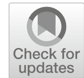
https://doi.org/10.1007/s00165-020-00517-0
BCS 2020
Formal Aspects of Computing (2020) 32: 361–393

Formal Aspects of Computing



An Event-B based approach for cloud composite services verification

Aida Lahouij[1], Lazhar Hamel[2], Mohamed Graiet[3] and Béchir el Ayeb[4]

[1]Université de Sousse, ISITCom, 4011 Hammam Sousse, Tunisia
[2]ISIMM, Monastir University, Monastir, Tunisia
[3]ENSAI RENNES, Bruz, France
[4]FSM, Monastir University, Monastir, Tunisia

Abstract. The verification of the Cloud composite services' correctness is challenging. In fact, multiple component services, derived from different Cloud providers with different service description languages and communication protocols, are involved in the composition which may raise incompatibility issues that in turn lead to a non-consistent composition. In this work, we propose a formal approach to model and verify Cloud composite services. Four verification levels are considered in this article; the structural, semantic, behavioral, and resource allocation levels. Therefore, we opted for the Event-B formal method that enables complex problems decomposition thanks to its refinement capabilities. The proposed approach has proven its efficiency for the modelling and verification of Cloud composite services. The proposed model comprises four abstract levels with respect to the four verification axes. A proof-based approach is applied to the model's verification. We also succeeded in the validation of the model thanks to the model animation provided by the PROB tool. The use of formal methods provides a rigorous reasoning and mathematical proofs on the correction of the model which ensures the elaboration of correct-by-construction composite services.

Keywords: Formal verification, Cloud composite services, Semantic verification, Behavioral verification, Resource allocation, Event-B

1. Introduction

In the last few years, there has been a growing interest in the verification of Cloud composite services especially with the emergence of the Cloud computing paradigm [FE10] and the additional challenges it brings. Service composition alone, without considering a Cloud environment, raises the need for design-time verification approaches to check the correctness of the interactions between the different components of a composite service. Indeed, the verification of the structural, semantic and behavioral matching of the component services involved in the composition is necessary in order to avoid the inconsistencies in the composite service and any possible deadlock situation that may occur during its execution. This task is not trivial especially with the deployment of the composite services in the Cloud, which adds the necessity to verify the Cloud resources allocated to the component services.

Correspondence to: Aida Lahouij, e-mail: aida.lahouij@gmail.com

In fact, the Cloud environment provides three types of resources: computing, networking, and storage. The resource's elasticity and scalability are key features of the Cloud computing paradigm. A resource is elastic if it can change its capacity at runtime. Elasticity is the ability to dynamically increase or decrease infrastructure resources as needed to adapt to workload changes in an autonomic manner, maximizing the use of resources. On the other hand, scalability handles the changing needs of an application within the confines of the infrastructure by statically adding or removing resources to meet the application's demands if needed. A resource is shareable if it can be allocated to many activities' instances. A multi-tenant resource can be shared between the involved component services provided that its capacity is able to handle the component service' requests at the same time.

For several years, great efforts have been devoted to the verification of web services and more recently, attention has shifted toward services deployed in the Cloud environment. Some verification approaches based on formal languages have been proposed (e.g., Event-B [GMBG17], SOG [BKS+17, KTD11], and Process Algebra [PF11], etc). Despite their efficiency when dealing with a particular verification problem, a key limitation of such research approaches is that they address each problem separately. In other words, none of them has proposed a full verification approach of the composite service. In this study, a new verification approach is suggested in order to cover several sides of a Cloud composite service. The proposed approach must guarantee that:

- The composite service provides the required functional level.
- The component services semantically match with respect to their inputs, outputs, preconditions, and effects. This is related to data type compatibility between the linked components where the output of service should be of the same type as the input of another resource.
- The component services protocols are compatible. This is related to the order of messages and the deadlock freeness of the composition i.e., the reachability of the desired final composition state from the initial state.
- The Cloud resources are correctly allocated to the component services in order to avoid any deadlock situation when accessing the resources. The resources' elasticity and shareability must be considered to perform this verification.

As we can notice, the verification requirements are complex and divided into many levels. Therefore, we opt for the Event-B [AM98] formal method that facilitates the modelling of complex systems through its refinement capabilities. Formal verification is necessary in order to avoid incorrect composition behavior and unnecessary executions of erroneous compositions. With first-order logic and set theory as underlying mathematical notations, the Event-B method allows us to specify and model software systems in a mathematically sound way. The use of formal methods is nowadays a necessity to create reliable software for critical and complex systems. In [LHGM18], we have introduced a part of this model where we verify the composite service behavior and resource allocation at runtime. The purpose of this paper is to extend the previous work by presenting a global verification approach that deals with all verification levels at both design and runtime. The objective is to verify the composite service's correctness on several levels at both design time and runtime to avoid any possible execution problems.

The remainder of this article is organized as follows; in the following section, we give a summary of existing related works. In Sect. 3, we present our motivations. In Sect. 4, we go over the main requirements considered in this paper. An overview of the Event-B method is given in Sect. 5. Our formal model is introduced in Sect. 6. The verification and validation approach is detailed in Sect. 7. Finally, we conclude and provide insights for future works in Sect. 8.

2. Related works

For several years, great efforts have been devoted to the study of the verification of composite services. In this section, we discuss some of the proposed approaches and compare them to ours.

2.1. Semantic verification approaches

Many studies have been conducted on the semantic verification of services. For instance, authors in [SMWZ15], propose a unified semantic Cloud Service Description Model (CSDM) extending the basic struc-

ture of the Unified Service Description Language (USDL) and defining cloud-service-specific attributes. The CSDM integrates a module design of service integration and composition to support both syntactic and semantic expressions of the service description, as well as to enable a variety of operations such as discovery and match-making. The authors of this work considered an informal description of the model supported by UML diagrams making the verification and validation of such a representation awkward. Besides, no mechanism of matching was evoked in this work. Formal representation of the CSDM is required for the verification and validation of the proposed model. In [ZGOH09], the authors propose a matching algorithm called SMA for automatic semantic web service composition. The service matching, in this algorithm, consists of matching the output parameters of a service operation to the input parameters of an operation of another service. Authors measure the matching degree $Math(P1, P2)$ between the different service operators op1 and op2 by calculating the semantic similarity between a concept set $P1$ in the output parameters of the operator op1 and a concept set $P2$ in the input parameters of the operator op2. Like this approach, the aim of this work is to propose a matching technique in order to verify the semantic compatibility between the component services.

Authors in [EMAZ15] introduce a new method called ComSDM to model the concept of service-oriented design in order to increase the reusability of the system and decrease its complexity while keeping the service composition considerations in mind. The ComSDM method provides a mathematical representation of the components of a service-oriented design using the graph-based theory to facilitate the design quality measurement. To demonstrate that the ComSDM method is also suitable for composite services, the authors implemented the case study of a smart home. The major drawback of this approach is the lack of matching techniques adding to the absence of model verification and validation. The previously stated studies on the semantic verification of composite services present certain weaknesses. Commonly, they are unsatisfactory because they don't give any proof on the model's reliability and consistency. The model's verification and validation have been extensively performed in our work thanks to the Event-B tools and features. Additionally, these studies ignore the preconditions and effects of the execution of services which makes it difficult to guarantee a correct invocation and execution of services and their composition. In [WDJ+16], authors deal with automatic web service composition focusing on uncertain effects. To do so, they have used a graph-plan-based approach consisting in introducing branch structures into composite solutions to cope with uncertainty in the service composition process. However, conditions propagation is not considered in this work. Authors in [AM16] propose a pattern-based orchestration model that considers IOPE (inputs, outputs, preconditions, and effects). Condition's propagation is considered in this work. It does not, however, deal with the uncertain effects problem. Our approach presents more advantages, compared to the works stated above, by dealing with both the uncertain effects problem and conditions propagation.

2.2. Behavior verification approaches

The recent years have witnessed the publication of several research papers dealing with the verification of either services or composite services. For instance, different semi-formal languages and supporting tools have been proposed in this context. In [LHJ+14], a semantic-aware model checking (SAMC) approach is proposed to capture the simple semantic information of services. The key limitation of this approach, which is the major drawback of all semi-formal verification methods in general, is the lack of formal semantics. Therefore, in this work, we opt for a formal method to model and verify Cloud composite services. The benefits of using formal methods to ensure the correctness of systems are well-proven. Authors in [FW12], present an abstract formalization of federated Cloud workflows using the Z notation [WD96]. In [CHH+12], the authors used the Labeled Transition System Analyser (LTSA) to present a λ calculus model for analyzing and verifying the resources used in web service applications in cloud computing environments. The work in [AMJS11] presented a model called Event Condition Action (ECA). It uses the SPIN model checker [Hol97] in order to verify the service agreement property. In [BSS19], authors verify the transactional behavior in Business Process as a Service (BPaaS) configurations. A formal approach is used to verify that the selected

configurable features do not violate any constraints defined by the client. Although these works use a formal method to model and verify the Cloud's open issues, they do not verify the correctness properties considered in this paper.

In [AM16], authors seek to prove the soundness of composite services using a formal language for process description called OFL and which takes into account resources and services relationships at different Cloud Service Layers. In addition, they solve the problem of semantics-based matching and analysis of composite processes by means of Condition Propagation. This work, however, does not cover the behavior and deadlock freeness verification of composite services. Moreover, early research efforts often focus on non-functional behavior of the composite service in terms of response time and financial cost [LHG+16, AGGH15]. Non-functional properties verification is necessary to ensure an efficient composite service. However, it is not sufficient. Proofs on the reliability of the composition are needed. In this paper, we are interested in checking the functional behavior's soundness with respect to relevant correctness properties.

2.3. Resource verification approaches

The work in [MKS13] uses High-Level Petri Nets (HLPN) to analyze and model the structural and behavioral properties of three open-source VM-based cloud management platforms: Open Nebula, Eucalyptus and Nimbus. Recently, the work in [NSG+14] has also used Markov Decision Processes (MDP) for the cloud elasticity modelling and then used the PRISM model checker in order to model and verify several elasticity decision policies that aim to maximize user-defined utility functions. In [KMCB17], authors model and verify the behavior of resources and their composition using Colored Petri Nets (CPNs). Although the proposed model considers composition behavior properties, it presents several limitations when the composition involves a large number of services. Model-checking suffers from the state-space explosion problem that makes the exhaustive verification very difficult for large and complex systems. In addition, it is computationally expensive to cover all the state space of the system's model. In fact, abstraction is a powerful technique that enables fitting big systems into model checkers, yet, it has not been well explored for the modelling and verification of Cloud systems. In [HSDV13, HSD13, BYO+14], the authors focused on workflow scheduling strategies and resource allocation algorithms using Cloud resources in an optimal way. They consider the elasticity property of processes. They used Cloud-based resources on the PaaS level which are computing resources (VMs) for the execution of business processes. In our work, we propose a verification approach of resource allocation that seeks to check Cloud resource properties (elasticity and sheareability) and to avoid deadlocks. Moreover, early research efforts often focus on non-functional behavior of the resource allocation in terms of response time and financial cost [BKKM11, CWG09, JCM19, MFBR15]. In this paper, however, we are interested in checking the functional behavior soundness with respect to Cloud properties.

Our work is inspired by the work in [GMBG17], where authors provide a formal description of the resource perspective in the existing business processes in order to ensure the correctness of Cloud resources allocation in business process modelling. The work uses the Event-B method to ensure the correctness of Cloud resources allocation. Nevertheless, it does not consider Cloud services behavior. [GHMT17] proposes an approach for the verification and deployment of elastic component-based applications based on the Event-B formal method. The work consists of informally modelling the component artefacts and the elasticity mechanisms (scaling up and down) for component-based applications. This work, however, does not consider the checking of the functional behaviour's soundness.

2.4. Discussion

The existent works generally focus only on one side of the verification. We find works that were interested only on the semantic verification [SMWZ15, ZGOH09, EMAZ15, WDJ+16, AM16]. Even when examining these works, we find those who have neglected the precondition and effects ([SMWZ15, ZGOH09, EMAZ15]) in their semantic verification despite their increased importance in establishing semantic reliability especially in the last few years. Other works [EMAZ15, WDJ+16, AM16] does not afford any matching technique to check the semantic matching among service. In the literature, we find also works that deal only with the behavior requirements ([LHJ+14, FW12, CHH+12, AMJS11, Hol97]). Others were interested only in the resource

allocation problem ([MKS13, NSG+14, KMCB17, HSDV13, HSD13, BYO+14, GMBG17, GHMT17]). Compared to these works, our work combines the modelling and the verification of the three requirements in the same model: Semantic, Behavior, and Resource requirements in Cloud composite services applications. Likewise, works that focus only on the non-functional behavior of the resource allocation [BKKM11, CWG09, JCM19, MFBR15] may guarantee an efficient set of resources but not a reliable one.

Adding to verification requirements, in the literature, we find a variety of verification methods. We distinguish semi-formal and formal languages. Semi-formal formal verification techniques [EMAZ15, WDJ+16, LHJ+14], in general, lack of formal semantics which makes the modelling and verification of reliable systems a tedious task. Among formal methods, we can find those who suffer from state-space explosion problems such as [FW12, MKS13, NSG+14, KMCB17]. In this work, our aim is to overcome the aforementioned verification limits by introducing a new approach, that takes advantage of the Event-B modelling method which enables the modelling of systems by means of abstraction techniques. The important point, in our work, is the incremental development of models for Cloud composite services verification. The use of the Event-B method permits, thanks to a refinement technique, to master the complexity of a system by introducing its details step by step. Moreover, the obtained model can be validated using the ProB animator/model-checker that help us detect and fix some errors before the proof activity that can be long and tedious.

In this paper, while we refer to our earlier work [LHG18], [GLA+15], [GHMT17], and [LHG15], the focus is different. The work proposed in this paper is new in the sense that it (1) *proposes a formal verification of the semantic properties of a composite service*, (2) *proposes a formal verification of the behavior properties of a composite service*, (3) *tackles the problem of the Cloud resource allocation*, (4) *is based on a formal model* and (5) *does not suffer from the problem of state-space explosion.*

3. Motivations and problem statement

Let us consider the required composite service (the travel booking composite service) that must provide the following functions:

1. *Webform display* (wfr): enables customers to enter the travel informations.
2. *Flight searching* (fs): enables customers to search for the requested flight.
3. *Hotel searching* (hs): enables customers to search for a hotel.
4. *Booking* (bg): used to book the flight and the hotel.
5. *Payment* (cp): after booking the flight and the hotel, the user proceeds to the payment.
6. *The reservation confirmation* (rc): a reservation confirmation is sent to the customer.

