A verification and deployment approach for elastic SCA-based applications

Mohamed Graiet[1], Lazhar Hamel[1], Amel Mammar[2] and Samir Tata[2]

[1]High School of Computer Science and Mathematics, Monastir, Tunisia
lazhar.hamel@gmail.com, mohamed.graiet@imag.fr
[2]Institut Mines-Telecom, TELECOM SudParis, UMR CNRS Samovar, Evry, France
amel.mammar@telecom-sudparis.eu, samir.tata@mines-telecom.fr

Abstract. Cloud environments are being increasingly used for the deployment and execution of complex applications and particularly for Service-Component Architecture (SCA) based applications. Among other characteristics, Cloud environments are expected to provide elasticity in order to allow a deployed application to rapidly change the amount of its allocated resources in order to meet the demands variation while ensuring a given QoS. However, ensuring a correct elastic SCA-based application is not guaranteed in cloud. Applying elasticity mechanisms should preserve functional properties and improve non functional properties related to QoS, performance, and resource consumption. In this paper, we propose an approach for verifying and deploying the elastic SCA-based application. Our approach is based on the Event-B formal method. To this aim, we formally model the SCA artifacts using Event-B and we define the Event-B events that model the elasticity mechanisms (duplication/consolidation) for SCA-based applications. In addition, we formally verify, using the Proof Obligations and the ProB [1] animator, that our approach preserves the semantics of the SCA compositions. Once the elastic SCA-based applications are validated, they can be deployed in a cloud environment using an elastic SCA deployment framework.

Keywords: Service Component Architecture, elasticity, Event-B, automatic verification, Cloud.

1 Introduction

The rapid development of processing and storage technologies and the success of the Internet opened the gate to emerge a new trend of IT services paradigm, called Cloud Computing. This paradigm enables ubiquitous on-demand network access to a shared pool of configurable computing resources [2]. Cloud Computing delivers services at different levels (i.e. infrastructure, platform, software, etc.) using resources such as networks, servers, applications, CPU and storage, etc. Cloud environments are increasingly used for the deployment and execution of complex applications and particularly for Service-Component Architecture (SCA) based applications. SCA is a set of specifications for the development of

applications based on service-oriented architecture (SOA). The main idea behind this approach is to define how to create components and what mechanism to use for describing the communication of these components.

In this paper, we are interested in another very important characteristic of Cloud Computing called *Elasticity*. Indeed, Cloud Computing is expected to provide a rapid elasticity at different service levels in order to allow a Cloud service to rapidly adapt the amount of its allocated resources and meet the demands variation while ensuring the required QoS [3]. Our goal is to build elastic SCA-based applications that are able to adapt to the workload changes, that is, scale-up by adding new components or scale-down by removing some of others ones. The problem consists in ensuring that the execution of the elasticity mechanisms on the deployed SCA-based application can preserve their functional and non-functional properties. By preserving functional properties, we mean given a client request the execution of an application and the execution of its related elastic application, provided with elasticity mechanisms, both involve the same component services executed in the same order. Thus any application and its related elastic application should have the same operational behavior. Nevertheless, the elastic application should provide better performance or QoS. As an example, when the number of invocations increases, the response time of a non-elastic application increases quicker than the response time of its related elastic application face to the same number of invocations.

To address the above-mentioned issues, we propose in this paper a novel approach for the formal modeling, verification and provisioning of elastic SCA-based applications. First, we identify the requirements for the modeling of SCA-based application and elastic SCA-based applications. Second, we define a formal framework for modeling, SCA applications and their elasticity mechanisms using Event-B. Third, we provide tools to check how the modeled SCA-based applications satisfy the identified requirements. Finally, we propose mechanisms for the deployment and provisioning of SCA-based application in the Cloud. Our approach will permit to significantly reduce development and maintenance cost, and also increase credibility of complex SCA-based applications deployed in Cloud.

The rest of this paper is organized as follows: Section 2 presents a summary of the related work, the basic knowledges of Event-B and a motivating example. In Section 3, we present an Event-B formal model of the elastic SCA-based applications. We propose, in Section 4 a transformation algorithm for automatic modeling of the SCA-based applications elasticity and the evaluation of the proposed approach. Section 5 concludes this paper and presents some future works.

2 Background and related work

In this section, we present a summary of related work and the main concepts of the SCA-based applications. Then, we introduce the basic knowledges of the Event-B method that are relevant for the comprehension of the paper.

2.1 Related work

One of the most relevant issues raised by the Cloud environment is the elasticity at different levels. Elasticity is the ability to determine the amount of resources to be allocated as efficiently as possible according to users requests. Many approaches based on predictive or reactive strategies have been proposed to address this issue [4, 5]. Reactive strategies [6–8] are based on Rule-Condition-Action mechanisms. While predictive strategies [9, 10] are based on predictive-performance models and load forecasts.

In [11], the authors introduce the Vienna Platform for Elastic Processes (ViePEP), a platform for realizing elastic processes in cloud environment. ViePEP allows the monitoring of the process execution and the reasoning about optimizing resources utilization using the current and future system landscape. To do this, the platform can carry out a set of elasticity actions. In [12, 13], the authors extend ViePEP with two works. The first extension consists of a prediction and reasoning algorithm, based on knowledge about the current and future process landscape, for elastic process execution. While the second consists of a scheduling and resources-allocation algorithms, based on user defined non-functional requirements, in order to optimize resources utilization. Though the authors discuss the elasticity at the process level as we did in our approach, the ViePEP approach seems to be difficult to use. In fact, since the ViePEP Reasoner ensures the elasticity of all the deployed processes, it may be not elastic. In [14], the authors present ElaaS, a service for managing elasticity in the Cloud. ElaaS is implemented as a SaaS application that can be used in any cloud environment and on any cloud-enabled application. It is composed of a set of pluggable components for managing applications, monitoring and business logic. Elasticity is insured based on the deployment graph of the considered application and its KPI. While the idea of pushing elasticity management to the applications is in line with our approach, the ElaaS is difficult to use since it assumes efforts from the application designer who will be in charge of delivering necessary information for the business logic manager for elasticity enforcement. In [15], the authors present a framework for modeling and reasoning about non-functional properties that should reflect elasticity of deployed business services. As defined the elastic properties and mechanisms can tackle any application, since the characteristics of business processes (structure or behavior) are not considered in the proposed approach. The elasticity mechanisms serve all the deployed processes which can lead to the overload of the elasticity mechanisms. In [16], the authors present an approach that consists in producing a model for an elastic Service-based Business Process (SBP) which is the result of the composition of the SBP model with models of mechanisms for elasticity. Unlike our approach, the proposed approach requires an effort from the designer. In addition, the proposed approach changes the nature of the considered SBP. In [17], the authors consider scaling at both the service and application levels in order to ensure elasticity. They discuss the elasticity at the service level as we did in our approach. Nevertheless, the correctness of the proposed mechanisms is not proved since the approach is not based on a formal model.

The formal specification is necessary to properly research, analyze and manipulate Web services. Several formalisms have been proposed, such as Process Algebra [18], Event Calculus [19], Petri Nets [20], or Event-B [21]. In [22], authors proposed a work consisting in translating a formal signature model and a behavior model for SCA to Promela, and Promela specifications are thereafter verified with the model checker SPIN. In [23], a formal language (SRML-P) is presented to specify the interaction protocol between components. SRML-P provides a mathematical framework, in which some service-modeling primitives are defined and application models can be reasoned about. In [24], authors proposed an MDE approach to obtain SCA models and to verify the properties of these models. They applied two transformations: the first one to obtain SCA models using UML 2.0 meta-model and the second transformation to ensure the verification of these properties of models using Event-B meta-model. Thus, they defined transformation rules in the ATL language to ensure the consistency of the model. In [25], authors proposes a framework for modeling services based applications. They proposes an executable language based on SCA standard and the ASM formal method for modeling service behavior and interactions in an abstract way. However, all these works are not suitable for Cloud environments since they do not support the elasticity of an SCA composition.

