it represents a set of interconnected components described as follows:

[1] http://www.event-b.org/install.html
[2] https://www.zimbra.com/email-server-software/

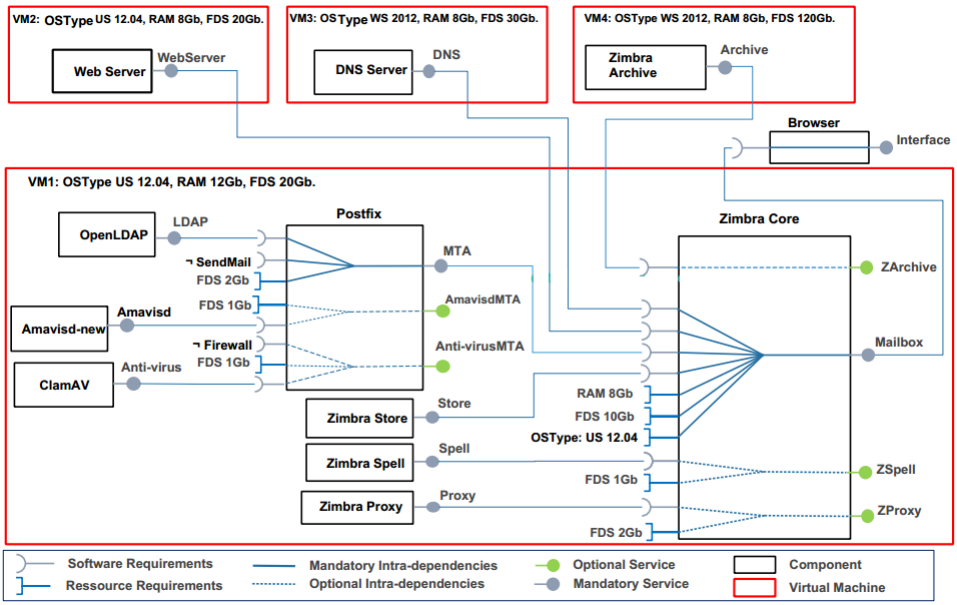

Figure 1: ZCS component-based architecture

- **Zimbra Core**: it is the main component of the application and provides a *Mailbox* service that ensures access to end-users and administrators by the browser and three optional services: *ZArchive*, *ZSpell* and *ZProxy*. Its requirements are described as follows:
  - The *Zimbra Core* must be installed on a VM that has an Ubuntu server 12.04 as an operating system (OS).
  - The mandatory service *Mailbox* requires four services: *MTA*, *DNS*, *WebServer* and *Store*, and hardware resources: 8Gb of RAM and 10Gb of free disk space (FDS).
  - When it is activated, the optional service *ZArchive* requires the service *Archive*.
  - When it is activated, the optional service *ZSpell* requires the service *Spell*.
  - When it is activated, the optional service *ZProxy* requires the service *Proxy*.
- **Zimbra Proxy**: it is a high-performance reverse proxy component for passing IMAP(S)/POP(S)/HTTP(S) client requests to other internal Zimbra services. It offers the *Proxy* service.
- **Zimbra Store**: it provides the *Store* service to the Zimbra core component. It is a storage space that includes Datastore, i.e., MySQL database

where internal mailbox IDs are linked with user accounts, the Message store, i.e., where all email messages and file attachments reside, and the Index store, i.e., where each message is indexed as it enters the system.

- **OpenLDAP**: it is an implementation of the Lightweight Directory Access Protocol (LDAP) used to guarantee user authentication. It provides the *LDAP* service.
- **Amavisd-new**: it is a reliable content filter used as a high-performance interface between MTA such as Postfix and one or more content checkers: virus scanners, and/or SpamAssassin. It provides the *Amavisd* service.
- **ClamAV**: it denotes an anti-virus scanner that protects against malicious files. It provides *Anti-virus* service.
- **Zimbra Spell**: it represents an open-source spell checker used on Zimbra. It provides the *Spell* service.
- **Postfix**: it is a mail transfer agent (MTA) that receives email via SMTP and routes each message to the appropriate Zimbra mailbox. It provides three services:
  - *MTA* service: it is a mandatory service and requires LDAP service and 2Gb of FDS.
  - *AmavisdMTA* service: it is an optional service and requires the Amavisd service.
  - *Anti-virusMTA* service: it is an optional service and requires the service Anti-virus from ClamAV component and the deactivation of the Firewall service, provided by the *OS Firewall* component, with which it is conflicting.
- **SendMail**: it is an open-source program that is responsible for the delivery and sending of emails. It supports a wide range of email transfer and delivery methods, including popular SMTP. This component provides the *SMTA* service, which requires an *LDAP* service and 2 Gb of FDS. *Sendmail* is conflicting with *Postfix* component.
- **OS Firewall**: this component is defined as a network security program that controls incoming and outgoing connections based on predetermined security rules. It provides a *Firewall* service that is in conflict with the service *Anti-virusMTA* of the *PostFix* component.
- **Web Server**: it denotes the web application server that Zimbra runs in. It must be installed in a separate VM that has: an Ubuntu server 12.04 OS, RAM 8Gb, and 20Gb of FDS.
- **DNS Server**: it is needed for Domain Name System. The purpose of the DNS is to translate the domain names to the IP addresses and vice-versa. The DNS is used by Zimbra to find out the mail server of the other side. It

requires a VM that has: Windows Server 2012 OS, RAM 8Gb, and 30Gb of FDS.

Component	Provided Service	Type	Service Requirements	
			Required Services	Resources
Zimbra Core	Mailbox	M	DNS, WebServer, MTA, Store	OSType: Ubuntu12.04, RAM 8Gb, FDS 10Gb
	ZSpell	O	Spell	FDS 1Gb
	ZProxy	O	Proxy	FDS 2Gb
	ZArchive	O	Archive	-
Postfix	MTA	M	LDAP	FDS 2Gb
	AmavisdMTA	O	Amavisd	FDS 1Gb
	Anti-virusMTA	O	Anti-virus	FDS 1Gb
SendMail	SMTA	M	LDAP	FDS 2Gb
OS Firewall	Firewall	M	-	-
OpenLDAP	LDAP	M	-	-
Amavisd-new	Amavisd	M	-	-
ClamAV	Anti-virus	M	-	-
Zimbra Proxy	Proxy	M	-	-
Zimbra Spell	Spell	M	-	-
Zimbra Store	Store	M	-	-
Web Server	WebServer	M	-	OSType: US12.04, RAM 8Gb, FDS 20Gb
DNS Server	DNS	M	-	OSType: WS2012, RAM 8Gb, FDS 30Gb
Zimbra Archive	Archive	M	-	OSType: WS2012, RAM 8Gb, FDS 120Gb
Browser	Interface	M	Mailbox	-

Table 2: ZCS intra-dependencies description

Table 2 summarizes all the Zimbra Collaboration Suite intra-dependencies. For each component of ZCS we describe its intra-dependencies, including for each provided service its type (mandatory 'M' or optional 'O'), its requirements in terms of conflicting constraints, resources (RAM, OS, FDS, etc.) or required services.

As we can notice, to ensure the correct deployment of such an application, we have to cope with several interrelated requirements. Doing this by hand would be a difficult and error-prone task. The next section introduces a generic Event-B modelling for the correct by-construction deployment of component-based applications. We also show how the defined generic modelling is used for the deployment of the Zimbra Collaboration Suite.

6. Event-B specification of component-based applications

This section introduces our modelling of elastic component-based applications using Event-B. In this paper, we describe the main modeling elements, the complete Event-B specification, whose architecture is depicted in Figure 2, is available at [28]. The built specification is composed of four levels (machines

Requirements	Component	Invariant/Event/Axiom
Req1	C1	Axiom *axm1*
Req2	C1	Axiom *axm1*
Req3	C1	Axioms *axm2-axm4*
	M1	Invariant *inv3*
Req4	M1	Invariant *inv2*
Req5	C2	Axioms *axm1-axm4*
Req6	M2	Invariant *inv1*
Req7	M2	Invariant *inv2*
Req8	M2	Events putActive and putNotActive
Req9(a-b, d-e)	C3 and M3	Axioms of C3 and the invariants of M3
Req9 (c)	M4	Invariants *inv2* and *inv4*
Req10	M3	Invariant *inv4*
Req11	C4	Axiom *axm1*
	M4	Invariant *inv1*
Req12	M4	Invariant *inv3*
Req13	M1	Event removeVm
Req14	M4	Event addVm
Req15	M4	Event updateRessource
Req16	M4	Invariant inv5

Table 3: Cross-reference between the components of our model and the requirements of Section 3

and contexts) linked with sees/extends/refines relations. In the first level (M1 + C1), we model the components along with VMs on which they can be deployed. The second level (M2 + C2) describes the services of components along with their activation/deactivation. Different constraints associated with services and components are modeled in the third level (M3 + C3). Finally, the last level (M4 + C4) deals with the elasticity mechanism. Table 3 relates the components of our model with the requirements listed in Section 3.