The required composite service takes as input the departure date, the return date, the departure city, and the arrival city and produces as outputs the airline, the flight number, the seat number, and the hotel. We use Business Process Model and Notation (BPMN) to describe our Travel booking composite service (see Fig. 1). It is constituted of the following component services:

- *Webform request* (WFR): a webform is used to enter travel information such as the departure date, the return date, the departure city, and the arrival city,
- *Search Flights* (SF): to search for the available flights at the required dates,
- *Search Hotels* (SH): to search for the available hotels at the required dates,
- *Booking* (BG): used to book the selected flight and hotel,
- *Credit card payment* (CP): the payment service for the booked flight and hotel,
- *The reservation confirmation* (RC): sends an e-mail to confirm the reservation.

To run the aforementioned component services, Cloud resources are required. First, the execution of the component service *Web form request* needs to communicate its inputs via a virtual networking Cloud resource (*Network1*). The component services *Search Flights* and *Search Hotels* share the virtual machine (*compute1*) with 4 GB of RAM. Moreover, they respectively need the storage resources *Store1* and *Store2*. The component service *Booking* stores its data in the storage resource Store3 which has a capacity of 5 GB of available storage size. The execution of the *Credit card payment* service is performed in the virtual machine (*compute2*) with 2 GB of RAM. Finally, the component service *Reservation confirmation* needs to communicate its outputs via a virtual networking Cloud resource (*Network2*).

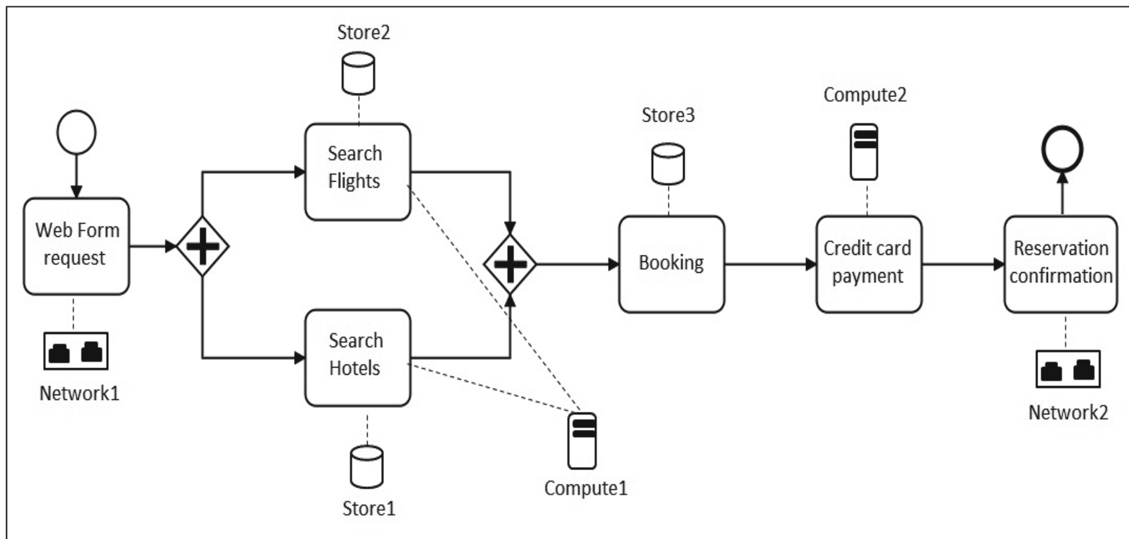


Fig. 1. The travel booking composite service

However, building the composition properly and ensuring its correct behavior is a difficult task. In fact, several problems may occur when building and/or executing the composition:

- **Non-interoperability:** Links between the output of a component service and the input of another may be invalid. This is due to the difference in data types that each service handles. For example, the component service *booking* only processes an array of values whose data type is a string. If it receives values from the previous services with a different data type, the composition will be erroneous. a data type mismatch causes interoperability issues between services.
- **Looping:** Starting the composition by invoking the first component service, the process may not reach the final expected result. This can be due to an end-loop occurring at a certain stage during the composition's execution, such as a loop in the execution of the preprocessing resource that can prevent the next resources from running.
- **Dead service:** A component service, such as *Credit card payment*, may not respond due to some technical problems, and thus the next related service *Reservation confirmation* involved in the composition process will not be invoked.
- **Resource conflict:** When dealing with parallel services such as *Search Flights* and *Search Hotels*, conflict in their execution may occur (deadlock). An exclusive shareable resource (*Compute1*) cannot be consumed by more than one component service at the same time. For instance, the composite service may end up in a deadlock situation, since *Compute1* is allocated simultaneously to *Search Flights* and *Search Hotels*.
- **Resource capacity:** A non-elastic resource, for example, *Store3*, cannot be allocated by the component service *Booking* while its available capacity (5 GB) does not fit the needed capacity (10 GB).

To overcome these problems, we propose an Event-B approach to model the behavior of Cloud component services and verify the correctness of their composition. Such a modelling language should cover the requirements listed below:

- **Semantic verification:** allows the checking of the composition's syntax and thus ensures better management of the links between the component services. It also involves the verification of the preconditions and effects of each component service. A matching technique is introduced in the following section.
- **Behavior verification:** In order to verify the correctness of the composition's behavior, the proposed approach should be able to verify the behavioral properties (Reachability, Liveness, and Persistence). This requirement is met via the analysis of the candidate service protocols to find possible mismatches.

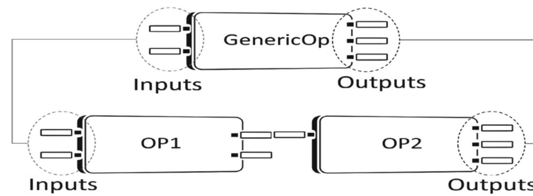


Fig. 2. Vertical matching

- **Resource allocation verification:** the proposed approach must check different Cloud resource properties (share-ability, elasticity, scalability)

In order to cope with the requirements mentioned above, we propose in this paper a formal approach based on the Event-B method to model Cloud services and their composition. With its formal syntax and semantics, the Event-B specification is able to validate the behavior of the built models through the execution of several verification properties.

4. Modelling requirements

In this section, we provide insight into the main concepts related to the verification of Cloud composite services. Modelling requirements are progressively extracted from the presented concepts.

4.1. Semantic requirements

In the following, we concentrate on analysing the semantic support from literature in order to give an overview of the basic semantic concepts, relations, and mappings schemes. Verifying the semantics of the component services is crucial in order to ensure the accuracy of the composite service. Therefore, in this work, we use a matching technique to verify the interoperability between the component services. This technique consists mainly of matching some of the outputs of a service to all of the inputs of the next service since it is assumed that a service cannot be executed if one of its inputs is not available. The service functionalities and interface information are given in terms of its inputs, outputs, preconditions, and effects (IOPEs) [Sub04]. Inputs and outputs denote the information transformation while preconditions and effects denote the changes produced by the service's execution.

Inputs and Outputs: In the following, we describe the possible matching cases:

- Vertical matching: in this matching scheme, we check whether a generic service *GenericS* (i.e composite service) can be substituted by a set of atomic services S1...Sn (component services). This matching consists in verifying whether the generic operation *GenericOp* of *GenericS* can be substituted by a set of atomic operations. A generic operation can be substituted by a sequence of operations only if the following requirements are met (Fig. 2):

Sem1. Op1 (the first operation in the sequence of operations) has the same inputs as the GenericOp.

Sem2. Opn (the last operation in the sequence) has a set of outputs that includes all of the outputs of the GenericOp.

For instance, in our motivating example, there must be an operation of the web form request service that has the same inputs as the Travel booking composite service (the departure date, the return date, the departure city, and the arrival city). And the Reservation confirmation service must have an operation that has a set of outputs that includes all of the outputs of the travel booking composite service (the airline, the flight number, the seat number, and the hotel).

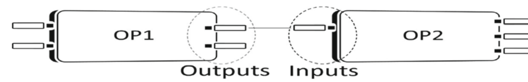


Fig. 3. Horizontal matching

- Horizontal matching: checks whether two services S1 and S2 can be linked together, which consists in checking whether their operations OP1 and OP2 can be coordinated together.

Sem3. OP1 and OP2 can coordinate only if the inputs of OP2 are included in the outputs set of OP1 (Fig. 3).

Back to our motivating example, the Booking service, for example, must provide the inputs of the Credit Card Payment service. The Booking service must have an operation that has a set of outputs (the price, the credit card number, the credit card balance, the credit card expiration date) that includes the inputs required by the operation of the Credit Card Payment service (the price, the credit card number, the credit card balance, the credit card expiration date).

Preconditions and Effects: Considering only the matching among the inputs and outputs is not enough when we are dealing with service composition [WDJZ14] because inputs and outputs only specify the data transformation produced by a service [Sub04], which contains the underlying functional knowledge of the service. In addition to these parameters, the actual execution of a service may still need to satisfy some prerequisites. The prerequisites and consequences, often known as the Preconditions and Effects of services, describe the execution conditions of a service and the changes resulting from its execution, thereby facilitating the automation of discovering and composing services. Such information was not well studied and fully utilized in the previous methods for automatic service composition. The preconditions and effects (PE) are defined by logic predicates and are difficult to manage. After the execution of a Cloud Service, some things may happen or change: these are the effects that the service produces. The execution of the Cloud services may change the truth value of some predicates: a running composite service may evaluate some predicates as true or false after the execution of any activity. Meeting the preconditions is assured by the previously executed services at any time in the composite process and condition values may change during its execution. This is an effect we call Condition Propagation and it provides a means to describe the semantics of the whole composite service.

For Condition Propagation, we assume that, in order to bind a real service to the workflow, it must meet all the preconditions and, after its execution, it must produce all the declared effects. The important point is to understand if a service can avoid meeting a precondition because of Condition Propagation. The preconditions of an activity depend on the effects of previously executed activities. The real problem is to understand if a composite service is correct in terms of IOPE matching. Therefore, we define the following requirements:

Sem4. The first component service must have the same preconditions as the request since a service cannot run unless its preconditions are satisfied.

Sem5. The composite service's effects must match the request's desired effects.

Sem6. Each component service's preconditions must not contradict the effects produced by previous component services.

4.2. Behavioral requirements

In this section, we explain how analyzing the services' protocols helps us to find out possible interoperability issues. In this work, the behavioral verification is performed at design time (i.e the discovery and the selection of component services) and at runtime. Therefore, we adopt the bidirectional compatibility (BC) notion introduced in [DOS12] to check the matching of the protocols. It is the most intuitive notion of compatibility. It requires that when one service can send a message, there is another service which eventually receives that message, and when one service is waiting to receive a message, then, there is another service which

must eventually send that message. Furthermore, the protocols must be deadlock-free. In order to interact properly, two services must be behaviorally compatible. Behavioral compatibility implies the compatibility of the protocols associated with each service. Two services have a behavioral mismatch if their protocols get stuck during their interactions. In fact, the component services communicate according to a business protocol that specifies the order in which the messages are exchanged. However, two component services can get stuck during their interaction due to behavioral constraints in their protocols. Behavioral mismatches are identified in the protocols by analyzing the ordering constraints of the exchanged messages and the existence of deadlocks. The following behavioral requirements are considered in this work:

Beh1. For each component service, when its protocol is waiting to receive a message there must be another component service that will send that message.

Beh2. For each component service, when its protocol is going to send a message there must be another component service that will receive that message.

Beh3. The receiver's Queue must be able to receive the message.

Beh4. The composite service must be deadlock-free (i.e starting from an initial state, we can reach the desirable final state).

4.3. Resource requirements

To execute a composite service, Cloud resources are required. The Cloud environment provides three types of resources: computing, networking, and storage. Resources' elasticity and scalability are key features of the cloud computing paradigm. A resource is elastic if it can change its capacity at runtime. Elasticity is the ability to increase or decrease infrastructure resources dynamically as needed to adapt to workload changes in an autonomic manner. However, scalability handles the changing needs of an application within the confines of the infrastructure via statically adding or removing resources to meet the application's demands if needed. A resource is shareable if it can be allocated to many activities' instances. The capacity of a multi-tenant resource, shared between the N services of the composition, must be able to handle the N services requests. To manage the resource allocation process correctly, different requirements should be satisfied. For instance, we consider the following shareability requirements:

Sh1. A non-shareable resource cannot be allocated to more than one service.

Sh2. At most one service can be running on a non-shareable resource.

Sh3. A shareable resource can be allocated to more than one service.

Sh4. One or more services can be running on a commonly shareable resource.

Sh5. At most one service can be running on an exclusively shareable resource at the same time.

Ra1. The resource is of the same type as the required resource and,

Ra2. there is enough resource capacity to handle the service or,

Ra3. there is not enough resource capacity to handle the service however the resource is Elastic or,

Ra4. there is not enough resource capacity to handle the service however the resource is Scalable.

Ra5. The resource is shareable or,

Ra6. the resource is not shareable and it is not allocated to any other service.

The structure of Ra1 up to Ra6 could be expressed explicitly by the formula $Ra1 \wedge (Ra2 \vee Ra3 \vee Ra4) \wedge (Ra5 \vee Ra6)$. The above requirements are expressed in the Event-B language presented in the following section.

5. Overview of the Event-B method

The B-method was developed by Jean-Raymond Abrial [Abr88] and has been used in major safety-critical system applications in Europe such as the Paris Metro Line 14. It has a robust, commercially available tool support for specification, design, proof, and code generation. Event-B is an evolution of the B-method also called classical B [Abr05]. Event-B [AM98] reuses the set-theoretical and logical notations of the B method and provides new notations for expressing abstract systems or simply models based on events[CM08].

Fig. 4. Event-B constructs

Fig. 5. Event-B event and refinement event

Through sequential refinement, this formal method enables the incremental development of software step by step from the most abstract level to more detailed levels and possibly to the code level. The complexity of a system is mastered thanks to the refinement concept allowing to gradually introduce the different parts that constitute the system starting from an abstract model to a more concrete one. A stepwise refinement approach produces a correct specification by construction since we prove the different properties of the system at each step. Event-B is supported by the Eclipse-based RODIN platform [ABH+10] on which different external tools (e.g. provers, animators, model-checkers) can be plugged in order to animate/validate a formal development.