The approaches for elasticity, mainly those we cite above, did not take into account SCA-based applications. The work we present in this paper is new in the sense that it (1) transforms a non-elastic SCA composition into an elastic one (2) tackles the problem of elasticity of SCA-based application at the SaaS level, (3) is based on a formal model and (4) proposes a formal verification of the elasticity mechanisms.

2.2 Service Component Architecture (SCA)

Service Component Architecture (SCA) [26] defines a general approach that describes how to create components and which mechanism to use for describing how those components communicate together. SCA defines a generalized notion of a component. It also specifies how those components can be combined into larger logical structures called composites. Indeed, the implementation defines the component's functions, e.g. Java class or BPEL process, while the SCDL configuration specifies how SCA components interact with each others, using a common set of abstractions including services, references and properties [27].

Each component exposes its business logic as one or more services and it uses references to invoke services from other components. Besides, a component can define some properties which contain values that can be read by that component when it's instantiated. A wire is an abstract representation of the relationship between a reference and some services that meet its needs. A set of components communicating between each other by means of services are called SCA composite. Though, composites can call external composite services through promoted references and they can expose their own services to external composites through promoted services. Services of components promoted by the composite

can be used as services from outside requestors, while references denote that the composite is served by outside composites services.

Each component has a limited capacity in terms of services it can offer. However, the demands can increase/decrease at any moment leading to over-provisioning or under-provisioning problem. To avoid such a problem, an elasticity mechanism should be used in order to adapt the offered capacity with respect to the current demands. Basically, we propose to duplicate (resp. consolidate) each overloaded (resp. under-overloaded) component when its demands become greater (resp. lower) than the capacity offered by the component itself and their created copies. Recall that the consolidation consists in removing the unnecessary copies of the under-overloaded components.

To sum up, the development of a SCA based-application should respect the following requirements:

Req1. A composite is made of a set of components. Each component belongs to a single composite and each composite must contain at least one component.

Req2. A composite is made of a set of services, references and properties. Each service (resp. reference, property) belongs to at the most one composite. A composite must have at least one service.

Req3. A component is made of a set of services, references and properties. Each service (resp. reference, property) belongs to at the most one component. A component must have at least one service/reference.

Req4. A service (resp. reference, property) cannot belong to a component and a composite at the same time.

Req5. A wire permits to link services and references of components of the same composite.

Req6. The service and the reference linked by a wire should belong to different components.

Req7. Each service (resp. reference) that does not involve in a wire is promoted as a service (resp. reference) of the composite.

Req8. Each property of a component is promoted as a property of the composite. Contrary to a service or a reference, each component property is promoted to a distinct composite property.

The elasticity mechanism must fulfill the following additional requirements:

Req9. A component may be a copy of a single component. A component may have several copies.

Req10. If a component A is a copy of another component B then B cannot be a copy of another component.

Req11. A copy and the original component should have similar functionalities: the same number of services (resp. references, properties).

Req12. A copy cannot be used in a composite if the original component is not present in this composite.

Req13. When a copy is used in a composite, its services (resp. references) are linked to the same elements (services or references) as its original components.

Req14. When a copy is used in composite, its property is promoted to a distinct composite property.

Req15. At each moment if the request on a component is greater than the offered capacity, it should be possible to honor the request in the future by duplicating the component.

Req16. At each moment if the request on a component is lower than the offered capacity, it should be possible to remove in the future the unnecessary copies of the component.

Figure 1 depicts an example of an online computer shopping application based on SCA composition. This example will be carried on along this paper.

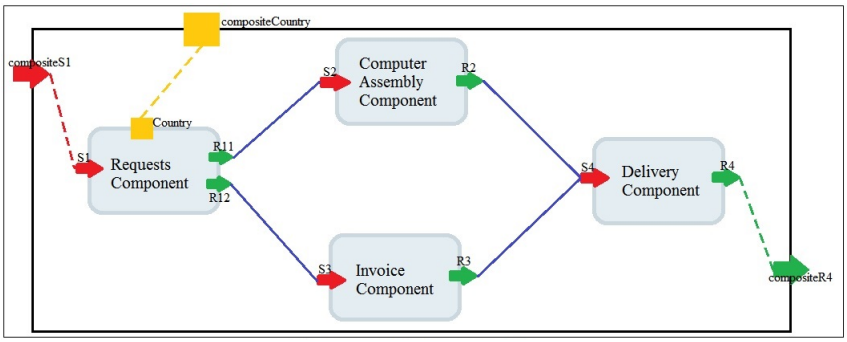

Fig. 1: Online Computer Shopping composite example

The online computer shopping composite is composed of four components:

- **Requests component:** receives requests to purchase a computer through its service "RequestsService" (S1). This service is promoted by the composite (compositeS1). The Requests component calls the Computer assembly component service and the Invoice component service via its references, e.g. "ComputerAssemblyReference"(R11) and "InvoiceReference"(R12). The property "Country" (Country) of this component indicates the country where the application is deployed.
- **Computer assembly component:** performs the assembly of computer components according to the requester desires. This functionality is presented by its "ComputerAssemblyService" (S2) service. The "AssemblyDeliveryReference" (R2) refers to the service of the Delivery component.
- **Invoice component:** creates the invoice related to the purchased computer. This component is accessible via its "InvoiceService" (S3) service. The "InvoiceDeliveryReference" (R3) refers to the service of the Delivery component.

– **Delivery component:** delivers the computer with its invoice to the customer. "DeliveryService" (S4) is the interface of the component. If the delivery is international, the composition of our application may need external service of international delivery. This is presented with the promotion relationship between the "InternationalDeliveryReference" (R4) of the current component and the "CompositeInternationalDeliveryReference" (compositeR4) of the composite.

In the next section, we present an Event-B model of an SCA-based application that respects all the above requirements. Let us remark that the two last requirements are dynamic ones since they involve states taken at different moments: the present and the future. Such requirements cannot be easily expressed as invariants. This is why we suggest to model them as CTL formulas and verify them using the approach introduced in [28].

2.3 Event-B

The Event-B [21] is a stepwise formal development method based on the theory of sets. Compared to other formal methods (Petri Net [20], Event-calculus[29], process algebra [18], etc.), the strong point of Event-B is that it offers a refinement process that permits to master the complexity of a system. This is the main reason for which we adopted Event-B as a formal method. Indeed, the designers start by abstractly specifying the requirements of the whole system by focusing more on its global goal/structure then details are gradually introduced by refinement. An Event-B specification is made of two elements, named *context* and *machine*, whose structures and links are depicted in Figure 2. A context describes the static part of an Event-B specification; it consists of constants C and sets (user-defined types) S together with axioms A 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 E. 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 has the following form: **ANY** X **WHEN** G **THEN** Act **END**. It can be executed if it is enabled, i.e. all the conditions G, named guards, prior to its execution hold. Among all enabled events, only one is executed. In that case, substitutions Act, called actions, are applied over variables. The execution of each event should maintain the invariant. To this aim, proof obligations are generated. For each event, we have to establish that:

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

where $[Act]Inv$ gives the weakest constraint on the *before* state such that the execution of Act leads to an *after* state satisfying Inv.