For a step-by-step validation purpose, at each level i, we validate the developed model M_i by creating a new machine M_iInst that refines Mi without introducing any modification. This machine sees a new context C_iInst that extends both contexts C_i and $C_{i-1}Inst$. More details about the validation phases are provided in Section 7.

The above modelling steps are illustrated through the case study described in the previous section. It is noteworthy that several solutions (architectures) are possible for modeling a system in Event-B. The main criterion that makes one of them better is the correctness proof complexity. A good strategy is to introduce a particular system characteristic at each refinement level. This is what we have done in the present work.

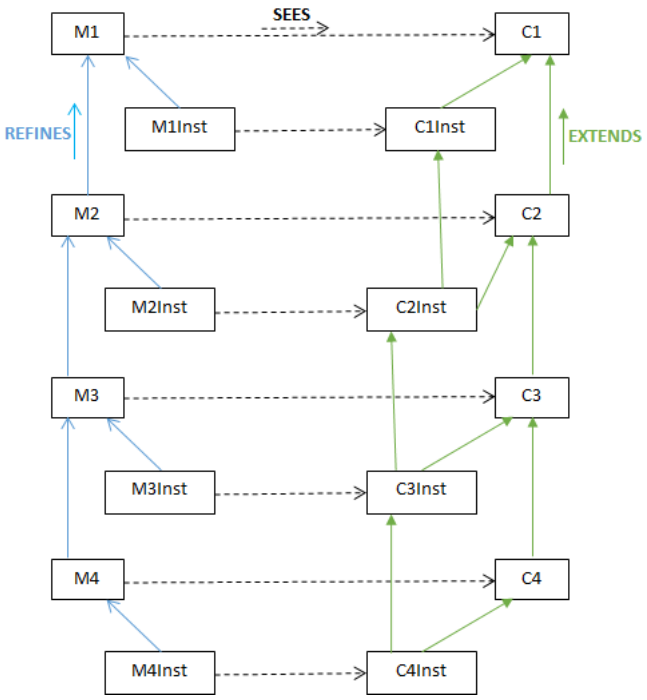

Figure 2: Architecture of the Event-B models

6.1. Modelling component deployment

At the abstract level, the component-based application is seen as a set of components deployed on a set of VMs (Requirements **Req1** and **Req2**). In Event-B, this is formalized as follows. We introduce a context C1 that defines two abstract sets *Components* and *VMs* to represent the set of all possible components and virtual machines, respectively (Axiom *axm1*). In this context, we define a constant *conflictComponents*, as a relation, to model the fact that a component is in conflict with some other components (Requirement **Req3**, see axioms *axm2*-*axm4* of Figure 3). Axiom *axm3* states that a component cannot be in conflict with itself; Axiom *axm4* specifies that the conflict property is symmetric.

CONTEXT *C1*
SETS
Components, VMS
CONSTANTS
conflictComponents
AXIOMS
axm1: $\mathbf{finite}(Components) \land \mathbf{finite}(VMS)$
axm2: $conflictComponents \in Components \leftrightarrow Components$
axm3: $\mathbf{id} \cap conflictComponents = \varnothing$
axm4: $conflictComponents = conflictComponents^{-1}$
END

Figure 3: EVENT-B context C1

To model the case study, the sets and constants of C1 are valued as follows:

CONTEXT *C1Inst*
EXTENDS *C1*
AXIOMS
axm1: $\mathbf{partition}(Components, \{ZimbraCore\}, \{ZimbraProxy\}, \{ZimbraStore\}, \{OpenLDAP\}, \{Amavisdnew\}, \{ClamAV\}, \{ZimbraSpell\}, \{Postfix\} \{SendMail\}, \{OSFirewall\}, \{WebServer\}, \{DNSServer\})$
axm2: $\mathbf{partition}(VMS, \{VM1\}, \{VM2\}, \{VM3\}, \{VM4\})$
axm3: $conflcitComponents = \{Sendmail \mapsto Postfix, Postfix \mapsto Sendmail, \ldots\}$
END

Context C1 is seen by Machine M1 that models the deployment of components of VMs (see Figure 4). For that purpose, we define two variables: Variable *vms* to represent the set of existing VMs at a given moment (Invariant *inv1*), and Variable *deployedOn* to model the deployment of a component on different VMs. Invariant *inv2* states that a component can be deployed on several VMs (Requirement **Req4**) whereas *inv3* prohibits the deployment of conflict components on the same VM (Requirement **Req3**).

MACHINE *M1*
SEES *C1*
VARIABLES
vms, deployedOn
INVARIANTS
inv1: $vms \subseteq VMS$
inv2: $deployedOn \in Components \leftrightarrow vms$
inv3: $\forall\, c1, c2.\ c1 \mapsto c2 \in conflictComponents \Rightarrow deployedOn[\{c1\}] \cap deployedOn[\{c2\}] = \varnothing$
END

Figure 4: Machine *M1*: component deployment

To make the state of the application evolve, four events are defined in Machine M1 in order to add/remove a VM and install/uninstall a component. Figure 5 gives the Event-B specification of the events that add a new VM or remove an existing one. Guard *grd*1 of Event addVm checks that the VM *vm* to add does not exist yet. Similarly, Guard *grd*2 of Event removeVM checks that no component is deployed on the VM *vm* that will be removed.

Event addVm $\widehat{=}$
any
vm
where
grd1: $vm \in VMS \setminus vms$
then
act1: $vms := vms \cup \{vm\}$
end
Event removeVm $\widehat{=}$
any
vm
where
grd1: $vm \in vms$
grd2: $vm \notin \mathbf{ran}(deployedOn)$
then
act1: $vms := vms \setminus \{vm\}$
end

Figure 5: Event-B events to add and remove a VM

Figure 6 gives the events that install and uninstall a component c on a given VM *vm*. Guard *grd3*, of Event Install, checks that no conflict component is already deployed on the related *vm*. Similarly, Guard *grd2*, of Event UnInstall, checks that the component c is actually deployed on the VM *vm*. For instance, to deploy the component *ZimbraCore* on the *VM1*, we trigger the event Install with parameters *ZimbraCore* and *VM1*. After, verifying that the guards are satisfied, the action *act1* is executed by adding the tuple $(ZimbraCore \mapsto VM1)$ to the variable *deployedOn*.

Event Install $\widehat{=}$
any
c, vm
where
grd1: $c \in Components \land vm \in vms$
grd2: $c \mapsto vm \notin deployedOn$
grd3: $conflictComponents[\{c\}] \cap deployedOn^{-1}[\{vm\}] = \varnothing$
then
act1: $deployedOn := deployedOn \cup \{c \mapsto vm\}$
end
Event UnInstall $\widehat{=}$
any
c, vm
where
grd1: $c \in Components \land vm \in vms$
grd2: $c \mapsto vm \in deployedOn$
then
act1: $deployedOn := deployedOn \setminus \{c \mapsto vm\}$
end

Figure 6: Installing and uninstalling a component on a VM

6.2. Modelling component services

The next step concerns the modelling of component services. So, we extend C1 by a new context C2 that defines an abstract set *Services* to represent the set of all possible services and a constant *mandProvSers* (resp. *optProvSers*) to denote the set of mandatory (resp. optional) services of a component (see Figure 7). Axiom *axm1* (resp. *axm3*) defines the mandatory (resp. optional) services of a component. Axiom *axm2* states that each component should have at least one mandatory service whereas *axm4* specifies that each service of a component is either mandatory or optional (Requirement **Req5**).

