Irum Rauf | Inna Vistbakka | Elena Troubitsyna

# Formal Verification of Stateful Services with REST APIs using Event-B

Turku Centre *for* Computer Science

TUCS Technical Report
No 1192, March 2018

# Formal Verification of Stateful Services with REST APIs using Event-B

Irum Rauf
Åbo Akademi University, Department of Computer Science, Turku, Finland
irum.rauf@abo.fi
Inna Vistbakka
Åbo Akademi University, Department of Computer Science, Turku, Finland
inna.vistbakka@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Computer Science, Turku, Finland
KTH, Sweden
elena.troubitsyna@abo.fi

TUCS Technical Report
No 1192, March 2018

**Abstract**

REST APIs are being increasingly used in the industry including their application in safety-critical applications and in the IoT world. They offer basic CRUD (create, retrieve, update, delete) interfaces. However, REST APIs can be used to build services with more advanced scenarios. Developing such services with REST constraints require rigorous approaches that are capable of creating services that can be trusted for their behavior. In this work, we present an approach based on formal verification technique for development of REST services and focus on deriving correct system architecture by refinement and on formal verification consistency of service design models. We use Event-B and its refinement approach for system development. We illustrate our approach on a Hotel Reservation System.

**Keywords:** REST API, Formal Verification, Event-B, Stateful Services, UML, Modeling.

**TUCS Laboratory**
Embedded Systems Laboratory (ESLAB)

# 1 Introduction

APIs (Application Programming Interface) are programmatic interfaces that offer public interfaces that can be used by other programmers to build applications on top of them. The simplicity of REST architectural style [11] has encouraged small and large enterprises to offer their service as REST APIs to offer highly consumable services [20].

REST aims at producing scalable and extensible services using technologies that play well with the existing tools and infrastructure of the web. It provides a uniform set of operation that can be used to invoke a CRUD interface (create, retrieve, update and delete) of a service. A programmer can use CRUD interface of REST API to develop an application that offers functionalities that go beyond CRUD. Automated systems, e.g., hotel reservation systems, provide advanced scenarios for stateful services that require a certain sequence of requests that must be followed in order to fulfill the service goals. Designing and developing such services for advanced scenarios with REST constraints of statelessness and extensibility require rigorous approaches that are capable of creating services that can be trusted for their behavior. Systems that can be trusted for their behavior can be termed as dependable systems. In addition, the use of API's in safety-critical applications [15] and in the IoT world [34], further motivates the use of rigorous development approaches to offer correct behavior throughout a service lifecycle. Unintentional design errors can lead to inconsistent designs leading to services implementations with unintended behavior.

In this paper, we present an approach based on formal verification technique for development of REST services that offer advanced and complex scenarios using REST APIs. In [24], we presented a model-based design approach that creates behavioral REST service interfaces by construction. The behavioral interfaces provide information on what methods can be invoked on a service and the pre- and post-conditions of these methods. In this work we particularly focus on deriving correct system architecture by refinement and on formal verification consistency of service design models. We use Event-B and its refinement approach for system development[2]. We have chosen Event-B to represent our design models to ensure scalability of the proposed framework and because it promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B also has a mature industrial-strength tool support – the Rodin platform [28]. In addition, our approach is supported by the partial code generation tool that creates code skeletons of REST services with method pre and post-conditions [23]. The partial code generation tool is implemented in Django web framework [12]. The details of the methods can be manually inserted by the developer as required. We do not target complete automation because we focus only on the interface aspects of the service.

We illustrate our approach on a Hotel Reservation System. We perform its formal development and while modelling the behavior of such a system we show how a refinement process allows us to derive its correct architecture that ensures required properties. Traditionally, such a verification is undertaken by abstracting implementation up to requirements level and model-checking satisfiability of goals. However, such an approach suffers from a state explosion that is especially prohibitive for such applications as multi-robotic systems [4]. The paper is organized as follows: section II gives necessary background for the methodology presented in section III. Section IV presents the design approach for creating stateful services. Section V presents formal development approach. Section VI and VII discuss related work and conclusions.

# 2 Background

## 2.1 REST APIs

REST APIs advocate stateless interaction between components, i.e., every request is independent of its previous request with no stored context on the server. This allows REST services to cater large number of clients resulting in system scalability since the provision of not having to store state between request allows the server to free resources rather quickly. This may affect the system as a design trade-off resulting in decreased network performance due to data repetitions. However, REST web services play well with existing infrastructure of the web that can help in improving efficiency of the network. REST is centralized around the concept of resources which are pieces of information that can be navigated through URIs. It offers the following main features: identifying resources with names, manipulating resources with a uniform interface, using hypermedia to link these resources and to use stateless interaction between client and server.