Refinement is a process of enriching or modifying a model in order to augment the functionality being modeled, or/and explain how some purposes are achieved. Both Event-B elements *context* and *machine* can be refined. A context can be

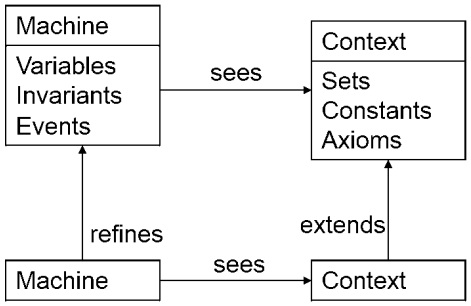

Fig. 2: Machine and context relationships

extended by defining new sets S_r and/or constants C_r together with new axioms A_r. A machine is refined by adding new variables and/or replacing existing variables by 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, the refined events have the same form: **ANY** X_r **WHEN** G_r **THEN** Act_r **END**. To prove that a refinement is correct, we have to establish the following two proof obligations (each element Y_r refines its corresponding one Y):

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

$$\forall(S, C, S_r, C_r, V, V_r, X, X_r).$$
$$(A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$$

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

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

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 an automatic and an interactive options to discharge the proof obligations.

3 Modeling elastic SCA-based application with Event-B

In this section, we present our approach for formalizing the elastic SCA-based application. The formalization approach consists of two steps. We start by modeling the SCA-based application without elasticity mechanisms. Then, we extend this model by introducing the elasticity mechanism of a SCA-based application.

[1] `http://www.event-b.org/install.html`

3.1 Modeling SCA-based applications

In a first step, we start by modeling the structure of a composite and a component together with the event that permits to create a new composite. For this aim, we define a context "SCAContext" that specifies some sets to represent all the possible *Composites* (resp. *Properties*, *References*, *Services*, *Components*). As we do not consider the evolution of the component's structure, the services (resp. references, properties) of a components are modeled as a constant partial function that maps a given service (resp. reference, property) into its potential component (Axioms $axm1$-$axm3$). The function is partial since a service may not belong to any component. Contrary to the function modeling the properties (Axiom $axm3$), the functions corresponding to services/references are surjective since each components must have at least one service/reference (Axioms $axm1$ and $axm2$). In other words, the context "SCAContext" models the requirement **Req3** (See Figure 3).

CONTEXT **SCAContext**
SETS
$Composites$
$Properties$
$References$
$Services$
$Components$
CONSTANTS
$ComponentsReferences$
$ComponentsProperties$
$ComponentsServices$
...
AXIOMS
axm1 : $ComponentsServices \in Services \twoheadrightarrow Components$
axm2 : $ComponentsReferences \in References \twoheadrightarrow Components$
axm3 : $ComponentsProperties \in Properties \rightarrow Components$
END

Fig. 3: An Event-B model for the SCA-based applications : the context

The above context is seen (clause **SEES**) by the machine "SCAMachine" that defines some variables to model the characteristics of each existing composite, namely its components, services, references and properties (See Figure 4). The set of the existing composites is modeled by the variable *composites* that is subset of the given set *Composites* (invariant $inv1$). Similarly to a component, the invariants $inv2$ and $inv3$ model the requirement regarding the services/references of a composite. The invariant $inv4$ models the requirement **Req4** regarding services, similar invariants have been defined for references/properties. Finally, the invariant $inv5$ models the components that constitute a composite (**Req1**).

The wire connecting the services and the references is represented by a relation defined between the services/references of the components that involve in a composite. Invariants $inv6$, $inv7$ and $inv8$, of Figure 5 cover the requirements (**Req5**) and (**Req6**).

MACHINE **SCAMachine**
SEES **SCAContext**
VARIABLES
$composites$
$CompositesReferences$
$CompositesServices$
...
INVARIANTS
inv1 : $composites \subseteq Composites$
inv2 : $CompositesServices \in Services \twoheadrightarrow composites$
inv3 : $CompositesReferences \in References \twoheadrightarrow composites$
inv4 : $dom(CompositesServices) \cap dom(ComponentsServices) = \varnothing$
inv5 : $contained \in Components \twoheadrightarrow composites$
...
END

Fig. 4: An Event-B model for the SCA-based applications : the machine (1)

INVARIANTS
inv6 : $wires \in ComponentsReferences^{-1}[dom(contained)] \leftrightarrow ComponentsServices^{-1}[dom(contained)]$
inv7 : $\forall x, y \cdot (x \mapsto y \in wires \Rightarrow contained(ComponentsReferences(x)) = contained(ComponentsServices(y))$
inv8 : $\forall x, y \cdot (x \mapsto y \in wires \Rightarrow ComponentsReferences(x) \neq ComponentsServices(y))$

Fig. 5: An Event-B model for the SCA-based applications : the machine (2)

Figure 6 depicts the Event-B invariants related to the promotion of services/references/properties of a component. The invariants $inv9$ and $inv10$ (resp. $inv11$ and $inv12$) model the promotion of services (resp. references) as a total function from the set of services (resp. references) that do not involve in wires into the services (resp. references) of the composite. They cover the requirement **Req7**. Requirement (**Req8**) related to the promotion of properties is modeled by a bijective function ($inv13$).

INVARIANTS
inv9: $promotedServs \in ComponentsServices^{-1}[contained^{-1}[composites]] \setminus ran(wires) \rightarrow CompositesServices^{-1}[composites]$
inv10: $\forall x, y \cdot (x \mapsto y \in promotedServs \Rightarrow contained(ComponentsServices(x)) = CompositesServices(y))$
inv11: $promotedRefs \in ComponentsReferences^{-1}[contained^{-1}[composites]] \setminus dom(wires) \rightarrow CompositesReferences^{-1}[composites]$
inv12: $\forall x, y \cdot (x \mapsto y \in promotedRefs \Rightarrow contained(ComponentsReferences(x)) = CompositesReferences(y))$
inv13: $promotedProps \in ComponentsProperties^{-1}[contained^{-1}[composites]] \rightarrowtail CompositesProperties^{-1}[composites]$
inv14: $\forall x, y \cdot (x \mapsto y \in promotedProps \Rightarrow contained(ComponentsProperties(x)) = CompositesReferences(y))$

Fig. 6: An Event-B model for the SCA-based applications : the machine (3)

To create a new composite, an event *AddComposite* is defined in the machine "SCAMachine". More details about the specification of this event can be found at [30]. According to the Event-B model presented in this section, the SCA-based application of Figure 1 is modeled by:

$Composites = \{composite1, \ldots\}$
$Components = \{RC, CAC, IC, DC, cRC, \ldots\}$
$Services = \{compositeS1, S1, S2, S3, S4, cS1, \ldots\}$
$References = \{R11, R12, R2, R3, R4, compositeR4, cR11, cR12, \ldots\}$
$Properties = \{compositeCountry, Country, cCountry, ccompositeCountry, \ldots\}$
$ComponentsProperties = \{Country \mapsto RC, cCountry \mapsto cRC, \ldots\}$
$ComponentsServices = \{S1 \mapsto RC, S2 \mapsto CAC, S3 \mapsto IC, S4 \mapsto DC, cS1 \mapsto cRC, \ldots\}$
$ComponentsReferences = \{R11 \mapsto RC, R12 \mapsto RC, R2 \mapsto CAC, R3 \mapsto IC,$
$R4 \mapsto DC, cR11 \mapsto cRC, cR12 \mapsto cRC, \ldots\}$

$CompositesReferences = \{compositeR4 \mapsto composite1, \ldots\}$
$CompositesServices = \{compositeS1 \mapsto composite1, \ldots\}$
$contained = \{RC \mapsto composite1, CAC \mapsto composite1, IC \mapsto composite1,$
$DC \mapsto composite1, \ldots\}$
$wires = \{R11 \mapsto S2, R12 \mapsto S3, R3 \mapsto S4, R2 \mapsto S4, \ldots\}$
$CompositesProperties = \{compositeCountry \mapsto composite1, \ldots\}$
$promotedProps = \{Country \mapsto compositeCountry, \ldots\}$
$promotedRefs = \{R4 \mapsto compositeR4, \ldots\}$
$promotedServs = \{S1 \mapsto compositeS1, \ldots\}$

Let us remark that the set $Components$ (resp. $Services$, $References$, $Properties$) includes the information about the original elements but also their copies. However at the current abstraction level, these copies are considered like other components.