CONTEXT *C2*
EXTENDS *C1*
SETS
Services
CONSTANTS
mandProvSers, optProvSers
AXIOMS
axm1: $mandProvSers \in Components \leftrightarrow Services$
axm2: $\mathbf{dom}(mandProvSers) = Components$
axm3: $optProvSers \in Components \leftrightarrow Services$
axm4: $mandProvSers \cap optProvSers = \varnothing$
END

Figure 7: Axioms of the Event-B context C2

For our case study, constants of C2 are instantiated as follows:

CONTEXT *C2Inst*
EXTENDS *C2, C1Inst*
AXIOMS
axm1: **partition**(*Services*, {*Mailbox*}, {*ZSpell*},{*ZProxy*}, {*ZArchive*},...)
axm2: $mandProvSers = \{ZimbraCore \mapsto Mailbox, \dots\}$
axm3: $optProvSers = \{ZimbraCore \mapsto ZSpell, ZimbraCore \mapsto ZProxy, ZimbraCore \mapsto ZArchive, \dots\}$
END

At this level, we model the activation/deactivation of services of a component when this latter is deployed on a given VM (Requirement **Req6**). To this aim, a new machine M2, which refines Machine M1, introduces a new variable *isActive* along with the invariant depicted in Figure 8. Invariant *inv1* models the activation of services as a subset of tuples $c \mapsto (vm \mapsto s)$ to state that the service s of the component c deployed on VM vm is active. This invariant uses the direct product operator ($\otimes$) whose formal definition id provided in Appendix A. Invariant *inv2* states that the mandatory provided services of a deployed component should be always activated (Requirement **Req7**).

MACHINE *M2*
REFINES *M1*
SEES *C2*
VARIABLES
isActive
INVARIANTS
inv1: $isActive \subseteq deployedOn \otimes (mandProvSers \cup optProvSers)$
inv2: $deployedOn \otimes mandProvSers \subseteq isActive$
END

Figure 8: Event-B invariants of Machine M2

To make the previous invariant preserved by the execution of the different events, we refine the event Install and UnInstall by adding actions that update the variable *isActivate*. For example, we add the following action for Event *Install* to make all the mandatory services of c activated:

$$isActive := isActive \cup (\{c\} \times (\{vm\} \times mandProvSers[\{c\}]))$$

Likewise, we add the following action to the event UnInstall to remove the corresponding services from the set *isActive*:

$$isActive := isActive \setminus (\{c\} \times (\{vm\} \times Services))$$

Moreover, Machine M2 introduces two additional events that activate/deactivate an optional service of a deployed component (Requirement **Req8**). As we can notice, both events check, in Guard grd1, that the component is deployed on the related VM; Guard grd2, of Event putActive, checks that the service s is optional since mandatory ones are always activated and cannot be deactivated

as specified in Guard gurd2 of Event putNotActive (see Figure 9). Let us notice that the **ANY** construct nondeterministically selects a service/component among a set of possible ones that verify the guards.

Event putActive $\widehat{=}$
  any
    c_0, s_0, vm_0
  where
    grd1: $c_0 \in Components \land c_0 \mapsto vm_0 \in deployedOn$
    grd2: $s_0 \in optProvSers[\{c_0\}] \land c_0 \mapsto (vm_0 \mapsto s_0) \notin isActive$
  then
    act1: $isActive := isActive \cup \{c_0 \mapsto (vm_0 \mapsto s_0)\}$
  end
Event putNotActive $\widehat{=}$
  any
    c_0, s_0, vm_0
  where
    grd1: $c_0 \in Components \land c_0 \mapsto vm_0 \in deployedOn$
    grd2: $s_0 \in optProvSers[\{c_0\}] \land c_0 \mapsto (vm_0 \mapsto s_0) \in isActive$
  then
    act1: $isActive := isActive \setminus \{c_0 \mapsto (vm_0 \mapsto s_0)\}$
  end

Figure 9: Activation/deactivation events in Event-B

Let us notice that, at this level, any service can be active even if the required services are not available and/or its constraints are not satisfied. This is made possible since such constraints are not specified yet, and they will be introduced in the next level.

6.3. Modelling service constraints

In the next level, we model the constraints that can be defined on the service of a component, which are some of those specified in Requirement **Req9** (a, b, d and e) related to a specific need/prohibition of a given service/component . To do that, a new context C3 is introduced by extending the Context C2. Each constraint on service is modelled by a constant in the context C3. To this purpose, we define, in Figure 10, the constants *conflictServ*, *conflictCompServ*, *neededServ* and *neededServComp* to model respectively the conflict between services (**Req9a**, Axiom *axm1*), between services of specific components (**Req9b**, Axiom *axm4*), the services required by a given service (**Req9e** with Axioms *axm8* and **Req9d** with *axm10*). Axiom *axm2* states each service cannot be in conflict with itself and that the relation *conflictServ* is symmetric. Axiom *axm3* specifies that a mandatory service cannot be in conflict with an optional/mandatory service of the same component. Axiom *axm5* states that if two services $s1$ and $s2$ are in conflict, then for any two components $c1$ and $c2$, we do not need to specify that the couples ($c1$, $s1$) and ($c2$, $s2$) are also in conflict with respect to *conflictCompServ*. Furthermore, the axiom *axm6* states that if two services are in conflict then at least one of them is optional. Axiom *axm7* specifies that

if two components $c1$ and $c2$ are in conflict, we do not need to specify that their services are in conflict too. Axiom *axm9* specifies that the required service should not be in conflict with the related service, Finally, Axiom *axm11* and *axm12* respectively specify that for each service t of a component z needed by the service y of the component x, the components x and z are not in conflict, and y and t are not in conflict.

CONTEXT *C3*
EXTENDS *C2*
CONSTANTS
$conflictServ$, $conflictCompServ$, $neededServ$, $neededServComp$
AXIOMS
axm1: $conflictServ \in Services \leftrightarrow Services$
axm2: $\mathbf{id} \cap conflictServ = \varnothing \wedge conflictServ = conflictServ^{-1}$
axm3: $\forall\ c.\ c \in Components \Rightarrow (mandProvSers\ [\{c\}] \times (mandProvSers \cup optProvSers)[\{c\}]) \cap conflictServ = \varnothing$
axm4: $conflictCompServ \in mandProvSers \cup optProvSers \leftrightarrow mandProvSers \cup optProvSers \wedge conflictCompServ = conflictCompServ^{-1} \wedge \mathbf{id} \cap conflictCompServ = \varnothing$
axm5: $\forall\ c,s.\ conflictCompServ[\{c \mapsto s\}] \cap (Components \times conflictServ[\{s\}]) = \varnothing$
axm6: $\forall\ s,\ c.\ s \in mandProvSers[\{c\}] \Rightarrow conflictCompServ[\{c \mapsto s\}] \subseteq optProvSers$
axm7: $\forall\ c1,\ c2.\ c1 \mapsto c2 \in conflictComponents \Rightarrow (\{c1\} \times Services) \times (\{c2\} \times Services)) \cap conflictCompServ = \varnothing$
axm8: $neededServ \in mandProvSers \cup optProvSers \leftrightarrow Services$
axm9: $\forall\ c,\ s.\ c \mapsto s \in \mathbf{dom}(neededServ) \Rightarrow neededServ[\{c \mapsto s\}] \cap conflictServ[\{s\}] = \varnothing$
axm10: $neededServComp \in mandProvSers \cup optProvSers \leftrightarrow mandProvSers \cup optProvSers$
axm11: $\forall\ x,y,z,t.\ (x \mapsto y) \mapsto (z \mapsto t) \in neededServComp \Rightarrow z \notin conflictComponents[\{x\}]$
axm12: $\forall\ x,y,z,t.\ (x \mapsto y) \mapsto (z \mapsto t) \in neededServComp \Rightarrow (x \mapsto y) \mapsto (z \mapsto t) \notin conflictCompServ$
END

Figure 10: Axioms of the EVENT-B context C3

Instantiating this context with the case study gives:

CONTEXT *C3Inst*
EXTENDS *C3, C2Inst*
AXIOMS

axm1: $conflictServ = \varnothing$

axm2: $conflictCompServ = \{(Postfix \mapsto AntiVirusMTA) \mapsto (OSFirewall \mapsto Firewall)\}$

axm3: $neededServ = \{ZimbraCore \mapsto Mailbox \mapsto MTA,$
$ZimbraCore \mapsto Mailbox \mapsto DNS,$
$ZimbraCore \mapsto Mailbox \mapsto Webserver,$
$ZimbraCore \mapsto Mailbox \mapsto Store,$
$ZimbraCore \mapsto ZArchive \mapsto Archive, \ldots\}$

axm4: $neededServComp = \{(Postfix \mapsto AntiVirusMTA) \mapsto (ClamAV \mapsto AntiVirus)\}$

END

To take the constraints specified in C3 into account, we refine Machine M2 by a new machine M3 that introduces some invariants without any new variable. These invariants state that the deployment of components and the activation of services must be performed with respect to these constraints.