## 2.2 Stateful Services as "REST APIs"

Services can have different service states that a service must go through during its lifecycle. A stateful service requires a certain sequence of method invocations that must be followed in order to fulfill the functionality a service promises to deliver to its users. For example, in a room booking service, the booking cannot be paid until a booking is made. This requires that a booking must be made first and then it should be paid. If the user of the service does not follow this protocol, it cannot expect the desired results. In a stateful service, the result of a (side-effect) method is dependent on the current state of the service or resource (in case of REST service). A method invoked on a service or a resource, may return different results depending on the state of

the service or resource. For example, consider the case of a service that allows only canceled bookings to be deleted. In such a case, the result of invoking a method that deletes a booking, on a booking instance (or resource) that is canceled, is different from the results of invoking the same method on a booking instance (or resource) that is not canceled. In the first case the booking is deleted but in later case the booking is not deleted as it is still an active booking. A state of the service is thus defined as a specific condition of the service at a certain time instance.

A stateful protocol requires that the server can associate a request with the previous requests and knows that they all come from the same user. On the other hand, a stateless protocol treats each request independently and unrelated to the previous request. The REST APIs uses HTTP as a stateless protocol for communication between the server and the client.

However, REST services come with the property of transfering state of the application (service in our case) from one resource to another. REST does that by providing resource representation in a machine processable format like JSON, response code and links in the representation of the resources [35]. When an HTTP method (GET, PUT, POST, DELETE) is invoked on a resource, it returns its representation along with the HTTP response code. The response code tells whether the request went well or not [1]. For example, if the response code is 200, this means that the resource exists and if it is 404, this means that the resource does not exist. The links in the representation contain information on what further links should be addressed so that the sequence of method invocations is maintained and also the state of the service is preserved. Thus using a stateless HTTP protocol, services that give stateful behavior can be constructed in this manner.

However, for a designer, creating stateful services using a stateless protocol such that each resource is designed according to the state it is expected to be in, is an interesting design challenge since there is no provision of passing or maintaining hidden session information over a sequence of events. We address the creation of such services with advanced scenarios using REST APIs such that it facilitates the designer in making better design decisions.

## 2.3 Modelling and Refinement in Event-B

In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The important system properties to be preserved are defined as model invariants. A machine usually has the accompanying component, called context. A context may include user-defined carrier sets, constants and their properties (defined as model axioms).

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$\mathsf{event} \;\widehat{=}\; \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, $a$ is the list of local variables, and (the event *guard*) $G_e$ is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation $R_e$. The event guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional system behaviour. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, refine old events as well as replace abstract variables by their concrete counterparts.

The consistency of Event-B models – verification of model well-formedness, invariant preservation as well as correctness of refinement steps – is demonstrated by discharging the relevant proof obligations. The Rodin platform [28] provides tool support for modelling and verification. In particular, it automatically generates all required proof obligations and attempts to discharge them. When the proof obligations cannot be discharged automatically, the user can attempt to prove them interactively using a collection of available proof tactics. Moreover, a user can also rely on verification by model checking supported by ProB plug-in.

# 3 Methodology

Our method is divided into four phases. Figure 1 gives an overview of the method. The colored boxes represent the phases addressed in this paper supported by other phases, details of which are presented in our previous works ([24][23]).

## 3.1 Design

In the first step, we provide an approach to design REST services and their behavioral interfaces in UML [24]. It is briefly presented in section 4. A REST API only provides information about the methods that can be invoked on it along with details of how to use it in text format in some cases. Our design approach provides a behavioral interface for services that can constrain the service user to invoke the service under right condition and

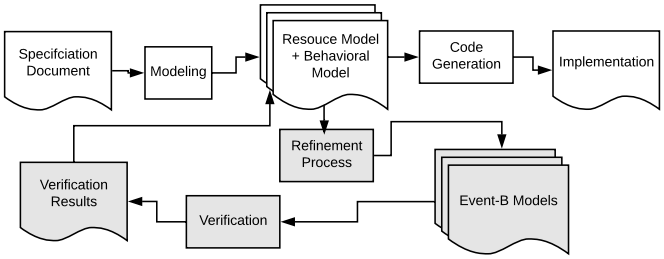


Figure 1: Our Approach to Developing Dependable System

also constraints the service developer to implement the right functionality by defining pre-conditions and post-conditions.

## 3.2 Development by Refinement

In this step, we perform a gradual development of the REST services by refinement in Event-B. With Event-B it is possible to develop a system which can be proved "correct" i.e, a system whose implementation is guaranteed to maintain the invariant stated at the most abstract level of its specification.

## 3.3 Formal Verification

During this phase, we verify the consistency of service design models with REST constraints. Moreover, we also check our model on deadlock freeness and verify the associated with a system safety properties.