3.2 Modeling the elasticity in SCA-based applications

Recall that ensuring SCA-based applications elasticity consists in providing Cloud environments with mechanisms that allow a deployed SCA-based application to scale up or down whenever needed. To scale up a SCA-based application, the elasticity mechanisms must duplicate the application in order to create as many instances as needed to handle the dynamically received requests. To scale down a SCA-based application, these mechanisms should consolidate the application in order to remove useless instances, thereby avoiding under-utilization of resources. In this section, we extend the formalization, developed in the previous section, with elasticity mechanisms in order to formally model the elastic SCA-based applications. The elasticity mechanisms are two operations namely duplicate and consolidate. The first operation is used to scale-up the SCA-based application, while the second one is used to scale it down.

In Event-B, the elasticity mechanism is introduced using the refinement. Thus, we create a second abstraction level that refines the first one. Basically,

we extend the first context by defining a new one *SCAcontextCopy* (See Figure 7) and a new machine $SCAMachineRef$ (See Figure 8) that refines the machine $SCAMachine$. The context *SCAcontextCopy* defines a set of constants together with their properties to model the copy relationship between some components:

1. Requirement (**Req9**) (resp. **Req10**, **Req11**) is covered with the axiom $axm1$ (resp. $axm2$ and $axm3$)
2. Axioms $axm4$ and $axm5$ matches the services of a components and those of its copy. Similar axioms have been defined for the references and the properties matching.
3. Axiom $axm10$ defines the capacity of each non-copy component. A copy component has implicitly a same capacity as the original one. This is why we choose to do not specify it.

CONTEXT **SCAcontextCopy**
EXTENDS **SCAContext**
CONSTANTS
copy copyServices copyReferences copyProperties
AXIOMS
axm1: $copy \in Components \nrightarrow Components$
axm2: $dom(copy) \cap ran(copy) = \varnothing$
axm3: $\forall x, y(x \mapsto y \in copy \Rightarrow$
$card(ComponentsProperties^{-1}[\{x\}]) = card(ComponentsProperties^{-1}[\{y\}]) \wedge$
$card(ComponentsServices^{-1}[\{x\}]) = card(ComponentsServices^{-1}[\{y\}]) \wedge$
$card(ComponentsReferences^{-1}[\{x\}]) = card(ComponentsReferences^{-1}[\{y\}]))$
axm4: $copyServices \in ComponentsServices^{-1}[dom(copy)] \rightarrow ComponentsServices^{-1}[ran(copy)]$
axm5: $\forall x, y(x \mapsto y \in copyServices \Rightarrow ComponentsServices(x) \mapsto ComponentsServices(y) \in copy)$
...
axm10: $ComponentsCapacity \in Components \backslash dom(copy) \rightarrow N1$

Fig. 7: An Event-B model for the SCA-based applications elasticity: the context

According to this previous formal specification, our example gives:

$copy = \{cRC \mapsto RC, \ldots\}$
$copyServices = \{cS1 \mapsto S1, \ldots\}$
$copyReferences = \{cR11 \mapsto R11, cR12 \mapsto R12, \ldots\}$
$copyProperties = \{cCountry \mapsto Country, \ldots\}$

The context *SCAcontextCopy* being defined, we model the elasticity mechanism, thanks to the refinement concept of Event-B, by introducing some additional variables and two new events *Duplicate* and *Consolidate*. The following variables have been added together with the invariants described in Figure 8:

– *containedCopy*: this partial function states the possible composite to which a component copy belongs to ($inv1$). Invariant ($inv2$) permits to distinguish the functions *containedCopy* and *contained* by specifying that *containedCopy* exclusively concerns the component copies whereas *contained* is related to the original one. Invariant ($inv3$) models the requirement (**Req12**).

- *wiresCopy*: this relation describes the possible links between the services/references of a component copy and the services/references of the other components ($inv4$). Invariants ($inv6$) and ($inv7$) covers the requirement (**Req13**) regarding the services of a component copy; a similar invariant is defined for the references.
- *CompositesPropertiesCopy*: this partial function states the possible composite to which a property copy belongs to.
- *promotedPropsCopy*: this total function states the promotion links between the properties copies of the components and the properties copies of the composites (Invariant ($inv8$)). It covers requirement (**Req14**).
- *promotedRefsCopy* (resp. *promotedServsCopy*): this relation states the promotion links between the services (resp. references) copies of the components and the services (resp. references) copies of the composites.
- *ComponentsNeed*: this total function specifies the application needs on each component (Invariant ($inv9$)).
- *ComponentsOffered*: this total function specifies the capacity offered by a component and the copies belonging to the composite. (Invariants ($inv10$) and ($inv11$))

MACHINE **SCAMachineRef**
REFINES **SCAMachineRef**
SEES **SCAcontextCopy**
INVARIANTS

inv1 : $containedCopy \in dom(copy) \nrightarrow composites$
inv2 : $dom(contained) \cap dom(copy) = \varnothing$
inv3 : $\forall x, y \cdot (x \mapsto y \in copy \wedge x \in dom(containedCopy) \Rightarrow$
$y \in dom(contained) \wedge containedCopy(x) = contained(y))$
inv4 : $wiresCopy \in ComponentsReferences^{-1}[dom(containedCopy)] \cup dom(wires) \leftrightarrow$
$ComponentsServices^{-1}[dom(containedCopy)] \cup ran(wires)$
inv5: $promotedServsCopy \in copyServices^{-1}[dom(promotedServs)] \cap$
$ComponentsServices^{-1}[dom(containedCopy)] \rightarrow ran(promotedServs)$
inv6 $\forall x, y (x \mapsto y \in copy \wedge x \in dom(containedCopy) \Rightarrow$
$(\forall z (z \in ComponentsReferences^{-1}[\{x\}] \Rightarrow$
$wiresCopy[\{z\}] = (wires \cup wiresCopy)[\{copyReferences(z)\}])))$
inv7: $\forall x, y (x \mapsto y \in copy \wedge x \in dom(containedCopy) \Rightarrow$
$(\forall z (z \in ComponentsServices^{-1}[\{x\}] \wedge copyServices(z) \in dom(promotedServs) \Rightarrow$
$z \in dom(promotedServsCopy) \wedge$
$promotedServsCopy(z) = promotedServs(copyServices(z))$
inv8: $promotedPropsCopy \in ComponentsProperties^{-1}[containedCopy^{-1}[composites]] \twoheadrightarrow$
$CompositesPropertiesCopy^{-1}[composites]$
inv9: $ComponentsNeed \in dom(contained) \rightarrow N1$
inv10: $ComponentsOffered \in dom(contained) \rightarrow N1$
inv11: $\forall x.x \in dom(contained) \Rightarrow ComponentsOffered(x) = ComponentsCapacity(x) \times$
$(card(dom(containedCopy)copy[\{x\}]) + 1)$
....