An Event-B specification is made of two elements: context and machine (Fig. 4). A context describes the static part of an Event-B specification. An Event-B context is optional and contains essentially the following clauses: the clause SETS that describes a set of abstract and enumerated types, the clause CONSTANTS that represents the constants of the model and the clause AXIOMS that contains all the properties of the constants and their types. A context can optionally extend another one by adding its name to the clause EXTENDS. A context is referenced by the machine in order to use its sets and constants by adding its name in the clause SEES. An Event-B machine describes the dynamic part of an Event-B specification. It is composed of a set of clauses organized as follow; the clause VARIABLES representing the state variables of the model, the clause INVARIANTS defining the invariant properties of the system which must allow, at least, the typing of the variables declared in the clause VARIABLES and finally the clause EVENTS containing the list of events related to the model. An event is modelled with a guarded substitution and fired when its guards are evaluated to true. The events occurring in an Event-B model affect the state described in the clause VARIABLES. A machine can optionally refine another one by adding its name in the clause REFINES. An event consists of a guard and a body (Fig. 5). When the guard is satisfied, the event can be activated. When the guards of several events are satisfied at the same time, the choice of the event to enable is deterministic. Refinement is a process of enriching or modifying a model in order to augment the functionality being modelled, or/and explain how some goals are achieved. Both Event-B elements, the context, and the machine can be refined. A context can be extended by defining new sets and/or constants along with new axioms. A machine is refined by adding new variables and/or replacing existing variables by new ones that are typed with an additional invariant. New events can also be introduced to implicitly refine a skip event. In this paper, refined events have the same form.

Proof-based development methods integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. We then gradually add details to this first model by building a sequence of more concrete ones. As such, an Event-B model is controlled by means of a number of proof obligations, which guarantee the correctness of the development. Different types of proof obligations exist in Event-B. Proof obligations types help to understand why each proof obligation is generated and to which part of the model the particular proof obligation is related. Note that the structure of a proof obligation name in Event-B is cause/proof-obligation-type. The cause could be an event, invariant or axiom. The proof-obligation-type is one of the predefined proof obligation types in Event-B.

- Well-Definedness (WD): partial functions are used in Event-B quite frequently. Using partial functions could result in reasoning about badly-defined expressions in proofs which can be difficult and tedious to work with [ABH+10]. Therefore it is necessary to prove that partial functions are applied only to arguments in their own domain. So WD proof obligations ensure that partial functions are never applied to arguments outside their set domain.

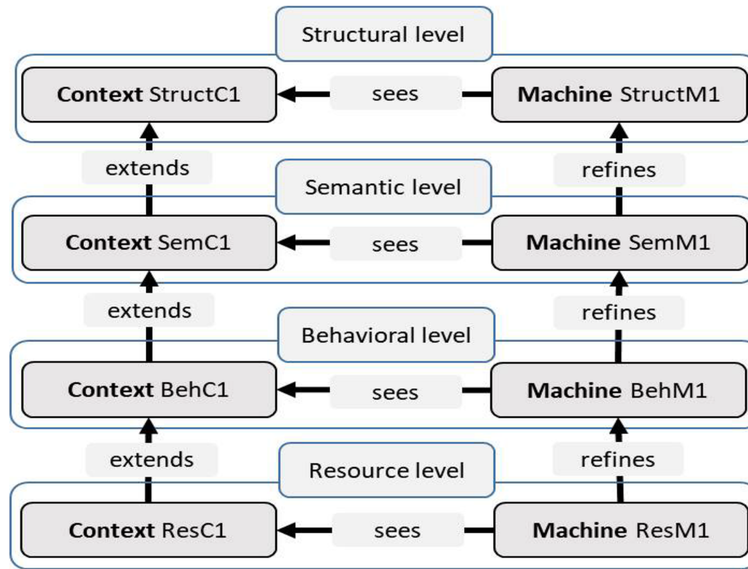


Fig. 6. CloudM specification

- Feasibility of non-deterministic events (FIS): Event-B requires actions to be feasible when their guards (preconditions) are true [ABH+10]. This means that an action must yield success if its preconditions hold. This is ensured by this type of proof obligations.
- Invariant Preservation (INV): ensures that the invariants hold over events. This means that it must be proved that invariants are always valid even after actions are applied.

Event-B includes other types of proof obligations for guards, witnesses, and theorems (derived axioms) as well, but they are not discussed here as they are not necessary for the understanding of the present work.

6. Modelling cloud composite services with Event-B

In this section, we detail our Event-B-based formal approach for the verification of Cloud composite services correctness. It covers the structural, semantic, behavioral and resource properties of a composite service.

6.1. The architecture of the Event-B specification

Figures 6 and 7 depict the formalization architecture of our Event-B model denoted by *CloudM* and the events' refinement. Our model's abstraction is provided in four levels:

- *StructM1* sees *StructC1* and models the structural properties of the Cloud composite service. This machine contains five events. The *SelectService* event to select the component services and the *RunComponent*, *RunComposite*, *Terminate* and *Time* events to run the selected component services.
- *SemM1* refines *StructM1* and introduces the semantic properties. The *SelectService* event is refined in this machine to handle the semantic requirements. The *RunComponent*, *RunComposite* and *Terminate* events are also refined in this machine to verify the semantic requirements at run time.
- *BehM1* refines *SemM1* by adding the behavioral properties of the composition. This machine contains the second refinement of the *SelectService* event to check the services' protocols compatibility. The *RunComponent*, *RunComposite* and *Terminate* events are also refined to verify the composite service execution. Additional events are defined at this level; the *Non_Blocking_Send*, *Blocking_Send*, *Send_Ack* and *Consume* events that manage the messages exchange between running component services.

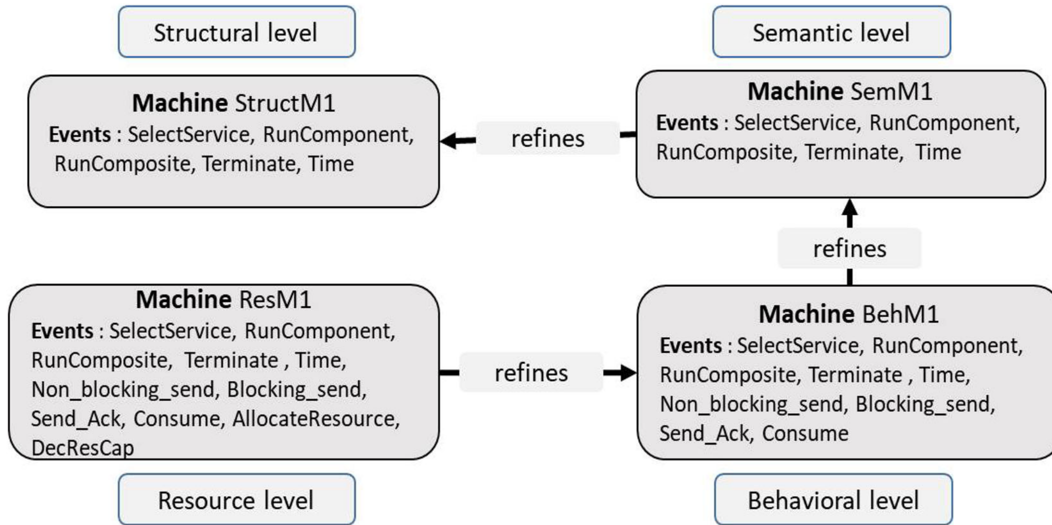


Fig. 7. The events' refinement

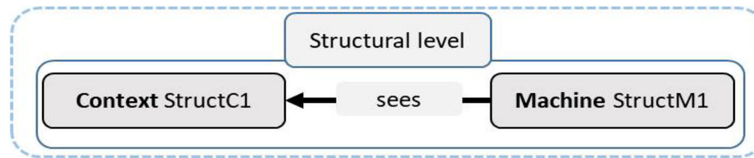


Fig. 8. The first abstraction level

- *ResM1* refines *BehM1*, where details about the resource allocation are added. At this level, this machine sees the context *ResC1*. Two events are introduced in this machine; the *AllocateResources* and *DecResCap* events in order to manage the resources allocation.

Parts of this model have been introduced in [LHGM18] and extended in this article. In this previous work, we have performed the verification of the composite service's behavior and resource allocation only at runtime. In this paper, we extend the previous verification levels by performing an additional verification at the candidate services selection time. We have also presented the structural and semantic verification at both design time and runtime. In the meanwhile, we have also made some rectifications to the model presented in [LHGM18]. The result is a complete model that covers all the verification sides and avoids the execution of erroneous service compositions. In the following, we present the formalization of our modelling requirements for each abstraction level.

6.2. Modelling the structural requirements

At this level, we model the structure of the composite service. It consists on modelling the elements constituting a composite service as well as an atomic service. For instance, it is axiomatic to define that a composite service is made of a non empty set of services, that a service must provide a business function, therefore, it must have at least one operation, etc. At this level, we have also to select the services constituting the composite service with regard to the required functionalities. Namely, the selected services must provide the set of required functions. Therefore, we have defined the *SelectService* event. This abstraction level must also guarantee that these functions are executed in the required order.

CONTEXT StructC1
SETS

Services Functions Operations Requests State

CONSTANTS

FctOfOp OpOfS ExeT

AXIOMS

axm1: $finite(Services) \wedge finite(State) \wedge finite(Functions) \wedge finite(Operations) \wedge finite(Requests)$
axm2: $FctOfOp \in Operations \rightarrow Functions$
axm3: $OpOfS \in Services \twoheadrightarrow \mathbb{P}_1(Operations)$
axm4: $ExeT \in Services \twoheadrightarrow \mathbb{N}_1$
axm5: $rankf \in Functions \rightarrow \mathbb{N}$

END

Fig. 9. The *StrcutC*1 context description

MACHINE StructM1
SEES StructC1
VARIABLES

Composites Components SerOf St ReqOf FctOfReq T t RunningS

INVARIANTS

inv1: $Composites \subset Services$
inv2: $Components \subseteq Services$
inv3: $partition(Services, Composites, Components)$

inv4: $SerOf \in Composites \twoheadrightarrow \mathbb{P}(Components)$
inv5: $St \in Services \rightarrow State$
inv6: $ReqOf \in Composites \twoheadrightarrow Requests$
inv7: $FctOfReq \in Requests \twoheadrightarrow \mathbb{P}(Functions)$
inv8: $T \in \mathbb{N}$
inv9: $t \in \mathbb{N}$
inv10: $RunningS \subseteq Services$

END

Fig. 10. The *StrcutM*1 machine description

This is ensured by the *RunComposite*, the *RunComponent* and the *Terminate* events. Any disorder in the execution of the services my lead to a deviation from the required business function.

In Event-B, this first abstraction level is constituted of the Event-B context *StructC*1 and the machine *StructM*1 (Fig. 8). The elements that describe a composite service are defined thanks to the Event-B sets, constants and variables. Relations between these elements are given in the Event-B axioms and invariants. The selection, at this level, is limited to the service's functions. In other words, for each function required by the composite service, a service providing this function is selected. In the *StructC*1 context, we have introduced the enumerated sets *Services*, *Functions*, *Operations*, *Requests*, and *State* in order to model the structure of a composite service (Fig. 9). The set *Services* models the set of Cloud services (composite services and component services). The set *Operations* models the set of operations provided by the services set. *Functions* is the set of available functions. The set *Requests* is used to type the composite services' requests. The set *State* models the set of states that a service goes through during its lifecycle. In the axioms clause, we have typed the following constants:

- The axiom *axm1* is added to specify that all the defined sets are finite.
- The constant *FctOfOp* denotes the function provided by each operation (*axm2*). It is modelled by a total function since each operation must provide at least one function.
- *OpOfS* is a constant that denotes the set of operations provided by each service (*axm3*).
- We have associated an execution time *ExecT* to each service in *axm4*.

```
Event SelectService ⟨ordinary⟩ ≙
 any
      s c f op
 where
      grd1: s ∈ Components ∧ St(s) = deployed
      grd2: c ∈ Composites ∧ c ∈ dom(ReqOf) ∧ ReqOf(c) ∈ dom(FctOfReq) ∧ c ∈
         dom(SerOf)
      grd3: f ∈ FctOfReq(ReqOf(c)) ∧ (op ∈ OpOfS(s) ∧ s ∈ dom(OpOfS) ∧
         FctOfOp(op) = f)
 then
      act1: SerOf(c) := SerOf(c) ∪ {s}
      act2: St(s) := readyToInvoke
      act3: FctOfReq(ReqOf(c)) := FctOfReq(ReqOf(c)) \ {f}
 end
```

Fig. 11. The *SelectService* event description

```
Event RunComposite ⟨ordinary⟩ ≙
 any
      c
 where
      grd1: c ∈ Composites ∧ c ∈ dom(SerOf)
      grd2: St(c) = required
      grd3: c ∈ dom(ReqOf) ∧ ReqOf(c) ∈
         dom(FctOfReq) ∧ FctOfReq(ReqOf(c)) = ∅
 then
      act1: St(c) := running
 end
```

```
Event RunComponent ⟨ordinary⟩ ≙
 any
      s c
 where
      grd1: s ∈ Components ∧ c ∈ dom(SerOf)
      grd2: (s ∈ SerOf(c)) ∧ (c ↦ running ∈ St)
      grd3: s ↦ readyToInvoke ∈ St
 then
      act1: St(s) := running
      act2: RunningS := RunningS ∪ {s}
      act3: t := T
 end
```

Fig. 12. The *RunComposite* and *RunComponent* events description

- *rankf* (*axm5*) is a total function denoting the order of the required functions.

The above context is seen (clause SEES) by the machine *StructM*1 that defines some variables to model the dynamic properties of the structural model (Fig. 10).

- We have defined the variables *Composites* and *Components* to respectively model the composite services set and the component services set.
- We have then partitioned the *Services* set in the invariant *inv3* into *Composites* and *Components*.
- The component services of a composite service are denoted by the partial function *SerOf* (*inv4*).
- *St* denotes the state of a service (*inv5*). It changes during the service's life cycle.
- To each required composite service, we have associated a request denoted by the function *ReqOf* (*inv6*)
- and to each request a set of required functions denoted by *FctOfReq* (*inv7*).

In this work, we perform the verification at both design time and runtime. For design-time verification, we have defined the *SelectService* event in Fig. 11. This event is enabled to select the component services with respect to some functional, semantical and behavioral requirements. At this first abstraction level, we only verify that the selected services provide the set of functions required by the composite service. s is a deployed component service (*grd1*). The required composite service is denoted by c (*grd2*). The component services must provide the functions specified by the request (*grd3*). For each required function ($f \in FctOfReq(ReqOf(c))$), a service s is selected such that it provides an operation with the required functionality ($op \in OpOfS(s) \wedge s \in dom(OpOfS) \wedge\ FctOfOp(op) = f$).

It is important to check the execution of the component services at runtime to detect eventual errors. Therefore, we have defined the *RunComposite* event that triggers the composite service's execution (Fig. 12). A composite service execution runs only if all the required component services have been selected.