MACHINE *M3*
REFINES *M2*
SEES *C3*
INVARIANTS

inv1: $\forall\, c, s, c1, s1, vm.\ s \mapsto s1 \in conflictServ \wedge c \mapsto (vm \mapsto s) \in isActive \Rightarrow c1 \mapsto (vm \mapsto s1) \notin isActive$

inv2: $\forall\, c, s, c1, s1, vm.\ c \mapsto (vm \mapsto s) \in isActive \wedge c1 \mapsto s1 \in conflictCompServ[\{c \mapsto s\}] \Rightarrow c1 \mapsto (vm \mapsto s1) \notin isActive$

inv3: $\forall\, c, s, s1, vm.\ c \mapsto (vm \mapsto s) \in isActive \wedge .\ s1 \in neededServ[\{c \mapsto s\}] \Rightarrow (\exists\, c1, vm1 \cdot\ c1 \mapsto (vm1 \mapsto s1) \in isActive$

inv4: $\forall\, c, s, c1, s1, vm.\ c \mapsto (vm \mapsto s) \in isActive \wedge c1 \mapsto s1 \in neededServComp[\{c \mapsto s\}] \Rightarrow (\exists\, vm1.\ c1 \mapsto (vm1 \mapsto s1) \in isActive)$

END

Figure 11: EVENT-B invariants of Machine M3

Figure 11 depicts the invariants related to the different constraints. Invariant *inv1* specifies that two conflicting services cannot be activated at the same time on the same VM (Requirement **Req9**.a). Invariant *inv2* states that for each couple of conflicting services $(s, s1)$ provided by two components c and $c1$ that are deployed on the same VM, then both cannot be activated at the same time (Requirement **Req9**.b). Invariant *inv3* states that, for each service $s1$ needed by another service s, it should exist a deployed component $c1$ that provides $s1$ (Requirement **Req9**.d). Finally, *inv4* specifies that if a service s needs a service $s1$ of a specific component $c1$, then a VM $vm1$ should exist on which $c1$

is deployed with the service $s1$ activated (Requirement **Req9**.e).

To make these invariants preserved after the execution of each event, we add adequate guards at each event that updates the variables involved in these invariants. For instance, to make the event Install preserve the invariants *inv1*, *inv2* and *inv4*, we add the guards *grd_inv1*, *grd_inv2* and *grd_inv4* as depicted by Figure 12. Guard *grd_inv1* states that each service $s1$ in conflict with a mandatory service s of the component c should be not active on the VM vm on which c is deployed. Guard *grd_inv2* ensures that any service $s1$, of a component $c1$ that is in conflict with a mandatory service s of the new installed component c, is not activated on the same VM. Guard *grd_inv4* specifies that for each service $s1$ of the component $c1$ needed by a mandatory service s of c, it should exist a VM *vm1* on which the component $c1$ is deployed with the service $s1$ activated.

Event Install $\widehat{=}$
  any
    ...
  where
    : ....
    grd_inv1: $\forall\, c1,\, s1,\, s.\ s \in mandProvSers[\{c\}] \land s \mapsto s1 \in conflictServ$
      $\Rightarrow$
      $c1 \mapsto (vm \mapsto s1) \notin isActive$ *//for Invariant inv1*
    grd_inv2: $\forall\, s,\, c1,\, s1 \cdot\ c \mapsto s \in mandProvSers \land$
      $c1 \mapsto s1 \in conflictCompServ[\{c \mapsto s\}]$
      $\Rightarrow$
      $c1 \mapsto (vm \mapsto s1) \notin isActive$ *//for Invariant inv2*
    grd_inv4: $\forall\, s,\, c1,\, s1.\ c \mapsto s \in mandProvSers \land$
      $c1 \mapsto s1 \in neededServComp[\{c \mapsto s\}]$
      $\Rightarrow$
      $(\exists\, vm1.\ c1 \mapsto (vm1 \mapsto s1) \in isActive)$ *//for Invariant inv4*
  then
    act1: ...
  end

Figure 12: Refinement of the event Install

The refinement of the event Uninstall is achieved in a similar manner by adding guards that permit to satisfy the invariants of the machine M3 . Figure 13 depicts the refinement of such an event. Two guards *grd_inv3* and *grd_inv4* are added in order to fulfill the invariants *inv3* and *inv4* respectively. Guard *grd_inv3* ensures that for each service s_0 of a component c_0 that needs a service s_1 of the component c to uninstall, there is a component c_1 installed on a VM vm_1 with the service s_1 activated. Component c_1 must be different from the component c or installed on a VM different from the VM vm on which c is installed ($c_1 \mapsto vm_1 \neq c \mapsto vm$). Guard *grd_inv4* ensures that for each active service s_1 of a component c_1 that needs a service s of the component c to uninstall, there is the same component c installed on a different VM vm_2 ($vm_2 \neq vm$) with the service s activated.

Event Uninstall $\widehat{=}$

any

...

where

: ....

grd_inv3: $\forall\, c_0,\, s_0,\, vm_0,\, s_1.\ c_0 \mapsto (vm_0 \mapsto s_0) \in isActive \wedge$
$s_1 \in (mandProvSers \cup optProvSers)[\{c\}] \wedge$
$s_1 \in neededServ[\{c_0 \mapsto s_0\}]$
$\Rightarrow$
$(\exists\, c_1,\, vm_1.\ c_1 \mapsto vm_1 \neq c \mapsto vm \wedge c_1 \mapsto (vm_1 \mapsto s_1) \in isActive)$
//for Invariant inv3

grd_inv4: $\forall\, s,\, c_1,\, s_1,\, vm_1.\ c \mapsto s \in neededServComp[\{c_1 \mapsto s_1\}] \wedge$
$c_1 \mapsto (vm_1 \mapsto s_1) \in isActive$
$\Rightarrow$
$(\exists\, vm_2 \cdot\ vm_2 \neq vm \wedge c \mapsto (vm_2 \mapsto s) \in isActive)$ *//for Invariant inv4*

then

act1: ...

end

Figure 13: Refinement of the event Uninstall

Likewise, we add the guards *grd_inv1* and *grd_inv4* to the event that activates an optional service. Guard *grd_inv1* specifies that any service $s1$ in conflict with s_0 cannot be active on the same VM vm_0. Guard *grd_inv4* states that for each service $s1$, of the component $c1$, needed by the service *s0*, it should exists a VM $vm1$ on which the component $c1$ is deployed with the service $s1$ active. This guard is similar to that of mandatory services when a component is deployed (Event Install).

Event putActive $\widehat{=}$

any

$c_0,\, s_0,\, vm_0$

where

grd1: ...

grd2: ...

grd_inv1: $\forall\, c1,\, s1.\ s \mapsto s1 \in conflictServ \Rightarrow$
$c1 \mapsto (vm_0 \mapsto s1) \notin isActive$ *//for Invariant inv1)*

grd_inv4: $\forall\, c1,\, s1.\ c1 \mapsto s1 \in neededServComp[\{c \mapsto s_0\}]$
$\Rightarrow$
$(\exists\, vm1.\ c1 \mapsto (vm1 \mapsto s1) \in isActive)$ *//for Invariant inv4)*

then

act1: ...

end

Let us suppose that the user would like to install the component *Zimbra-Core*. This is not possible since Gaurd $grd1$ of Event Install requires an existing VM on which the component will be deployed. Thus, a prior step consists of adding a new VM, for instance, $VM1$. Now, if the user tries to install this component on $VM1$, the above guard grd_inv3 prohibits such an installation since there is no component $c1$ that would provide the service *DNS* required by the mandatory service $Mailbox$. In other words, the guards establish an order on the actions/events that can be carried out at each moment and avoid inconsistent states, for instance in the case of activating a service without its

required services. In this particular example, the user has to install all the components that would offer the services required by *Mailbox* before a successful installation of the *ZimbraCore* component.

6.4. Modelling elasticity management

The last step of the Event-B modelling of component-based applications and their deployment deals with the resources and elasticity mechanisms. For this sake, we extend the context C3 by a new context C4, that introduces two sets *Resources* and *osType* to respectively denote the set of all possible resources and the different operating systems that equipped a VM, and have to be managed during the deployment (see Figure 14). Axiom *axm1* states that we distinguish two kinds of resources, Axiom *axm2* associates each VM with its operating system, Axiom *axm3* (resp. *axm4*) states the needs and requirements of each service in terms of resources (OS type, RAM, disk space, etc.). Finally, the invariant *inv*5 states the maximum number of installed VMs.