Fig. 8: The refined machine

Duplication Excessive invocations of a component can exceed its capacity, which directly affect its Quality of Service (Qos). As a solution to overcome

the overload problem, we suggest to add a copy of this component into the related composite. This is modeled by the event *Duplicate* of Figure 9 where a component *comp* is duplicated by adding a copy *copycomp* (Guards $grd1$, $grd2$ and $grd3$). This event occurs when the needed capacity of the concerned component exceeds the capacity provided by the component itself and all the copies already present in the composite (Guard $grd4$). In that case, *copycomp* is added to the composite of the *comp* component ($act1$). The services (resp. references) of *copycomp* are linked to the same elements as those of *comp* (actions $act2$, $act3$ and $act4$). Also, new properties are added to the related composite and links are created to promote the properties of *copycomp* (actions $act5$ and $act6$). Finally, the capacity offered by the component is updated (action $act7$).

Duplicate $\widehat{=}$
ANY comp copycomp pros promProp
WHERE
grd1: $comp \in Components \wedge comp \in dom(contained)$
grd2: $copycomp \in Components \wedge copycomp \notin dom(contained \cup containedCopy)$
grd3: $copycomp \mapsto comp \in copy$
grd4: $ComponentsNeed(comp) > ComponentsOffered(comp)$
grd5: $pros \subseteq Properties \setminus (dom(CompositesProperties \cup CompositesPropertiesCopy) \cup dom(ComponentsProperties)) \wedge card(pros) = card(ComponentsProperties[\{copycomp\}])$
grd6: $promProp \in ComponentsProperties^{-1}[\{copycomp\}] \rightarrowtail\!\!\!\!\!\!\rightarrow pros$
THEN
act1: $containedCopy(copycomp) := contained(comp)$
act2: $wiresCopy := wiresCopy \cup$
$(\bigcup ref \cdot (ref \in ComponentsReferences^{-1}[\{copycomp\}] \wedge copyReferences(ref) \in dom(wires \cup wiresCopy)) | \{ref\} \times (wires \cup wiresCopy)[\{copyReferences(ref)\}])$
$\cup$
$(\bigcup ser \cdot (ser \in ComponentsServices^{-1}[\{copycomp\}] \wedge copyServices(ser) \in ran(wires \cup wiresCopy)) | (wires \cup wiresCopy)^{-1}[\{copyServices(ser)\}] \times \{ser\})$
act3: $promotedRefsCopy := promotedRefsCopy \cup$
$(\bigcup ref \cdot (ref \in ComponentsReferences^{-1}[\{copycomp\}] \wedge copyReferences(ref) \in dom(promotedRefs)) | \{ref \mapsto (promotedRefs)(copyReferences(ref))\})$
act4: $promotedServsCopy := promotedServsCopy \cup$
$(\bigcup ser \cdot (ser \in ComponentsServices^{-1}[\{copycomp\}] \wedge copyServices(ser) \in dom(promotedServs)) | \{ser \mapsto promotedServs(copyServices(ser))\})$
act5: $CompositesPropertiesCopy := CompositesPropertiesCopy \cup (pros \times \{contained(comp)\})$
act6: $promotedPropsCopy := promotedPropsCopy \cup promProp$
act7: $ComponentsOffered(comp) := ComponentsOffered(comp) + ComponentsCapacity(comp)$
END

Fig. 9: Modeling the duplication operation

Let us consider that the capacity of the component *RC* is equal to 10 and the need of the application is 15. So to meet this demand, the event *Duplicate* is

executed resulting in adding the copy cRC to the composite. Figure 10 depicts the initial composite after the duplication of the component RC. From the Event-B modeling point of view, the variable added in the refined machine are as follows:

$containedCopy := \{copycomp \mapsto composite1\}$
$wiresCopy := \{CR11 \mapsto S2, CR12 \mapsto S3, CR3 \mapsto S4, CR2 \mapsto S4, \ldots\}$
$CompositesPropertiesCopy = \{CcompositeCountry \mapsto composite1, \ldots\}$
$promotedPropsCopy = \{CCountry \mapsto compositeCountry, \ldots\}$
$promotedServsCopy = \{CS1 \mapsto compositeS1, \ldots\}$

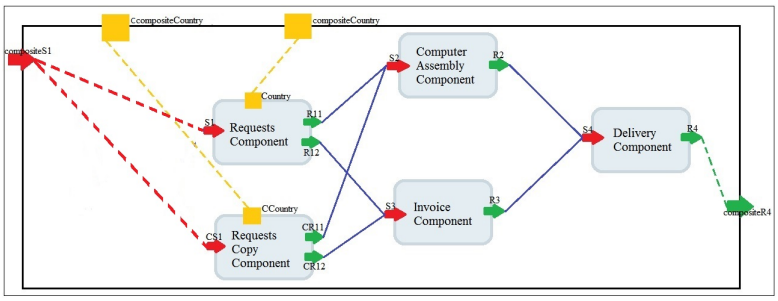

Fig. 10: Online Computer Shopping composite example after duplicating RC

Consolidation Figure 11 depicts the Event-B specification of the event *Consolidate*. This event is enabled when the needed capacity on a component can still be honored even if a copy is removed (guard $grd3$). Thus, this event consists in removing a copy *copycomp* of a component from a composite. Moreover in order to preserve the different invariants; all the variables are updated such as each pair involving *copycomp* is deleted from these variables (actions $act1$-$act6$). Finally, the capacity offered by the component is updated (action $act7$). The consolidation operation is defined as dual to the duplication operation that removes a copy of a component whenever this copy is under-provisioned. After a finite number of consolidation we restore the original SCA composition. Of course to make the needed capacity on a component evolve (increase/decrease), we have defined the event *SetNeed* that set the variable *ComponentsNeed* (See Figure 12).
The global architecture of the Event-B development is depicted in Figure 13:

Consolidate $\widehat{=}$
ANY copycomp
WHERE
grd1: $copycomp \in Components \wedge copycomp \in dom(containedCopy)$
grd2: $copycomp \in dom(copy)$
grd3: $ComponentsNeed(copy(copycomp)) \leq ComponentsOffered(copy(copycomp))$- $ComponentsCapacity(copy(copycomp))$
THEN
act1: $CompositesPropertiesCopy :=$
$promotedPropsCopy[ComponentsProperties^{-1}[\{copycomp\}]] \lhd CompositesPropertiesCopy$
act2: $promotedPropsCopy := ComponentsProperties^{-1}[\{copycomp\}] \lhd promotedPropsCopy$
act3: $wiresCopy := ComponentsServices^{-1}[\{copycomp\}] \lhd wiresCopy \rhd ComponentsReferences^{-1}[\{copycomp\}]$
act4: $containedCopy := \{copycomp\} \lhd containedCopy$
act5: $promotedRefsCopy := ComponentsReferences^{-1}[\{copycomp\}] \lhd promotedRefsCopy$
act6: $promotedServsCopy := ComponentsServices^{-1}[\{copycomp\}] \lhd promotedServsCopy$
act7: $ComponentsOffered(copy(copycomp)) := ComponentsOffered(copy(copycomp))$- $ComponentsCapacity(copy(copycomp))$
END

Fig. 11: Modeling the consolidation operation

SetNeed $\widehat{=}$
ANY comp need
WHERE
grd1: $comp \in dom(contained)$
grd2: $need \in N1$
THEN
act1: $ComponentsNeed(comp) := need$
END

Fig. 12: Modeling the SetNeed event

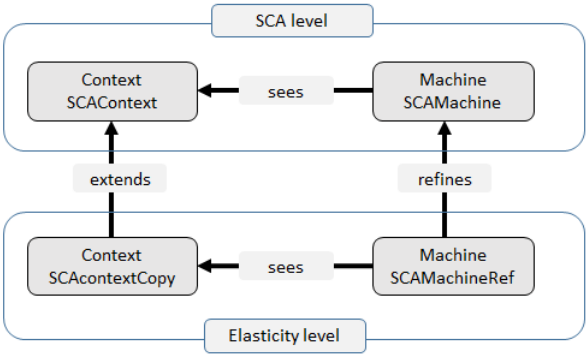

Fig. 13: The Event-B architecture model

4 Implementation and evaluation

In the previous section, we detailed the Event-B model of a SCA-based application and how we formally modeled the artifacts of the SCA specification (composite, components, services, references, properties, wires, etc.). Also, we

presented our modeling approach of the elasticity of the SCA-based applications. In this section, we describe the validation of the Event-B model, the tool that implements the translation approach together with its evaluation.