## 3.4 Code-Generation Tool

The fourth phase is supported by our partial code-generation tool [23] that generates behavioral interface as code-skeletons implemented in Django web framework [13]. The inconsistency problems and other design errors revealed during Event-B refinement and formal verification phases are reported back to models. The design models are fixed based on these results. This is an important step, since the behavioral interfaces generated from the models have several implication in later stages of development. It is therefore critical that these interface skeletons are generated from verified design models.

The *correct* models are used to generates code skeletons with the pre- and postconditions for the service methods. An automated process that can create behavioral interface skeletons of REST services can facilitate the service developer in the creation of REST services in an automated manner. These behavioral interface skeletons have different applications. In [22], we present our approach to validate service implementation and provide conformance testing of service compositions using behavioral interface skeletons

generated using our code-generation tool and design models. The generated code skeletons can also be can also be added as a proxy interface to already developed and deployed services to monitor their functioning. This facilitates location of the fault in an application by observing the conditions that are not being met and by which methods. The application of our approach as cloud monitors [26] and for securing open source software [25], further motivates the need for rigorous and formal development approaches to offer correct service behavior throughout its lifecycle.

In order to demonstrate our design and refinement approach, we use Hotel Booking (HB) service that is implemented in REST architectural style. The HB service offers its users a service to book a room and pays for it. It consists of following scenarios:

- **Booking:** A customer can book a room in a hotel by accessing the booking service.
- **Payment:** If the user pays for the booking then the service processes the payment by invoking a third party payment service and waits for the confirmation. If the payment is not confirmed for 2 hours, the booking is considered unpaid implying a network problem.
- **Confirmation:** The service marks the booking as confirmed once the payment confirmation is received.
- **Cancel:** A booking can be canceled anytime except when it is processing the payment.
- **Refund:** A canceled booking that has been paid should be refunded.
- **Delete:** An inactive booking can only be deleted if it is unpaid or has been refunded.

In the next sections, we will demonstrate how the design models are constructed for such a service using step-wise development in Event-B.

# 4 Designing Stateful Services with Stateless Protocol

## 4.1 Service States with REST APIs

All REST services are stateful by nature since CRUD (create, retrieve, update and delete) methods can be called on every resource to change or retrieve their state. However, from the developer's perspective, when REST APIs are used in advanced scenarios, it may become a challenge to design them in a

consistent and verifiable manner, since more advanced the scenarios, the more careful design efforts are needed to communicate the right information to the right users.

For example, the response of invoking a PUT on cancel resource for an unpaid booking is different from the response received by invoking PUT on cancel resource of a paid and confirmed booking. The former will cancel the booking, give links to rebook the room and can also be deleted if required, however in the latter case, the response would provide links to get the payment refunded or autonomously initiate a payment refund service giving links to either browse elsewhere while waiting for the payment confirmation or give a confirmation response (depending on the design of the service) and in addition, it requires that a canceled booking that has been paid cannot be deleted until it is refunded. In both the cases, and other similar scenarios, the resources need to be carefully designed so that they transfer the right state of the service, i.e., service state. We define service state as a predicate over resources.

## 4.2 Design Approach

In this section we briefly explain how the REST compliant design models are constructed. These models provide a graphical representation of the service specifications, include all the information required to build behavioral interfaces for REST APIs and can be comprehended and communicated with relative ease among different stakeholders. We use UML (Unified Modeling Language) [33] which is well accepted in the industry and academia and has many well-known and mature tools with a wide user base. Also, it can target design requirements independently of the implementation details.

The concept of a *resource* is central to Resource Oriented Architecture (ROA). ROA is a structural design that fulfills design criteria presented by REST [27]. A resource is something that can be referred to and can have an address. Any important information in a service interface is exposed as a resource.The standard HTTP methods (e.g. GET, PUT, POST, DELETE) methods can be invoked on these resources that may cause state change in resources. We model these and other properties of ROA with UML.

The starting point of our approach is an informal service specification in natural language that is used to build the resource and behavioral model of the service. We use UML class diagram to represent the resource model and UML protocol state machine with state invariants to represent the behavioral model of a REST service.

### 4.2.1 Resource Model

The resource model of HB service is shown in Figure 2. The resource model represents the resources of the service along with their data attributes, links between them and the different properties of these links. We are using UML class diagram with additional design constraints to represent resources, their properties, and relation to each other. We have used the term *resource definition* to define resource entity such that its instances are called resources. This is analogous to the relationship between a *class* and its *objects* in object-oriented paradigm. In our resource model, we represent *resource definitions* as classes. Each association has a name and minimum and maximum cardinalities. These cardinalities define the minimum and the maximum number of resources that can be part of the association. We require that every association must have a role name in order to form URI addresses.The attributes of classes must be public since the representation of a resource is available for manipulation and they must have a type since they represent a document containing information of the resource, i.e. an XML document or a JSON serialized object. For a detailed explanation of resource model, readers are referred to [24].