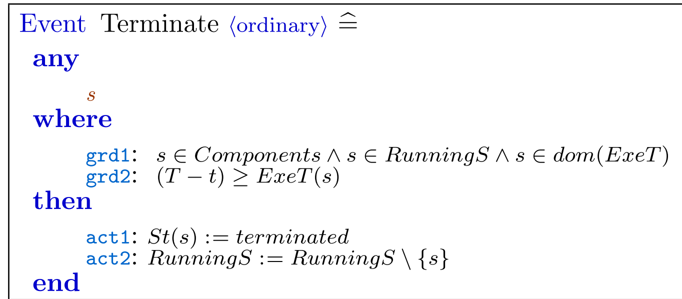


Fig. 13. The *Terminate* event description

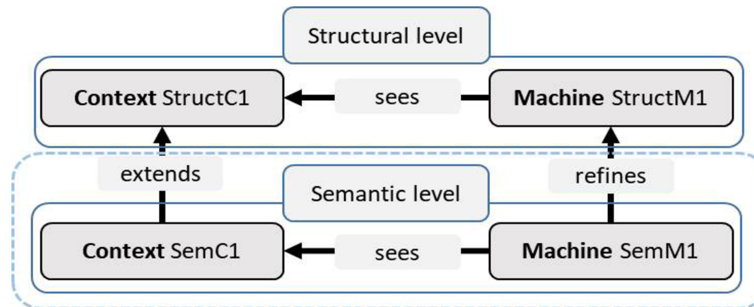


Fig. 14. The second abstraction level

This event has a parameter c that denotes a composite service ($c \in Composites$). The component services of the composite c must be already selected i.e $c \in dom(SerOf)$ (*grd1*) and $FctOfReq(ReqOf(c)){=}\varnothing$ (*grd3*). $FctOfReq(ReqOf(c)){=}\varnothing$ means that a service is already selected for all the required functions. In the previously introduced event *SelectService*, each time we select a service for a required function, this function is deleted from the variable *FctOfReq* representing the set of required functions for a given request. The state of the composite service changed from *required* (*grd2*) to *running* (*act1*).

After executing this event, the *RunComponent* event is enabled so that we can execute the component services of this composition (Fig. 12). To execute a component service, at this level, the composite service it belongs to must be running. This event has two parameters s and c that respectively denote the component service ($s \in Components$ (*grd1*)) to be executed and the composite service it belongs to ($s \in SerOf(c)$ (*grd2*)). The composite service c must be already running ($c \mapsto running \in St$ (*grd2*)) and the component service is ready to invoke ($s \mapsto readyToInvoke \in St$ (*grd3*)). After the execution of this event, the state of the component service is changed to *running* (*act1*) and it is added to the set of running services (*act2*). The execution start time is denoted by T (act3). The time is incremented in the *Time* event. Each time it is triggered, it increments the time T ($T = T + 1$). It can be triggered as much as needed. Once the accorded execution time is reached, the *Terminate* event (Fig. 13) is enabled to terminate the component service execution. The component service execution is terminated only if its execution time is completed. The event has only one parameter s that denotes the component service to terminate ($s \in Components$). The service's state must be running ($s \in RunningS$). In the guard *grd*2, the current running time ($T - t$) must exceed the execution time ($ExeT(s)$) estimated for the service s. The state of s is set to *terminated* (*act1*), and s is deleted from the set of running services (*act2*).

CONTEXT SemC1
EXTENDS StructC1
SETS

Parameters Types PE

CONSTANTS

In Out I O TypeOf

AXIOMS

axm1: $finite(Parameters) \wedge finite(Types) \wedge finite(PE)$
axm2: $In \subseteq Parameters$
axm3: $Out \subseteq Parameters$
axm4: $I \in Operations \rightarrow\!\!\!\!\!\!+ \mathbb{P}(In)$
axm5: $O \in Operations \rightarrow\!\!\!\!\!\!+ \mathbb{P}_1(Out)$
axm6: $TypeOf \in Parameters \rightarrow Types$

END

Fig. 15. The $SemC1$ context description

6.3. Modelling the semantic requirements

So far, we have modelled the basic concepts of a composite service. In this section, we focus on the modelling of the semantic features and requirements of a composite service. These features consist mainly on the parameters of the service's operations and their types and the preconditions and effects of the service's operations. The aim here is to find a matching between the inputs and outputs of the related component services and to check gradually the consistency of the operations' preconditions and effects. The verification of these semantic relations is performed at both design and run time. At design time, we have refined the *SelectService* event by introducing the semantic matching rules (see Sect. 4). For instance, the selected services must have the same inputs as the required composite service and must produce the same outputs as the required composite service. A service can not be executed without its required inputs. Therefore a component service is selected only if the set of inputs it requires are performed by the previously selected service(s). For the run time verification, we have refined the *RunComponent* event to ensure the semantic correctness during the component services execution. Namely, we have to ensure that each component service meets all the preconditions and produces the declared effects.

For this purpose, we extend the $StructC1$ context by the $SemC1$ context and refine the $StructM1$ machine by the $SemM1$ machine (Fig. 14). In the following, we start by representing the $SemC1$ context (Fig. 15). New sets are added to the model, namely *Parameters*, *Types*, and *PE* (preconditions and effects). This context elements are defined as follows:

- *Parameters* denote the set of an operation's parameters. It is divided into two sets *In* and *Out* (*axm2* and *axm3*).
- *I* in *axm4* refers to the inputs of an operation. It is a partial function from the set *Operation* to a power set of inputs *In*. This means that an operation may have zero or many inputs.
- *O* in *axm5* refers to the outputs of an operation, it is a partial function from the set of *Operation* to a non-empty power set of outputs *Out*. Which means that an operation must produce at least one output.
- *TypeOf* is a total function that assigns a type to each parameter (*axm6*).
- We have previously introduced services preconditions and effects that are modelled by the *PE* set.
- All sets are set to finite in the axiom *axm1*.

The context $SemC1$ is seen by the machine $SemM1$ that refines the $StructM1$ machine and models the dynamic part of the semantic model (Fig. 16). It is described as follow:

- The variables *ReqIn* and *ReqOut* models the request's inputs and required outputs, respectively (*inv1* and *inv2*).
- A rank is associated with each component service to represent its order of execution in the composite service (*inv3*). Parallel services have the same rank.

MACHINE SemM1
REFINES StructM1
SEES SemC2
VARIABLES

ReqIn ReqOut rank PR EFF precsR precsOp effsR effsOP AcEff

INVARIANTS

inv1: $ReqIn \in Requests \rightarrow \mathbb{P}(In)$
inv2: $ReqOut \in Requests \rightarrow \mathbb{P}(Out)$
inv3: $rank \in Components \rightarrow \mathbb{N}$
inv4: $PR \subseteq PE$
inv5: $EFF \subseteq PE$
inv6: $precsR \in Requests \nrightarrow \mathbb{P}(PR \rightarrow BOOL)$
inv7: $effsR \in Requests \nrightarrow \mathbb{P}(EFF \rightarrow BOOL)$
inv8: $precsOp \in Operations \nrightarrow \mathbb{P}(PR \rightarrow BOOL)$
inv9: $effsOP \in Operations \nrightarrow \mathbb{P}(EFF \rightarrow BOOL)$
inv10: $AcEff \subseteq (EFF \rightarrow BOOL)$

END

Fig. 16. The *SemM1* machine description

Event SelectService ⟨ordinary⟩ $\widehat{=}$
extends SelectService
where

...: ...
grd4: $finite(SerOf(c)) \land finite(FctOfReq(ReqOf(c)))$
grd5: $op \in dom(I)$
grd6:

$$(card(SerOf(c)) = 0 \land (\forall i \cdot i \in ReqIn(ReqOf(c)) \Rightarrow (\exists i' \cdot op \in OpOfS(s) \land s \in dom(OpOfS) \land op \in dom(I) \land i' \in I(op) \land TypeOf(i) = TypeOf(i')))) \quad (1)$$

$\lor$

$$(card(SerOf(c)) > 0 \land (\forall i \cdot i \in I(op) \Rightarrow (\exists o, op', s' \cdot s' \in SerOf(c) \land s' \in dom(OpOfS) \land op' \in OpOfS(s') \land op' \in dom(O) \land o \in O(op') \land TypeOf(i) = TypeOf(o)))) \quad (2)$$

$\lor$

$$(card(FctOfReq(ReqOf(c))) = 1 \land (\forall o \cdot o \in ReqOut(ReqOf(c)) \Rightarrow (\exists o' \cdot op \in OpOfS(s) \land s \in dom(OpOfS) \land op \in dom(O) \land o' \in O(op) \land TypeOf(o) = TypeOf(o')))) \quad (3)$$

then

...: ...
act4: $rank(s) := rank(f)$

end

Fig. 17. The first refinement of the *SelectService* event

- The PE set is comprised of preconditions and effects (*inv4* and *inv5*).
- *precsR* denotes the set of preconditions of the request (*inv6*). It is modelled by a partial function from the *Requests* set to the power set $\mathbb{P}(PR \rightarrow BOOL)$, which means that a request can have zero or many preconditions and each predicate is evaluated to TRUE or FALSE. Similarly, for the effects, a partial function *effsR* is defined in the invariant *inv*7.
- In the invariant *inv*8, *precsOp* defines the set of preconditions imposed by each service operation. The same holds for the effects of an operation, a partial function *effsOP* is defined in the invariant *inv*9.
- In the invariant *inv*10, *AcEff* denotes the set of effects accumulated during the composite service's execution. *AcEff* is used in order to model the conditions propagation since the preconditions of an action depend on the effects of the previously executed activities.

In the following, we refine the *SelectService* event (Fig. 17) by adding semantic constraints. The selection here is based on the semantic matching between component services.

```
Event RunComponent ⟨ordinary⟩ ≙
extends RunComponent
 any
     op
 where
     ...: ...
     grd4: s ∈ dom(OpOfS) ∧ op ∈ OpOfS(s) ∧ op ∈ dom(effsOP) ∧ op ∈ dom(precsOp) ∧ c ∈ dom(ReqOf) ∧
        ReqOf(c) ∈ dom(precsR) ∧ ReqOf(c) ∈ dom(effsR) ∧ s ∈ dom(rank) ∧ ((rank(s) = 1) ∨ (∀p·(p ∈
        Components) ∧ (p ∈ dom(rank)) ∧ (rank(p) = (rank(s) − 1)) ⇒ p ↦ terminated ∈ St))
     grd5: (rank(s) = 1 ∧ (∀i·i ∈ ReqIn(ReqOf(c)) ⇒ (∃i′·op ∈ dom(I) ∧ i′ ∈ I(op) ∧ TypeOf(i) =
        TypeOf(i′)))) ∨ (rank(s) > 1 ∧ op ∈ dom(I) ∧ (∀i·i ∈ I(op) ⇒ (∃s′, op′·s′ ∈ SerOf(c) ∧ s′ ∈
        dom(OpOfS) ∧ op′ ∈ OpOfS(s′) ∧ (s′ ∈ dom(rank)) ∧ op′ ∈ dom(O) ∧ (rank(s′) = (rank(s) − 1)) ∧ (∃o·(o ∈
        O(op′)) ∧ (TypeOf(i) = TypeOf(o))))))
     grd6:
        (rank(s) = 1 ∧ (∀p·p ∈ precsR(ReqOf(c)) ⇒ (p ∈ precsOp(op))) ∧ finite(precsR(ReqOf(c)))
        ∧ finite(precsOp(op)) ∧ card(precsR(ReqOf(c))) = card(precsOp(op)))                (4)
        ∨
        (rank(s) > 1 ∧ (∀p·p ∈ precsOp(op) ⇒ ¬(∃e·e ∈ AcEff ∧ dom(e) = dom(p)
        ∧ ran(e) ≠ ran(p))))                                                               (5)
        ∨
        (finite(SerOf(c)) ∧ rank(s) = card(SerOf(c)) ∧ (∀e·e ∈ effsR(ReqOf(c))
        ⇒ (e ∈ effsOP(op))))                                                               (6)
 then
     ...: ...
     act4: AcEff := AcEff ∪ effsOP(op)
 end
```

Fig. 18. The *RunComponent* event description

A component service is selected only if it semantically matches with other component services with respect to matching schemes previously introduced in Sect. 4.1. Namely, we have to verify that the selected component services have the same inputs and produce the same outputs as the required composite service and that each selected service inputs will be provided by the ancestor selected services. The three dots refer to the guards of the previous abstraction level. To model the Vertical matching 1:N (**Sem1**), we assume that the first service must have the same inputs as the required composite service since a service can not be executed without its required inputs (The formula (1) in *grd6*). For each input i of the request of the composite c, the candidate component service must have an operation op whose input i' has the same type as i. The formula (2) in *grd6* models the horizontal matching. It serves to check that the operations of the component services can be coordinated. Therefore, for the rest of the services, we assume that a component service is selected only if the set of inputs it requires are performed by the previously executed service(s) ($TypeOf(i){=}TypeOf(o)$) (**Sem3**). Therefore for each input i of the candidate service s, there must be a component service s' that has been previously selected $s' \in SerOf(c)$, $SerOf(c)$ is the set of component services of the composite service and has an operation op' whose output o is of the same type as of the input i. Namely, the component service must get its inputs from the previously executed component services. To finish modelling the Vertical matching 1:N, we have defined the formula (3) where we assume that the last component service must produce the required outputs (i.e the same type of outputs defined in the request) (**Sem2**). For each output o of the request, the candidate service s must have an operation op whose output o' is of the same type as o. In other words, the final component service must produce exactly the same outputs as the required composite service or more.

Then, we have proceeded to the composite service's verification at runtime. The semantic constraints must be preserved in order to ensure the semantic correctness of the composite service during its execution. Mainly the condition propagation, where each component service must meet all the preconditions and, after its execution, it must produce all the declared effects. Therefore, we have refined the *RunComponent* event (Fig. 18). A third parameter op denoting the service operation is added to this event's parameters.