4.1 Validation

To validate the Event-B model built in the previous section, we have operated into two steps. In the first step, we have used the ProB [1] animator/model checher that permits to play different scenarios and check the behavior of the Event-B models with respect to the desired system by showing at each step the values of each variable, which events are enabled and which are not. In a next step, we used ProB in order to ensure that the invariant is not trivially falsified. When ProB finds a counter-example for the invariant, it provides the sequence of the operations that leads to an invalid state from the initial state. In such cases, we reworked and fixed our Event-B specification by adding/modifying some guards/actions. When ProB found no counter-example on the whole Event-B models; this gave us some confidence about the correctness of the specification before performing the proof activity. The proof obligations that are generated to ensure, on the one hand, that each event preserve the invariants and, on the other hand, to verify that refinement is correct. To this end, 140 proof obligations have been generated: 70% of them are automatically discharged by the automatic prover, the remainder proofs are not very difficult, the automatic prover fails to discharge them because they require several steps. To carry out their proof, we use the interactive prover by helping it find the right steps and rules to apply. Moreover, we have expressed and verified the requirements **Req15** and **Req16** as CTL formulas. For instance, **Req15** is modeled by:

$$
\begin{array}{l}
\mathsf{AG}\ (comp \in dom(contained) \wedge \\
\quad ComponentsNeed(comp) > ComponentsOffered(comp) \wedge \\
\quad card(copy^{-1}[\{comp\}]) \times ComponentsCapacity(comp) \geq \\
\qquad (ComponentsOffered(comp) - ComponentsNeed(comp)) \\
\Rightarrow \\
\quad \mathsf{EF}\ ComponentsNeed(comp) \leq ComponentsOffered(comp))
\end{array}
$$

The second condition ensures that there is enough copies to satisfy the demand. To verify the above formula, we have applied the proof-based approach introduced in [28]. To prove a CTL formula $\mathsf{AG}\ (\psi \Rightarrow \mathsf{EF}\ \psi)$, the approach consists in exhibiting a program that starting from a ψ-state terminates in a state that satisfies ϕ. For the above formula, the program would be:

WHILE $(ComponentsNeed(comp) > ComponentsOffered(comp))$
DO $Duplicate(comp)$
INVARIANT ψ
VARIANT $max(0, ComponentsNeed(comp) - ComponentsOffered(comp))$
END

where ψ is equal to:

$$comp \in dom(contained) \wedge$$
$$ComponentsNeed(comp) > ComponentsOffered(comp) \wedge$$
$$card(copy^{-1}[\{comp\}]) \times ComponentsCapacity(comp) \geq$$
$$(ComponentsOffered(comp) - ComponentsNeed(comp))$$

To prove that this program establishes the formula, we have to verify:

1. the variant of the loop denotes a natural number:

$$max(0, ComponentsNeed(comp) - ComponentsOffered(comp)) \in NAT$$

2. to ensure the termination of the loop, the execution of the event $Duplicate(comp)$ should make the variant decrease:

$$\forall n.(n \in NAT \wedge$$
$$n = max(0, ComponentsNeed(comp) - ComponentsOffered(comp))$$
$$\Rightarrow$$
$$[Duplicate(comp)]$$
$$(max(0, ComponentsNeed(comp) - ComponentsOffered(comp)) < n)$$

3. the invariant ψ is preserved by each iteration of the loop :

$$\psi \Rightarrow [Duplicate(comp)]\psi$$

All the above formulas have been established as correct which validates the requirement **Req15**.

4.2 Transformation algorithm for automatic verification of the elastic SCA-based application

Here, we proceed at extending the SCA2B plug-in [31]. This tool automates the transformation of the SCDL description, i.e. the ".composite" file, of a SCA-based application to a formal Event-B model. Thus, to automate the modeling of the elasticity events, we extend the SCA2B plug-in by creating a new Machine that refines the abstract machine and we add to the refined Machine two methods where each one generates one of the elasticity events.

In this section, we present an approach for automatic formalizing the elastic SCA-based application. Our approach consists in two steps:

- (i) transform an elastic SCA-based application to an Event-B model;
- (ii) check the correctness of the elastic SCA-based application;

We introduce an algorithm for automatic generating an event-B model of the elastic SCA-based applications (see Algorithm 1).

The algorithm that we propose is generic in the sens that it takes as input any SCA-based application (SCAFile). It returns an Event-B model composed of three components type namely a context, a machine and the refined machine.

Algorithm 1 SCATranslator Algorithm

Require: *SCAFile*
Ensure: *Event-B project*
1: $ContextFile \leftarrow createContext(SCAFile)$
2: $MachineFile \leftarrow createMachine(SCAFile, ContextFile)$
3: $RefinedMachineFile \leftarrow refineMachine(MachineFile, SCAFile)$

Our transformation algorithm is executed in three main steps: generate an abstract model composed of a machine and a context (see lines 1-2), and (ii) refine this abstract model by introducing the concept of the elasticity (see line 3). At the first step, we use the following two functions: *createContext* and *createMachine* . *createContext* firstly reads the input file (e.i. the ".composite" file that holds the description of an example of an SCA-based application). Then, it creates the SCA artifacts abstract sets and the related axioms. In other words, each artifact of the SCA specification (e.g. components, services, references, etc.) is represented as we have defined in the previous section.

The *createMachine* function, after reading the SCA descriptor file (*SCA-CompositeFile*), creates the set of variables that are then taken as parameters for the creation of the invariants. These lasts define the type of variables and the relationship between the different artifacts of the SCA specification (e.g. *ComponentsServices, ComponentsReferences, promotedServs, wires, etc.*).

The second step of our transformation algorithm allows to extends the created model with elasticity. This is done by using a function refineMachine. This algorithm deals with the creation of a new machine that includes the elasticity mechanisms. Indeed, like the "creatMachine" , it starts by reading the SCA descriptor file (e.i. ".composite" file) then it creates the set of variables on which we apply gluing invariants in order to maintain the same state and semantic of the first machine. Finally, we add procedures where each one focus on creating one of the two proposed elasticity events (e.i. Duplication, Consolidation). These last ones deal with the current input SCA descriptor file and create the adequate actions depending on the composite example.

4.3 Deployment approach of the elastic SCA-based application

The goal of this section is to give an overview about the deployment of elastic SCA-based application after a correctness verification process. In [32], authors propose a deployment framework for non-elastic SCA-based applications. This deployment framework can not be used for deploying elastic SCA-based application that are validated using our verification approach.