CONTEXT *C4*
EXTENDS *C3*
SETS
Ressources, oSType
CONSTANTS
ram, *diskspace*, *osType*, *serviceNeedCapacity*, *serviceNeedOs*, *NB*
AXIOMS
axm1: **partition**(*Ressources*,{*ram*},{*diskspace*})
axm2: $osType \in VMS \rightarrow oSType$
axm3: $serviceNeedCapacity \in Services \times \{ram, diskspace\} \rightarrow \mathbb{N}$
axm4: $serviceNeedOs \in Services \rightarrow oSType$
axm5: $NB \geq 1$
END

Figure 14: Axioms of the Event-B context C4

Instantiating the context *C4* on the case study gives:

CONTEXT *C4Inst*
EXTENDS *C3Inst*, *C4*
AXIOMS
axm1: **partition**(*oSType*,{*UbuntuS1204*}, {*US1204*}, {*WS2012*})
axm2: $osType = \{VM1 \mapsto WS2012, VM2 \mapsto WS2012, VM3 \mapsto WS1204, VM4 \mapsto WS1204\}$
axm3: $serviceNeedCapacity = \{Mailbox \mapsto ram \mapsto 8, Mailbox \mapsto diskpace \mapsto 10, MTA \mapsto diskpace \mapsto 2, Archive \mapsto ram \mapsto 8, Archive \mapsto diskspace \mapsto 120, \dots\}$
axm4: $serviceNeedOs = \{Mailbox \mapsto US1204, Archive \mapsto WS2012, \dots\}$
END

To model the service resource needs, we refine the machine M3 by introducing a new machine M4 that sees the context C4. Machine M4 defines a single variable *vmRessources* along with four invariants (see Figure 15). For each VM, the invariant *inv1* states the provided capacity on each resource (Requirement

Req11). Invariant *inv2* states that the capacity of a VM should be sufficient to cope with the need of each active service deployed on it. Invariant *inv3* ensures that the total disk space needed by all the active services is less or equal to the disk space of the related VM. This invariant uses a function *SUM* specified in the theory *SUMandPRODUCT* [25]. *SUM* takes as input a set of services, active on a given `VM`, and returns the total of disk space needed by them. Invariant *inv4* ensures that each service is deployed on the adequate operating system, i.e., the required operating system of the service matches with the one of the VM on which it is deployed. Finally, the last invariant *inv5* models the requirement **Req16** related to the maximum number of installed VMs.

MACHINE *M4*
REFINES M3
SEES *C4*
VARIABLES
vmRessources
INVARIANTS
inv1: $vmRessources \in vms \rightarrow (\{ram,\ diskspace\} \rightarrow \mathbb{N})$
inv2: $\forall\ vm,\ s.\ vm \in VMS \wedge vm \mapsto s \in \mathbf{ran}(isActive) \Rightarrow serviceNeedCapacity(s \mapsto ram) \leq (vmRessources(vm))(ram)$
inv3: $\forall\ vm.\ vm \in vms \Rightarrow \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) \leq (vmRessources(vm))(diskspace)$
inv4: $\forall\ s, vm.\ s \in \mathbf{dom}(serviceNeedOs) \wedge vm \in vms \wedge vm \mapsto s \in \mathbf{ran}(isActive) \Rightarrow serviceNeedOs(s) = osType(vm)$
inv5: $\mathbf{card}(vms) \leq Nb$
END

Figure 15: Invariants of the Event-B machine M4

To preserve the invariant of the machine M4, we refine each event by adding adequate guards. For instance, we add adequate guards to the event putActive in order to preserve the invariants *inv*2, *inv*3, and *inv*4. Guard *grd8* ensures that the capacity of the VM in terms of RAM is sufficient to meet the service's need, Guard *grd9* indicates that the size of the available disk space is sufficient for all services already active including this service which will become active later on. Finally, Guard *grd10* checks that the operating system of the VM matches the one needed by the service.

Event putActive $\widehat{=}$
any
$c_0,\ s_0,\ vm_0$
where
grd1: ...
grd2: ...
grd_inv1: ...
grd_inv3: ...
grd8: $serviceNeedCapacity(s \mapsto ram) \leq (vmRessources(vm))(ram)$
grd9: $\mathbf{SUM}(((\mathbf{ran}(isActive)[\{vm\}] \cup \{s\}) \times \{diskspace\}) \lhd serviceNeedCapacity) \leq (vmRessources(vm))(diskspace)$
grd10: $serviceNeedOs(s) = osType(vm)$
then

act1: ...
end

Elasticity in component-based applications consists in adding a new VM or removing an existing one (horizontal elasticity) or increasing/decreasing the resource capacity of an existing VM (vertical elasticity) [**37**]. In our case, a new VM is added when the resources of all the existing VM are over-used, that is more than 80% of their disk capacities is used. Therefore, for example, we refine the events addVM and removeVM by adding respectively the following guards:

$$\begin{array}{c} \forall\ vm1.\ vm1 \in \mathbf{ran}(deployedOn) \Rightarrow \\ \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) \\ > \\ 80 \times (vmRessources(vm1))(diskspace) \div 100 \end{array}$$

and

$$\begin{array}{c} \forall\ vm1.\ vm1 \in \mathbf{ran}(deployedOn) \Rightarrow \\ \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) \\ < \\ 20 \times (vmRessources(vm1))(diskspace) \div 100 \end{array}$$

For instance, when starting the deployment of the component-based application of Figure 1, the user will install, on *VM1*, the components that do not require any other component like *openLDAP*, *AmasvidNew*, *ClamAV*, and at last *Postfix*. Now, to complete the installation of the *ZimbraCore* component, the virtual machine $VM1$ does not have enough free disk space required by the component *ZimbraArchive*. In that case, a horizontal elasticity is performed by adding a new VM $VM4$. The same process is applied for the installation of the other required components by adding $VM2$ and $VM3$. Furthermore, another example of horizontal elasticity is carried out when the Web Server gets an attack of DDoS, a Distributed Denial of Service that can saturate the RAM of $VM2$ where the Web Server is installed and make the Zimbra application unavailable to the users. In that case, a horizontal elasticity is performed by adding a new VM $VM5$ with a new instance of the Web Server component.

Likewise, to model the vertical elasticity, Machine M4 introduces one additional event updateRessource that updates the value of a given resource of a VM (see Figure 16). Guard *grd2* (resp. *grd3*) checks that the new value of RAM (resp. disk space) is sufficient to cover the needs of the services expressed by the invariant *inv2* (resp. *inv3*) of Machine M4. The vertical elasticity mechanism is specified by guards *grd4* and *grd5* that respectively state that the resource capacity should be decreased (resp. increased) when less (resp. more) 20% (resp. 80) of this resource is used.

7. Verification and validation

To validate and verify the correctness of the Event-B models for the deployment of component-based applications, we propose a strategy that includes three complementary steps (see Figure 17).

Event updateRessource $\widehat{=}$

any

vm, res, val

where

grd1: $vm \in vms \land res \in Resources \land val \in \mathbb{N}$

grd2: $res{=}ram \Rightarrow (\forall\ s.\ s \in Services \land s \in \mathbf{ran}(isActive)[\{vm\}] \Rightarrow serviceNeedCapacity(s \mapsto ram) \leq val)$

grd3: $res = diskspace \Rightarrow \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) \leq val$

grd4: $res = diskspace \land \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) < (20 \times ((vmRessources(vm))(diskspace))) \div 100 \Rightarrow val < vmResources(vm)(diskspace)$

grd5: $res = diskspace \land \mathbf{SUM}((\mathbf{ran}(isActive)[\{vm\}] \times \{diskspace\}) \lhd serviceNeedCapacity) \geq (80 \times ((vmRessources(vm))(diskspace))) \div 100 \Rightarrow val > vmResources(vm)(diskspace)$

then

act1: $vmRessources := vmRessources \Leftarrow \{vm \mapsto (vmRessources(vm) \Leftarrow \{res \mapsto val\})\}$

end

Figure 16: The EVENT-B specification of the vertical elasticity

7.1. Model checking using ProB

In the first step, we check the correctness of the models by using the ProB model checker for three purposes:

- Axioms satisfiability: for each context, ProB checks the satisfiability of the axioms that it contains. This means that ProB must find a valuation of sets and constants that satisfy all the axioms. In our case, ProB checks that each context *CInst* that instantiates the generic context C is correct, that is, it satisfies all the axioms defined in C.