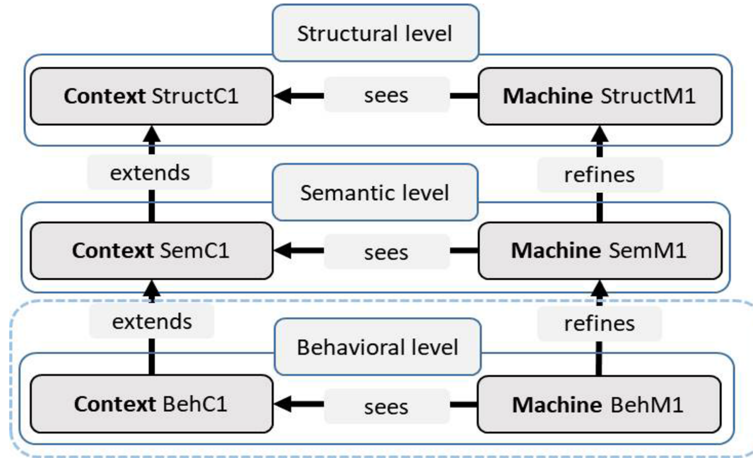


Fig. 19. The third abstraction level

To run the component service s, the execution of the previous component services must be terminated $(\forall p \cdot (p \in Components) \wedge (p \in dom(rank)) \wedge (rank(p)=(rank(s)-1)) \Rightarrow p \mapsto terminated \in St)))$ $(grd4)$. rank(p) is the rank of the component service p preceding the component service s. The service s is going to be executed therefore the previous component service p which rank is equal to $rank(s) - 1$ must be already executed and its state must be terminated. In the event guard $grd5$, we formalize **Sem1**, **Sem2**, and **Sem3** as previously described in the *SelectService* event guard. We presume that a component service cannot be executed if it does not meet these requirements. In the guard $grd6$ (Fig. 18), we have defined the PE requirements. Similarly to the inputs and outputs, each service precondition must be satisfied by the previous services' effects. We presume that a service cannot be executed if its preconditions are not satisfied. In formula (4), we assume that the first component service must have the same preconditions as the composite service. It denotes that for all preconditions p such that p is a precondition of the required composite service $(p \in precsR(ReqOf(c)))$, p must belong to the set of preconditions of the operation of the service s $(p \in precsOp(op))$. $precsR(ReqOf(c))$ and $precsOp(op)$ are finite sets and the number of elements of the set $precsR(ReqOf(c))$ is equal to the number of elements of the set $precsOp(op)$. In other words, the first component service must have exactly the same preconditions as the request. Additional preconditions may disable the execution of the component service. We assume that for each precondition p, there is no effect e of any previously executed service at any time in the composite service execution that may contradict p (formula (5)). For each precondition of the operation op of the service s, there should not be an effect e element of the actual effects set $AcEff$, such that it contradicts p. For instance the range of p is TRUE and the range of e is FALSE. In formula (6), we assume that the last component service produces the same effects as the required composite service. It denotes that for each effect e of the composite service c, there should be an element of the $effsOP(op)$ set of the last component service. $effsOP(op)$ denotes the set of effects of the service operation op. In the actions clause, the action $act4$ is defined to add the effects of the actual service execution to the effects of the previously executed services.

6.4. Modelling the behavioral requirements

In this section, we model the behavioural requirements that a composite service must meet. Therefore, features related to the composite behavior have been added to the model. For instance, we have introduced the component services' protocols, the protocol messages, the reception capabilities and the patterns used for the messages interchange. In this work, we content to model the blocking and non-blocking send patterns. At this refinement level, we model and verify the behavior of the component services. The selected services must be compatible in terms of interacting protocols in order to avoid blocking situations. Wherefore, in this section, we refine the *SelectService* in order to introduce behavior requirements. For instance, we assume that for each message that will be sent/received, there is a component service that will respectively receive/send that message.

CONTEXT BehC1
EXTENDS SemC1
SETS

Messages Protocols Queues Type

CONSTANTS

ProtOf RMsg SMsg Id

AXIOMS

axm1: $ProtOf \in Services \rightarrow Protocols$
axm2: $RMsg \in Protocols \nrightarrow \mathbb{P}(Messages)$
axm3: $SMsg \in Protocols \nrightarrow \mathbb{P}(Messages)$
axm4: $Id \in Messages \rightarrow \mathbb{N}$

END

Fig. 20. The $BehC1$ context description

MACHINE BehM1
REFINES SemM1
SEES BehC1
VARIABLES

Queue Qsize SReqM RReqM send receive STP Ack WaitForAckOf Consumed

INVARIANTS

inv1: $Queue \in Components \rightarrow \mathbb{P}(Messages)$
inv2: $Qsize \in Components \rightarrow \mathbb{N}$
inv3: $SReqM \in Requests \rightarrow \mathbb{P}(Messages)$
inv4: $RReqM \in Requests \rightarrow \mathbb{P}(Messages)$
inv5: $send \in Messages \nrightarrow Components$
inv6: $receive \in Messages \nrightarrow \mathbb{P}(Components)$
inv7: $STP \in send \rightarrow Type$
inv8: $Ack \in Messages \nrightarrow Components$
inv9: $WaitForAckOf \in Ack \nrightarrow BOOL$
inv10: $Consumed \in Ack \nrightarrow BOOL$

END

Fig. 21. The $BehM1$ machine description

Such constraints enable us to avoid any blocking situation where for example a component service is waiting to receive a message however none of the other component services is willing to send it, dragging the composite service execution to failure. This design-time verification is enhanced by a runtime verification performed by the refined events: *RunComposite*, *RunComponent* and *Terminate*. We also introduced new events (*Non_blocking_send*, *Blocking_send*, *Send_Ack* and *Consume*) in order to model the send patterns. This runtime verification is performed to ensure reliable communication between interacting protocols. Therefore, we have extended the $SemC1$ context by the $BehC1$ context and refined the $SemM1$ machine by the $BehM1$ machine (Fig. 19). We start by representing the $BehC1$ context (Fig. 20). We mainly define new sets namely the *Protocols*, *Messages*, *Queues*, and *Type* sets. These sets are used to type constants in the AXIOMS clause as follows:

- The constant *ProtOf* is introduced to represent the protocol of each service ($axm1$). It is a total function from the *Services* set to the *Protocols* set. It means that each service has exactly one protocol.
- *RMsg* represents the set of messages that a component service requires ($axm2$). It is a partial function from the *Services* set to the power set of *Messages*.
- *SMsg* models the set of messages that a component service has to send ($axm3$). It is a partial function from the *Services* set to the non-empty power set *Messages*.
- *Id* represents a unique identifier for each message ($axm4$).

The above context is seen, in the clause SEES, by the machine $BehM1$ (Fig. 21). In this second level of refinement, we define new variables as follows:

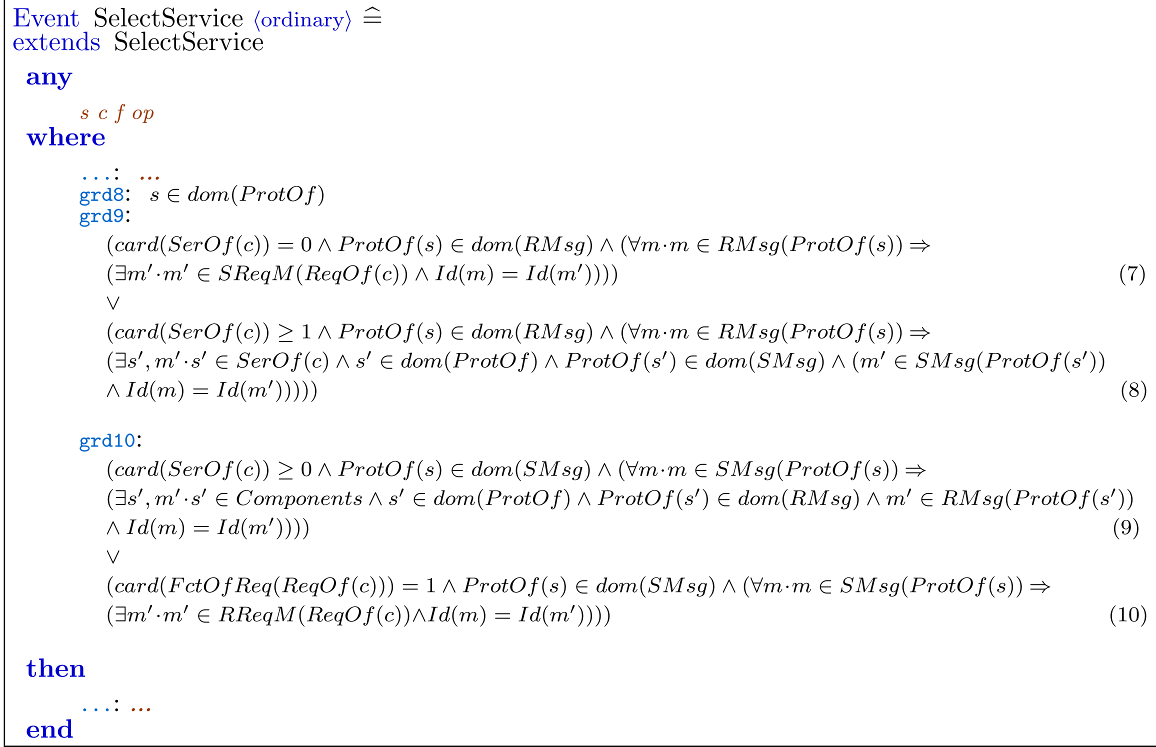


Fig. 22. The second refinement of the *SelectService* event

- *Queue*, in the invariant *inv1*, is a total function that returns the set of messages waiting in the service queue. Each component service can have zero or many services in its queue.
- The queue size (Qsize) of each component service is denoted by a total function *Qsize* (*inv2*).
- Each composite service request is created after receiving a message from the consumer and is supposed to send the request's result which is the composite service's execution result. This is modelled in Event-B via the total functions *SReqM* and *RReqM* (*inv3* and *inv4*) that respectively denote the set of messages sent by the requester and the set of messages he expects to receive.
- *Send*, in the invariant *inv*5, refers to the send actions. It is a partial function between the message to be sent and the component service that waits for this message.
- The same goes for the receive action in the invariant *inv*6.
- *STP* is a total function defined for each send's action to denote the type of send (whether it is a blocking or a non-blocking send).
- In a blocking send action, the service sends the message and stays blocked until receiving the message acknowledgment. Therefore, we have defined in *inv*8 and *inv*9 the *Ack* and the *WaitForAckOf* functions to manage the blocking send actions. They respectively refer to the message acknowledgment required by the sender and the state of the sender (whether it is waiting for an acknowledgment or not).
- *Consumed*, in the invariant *inv*10, denotes whether the received message is consumed by the component service or not.

The component services' protocols must be compatibles in order to avoid blocking situations. Namely, for each required message there will be a sender and for each message to be sent there will be a receiver. Such verification enables us to avoid deadlock situations. We proceed to the description of the second refinement of the *SelectService* event (Fig. 22). At this level, we check the protocols compatibility between candidate services at the selection process. In the guard *grd*9, in formula (7), we assume that the set of messages required by the first component service matches with the set of messages received by the request.

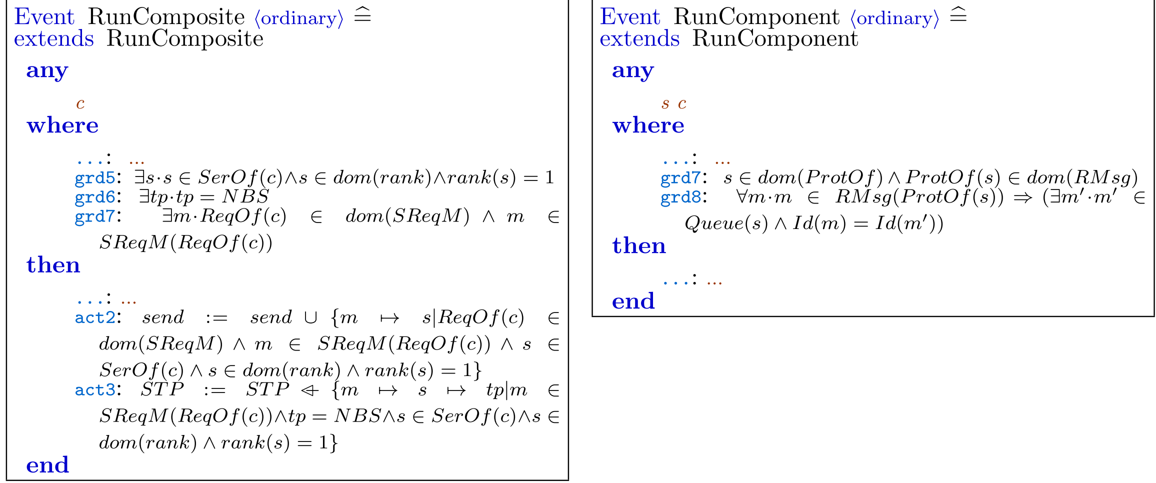


Fig. 23. The second refinement of the *RunComposite* and *RunComponent* events

It denotes that for each message m such that m is requested by the protocol of s ($ProtOf(s)$), there exists m' a message of the request such that ($Id(m){=}Id(m')$). In other words, the first component service must require the same messages as the composite service since a service cannot be executed unless it have received all its required messages. For the rest of the component services, we have defined the formula (8) where for each message m requested by the candidate service s, there must be a service s' preceding s ($s' \in SerOf(c)$) and willing to send a message m' ($m' \in SMsg(ProtOf(s'))$ such that $Id(m){=}Id(m')$. Namely, a component service must dispose of its required messages in order to execute. These messages must be provided by previously executed services. The same goes for the send action messages. When a message is going to be sent, there must be a service that is waiting for that message (*grd10*). In the description above the event guard $grd9$ models the behavioral requirement **Beh1** and the guard $grd10$ models the behavioral requirement **Beh2**. The *runComposite* event is refined in Fig. 23. The composition's execution starts by sending the messages of the request to the first component service. These messages generally contain the inputs of the composition. Namely, to run a composite service at this level the first component service must be selected and the send type and the message(s) of the request must be defined. This event is triggered only if the guards $grd5$, $grd6$, and $grd7$ are evaluated to TRUE. In the guard $grd5$, we presume that there exists a component service s already selected ($s \in SerOf(c)$) and with rank equal to 1 ($rank(s){=}1$). In the guard $grd6$, we presume that there exists a non-blocking (NBS) send type tp. Here we use the non-blocking send type for the messages exchange between the requester and the composite service. In the guard $grd7$, we presume that there exists a message m of the request. This message(s) communicates the inputs of the request. The execution of the composite service should result in defining the messages to send to the first component service. In the actions clause, we define the actions to perform once the guards are evaluated to TRUE. The message m and the service s are added to the *send* function in *act2*. Here each message of the request is added to the *send* set representing the set of messages that are ready to be sent. s is the first component service and the recipient of this message(s). The type of the send action is set to non-blocking send (NBS) in the action *act3*. To run each component service, we presume that each required message must be received in its messages' queue. This requirement is modelled in $grd8$ of the *RunComponent* event (Fig. 23).

We have considered two sending patterns in this work: the non-blocking and the blocking send patterns defined respectively by the *Non_blocking_send* and *Blocking_send* events (Fig. 24). In the *Non_blocking_send* situation, the sender sends the message without requesting any acknowledgment from the receiver. However, in the *Blocking_send* situation, the sender sends the message and stays blocked until the reception of the message acknowledgment from the message receiver. The *Non_blocking_send* event has three parameters: the composite service c, the component service s, and the message m. It is triggered when the message m is going to be sent by the service s with a non blocking. send type $STP(m \mapsto s){=}NBS$. The queue of s is able to receive the message m ($card(Queue(s)) \leq Qsize(s)$) (**Beh3**). On the other hand, the *Blocking_send* event has two parameters: the component service s and the message m.