In our approach, we propose an elastic deployment framework for deploying elastic SCA-based application. Our deployment framework is an extension of [32] to deal with the elasticity properties of SCA-based applications. To achieve our goal, we extend the packaging framework by adding generic modules implementing our elasticity mechanisms, i.e. Duplication/Consolidation. The figure 14 shows the extended framework

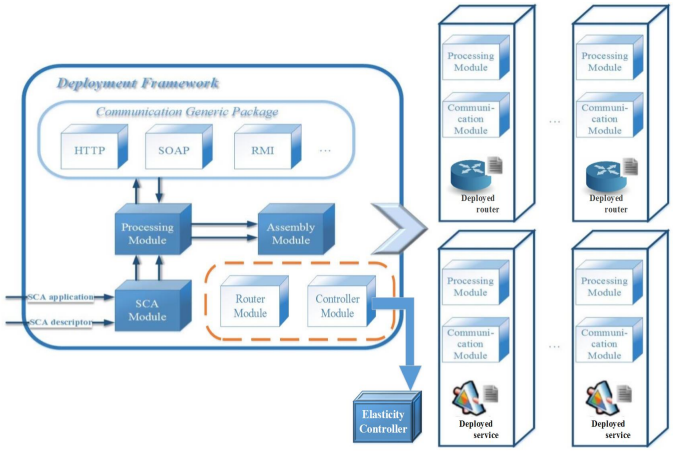

Fig. 14: Extended Packaging Framework

– Router Module: Through this module, we can dedicate a router for each service Micro-Container composing the SCA-based application. Hence, invocation between services Micro-Containers is done via routers. Thus, the router task is to manage the different copies of its attached service Micro-Containers. To do so, we provided the router with a routing table which contains the running copies. The routing table is dynamically updated whenever an elasticity action is executed.
– Controller Module: In order to manage the duplication/consolidation mechanisms, we develop a new generic module, called "Controller Module". Thus, this module generates the elasticity controller which is responsible of analyzing work-flow data of the SCA-based application and performing elasticity actions (duplication/consolidation). At this point, we explain how we provide services Micro-Containers with elasticity modules that ensure the execution of elasticity actions (Duplication/Consolidation) whenever a duplication or consolidation condition is verified i.e., a component is overloaded or underloaded. On one hand, if a component is overloaded and the duplication condition verified, the controller deploys a new copy of the concerned component. Consequently, the component provides its correspondent router with its coordinates in order to inform him that he is active. Thus, the router updates its routing table by adding this new entry (Figure 15-(A)). On the other hand, when a component is underloaded and the consolidation condition verified, the controller proceeds to execute the consolidation action. Indeed, it communicates with the related router in order to update its routing table by removing the unnecessary entry. Then, the controller removes the concerned copy of service (Figure 15-(B)).

As shown previously, our packaging framework generates a set of Micro-Containers and an elasticity controller container. The generated chain execution

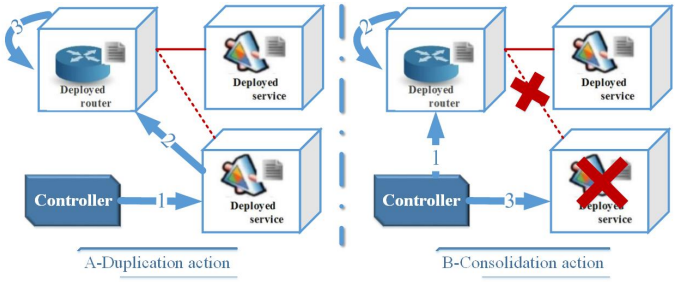

Fig. 15: The Duplication and Consolidation Controller Actions

is equivalent to the execution of the original SCA-based application and it leads to the same expected results. Thus, our deployment approach preserves the semantic of the SCA-based application, since no modifications were made on the functionnalities of the components. Hence, the elasticity operations executed by the controller do not violate the semantics of the concerned SCA-based application, because adding a new copy of a service consists in instantiating other Micro-Container of the same packaged service and removing an unnecessary copy means releasing its dedicated resources.

We use the response time as a metric for our elasticity mechanisms. Indeed, we choose to define for each component of the SCA-based application two threshold response time values namely tmax and tmin. The threshold tmax define a lowest acceptable response time for the overall application. The second one define a maximal acceptable response time for SAC-based application. The reason behind using these two thresholds is to ensure a certain QoS level and to optimally manage the used resources. In our use case, if the response time is over the maximum threshold, a duplication action is triggered (i.e. adding a new copy of the concerned service). In case that the response time goes lower than the defined minimum response time, a consolidation action is executed (i.e. removing a service copy).

4.4 Evaluation

In this section, we evaluate the efficiency of our approach. To achieve this goal, we compare the behavior of the SCA-based application before and after adding our elasticity mechanisms. Regarding the calls arrival scenario, we launch several simultaneous clients where each one sends sequentially a number of requests.

Experimental setting. The experimentation was performed on a computer with Pentium 4, 2.8 GHz, 4 GB of RAM, Windows 8 professional edition.

In this section, we present an example of an online computer shopping application based on SCA composition. This example will be carried on along this paper. As depicted in figure 1, the online computer shopping composite is composed of four components:

As case study, we consider the online computer shopping application. The architecture of the online computer shopping application is shown in figure 1. The online computer shopping application is elastic and its correctness is validated using our approach. It is deployed in a cloud environment using our deployment framework. For this particular application, we set the value of tmin and tmax to 800 and 8000 seconds respectively.

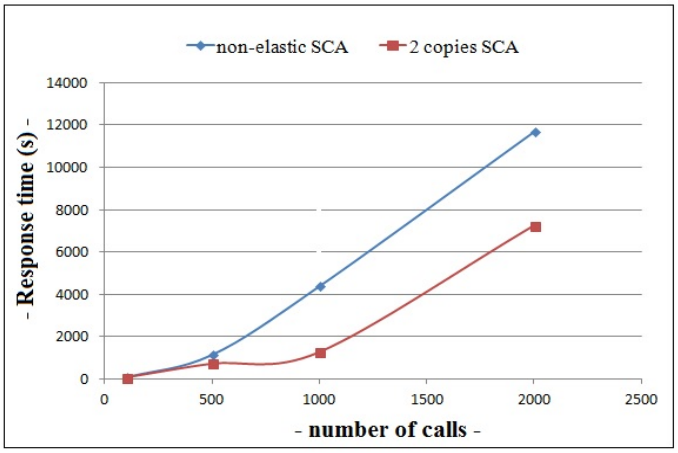

Fig. 16: Response Time of Non-elastic SCA-based Application and of Two Instantiated Copies

Experimental result. Our first deployment scenario is simply running the non-elastic version of our deployed online computer shopping application. The obtained experimental results show that the response time of the non-elastic SCA-based application is increasing in analogy with the number of clients calls, as shown on the figure 16.

The second curve on the same figure shows the result of our second experiment scenario. In this last one, we instantiate from the start two copy of our SCA-based application. In this case, we can see that the response time of the SCA-based application has significantly decreased and almost been linear in the interval [500,1000] of clients calls.

Also, we note that after 1000 calls, the curve started to increase exponentially, while always maintaining a lower response time than of the non-elastic SCA-based application. This can be understood and explained by the fact that the number of copies (two copies in this scenario) is no longer sufficient to treat the incoming calls in a linear manner. However, this result reassures as that our elasticity mechanisms are working and giving better results.

5 Conclusion

In this paper, we proposed an approach for verifying and deploying the elastic SCA-based application. To perform an elastic SCA composition, we formally modeled and verified the elasticity mechanisms, i.e. Duplication event and consolidation event, using the Event-B method. The proposed Event-B model allows designers to check the correctness of the associated application. Our approach is implemented as an Eclipse plug-in, called Elastic SCA2B and available in [30], together with the SCA designer plug-in. Once the elastic SCA-based applications are validated, they can be deployed in a cloud environment using an elastic SCA deployment framework. Figure 17 depicts a screenshot on the application of the plug-in on the case study. The tool creates a Event-B project containing mainly the specification described in Section 3 where the initialization of the first machine contains the structure of the case study with the different elements (components, references, services, etc.) are defined as constants in the contexts.