- Invariant correctness: ProB has to detect any scenario that violates invariants. A scenario is a sequence of events that, starting from the initial state, reaches a state that violates one or several invariants. In that case, the specification should be revised by adding guards to events or correcting the invariants. For instance, according to the requirement **Req16**, we have specified that the number of installed VMs should be less than 5 ($Nb = 4$) and have forgotten to add a guard to the event **addVm** to specify that the number of the existing VMs is less than (Nb). So, ProB returns a counterexample where 5 VMs are deployed, which violates the invariant $inv5$ of the machine $M4$. To fix this error, we have added the guard ($card(vms) < Nb$) to the event **addVm**. Thus, model checking permits us to define and exhibit some forgotten guards and invariants. The use of ProB is particularly useful for specifications with several invariants that produce a large number of proof obligations for which it

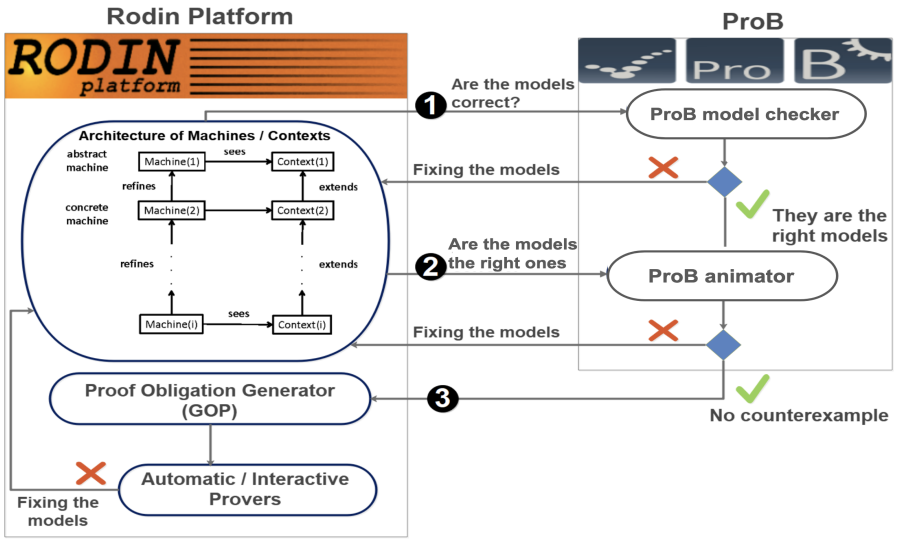

Figure 17: Validation and verification process of Event-B models

becomes very difficult to establish which ones are incorrect/correct.

- Absence of deadlock: in this step, we check that at any moment there is at least one enabled event that can be executed to make the system evolve.

Step	Action	Event-B event	Parameters
1	VM installation	*addVm*	*vm = VM1, ram=8, diskspace=5*
2	**Component installation**	***Install***	***c=ZimbraCore, vm=VM1***
3	VM installation	*addVm*	*vm=VM3, ram=8 diskspace=30*
4	Component installation	*Install*	*c=DNSServer, vm=VM3*
5	VM installation	*addVm*	*vm = VM2, ram=8, diskspace=20*
6	Component installation	*Install*	*c=WebServer, vm=VM2*
7	Component installation	*Install*	*c=OpenLDAP, vm=VM1*
8	Component installation	*Install*	*c=Postfix, vm=VM1*
9	Component installation	*Install*	*c=ZimbraStore, vm=VM1*
10	**Component installation**	***Install***	***c=ZimbraCore, vm=VM1***
12	Vertical Elasticity	*updateRessource*	*val=15, res=diskspace, vm=VM1*
13	Component installation	*Install*	*c=ZimbraCore, vm=VM1*

Table 4: Animating a scenario with ProB

7.2. Validation using ProB

In this step, the ProB model checker/animator is used to validate the models by playing a set of scenarios in order to ensure that we built the right models. At each step of validation, ProB provides us with the list of the enabled events, that are, the events whose guards are satisfied. To make this task easier, we follow a step-by-step validation approach by checking each level according to the architecture depicted in Figure 2:

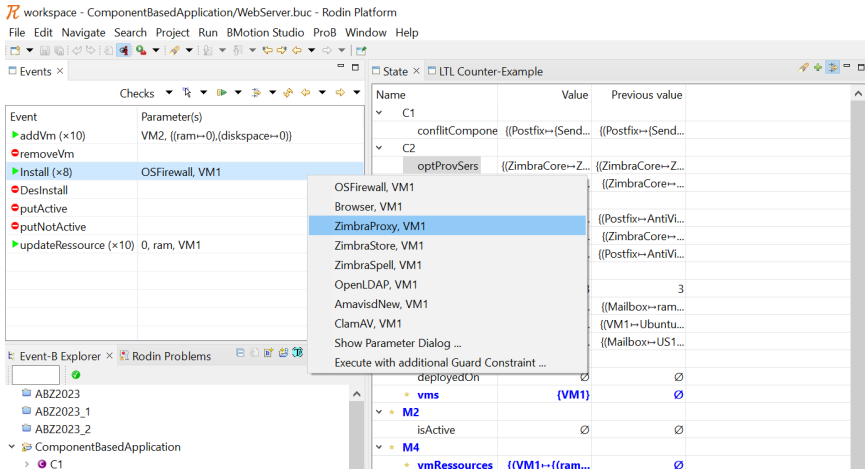

Figure 18: A step-by-step validation using ProB

1. Machine M1: we animate the machine M1_Anim to validate scenarios that add/remove a VM and install/uninstall .a component on a VM. We ensure for instance that two conflict components cannot be installed on the same VM. So, we have deployed the component *SendMail*. Then, we have ensured that no event is enabled to install the component *Postfix* because they are in conflict,

2. Machine M2: we animate the machine M2_Anim to validate scenarios that deploy components on VMs. We mainly verify that all the mandatory services of a component are activated when this later is deployed on a VM. For example, the deployment of the component *ZimbraCore* makes the service *Mailbox* automatically activated.

3. Machine M3: we animate the machine M3_Anim to validate scenarios that activate services that can be in conflict. We mainly verify that a service in conflict with an other service or a service of a given component cannot be active on the same VM. For instance, we have deployed the component $SendMail$. Then, we have ensured that no event is enabled to activate the service MTA of the component $Postfix$, because it is in conflict with the component $SendMail$. We also verify that we cannot activate a service if at least one of its needed services is not active on the same VM. Finally, we validate that when a service is made active, all its needed services are also activated.

4. Machine M4: we animate the machine M4_Anim to validate more complex scenarios like the deployment of the component*ZimbraCore* to show which actions are possible at each deployment step as depicted in Table 4 where unsuccessful actions are in bold. Naturally, the first step is to

install a VM on which a component can be deployed. So in Step **(1)**, we choose to install VM1 since it has the adequate OS required by *ZimbraCore*. After executing this event, ProB displays the enabled events in green and the others are in red to state that it cannot be executed at all (see Figure 18). ProB gives some parameters of each enabled event, but it is possible to give ours by selecting the option "*Execute with additional Guard Constraint*. It is what we do in Step **(2)**, when we tried to install the component *ZimbraCore* on VM1. So, we get an error message that states that the guards of the event are not satisfied since indeed the required services *DNS*, *WebServer*, *MTA*, *Store* are not available. To meet such requirements, we have installed the related components (*DNSServer*, *WebServer*, *ZimbraStore*) that provide them on the VMs with adequate OS and resources (see Steps **3-9**). Finally, in Step **10**, we succeeded to install the component *ZimbraCore* on VM1. In Step **11**, we tried to install the component *ZimbraCore* on VM1. This failed because there is no sufficient disk space as this component requires at least 12 Gb (10 Gb for *Mailbox* and 2 Gb for *MTA*). So, we operate a vertical elasticity action in Step **12** on VM1. This action makes the deployment of *ZimbraCore* possible on VM1 (Step **13**).

As one can notice, ProB permits the validation of the proposed model to ensure that it behaves as expected by prohibiting the actions/events that violate the requirements.