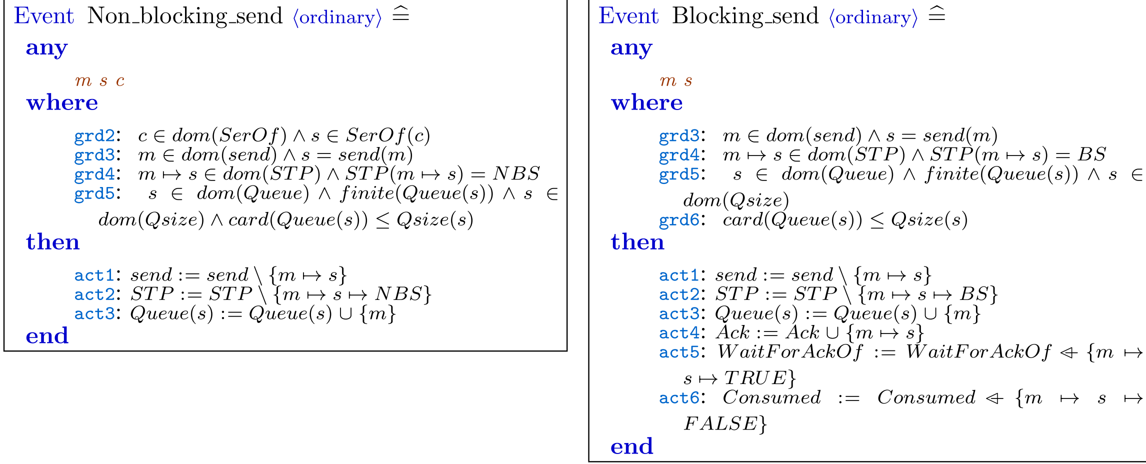


Fig. 24. The *Non_blocking_send* and *Blocking_send* events description

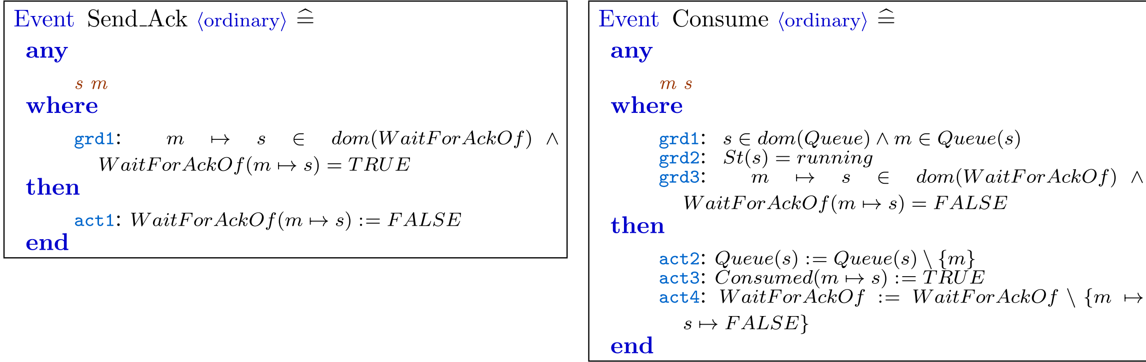


Fig. 25. The *Send_Ack* and *Consume* events description

It is triggered when the message m is going to be sent by the service s with a blocking send type. In the guard $grd6$, the queue of s is able to receive the message m (**Beh3**). In the action $act4$, the pair (m,s) is added to the Ack set. This means that the sender is waiting to receive the acknowledgment of the message m sent by s.

We have defined the *Send_Ack* event (Fig. 25) that is triggered when a component service receives a message with a blocking acknowledgment. This event's execution sends an acknowledgment to the message's sender. *WaitForAckOf* is then set to false (*act1*). Once the acknowledgment (*grd3*) is sent, the message can be consumed via the *Consume* event (Fig. 25). The message m is deleted from the queue of the service and *Consumed* is set to true (*act3*). The second refinement of the *Terminate* event is given in Fig. 26. The service's execution ends by deleting the received messages from the *Ack* and *Consumed* sets (*act4, act5*). The outputs of the service are sent to the next service in (*act6*) and the type of send is set to blocking *BS* (*act7*). The requirement **Beh4** is met if all the component services' executions terminate correctly.

6.5. Modelling the resource requirements

In the Cloud environment, services are hosted in the virtual machines running on the Cloud's infrastructure. Therefore, a set of physical resources are allocated to each component service in order for it to execute, which is not a trivial task. A set of constraints should be considered in order to perform correct and efficient resource allocation. In this section, we model the resource properties of a Cloud composite service. Therefore we have added the resources and their features to the model. For instance the resource state, type, sharing properties and capabilities.

Event Terminate ⟨ordinary⟩ $\widehat{=}$
extends Terminate
any
$s\ c$
where
...: ...
grd4: $c \in dom(SerOf) \wedge s \in SerOf(c) \wedge s \in dom(ProtOf) \wedge ProtOf(s) \in dom(SMsg) \wedge ProtOf(s) \in dom(RMsg)$
grd5: $\forall m \cdot m \in RMsg(ProtOf(s)) \Rightarrow (\exists m' \cdot m' \mapsto s \in dom(Consumed) \wedge Consumed(m' \mapsto s) = TRUE \wedge Id(m) = Id(m'))$
grd6: $(\exists tp, ser \cdot ser \in SerOf(c) \wedge ser \in dom(rank) \wedge rank(ser) = rank(s) + 1 \wedge tp = BS) \vee (rank(s) = max(ran(rank)))$
then
...: ...
act4: $Ack := Ack \setminus \{m \mapsto ser | m \mapsto ser \in dom(Consumed) \wedge Consumed(m \mapsto ser) = TRUE \wedge ser = s \wedge m \mapsto ser \notin dom(WaitForAckOf)\}$
act5: $Consumed := Consumed \setminus \{m \mapsto ser \mapsto TRUE | ser \in SerOf(c) \wedge m \mapsto ser \in dom(Consumed) \wedge Consumed(m \mapsto ser) = TRUE \wedge ser = s \wedge m \mapsto ser \notin dom(WaitForAckOf)\}$
act6: $send := send \lhd\!\!\!- \{m \mapsto ser | ProtOf(s) \in dom(SMsg) \wedge m \in SMsg(ProtOf(s)) \wedge c \in dom(SerOf) \wedge ser \in SerOf(c) \wedge ser \in dom(rank) \wedge rank(ser) = rank(s) + 1 \wedge ((\exists m' \cdot m' \in RReqM(ReqOf(c)) \wedge Id(m) = Id(m')) \vee (\exists m' \cdot m' \in RMsg(ProtOf(ser)) \wedge Id(m) = Id(m')))\}$
act7: $STP := STP \lhd\!\!\!- \{m \mapsto ser \mapsto tp | m \in SMsg(ProtOf(s)) \wedge ser \in SerOf(c) \wedge ser \in dom(rank) \wedge rank(ser) = rank(s) + 1 \wedge tp = BS \wedge ((\exists m' \cdot m' \in RReqM(ReqOf(c)) \wedge Id(m) = Id(m')) \vee (\exists m' \cdot m' \in RMsg(ProtOf(ser)) \wedge Id(m) = Id(m')))\}$
end

Fig. 26. The *Terminate* event description

We have also defined for each component service the type of resource it needs and the required capacity. At this level, the aim is to manage reliable resource allocation and to avoid blocking situations that may occur when allocating resources to services. Therefore we have introduced constraints on the resource sharing between component services through the model invariants. Namely, a non-shareable resource should not be allocated to more than one single service. If a non-shareable resource is allocated simultaneously to two parallel services, for example, the execution of one of the services will fail. We have also introduced new events related to the resource perspective. The *AllocateResource* event conducts the resources allocation to different component services. The *DecResCap* decreases the capacity required by the service from the available resource capacity. Adding to the refinement of the ancestor events (*RunComponent* and *Terminate*). This fourth level of abstraction (Fig. 27) is constituted of the context *ResC1* that extends *BehC1* (Fig. 28) and the machine *ResM1* that refines *BehM1* (Fig. 29).

The **ResC1** context models the static properties of a Cloud resource. Its constituting elements are described as follows:

- The set *Resources* represents the set of resources.
- The set *Sh* represents an enumeration of a resource's possible sharing types. A resource can be exclusively shareable (*IsExcSh*), commonly shareable (*IsComSh*) or not shareable (*NotSh*). A partition of the set *Sh* is then given in the axiom $axm2$.
- The set *TypeR* represents an enumeration of a resource's possible types. A resource can be a computing resource (*CompRes*), a storage resource (*StorRes*) or a network resource (*NetRes*). A partition of the set *TypeR* is therefore given in the axiom $axm9$.
- The set *StateR* represents an enumeration of a resource's possible states. A resource can be either available or not ($axm1$). A resource is considered unavailable if all its capacity is consumed.
- *TypeOfr*, in the axiom $axm8$, is a total function assigns a type to each resource.
- *GCa* in axiom $axm10$ denotes the global capacity of a resource.

The machine *ResM1* sees the context *ResC1* and models the resource allocation requirements (Fig. 29). We have extended the model by adding variables related to the resource perspective as follows:

- *ACa* denotes the available capacity of a resource. It is represented by a partial function from the set *Resources* to the set of positive numbers $\mathbb{N}$ ($inv1$).

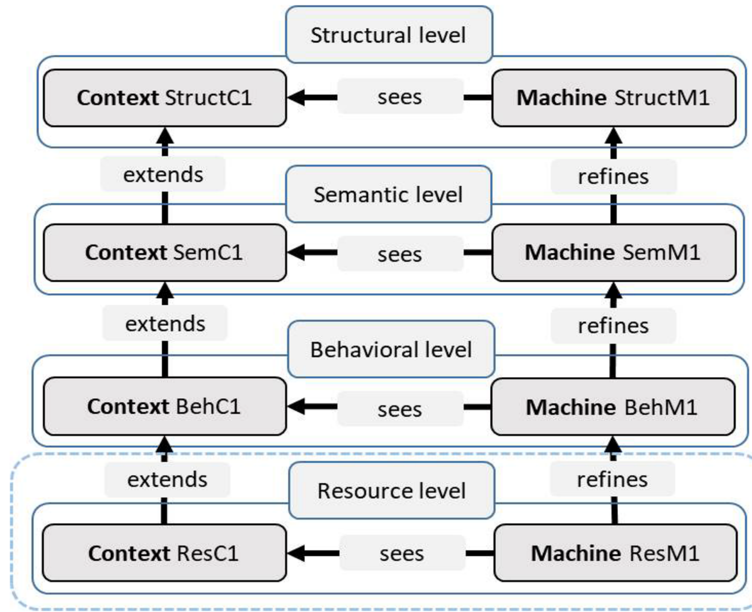


Fig. 27. The fourth abstraction level

CONTEXT ResC1
EXTENDS BehC1
SETS

Resources Sh TypeR StateR

CONSTANTS

Nav Av IsExcSh IsComSh NotSh CompRes StorRes NetRes TypeOfr GCa

AXIOMS

axm1: $partition(StateR, \{Nav\}, \{Av\})$
axm2: $partition(Sh, \{IsExcSh\}, \{IsComSh\}, \{NotSh\})$
axm9: $partition(TypeR, \{CompRes\}, \{StorRes\}, \{NetRes\})$
axm8: $TypeOfr \in Resources \rightarrow TypeR$
axm10: $GCa \in Resources \nrightarrow \mathbb{N}_1$

END

Fig. 28. The $ResC1$ context description

– *IsSh*, *IsEl*, and *IsSca* are total functions denoting, respectively, a resource's shareability, elasticity and scalability (*inv*7, *inv*8, *inv*9).
– A resource can be allocated either statically or dynamically (at runtime) to some component services. The set of services a resource is allocated to is defined by the partial function *AllocTo* from the *Resource* set to the power set of Components (*inv*6).
– The set of services that are actually running on the resource are given by the total function *RrunnigSer* (*inv*5).
– We associate to each service a set of required resource types denoted by the partial function *ReqTyR* (*inv*2).
– The capacity amount that a component service requires from each required resource type is denoted by the partial function *RCa* (*inv*3).

To correctly manage the resource allocation, different requirements should be satisfied. In this paper, we consider the following shareability requirements defined in *inv*10, *inv*11, *inv*12, *inv*13 and *inv*14 (Fig. 29):

MACHINE ResM1
REFINES BehM1
SEES ResC1
VARIABLES

ACa ReqTyR ResSt RrunnigSer AllocTo IsEl IsSca IsSh RCa

INVARIANTS

inv1: $ACa \in Resources \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
inv2: $ReqTyR \in Components \rightarrow\!\!\!\!\!\!+ \mathbb{P}(TypeR)$
inv3: $RCa \in (Components \rightarrow\!\!\!\!\!\!+ TypeR) \rightarrow\!\!\!\!\!\!+ \mathbb{N}_1$
inv4: $ResSt \in Resources \rightarrow StateR$
inv5: $RrunnigSer \in Resources \rightarrow\!\!\!\!\!\!+ \mathbb{P}(Components)$
inv6: $AllocTo \in Resources \rightarrow\!\!\!\!\!\!+ \mathbb{P}(Components)$
inv7: $IsSh \in Resources \rightarrow Sh$
inv8: $IsEl \in Resources \rightarrow BOOL$
inv9: $IsSca \in Resources \rightarrow BOOL$
inv10: $\forall r \cdot r \in Resources \wedge IsSh(r) = NotSh \wedge r \in dom(AllocTo) \wedge finite(AllocTo[\{r\}]) \Rightarrow card(AllocTo[\{r\}]) \leq 1$
inv11: $\forall r \cdot r \in Resources \wedge IsSh(r) = NotSh \wedge r \in dom(RrunnigSer) \wedge finite(RrunnigSer[\{r\}]) \Rightarrow card(RrunnigSer[\{r\}]) \leq 1$
inv12: $\forall r \cdot r \in Resources \wedge IsSh(r) \neq NotSh \wedge r \in dom(AllocTo) \wedge finite(AllocTo[\{r\}]) \Rightarrow card(AllocTo[\{r\}]) \geq 0$
inv13: $\forall r \cdot r \in Resources \wedge IsSh(r) = IsComSh \wedge r \in dom(RrunnigSer) \wedge finite(RrunnigSer[\{r\}]) \Rightarrow card(RrunnigSer[\{r\}]) \geq 0$
inv14: $\forall r \cdot r \in Resources \wedge IsSh(r) = IsExcSh \wedge r \in dom(RrunnigSer) \wedge finite(RrunnigSer[\{r\}]) \Rightarrow card(RrunnigSer[\{r\}]) \leq 1$