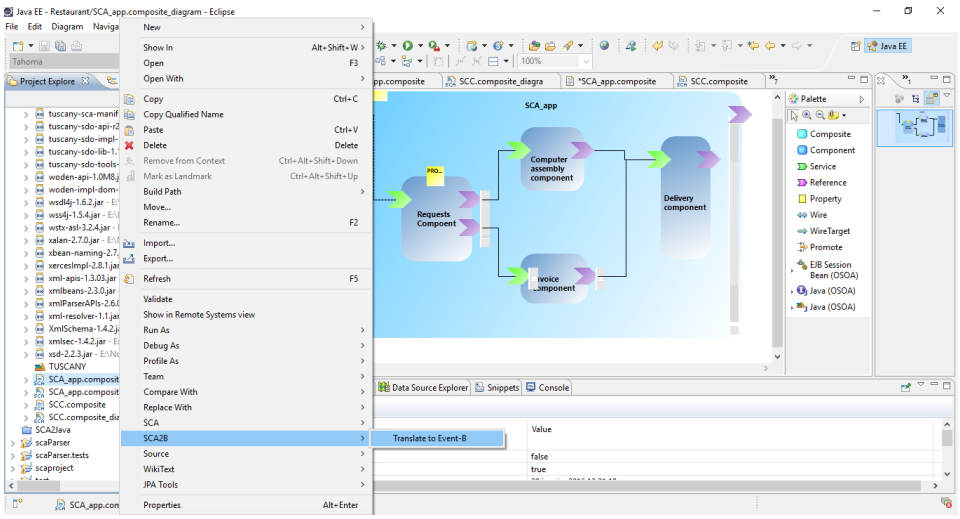

Fig. 17: A SCA2EventB plug-in for the verification of SCA applications

As perspectives of this work, we plan on going further in the evaluation phase for our deployed elasticity mechanisms. We believe that future and more exhaustive experiments will allow us to explore in details the benefits of using elasticity mechanisms for SCA-based application.

Another axis on which other future work may be conducted is the extension of our modeling approach. Indeed, more details (refining the condition of the execution of the elasticity mechanisms) can be added in the XSD File of the

service-component architecture. These information can help refining the condition of the execution of the elasticity mechanisms and can give an idea about the consumption of the resources and the execution time.

References

1. Butler M. Leuschel M. Prob: A model checker for b. *In K. Araki, S. Gnesi, D. Mandrioli (eds)FME 2003: Formal Methods*, LNCS 2805, Springer-Verlag:855–874, 2003.
2. Lu cheng Qi zhang and Raouf Boutaba. Cloud computing: state of the art and research challenges. In *Journal Internet Serv Appl (2010)*, pages 7–18. USENIX, 2010.
3. Samuel Kounev Nikolas Roman Herbst and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX.
4. G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *IEEE International Conference on Utility and Cloud Computing (UCC)*, pages 263–270, 2012.
5. Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52, January 2011.
6. Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *ICEBE*, 2009.
7. Sijin He, Li Guo, Yike Guo, Chao Wu, Moustafa Ghanem, and Rui Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *AINA*, Los Alamitos, CA, USA, 2012. IEEE Computer Society.
8. Rodrigo N. Calheiros, Christian Vecchiola, Dileban Karunamoorthy, and Rajkumar Buyya. The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds. *Future Gener. Comput. Syst.*, 28(6):861–870, June 2012.
9. Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE CLOUD*, Los Alamitos, CA, USA, 2011. IEEE Computer Society.
10. S. Dutta, S. Gera, Akshat Verma, and B. Viswanathan. Smartscale: Automatic application scaling in enterprise clouds. In *IEEE CLOUD*, pages 221–228, 2012.
11. Stefan Schulte, Philipp Hoenisch, Srikumar Venugopal, and Schahram Dustdar. Introducing the vienna platform for elastic processes. In *ICSOC Workshops*, pages 179–190, 2013.
12. Philipp Hoenisch, Stefan Schulte, Schahram Dustdar, and Srikumar Venugopal. Self-adaptive resource allocation for elastic process execution. In *IEEE CLOUD*, pages 220–227, 2013.
13. Philipp Hoenisch, Stefan Schulte, and Schahram Dustdar. Workflow scheduling and resource allocation for cloud-based execution of elastic processes. In *SOCA*, pages 1–8, 2013.
14. Pavlos Kranas, Vasilios Anagnostopoulos, Andreas Menychtas, and Theodora A. Varvarigou. ElaaS: An Innovative Elasticity as a Service Framework for Dynamic Management across the Cloud Stack Layers. In *CISIS*.

15. Lam-Son Lê, Hong Linh Truong, Aditya Ghose, and Schahram Dustdar. On elasticity and constrainedness of business services provisioning. In *IEEE SCC*, pages 384–391, 2012.
16. Kais Klai and Samir Tata. Formal modeling of elastic service-based business processes. In *IEEE SCC*, pages 424–431, 2013.
17. Wei-Tek Tsai, Xin Sun, Qihong Shao, and Guanqiu Qi. Two-tier multi-tenancy scaling and load balancing. In *ICEBE*, pages 484–489, 2010.
18. A. Cau M. Solanki and H. Zedan. Asdl : A wide spectrum language for designing web services. *15th International World Wide Web Conference (WWW2006)*, 2006.
19. R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing*, 4:67–95, 1986.
20. Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. *Fourteenth Australasian Database Conference (ADC2003)*, 17of CRPIT:191–200, 2003.
21. J.R Abrial. *Modeling in Event-B: System and Software Engineering,*. cambridge edn,Cambridge University Press, 2010.
22. Zuohua Ding, Mingyue Jiang, and Jing Liu. Model checking service component composition by spin. In *ACIS-ICIS*, pages 1029–1034, 2009.
23. José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A formal approach to service component architecture. In *WS-FM*, pages 193–213, 2006.
24. Soumaya Louhichi, Mohamed Graiet, Mourad Kmimech, Mohamed Tahar Bhiri, Walid Gaaloul, and Eric Cariou. Mde approach for the generation and verification of sca model. In *iiWAS*, pages 317–320, 2011.
25. Elvinia Riccobene and Patrizia Scandurra. A formal framework for service modeling and prototyping. *Formal Asp. Comput.*, 26(6):1077–1113, 2014.
26. David Chappell and associates. *Introduction SCA*. 2007.
27. Sanjay Patil Michael Beisiegel, Anish Karmarkar and Michael Powley. *Service Component Architecture Assembly Model Specification Version 1.1*. 2011.
28. Amel Mammar and Marc Frappier. Proof-based verification approaches for dynamic properties: application to the information system domain. *Formal Asp. Comput.*, 27(2):335–374, 2015.
29. Walid Gaaloul, Sami Bhiri, and Mohsen Rouached. Event-based design and runtime verification of composite service transactional behavior. *IEEE T. Services Computing*, 3(1):32–45, 2010.
30. Amel Mammar Mohamed Graie, Lazhar Hamel and Samir Tata. A verification and deployment approach for elastic sca-based applications. `http://www-public.tem-tsp.eu/~mammar_a/FACElasticity.html`, 2016.
31. Aida Lahouij, Lazhar Hamel, and Mohamed Graiet. Formal verification of sca assembly model with event-b. In *Semantics, Knowledge and Grids (SKG), 2013 Ninth International Conference on*, pages 44–51, Oct 2013.
32. S. Yangui, M. Ben Nasrallah, and S. Tata. Paas-independent approach to provision appropriate cloud resources for sca-based applications deployment. In *Semantics, Knowledge and Grids (SKG), 2013 Ninth International Conference on*, pages 14–21, 2013.