7.3. Correctness Proofs

Making the specification counterexample during the model checking phase does not mean that the specification is correct. Indeed, ProB can fail to find a scenario that violates the invariant for different reasons like a timeout on the model-checking process. This is why a proof activity should be performed, in the third step, to ensure definitely the correctness of the models. To prove the correctness of the development, that are the first machine and all the refinement levels, 86 proof obligations have been generated, and 61% (53/86) of them are automatically discharged by the automatic prover. The remaining proofs are discharged manually using the interactive prover since they require more deduction steps and are particularly related to the quantified invariants. To illustrate such proof obligation, we give the following example. Let us consider Invariant $inv3$ of Machine M3 and prove that it is preserved by the event putActive: $(inv3 \wedge gard_{putActive} \Rightarrow [putActive]inv3)$ where $[S]P$ denotes the weakest-precondition obtained by replacing each variable in P with its after-value obtained by executing S. We have to prove that for each service $s1$ needed by a service s, of a component c, that is active after updating the variable $isActive$ with the tuple $(\{(c_0 \mapsto vm_0 \mapsto s_0) \mapsto TRUE\})$, there should exist a component $c1$, deployed on a VM $vm1$, that offers the service $s1$ (G1):

$c \mapsto s \in mandProvSers \cup optProvSers \wedge$
$c \mapsto vm \in deployedOn \wedge$

$(isActive \lhd\!+ \{(c_0 \mapsto vm_0 \mapsto s_0) \mapsto \mathbf{TRUE}\})$
$(c \mapsto vm \mapsto s) = TRUE \wedge$
$s1 \in neededServ[\{c \mapsto s\}]$
$\Rightarrow$
$(\exists c1, vm1.\ c1 \mapsto vm1 \in deployedOn \wedge$
$c1 \mapsto s1 \in mandProvSers \cup optProvSers \wedge$
$(isActive \lhd\!+ \{(c_0 \mapsto vm_0 \mapsto s_0) \mapsto TRUE\})$
$(c1 \mapsto vm1 \mapsto s1) = TRUE)$

Under hypotheses (H1) and (H2) that respectively correspond to the invariant of M4 and the guard of the event putActive:

H1= $\forall\ c, s, s1, vm.\ c \mapsto s \in mandProvSers \cup optProvSers \wedge c \mapsto vm \in deployedOn \wedge$
$isActive(c \mapsto vm \mapsto s) = TRUE \wedge\ s1 \in neededServ[\{c \mapsto s\}]$
$\Rightarrow$
$(\exists\ c1, m1.\ c1 \mapsto vm1 \in deployedOn \wedge c1 \mapsto s1 \in mandProvSers \cup optProvSers \wedge$
$isActive(c1 \mapsto vm1 \mapsto s1) = TRUE)$

H2= $\forall\ s2.\ s2 \in neededServ[\{c_0 \mapsto s_0\}] \Rightarrow$
$(\exists\ c2, vm2.\ c2 \mapsto vm2 \in deployedOn \wedge c2 \mapsto s2 \in mandProvSers \cup optProvSers \wedge$
$isActive(c2 \mapsto vm2 \mapsto s2)$=TRUE)

So, we have to prove that (G2)

$(\exists\ c1, vm1.\ c1 \mapsto vm1 \in deployedOn \wedge c1 \mapsto s1 \in mandProvSers \cup optProvSers \wedge$
$(isActive \lhd\!+ \{(c_0 \mapsto vm_0 \mapsto s_0) \mapsto \mathbf{TRUE}\})$
$(c1 \mapsto vm1 \mapsto s1) = \mathbf{TRUE})$

With the additional hypothesis (H3):

H3= $c \mapsto s \in mandProvSers \cup optProvSers \wedge$
$c \mapsto vm \in deployedOn \wedge$
$(isActive \lhd\!+ \{(c_0 \mapsto vm_0 \mapsto s_0) \mapsto \mathbf{TRUE}\})$
$(c \mapsto vm \mapsto s) = \mathbf{TRUE} \wedge$
$s1 \in neededServ[\{c \mapsto s\}]$

To prove (G2), we distinguish two cases:

1. $(c \mapsto vm \mapsto s = c_0 \mapsto vm_0 \mapsto s_0)$: the hypothesis (H3) becomes:

   H3= $c_0 \mapsto s_0 \in mandProvSers \cup optProvSers \wedge$
   $c_0 \mapsto vm_0 \in deployedOn \wedge$
   $s1 \in neededServ[\{c_0 \mapsto s_0\}]$

   Goal (G2) is discharged for the following values: $c1 = c_0$ and $vm1 = vm_0$.

2. $(c \mapsto vm \mapsto s \neq c_0 \mapsto vm_0 \mapsto s_0)$: hypothesis (H3) becomes:

H3= $c \mapsto s \in mandProvSers \cup optProvSers \wedge$
$c \mapsto vm \in deployedOn \wedge$
$isActive(c \mapsto vm \mapsto s) = TRUE \wedge$
$s1 \in neededServ[\{c \mapsto s\}]$

By instantiating (H1) with c, vm and s, then applying Modus-Ponens with (H3), we obtain:

$(\exists c1, vm1.\ c1 \mapsto vm1 \in deployedOn \wedge$
$c1 \mapsto s1 \in mandProvSers \cup optProvSers \wedge$
$isActive(c1 \mapsto vm1 \mapsto s1) = TRUE)$

Goal (G2) is discharged for the same values that verify this last hypothesis.

Let us note that our Event-B specification is generic and can thus be reused as such for any particular application. The user has only to instantiate the structural aspect of the application (valuation of the different elements defined in the Event-B contexts). Then, he/she can apply the different events of the formal specification (the behavior part) to make the application evolve but without redoing any proof. Indeed, the proofs have been discharged independently from any particular application. In other words, what differs from one specific application to another is the static part and all the Event-B models (machines and refinement) can be reused as such without any modification.

8. Conclusion and future work

In this paper, we have presented an eventB-based approach for modelling and verification of the deployment of component-based applications. The built models consist of four levels; each of them stresses a specific aspect of the application (VMs, Components, deployment, resources). The validation and the verification of the Event-B models are carried out by animating the models using the ProB model checker/animator and also by discharging proof obligations generated by the proof obligations generator (POG) of the Rodin platform. The proposed approach has been illustrated through a case study.

From this experience, we drew the following lessons:

1. the Event-B refinement permits us to cope with the complexity of cloud applications by incrementally introducing the different elements/constraints. The considered aspects include the application architecture (components dependencies and conflicting constraints), the deployment phases (installation, uninstalling), and elasticity strategies (Horizontal and vertical elasticity);

2. defining several refinement levels makes the proof phase easier to deal with a specific aspect at each level;

3. the degree of abstraction offered by Event-B makes it possible to build generic models that do not depend on any specific case study. In other words, what would differ from one case study to another is only the valuations of different constants defined in contexts. The other parts of the Event-B specifications remain the same and thus are reused as such.

4. the validation and proof phases permit early error detection. Having several abstraction levels facilitate modeling and also error detection.

It is noteworthy that the Event-B specifications obtained at the last refinement level are close enough to a programming language like JAVA. A non Event-B expert can use plugins, like EventB2JAVA [35], to generate a correct JAVA application that verifies the different requirements without any additional verification or test.

Currently, we are working on the implementation of this approach that consists in automatically generating the context C5 corresponding to a specific application. Indeed, all the other models (M_i and C_i) are generic and can be reused as such regardless of a specific application.

As a future work, we aim at enlarging this approach to take into account the reconfiguration in a runtime environment and apply other elasticity strategies (such as migration, load balancing, etc.). We also plan to consider security issues by, for instance, formalizing and proving access control models defined in [27, 34, 36]. Thanks to the refinement concept of Event-B, we think that on one hand that such aspects can be integrated by refining the last refinement component. On the other hand, we can prevent conflict and deployment problems by verifying in advance deployment operations and configuration plans before execution. In this case, we can have a recommendation of the best deployment plans for such applications.

Appendix A. B symbols

Table A.5 gives the semantics of the different mathematical symbols used in the paper where:

- A and B denote any sets of elements,
- If a and b are elements of A and B respectively, $a \mapsto b$ denotes the tuple (a, b),
- A_1 and B_1 denote any subsets of A and B respectively,
- P denotes a predicate,
- S denotes any set expression.

Concept	Notation	Semantics
$A_1, \ldots, A_n$ is a partition of A	**partition** $(A, A_1, \ldots, A_n)$	$A_i \subseteq A \wedge \bigcup_i A_i = A$ $\forall i, j.i \neq j \Rightarrow A_i \cap A_j = \varnothing$
Set of parties of A	$\mathbb{P}1(A)$	$\mathbb{P}1(A) = \{A_1 \cdot A_1 \subseteq A \wedge A_1 \neq \varnothing\}$
R is a relation from A to B	$R \in A \leftrightarrow B$	$R \subseteq \{a \mapsto b \cdot a \in A \wedge b \in B\}$
R^{-1} is the inverse of R	R^{-1}	$R^{-1} = \{b \mapsto a \cdot a \mapsto b \in R\}$
Difference	$R_1 \backslash R_2$	if $R_1 \in A \leftrightarrow B$ and $R_2 \in A \leftrightarrow B$ then, $R_1 \backslash R_2 = \{a \mapsto b \cdot a \mapsto b \in R_1 \wedge a \mapsto b \notin R_2\}$
Override of R_1 by R_2	$R_1 \Leftarrow R_2$	if $R_1 \in A \leftrightarrow B$ and $R_2 \in A \leftrightarrow B$ then, $R_1 \Leftarrow R_2 = \{a \mapsto b \cdot a \mapsto b \in R_2 \vee (a \mapsto b \in R_1 \wedge a \notin dom(R_2))\}$
Direct product of R_1 abd R_2	$R_1 \otimes R_2$	if $R_1 \in A \leftrightarrow B$ and $R_2 \in A \leftrightarrow C$ then, $R_1 \otimes R_2 = \{a \mapsto (b \mapsto c) \cdot a \mapsto b \in R_1 \wedge a \mapsto c \in R_2\}$
Image of A_1 by R	$R[A_1]$	$R[A_1] = \{b_1 \cdot (b_1 \in B \wedge \exists a_1 \cdot (a_1 \in A_1 \wedge a_1 \mapsto b_1 \in R))\}$
Domain of R	$dom(R)$	$dom(R) = \{a_1 \cdot (a_1 \in A \wedge \exists b_1 \cdot (b_1 \in B \wedge a_1 \mapsto b_1 \in R))\}$
Range of R	$ran(R)$	$ran(R) = \{b_1 \cdot (b_1 \in B \wedge \exists a_1 \cdot (a_1 \in A \wedge a_1 \mapsto b_1 \in R))\}$
Domain subtraction of R	$A_1 \lhd R$	$A_1 \lhd R = \{a \mapsto b \cdot (a \mapsto b \in R \wedge a \notin A_1)\}$
Partial function f	$f \in A \nrightarrow B$	$f \in A \leftrightarrow B \wedge \forall a \cdot (a \in A \Rightarrow card(f[\{a\}]) \leq 1)$
Total function f	$f \in A \rightarrow B$	$f \in A \nrightarrow B \wedge dom(f) = A$
Guard of an event E	$grd(E)$	If E=**ANY** X **WHERE** G **THEN** S then, $grd(E) = G$