END

Fig. 29. The *ResM*1 machine description

Event AllocateResource ⟨ordinary⟩ $\widehat{=}$
any

r s tp

where

grd1: $s \mapsto deployed \in St \wedge s \in dom(ReqTyR) \wedge tp \in ReqTyR(s) \wedge TypeOfr(r) = tp \wedge r \in dom(AllocTo)$
grd2: $r \in dom(ACa) \wedge \{s \mapsto TypeOfr(r)\} \in dom(RCa) \wedge (RCa(\{s \mapsto tp\}) \leq ACa(r) \vee (RCa(\{s \mapsto tp\}) \geq ACa(r) \wedge IsEl(r) = TRUE) \vee (RCa(\{s \mapsto tp\}) \geq ACa(r) \wedge IsSca(r) = TRUE))$
grd3: $(IsSh(r) = IsComSh) \vee (IsSh(r) = IsExcSh \wedge (AllocTo(r) = \varnothing \vee (\forall ser \cdot ser \in AllocTo(r) \wedge ser \in dom(rank) \wedge s \in dom(rank) \wedge rank(ser) \neq rank(s)))) \vee (IsSh(r) = NotSh \wedge (AllocTo(r) = \varnothing))$

then

act1: $AllocTo := AllocTo \lhd\!\!\!- \{r \mapsto AllocTo(r) \cup \{s\}\}$
act2: $ReqTyR(s) := ReqTyR(s) \setminus \{TypeOfr(r)\}$

end

Fig. 30. The *AllocateResource* event description

- In the invariant *inv*10, a non shareable resource ($IsSh(r){=}NotSh$) can not be allocated to more than one service ($card(AllocTo[\{r\}]) \leq 1$) (**Sh1**).
- In the invariant *inv*11, we presume that at most one service ($card(RrunnigSer[\{r\}]) \leq 1$) can be running on a non shareable resource ($IsSh(r){=}NotSh$) (**Sh2**).
- In the invariant *inv*12, a shareable resource ($IsSh(r) \neq NotSh$) can be allocated to more than one service ($card(AllocTo[\{r\}]) \geq 0$)(**Sh3**).
- In the invariant *inv*13, we presume that one or more services ($card(RrunnigSer[\{r\}]) \geq 0$) can be running on a commonly shareable resource ($IsSh(r){=}IsComSh$) (**Sh4**).
- In the invariant *inv*14, we presume that at most one service ($card(RrunnigSer[\{r\}]) \leq 1$) can be running on an exclusively shareable resource($IsSh(r){=}IsExcSh$) at the same time (**Sh5**).

The *AllocateResource* event (Fig. 30) formalizes the resource allocation process. To properly allocate a resource to a service, we consider the following requirements:

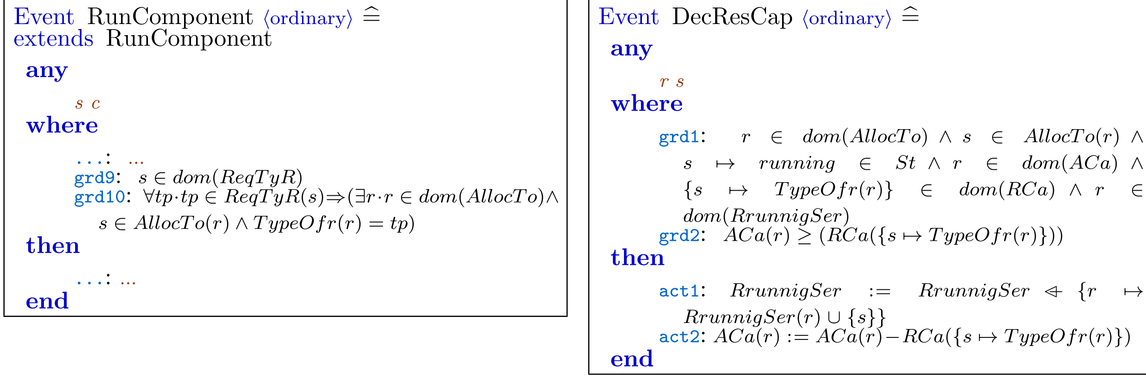


Fig. 31. The *RunComponent* and *DecResCap* events description

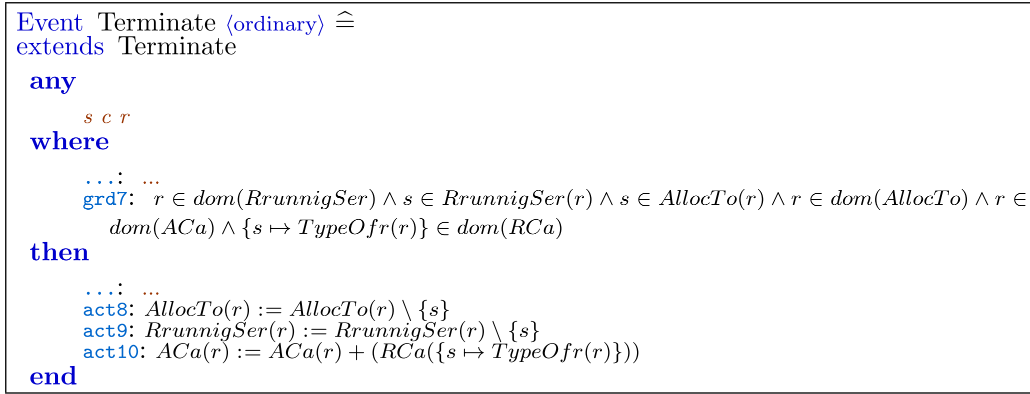


Fig. 32. The *Terminate* event description

- In the guard $grd1$, the resource is of the same type as the required resource type ($tp \in ReqTyR(s) \wedge TypeOfr(r){=}t$) (**Ra1**) and,
- there is enough resource capacity to handle the service ($RCa(\{s \mapsto tp\}) \leq ACa(r)$) ($grd2$) (**Ra2**) or,
- there is not enough resource capacity to handle the service ($RCa(\{s \mapsto tp\}) \geq ACa(r)$), however the resource is Elastic ($IsEl(r){=}TRUE$) ($grd2$) (**Ra3**) or,
- there is not enough resource capacity to handle the service ($RCa(\{s \mapsto tp\}) \geq ACa(r)$), however the resource is Scalable ($IsSca(r){=}TRUE$) ($grd2$) (**Ra4**).
- The resource is shareable ($IsSh(r) \not= NotSh$) ($grd3$) (**Ra5**) or,
- the resource is not shareable ($IsSh(r){=}NotSh$) and it is not allocated to any other service ($AllocTo(r){=}\varnothing$) ($grd3$) (**Ra6**).

Once the above requirements are fulfilled, the action $act1$ is triggered, and the resource is allocated to the service. The type of the required resource is deleted from the set of required types ($act2$). After defining the resource allocation event, we have to refine the events defined in *BehM1* in order to verify the resource properties of the running composite service. We first start by refining the *RunComponent* event (Fig. 31). To run a component service, for each resource type it requires there must be a set of resources allocated to this service with the required capacity ($grd10$). Once the service is running on the adequate resource, the resource's capacity is decreased by the event *DecResCap* (Fig. 31) ($act2$). In $act1$, the service s is added to the set of services running on the resource r.
We then, refine the *Terminate* event (Fig. 32). Once terminated, the service releases the allocated resources.

The terminated component service is deleted from the set of services running on the resource and the set of services to which the resource is allocated ($act8$, $act9$). Also, the resource's capacity occupied by the service is released ($act10$). The proposed resource allocation verification allows avoiding deadlocks when parallel component services are executed simultaneously.

6.6. Summary of the development

To sum up, the model presented above (see the model specification in Fig. 6 and the events' refinement in Fig. 7) addresses the verification of the composite services. Its aim is to compose and execute component services in a consistent way. The developed model is able to select component services that meet the requirements thanks to the *SelectService* event. Namely, it selects services that provide the required functions and semantically match. It also guarantees that the selected service' protocols match in order to prevent deadlock situations. Resources are allocated to selected component services in a rigorous way thanks to the *AllocateResource* event. The present work handles also the runtime verification of the composite service. The selected component services are accurately executed in the *RunComponent* event. A component service is executed only if it provides the required function, its preconditions are met and its inputs are received. Inputs are communicated thanks to the behavior refinement where we have defined the send patterns (Blocking send and non-blocking send patterns) in the *Blocking_send* and *Non_blocking_send* events. These patterns ensure a reliable communication of messages between component services. Thanks to the presented features, our model deals with the composite service verification problem in a comprehensive and consistent way.

7. Verification and validation

In this section, we describe the steps followed in order to verify and validate our model. The verification covers the static and dynamic properties of the model. The static properties are expressed through the invariants. The invariants of the model must hold for all states of the model; they must hold at the initial state and must be preserved by each event. Dynamic properties refer to the temporal properties of the system. Such properties could not be expressed through invariants. They express the different states of the system at different animation times.
Indeed, the validation consists in observing the specification's behavior. The Rodin platform[1] provides the plugin ProB [LB03] for the animation and validation of Event-B specifications. This plugin gives us the possibility to play different scenarios by showing, at each stage, the values of each variable and distinguishing the enabled events from the disabled ones.

7.1. Proof-based verification

The term proof obligation is mentioned in this section regularly. It is in fact, a theorem that needs to be proved in order to verify the correctness of the model [Pad11]. The proof obligations (POs)[2] are automatically generated by the Proof Obligation Generator tool of the Rodin Platform. The generated proof obligations were of types well-definedness (WD) and Invariant preservation (INV) (for the invariants of the model). The INV POs ensure that each event preserves the invariants. The name of an INV PO is of the form evt/inv/INV where for each event, we have to establish that:

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

where the event actions Act must preserve the invariant *Inv*. In other words, INV POs guarantee that:

- The initialization of a machine leads to a state where the invariants are valid.
- Assuming that the machine is in a state where the invariants are preserved, every enabled event leads to a state where the invariants are still preserved.

[1] http://www.event-b.org/.
[2] For more details on proof obligations rules please refer to the B-Book [Abr05] on page 190.

Table 1. Formal definition of the well-definedness PO (WD) [Abr05]

Mathematical expression	Well-definedness condition
inter (S)	$S \neq \varnothing$
$\cap x \cdot P \mid T$	$\exists x \cdot P$
f(E)	f is a partial function and $E \in dom(f)$
E/F	$F \neq 0$
E mod F	$0 \leq E \wedge 0 < F$
card(S)	finite(S)
min(S)	$S \neq 0 \wedge \exists x \cdot (\forall n \cdot n \in S \Rightarrow x \leq n)$
max(S)	$S \neq 0 \wedge \exists x \cdot (\forall n \cdot n \in S \Rightarrow x \geq n)$

Fig. 33. Proof obligations of the model

The well-definedness proof obligation rule (WD) ensures that a potentially ill-defined axiom, theorem, invariant, guard, action, variant, or witness is indeed well defined [Abr05]. For a given modelling element (axm, thm, inv, grd, act or a variant, or a witness x in an event evt), the names are: axm/WD, thm/WD, inv/WD, grd/WD, act/WD, VWD , evt/x/WWD. The specific form of this proof obligation rule depends on the potentially ill-defined expression. This is indicated in Table 1. A PO can be either automatically or interactively discharged (green symbol), or undischarged (orange symbol). The symbol "A" means that the PO is automatically discharged. Figure 33 reports some POs that are generated while proving the consistency of the model.

Modelling in Event-B relies entirely on the interplay between editing models and analyzing their proof obligations. Proof obligations are generated not only to ensure that each event preserves the invariants, but also to verify that the refinement had been correctly performed. To prove that a refinement is correct, 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:

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

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

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

In other words, to ensure a correct refinement, we must prove two things:

- The concrete events can only occur when the abstract ones occur.
- If a concrete event occurs, the abstract event can occur in such a way that the resulting states correspond again, i.e. the gluing invariant remains true.

For Feasibility POs we have to ensure that the action of an event is always feasible whenever the event is enabled. In other words, there are always possible after values for the variables, satisfying the before-after predicate. In practice, we prove feasibility for individual assignment of the event action. For deterministic assignments, feasibility holds trivially [Hoa13].

To summarize, 157 proof obligations have been generated: 83 of them are automatically discharged by the automatic prover. It fails to discharge the remaining proofs due to the numerous steps they require and not on account of their difficulty. To finish discharging these proofs, we resorted to the interactive prover and helped it find the right steps and rules to apply. The proof statistics are given in Fig. 34.

Element Name	Total	Auto	Manual	Reviewed	Undischarged
CloudM	**157**	**83**	**74**	**0**	**0**
BehM1	64	32	32	0	0
ResM1	51	24	27	0	0
SemM1	17	4	13	0	0
StructM1	25	23	2	0	0

Fig. 34. Proof statistics

7.2. Validation by animation

For model validation, we use the ProB animation [LB03]. ProB is an animator, constraint solver and model checker for the B-Method. It allows fully automatic animation of B specifications and can be used to systematically check a specification for a wide range of errors. The constraint-solving capabilities of ProB can also be used for model finding, deadlock checking and test-case generation [LTZ+13]. We use animation to execute specifications. Thanks to ProB we have played and observed different scenarios in order to check the behavior of our model. The animation is performed on a concrete Event-B model. For this purpose, we have given values to the carrier sets, constants, and variables of the model. To do so, we have considered the motivating example introduced in Sect. 3. The animation consists of the following steps (Fig. 35):

- **Step1.** we start by firing the SETUP-CONTEXT event that gives values to the constants and carrier sets in the context,
- **Step2.** we then fire the INITIALISATION event to set the model into its initial state,
- **Step3.** we, finally, proceed to the steps of the scenario to check. At each step, the animator computes all guards of all events, and enables the ones with true guards, and shows parameters which make these guards true. After event firing, substitutions are computed and the animator checks if the invariants still hold.

For instance, we animated the complete behavior of the composite service while verifying the different states in which it may move. We have successfully applied the animation of ProB on our final level of refinement as follows:

- **Step3-1.** We start by firing the *SelectService* event which selects a set of component services according to the previously defined requirements.
- **Step3-2.** Before running, the required resources are allocated to each component service using the *ResourceAllocation* event, with respect to the previously defined resource allocation and shareability requirements.
- **Step3-3.** The *RunComposite* event is then enabled to run the composite service.
- **Step3-4.** The request inputs are sent to the first component service by means of the non-blocking send pattern using the *Non_Blocking_Send* event.
- **Step3-5.** The *RunComponent* event is then enabled,
- **Step3-6.** The resource capacity is decreased (*DecResCap* event) and the received messages are consumed (*Consume* event).
- **Step3-7.** The execution of the component service is terminated (*Terminate* event) after the completion of its execution time (*Time* event: (T-t)=ExeT(s)) and the allocated resources are released.
- **Step3-8.** The outputs of the component service are sent either using the non-blocking send pattern (*Non_Blocking_Send* event) or the blocking send pattern (*Blocking_Send* event), to the next component service.
- **Step3-9.** The *RunComponent* event is then enabled again.
- **Step3-10.** If there is a message sent according to the blocking send pattern, the *Send_Ack* event is enabled and the execution stays blocked until the reception of the acknowledgment by the sender and the animation resumes from **Step3-6**.