Table A.5: Some EVENT-B symbols and their semantics

References

[1] Abbassi, I., Graiet, M., Jlassi, S., Elkhalfa, A., Sliman, L.: A Formal Approach for Correct Elastic Package-Based Free and Open Source Software Composition in Cloud. In: On the Move to Meaningful Internet Systems. pp. 732–750. Springer International Publishing (2017)

[2] Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)

[3] Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)

[4] Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (2010)

[5] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in Cloud Computing: State of the Art and Research Challenges. IEEE Transactions on Services Computing **11**, 430,447 (3 2018)

[6] Arshad, N., Heimbigner, D., Wolf, A.L.: Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. Software Quality Journal **15**(3), 265–281 (5 2007)

[7] Belguidoum, M.: Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels. Ph.D. thesis, Télécom Bretagne, Télécom Bretagne (2008)

[8] Belguidoum, M., Dagnat, F.: Dependency Management in Software Component Deployment. Electronic Notes in Theoretical Computer Science **182**, 17–32 (2007)

[9] Ben Hafaiedh, I., Ben Hamouda, R., Robbana, R.: A model-based approach for formal verification and performance analysis of dynamic load-balancing protocols in cloud environment. Cluster Computing **24**(4), 2977–2994 (2021)

[10] Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: A Formal Approach to Microservice Architecture Deployment, pp. 183–208. Springer International Publishing, Cham (2020)

[11] Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator – Version 5.4 (2011)

[12] Chardet, M., Coullon, H., Pertin, D., Pérez, C.: Madeus: A Formal Deployment Model. In: 5th International Symposium on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018). pp. 1–8 (2018)

[13] Etchevers, X., Salaün, G., Boyer, F., Coupaye, T., De Palma, N.: Reliable Self-deployment of Distributed Cloud Applications. Software: Practice and Experience **47**(1), 3–20 (2017)

[14] Fakhfakh, F., Kallel, S., Cheikhrouhou, S.: Formal verification of cloud and fog systems: A review and research challenges. J. Univers. Comput. Sci. **27**(4), 341–363 (2021)

[15] Fox, A., Griffith, B., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: Above the clouds: A berkeley view of cloud computing. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS **28**(13), 2009 (2009)

[16] Galante, G., de Bona, L.E.: A Survey on Cloud Computing Elasticity. In: Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on. pp. 263–270. IEEE (2012)

[17] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. vol. 6605, pp. 372–387. Springer (2011)

[18] Graiet, M., Hamel, L., Mammar, A., Tata, S.: A Verification and Deployment Approach for Elastic Component-Based Applications. Formal Aspects of Computing **29**(6), 987–1011 (Nov 2017)

[19] Herbst, N.R., Kounev, S., Reussner, R.H.: Elasticity in Cloud Computing: What It Is, and What It Is Not. In: ICAC. vol. 13, pp. 23–27 (2013)

[20] Jarraya, Y., Eghtesadi, A., Debbabi, M., Zhang, Y., Pourzandi, M.: Cloud calculus: Security verification in elastic cloud computing platform. In: 2012 international conference on collaboration technologies and systems (CTS). pp. 447–454 (2012)

[21] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: Abs: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. pp. 142–164. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[22] Kamel, O., Chaoui, A., Diaz, G., Gharzouli, M.: SLA-Driven modeling and verifying cloud systems: A Bigraphical reactive systems-based approach. Computer Standards & Interfaces **74**, 103483 (2021)

[23] Karam, Y., Baker, T., Taleb-Bendiab, A.: Security support for intention driven elastic cloud computing. In: 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation. pp. 67–73. IEEE (2012)

[24] Khebbeb, K., Hameurlain, N., Belala, F.: Formal modeling and verification of cloud elasticity with maude and ltl. In: Attiogbé, C., Ferrarotti, F., Maabout, S. (eds.) New Trends in Model and Data Engineering. pp. 64–77. Springer International Publishing, Cham (2019)

[25] Leuschel, M.: Available at `https://prob.hhu.de/w/index.php?title=Event-B_Theories` (2021)

[26] Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods. pp. 855–874. Springer Berlin Heidelberg (2003)

[27] Li, Z., Wang, D.: Achieving one-round password-based authenticated key exchange over lattices. IEEE Transactions on Services Computing pp. 1–1 (2019). https://doi.org/10.1109/TSC.2019.2939836

[28] Mammar, A., M. Belguidoum, S.H.H.: A Formal Approach for the Deployment Verification of Cloud Applications. Available at `http://www-public.imtbs-tsp.eu/~mammar_a/SCP/CBAWithEventB.html` (July 2021)

[29] Mell, P.M., Grance, T.: The NIST Definition of Cloud Computing. Tech. rep., Gaithersburg, MD, United States (2011)

[30] de Moura, L.M., Bjorner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)

[31] Muniasamy, K., Srinivasan, S., Vain, J., Sethumadhavan, M.: Formal methods based security for cloud-based manufacturing cyber physical system. IFAC-PapersOnLine **52**(13), 1198–1203 (2019)

[32] Naskos, A., Gounaris, A., Mouratidis, H., Katsaros, P.: Online analysis of security risks in elastic cloud applications. IEEE Cloud Computing **3**(5), 26–33 (2016)

[33] Nawaz, M.S., Sun, M.: Using PVS for Modeling and Verifying Cloud Services and Their Composition. In: 2018 Sixth International Conference on Advanced Cloud and Big Data (CBD). pp. 42–47 (2018)

[34] Qiu, S., Wang, D., Xu, G., Kumari, S.: Practical and provably secure three-factor authentication protocol based on extended chaotic-maps for mobile lightweight devices. IEEE Transactions on Dependable and Secure Computing pp. 1–1 (2020)

[35] Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code Generation for Event-B. International Journal on Software Tools for Technology Transfer **19**(1), 31–52 (2017)

[36] Roy, S., Das, A.K., Chatterjee, S., Kumar, N., Chattopadhyay, S., Rodrigues, J.J.P.C.: Provably secure fine-grained data access control over multiple cloud servers in mobile cloud computing based healthcare applications. IEEE Transactions on Industrial Informatics **15**(1), 457–468 (2019)

[37] Sotiriadis, S., Bessis, N., Amza, C., Buyya, R.: Vertical and horizontal elasticity for dynamic virtual machine reconfiguration. IEEE Transactions on Services Computing **PP**, 1–1 (12 2016)

[38] Souri, A., Navimipour, N.J., Rahmani, A.M.: Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review. Computer Standards & Interfaces **58**, 1–22 (2018)

[39] Yadav, M.P., Pal, N., Yadav, D.K.: Verification of cloud system elasticity using bigmc. International Journal of System Assurance Engineering and Management **13**(5), 2208–2220 (2022)

[40] Ye, L., Zhang, H., Shi, J., Du, X.: Verifying cloud service level agreement. In: 2012 IEEE Global Communications Conference (GLOBECOM). pp. 777–782. IEEE (2012)

[41] Zhang, X., Sun, M.: SMT-Based Modeling and Verification of Cloud Applications. In: Xia, Y., Zhang, L.J. (eds.) Services 2019. pp. 1–15. Springer International Publishing (2019)