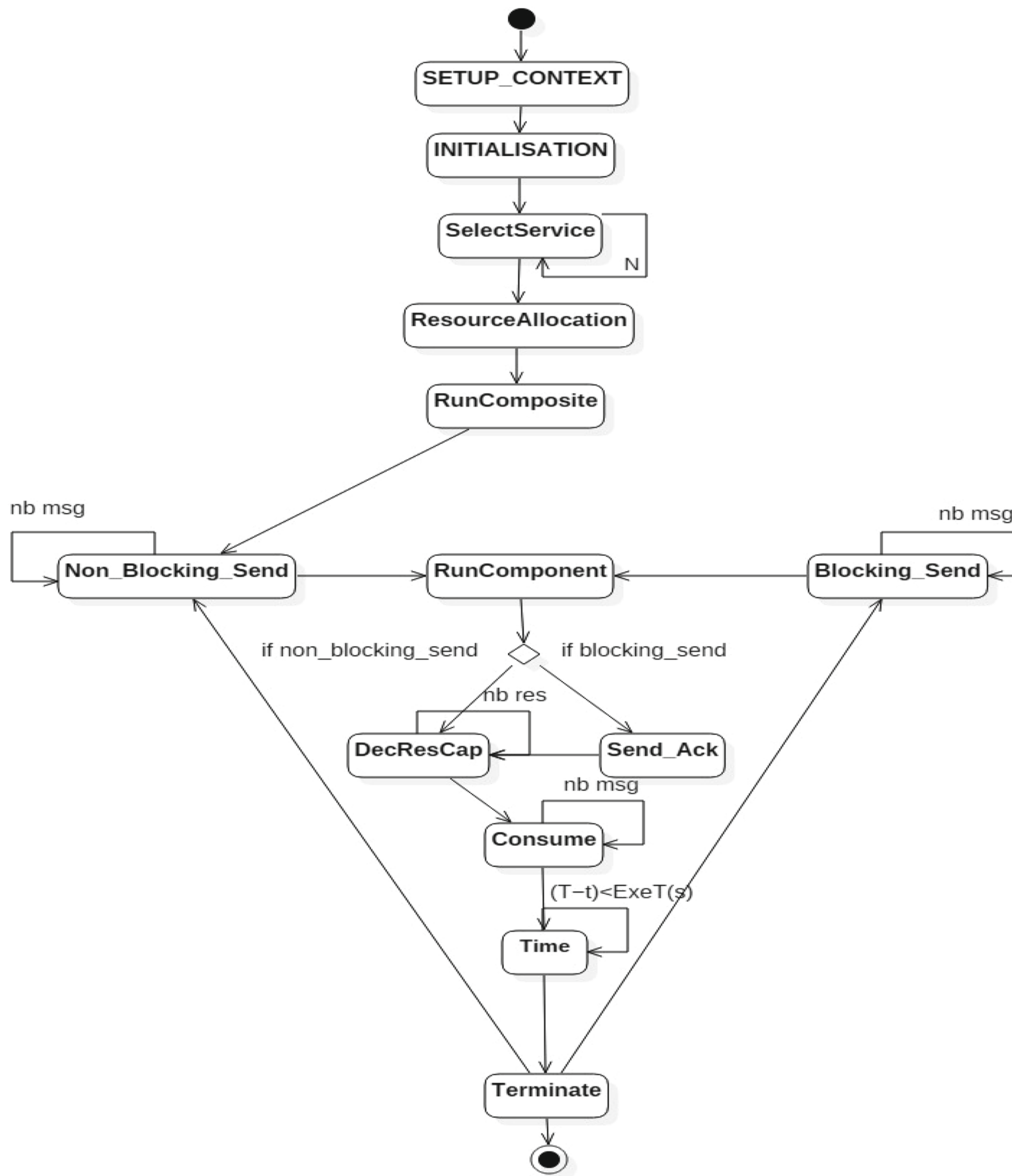


Fig. 35. Animation scenario

If the composite service is behaviourally incorrect, the animation stops without reaching the final service which incites the designer to make the necessary modifications to the composite service to avoid any deadlock situations. Compared to existing works, the proposed approach is complete. It combines the modelling and the verification of the structure, semantics, behavior of composite service and allocated resources in the Cloud context. Our approach is an incremental approach of modelling and verification of composite services in the Cloud context. The use of Event-B allows us to master the complexity of the composite service by introducing its details step by step. Event-B is well-known by its mathematical bases for the specification, development, and verification of complex applications. The correctness and the consistency of our model are validated by the use of the ProB animator/model checker and the proof activity. It is noted also that the Event-B method does not suffer from the state explosion problem which is not the case for other verification methods used in the literature.

8. Conclusion

In this work, the focus of attention was on the verification of the correctness of Cloud composite services. The proposed approach is based on the Event-B formal method. The designed formal model comprises four abstraction levels to model the structural, semantic, behavioral and resource allocation requirements that must be considered in order to avoid inconsistent composite services. A proof-based approach, coupled with a model animation, is performed in order to verify and validate the proposed model. Thanks to the proposed approach, we succeeded to rigorously design and verify Cloud composite services. On the basis of the promising findings presented in this paper, work on the remaining issues is continuing and will be presented in future papers. For instance, we aim, in the near future, to extend this work by considering the resource elasticity problems. We are also developing an Eclipse plugin to automate the present approach in order to be readily used in practice.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

[ABH+10] Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in event-b. Int J Softw Tools Technol Transf 12(6):447–466
[Abr88] Abrial JR (1988) The B tool (Abstract). Springer, Berlin, pp 86–87
[Abr05] Abrial J-R (2005) The B-book—assigning programs to meanings. Cambridge University Press, Cambridge
[AGGH15] Abbassi I, Graiet M, Gaaloul W, Hadj-Alouane NB (2015) Genetic-based approach for ATS and sla-aware web services composition. In: Web information systems engineering—WISE 2015—16th international conference, Miami, FL, USA, November 1–3, 2015, Proceedings, Part I, pp 369–383
[AM98] Abrial J-R, Mussat L (1998) Introducing dynamic constraints in B. In: B'98: recent advances in the development and use of the B method, second international B conference, Montpellier, France, April 22–24, 1998, Proceedings, pp 83–128
[AM16] Amato F, Moscato F (2016) Pattern-based orchestration and automatic verification of composite cloud services. Comput & Electr Eng 56:842–853
[AMJS11] Abdelsadiq A, Molina-Jimenez C, Shrivastava S (2011) A high-level model-checking tool for verifying service agreements. In: Proceedings of 2011 IEEE 6th international symposium on service oriented system (SOSE), pp 297–304
[BKKM11] Byun E-K, Kee Y-S, Kim J-S, Maeng S (2011) Cost optimized provisioning of elastic resources for application workflows. Future Gener Comput Syst 27(8):1011–1026
[BKS+17] Boubaker S, Klai K, Schmitz K, Graiet M, Gaaloul W (2017) Deadlock-freeness verification of business process configuration using SOG. In: Maximilien M, Vallecillo A, Wang J, Oriol M (eds) Service-oriented computing. Springer, Cham, pp 96–112
[BSS19] Bourne S, Szabo C, Sheng QZ (2019) Transactional behavior verification in business process as a service configuration. IEEE Trans Serv Comput 12(2):290–303
[BYO+14] Bessai K, Youcef S, Oulamara A, Godart C, Nurcan S (2014) Scheduling strategies for business process applications in cloud environments. Int J Grid High Perform Comput, 5:65–78, 01
[CHH+12] Chen J, Huang L, Huang H, Yu C, Li C (2012) A formal model for resource protections in web service applications. In: 2012 international conference on cloud and service computing, pp 111–118
[CM08] Cansell D, Méry D (2008) The event-b modelling method: concepts and case studies. pp 47–152
[CWG09] Cao Q, Wei Z, Gong W (2009) An optimized algorithm for task scheduling based on activity based costing in cloud computing. In: 2009 3rd international conference on bioinformatics and biomedical engineering, pp 1–3
[DOS12] Durán F, Ouederni M, Salaün Gwen (July 2012) A generic framework for n-protocol compatibility checking. Sci Comput Program 77(7-8):870–886
[EMAZ15] Elhag AAM, Mohamad R, Aziz MW, Zeshan F (2015) A systematic composite service design modeling method using graph-based theory. PLoS One 10(4):1–26, 04
[FE10] Furht Bo, Escalante A (2010) Handbook of cloud computing. 1st edition. Springer, Berlin
[FW12] Freitas L, Watson P (2012) Formalising workflows partitioning over federated clouds: multi-level security and costs. In: 2012 IEEE eighth world congress on services, pp 219–226
[GHMT17] Graiet M, Hamel L, Mammar A, Tata S (2017) A verification and deployment approach for elastic component-based applications. Formal Asp Comput 29(6):987–1011
[GLA+15] Graiet M, Lahouij A, Abbassi I, Hamel L, Kmimech M (2015) Formal behavioral modeling for verifying SCA composition with event-b. In: 2015 IEEE international conference on web services, ICWS 2015, New York, NY, USA, June 27–July 2, 2015, pp 17–24
[GMBG17] Graiet M, Mammar A, Boubaker S, Gaaloul W (2017) Towards correct cloud resource allocation in business processes. IEEE Trans Serv Comput 10(1):23–36
[Hoa13] Hoang TS (2013) An introduction to the event-B modelling method, pp 211–236. 07
[Hol97] Holzmann GJ (1997) The model checker spin. IEEE Trans Softw Eng 23(5):279–295

[HSD13] Hoenisch P, Schulte S, Dustdar S (2013) Workflow scheduling and resource allocation for cloud-based execution of elastic processes. In: 2013 IEEE 6th international conference on service-oriented computing and applications, pp 1–8

[HSDV13] Hoenisch P, Schulte S, Dustdar S, Venugopal S (2013) Self-adaptive resource allocation for elastic process execution. In: 2013 IEEE sixth international conference on cloud computing, pp 220–227

[JCM19] Jana B, Chakraborty M, Mandal T (2019) A task scheduling technique based on particle swarm optimization algorithm in cloud environment. In: Proceedings of SoCTA 2017, pp 525–536. 01

[KMCB17] Kallab L, Mrissa M, Chbeir R (2017) Bourreau Pierre Using colored petri nets for verifying restful service composition. In: Panetto H, Debruyne C, Gaaloul W, Papazoglou M, Paschke A, Ardagna CA, Meersman R (eds) On the move to meaningful internet systems. OTM 2017 conferences. Springer International Publishing, Cham, pp 505–523

[KTD11] Klai K, Tata S, Desel J (2009) Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. Data Knowl Eng 70(5):467–482. In: Business Process Management, 2011

[LB03] Leuschel M, Butler M (2003) Prob: a model checker for b. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods. Springer, Berlin, pp 855–874

[LHG15] Lahouij A, Hamel L, Graiet M (2015) Formal modeling for verifying SCA dynamic composition with event-b. In: 24th IEEE international conference on enabling technologies: infrastructure for collaborative enterprises, WETICE 2015, Larnaca, Cyprus, June 15–17, 2015, pp 29–34

[LHG+16] Lahouij A, Hamel L, Graiet M, Elkhalfa A, Gaaloul W (2016) A global sla-aware approach for aggregating services in the cloud. In: On the move to meaningful internet systems: OTM 2016 conferences—confederated international conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24–28, 2016, Proceedings, pp 363–380

[LHG18] Lahouij A, Hamel L, Graiet M (2018) Deadlock-freeness verification of cloud composite services using event-b. In: On the move to meaningful internet systems. OTM 2018 conferences—confederated international conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part I, pp 604–622

[LHGM18] Lahouij A, Hamel L, Graiet M, Malki ME (2018) A formal approach for cloud composite services verification. In: 11th IEEE conference on service-oriented computing and applications, SOCA 2018, Paris, France, November 20–22, 2018, pp 161–168

[LHJ+14] Leesatapornwongsa T, Hao M, Joshi P, Lukman JF, Gunawi HS (2014) Samc: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proceedings of the 11th USENIX conference on operating systems design and implementation, OSDI'14, pp 399–414, Berkeley, CA, USA. USENIX Association.

[LTZ+13] Laili Y, Tao F, Zhang L, Cheng Y, Luo Y, Sarker BR (2013) A ranking chaos algorithm for dual scheduling of cloud service and computing resource in private cloud. Comput Ind 64(4):448–463

[MFBR15] Mastelic T, Fdhila W, Brandic I, Rinderle-Ma S (2015) Predicting resource allocation and costs for business processes in the cloud. In: 2015 IEEE world congress on services, pp 47–54

[MKS13] Malik SUR, Khan SU, Srinivasan SK (2013) Modeling and analysis of state-of-the-art vm-based cloud management platforms. IEEE Trans Cloud Comput 1(1):1–1

[NSG+14] Naskos A, Stachtiari E, Gounaris A, Katsaros P, Tsoumakos D, Konstantinou I, Sioutas S (2014) Cloud elasticity using probabilistic model checking. 05

[Pad11] Padidar S (2011) A study in the use of event-b for system development from a software engineering viewpoint

[PF11] Papapanagiotou P, Fleuriot J (2011) Formal verification of web services composition using linear logic and the pi-calculus. In: 2011 IEEE ninth European conference on web services, pp 31–38

[SMWZ15] Sun L, Ma J, Wang H, Zhang Y (2015) Cloud service description model: an extension of usdl for cloud services. IEEE Trans Serv Comput, (99):1–1

[Sub04] W3C Member Submission (2004) Owl-s: semantic markup for web services. https://www.w3.org/Submission/OWL-S/.

[WD96] Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA

[WDJ+16] Wang P, Ding Z, Jiang C, Zhou M, Zheng Y (2016) Automatic web service composition based on uncertainty execution effects. IEEE Trans Serv Comput 9(4):551–565

[WDJZ14] Wang P, Ding Z, Jiang C, Zhou M (2014) Constraint-aware approach to web service composition. IEEE Trans Syst Man Cybern Syst 44(6):770–784

[ZGOH09] Zeng C, Guo X, Ou W, Han D (2009) Cloud computing service composition and search based on semantic. In: Proceedings of the 1st international conference on cloud computing, CloudCom '09, Springer, Berlin, pp 290–300

Received 15 March 2019

Accepted in revised form 28 July 2020 by Michael Butler

Published online 19 September 2020