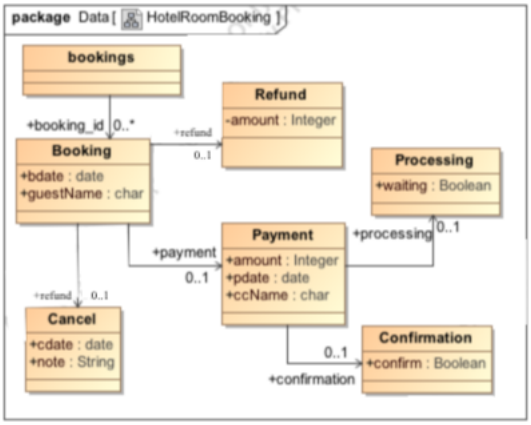


Figure 2: Hotel Booking Service: Resource Model

### 4.2.2 Behavioral Model

The behavioral model of the service, represented by a UML protocol state machine with state invariant, define different states of the service and show how these service states change when its interface methods are invoked. A UML state-machine has transitions that are triggered by method calls and each state has a *state invariant*. State invariant is a boolean condition that evaluates to true when the service is in that particular state. Otherwise, it evaluates to false.

The behavioral model of the HB service is presented in Figure 3. The parameters of method calls represent url for that resources, e.g. POST(booking) represents a POST method on the url of booking resource, i.e. */../booking/{booking_id}/*. The trigger methods in the behavioral model are restricted to HTTP methods PUT, POST and DELETE since these methods can make a change in the resource and GET method is used only to retrieve the state of the resource with no side-effects. We are able to define states of the service without compromising the requirements of stateless protocol by defining states as predicates over states of the resources. The representation of the resource, returned as a result of method invocation, is given by the attributes of resource defined in the resource model and the links that can be navigated further. These links are defined by observing the outgoing transitions from the target service state of the transition invoked by the interface method.

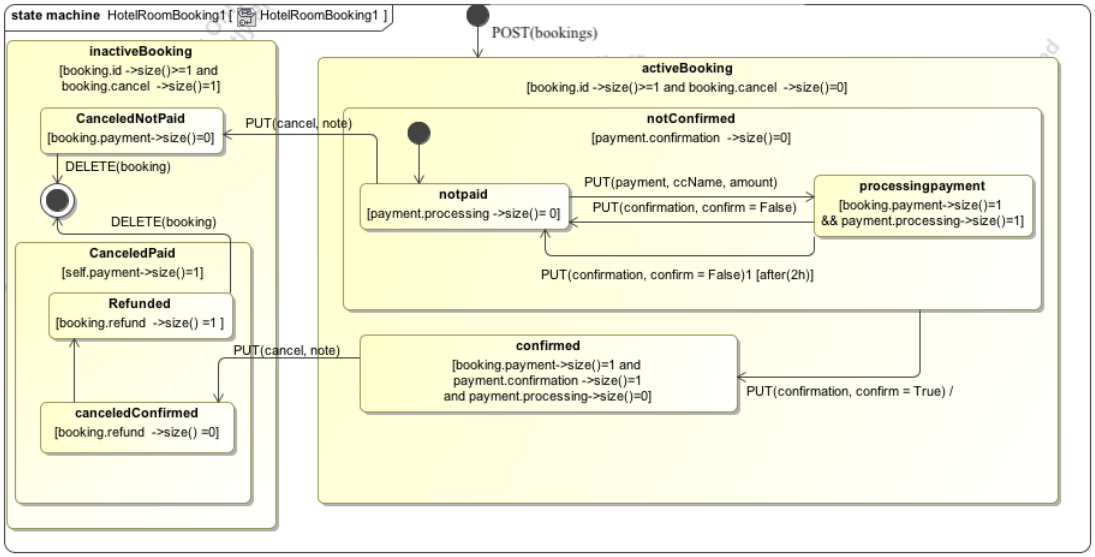


Figure 3: Hotel Booking Service: Behavioral Model

In Figure 3, a user can invoke a *booking* resource with a POST method to create a booking resource. Both the composite states *aciveBooking* and *inactiveBooking*, further consist of one composite state and one normal state, each. The state invariants of composite states are inherited by all its substates. We define the invariant of a state using OCL [19] as a boolean expression over addressable resources. In this way, the stateless nature of REST remains uncompromised since no hidden information about the state of the service is being kept between method calls.

In Figure 3, state invariant for state *activeBooking* is written as an OCL expression: $booking.id- > size() = 1 \quad and \quad booking.cancel- > size() = 0$. This implies that a particular booking exists and its cancel

resource does not exist. Here, $booking.cancel- > size() = 0$ implies that the response for invoking GET on *cancel* resource for the booking was not 200, meaning either the resource does not exist or is not reachable to infer anything about its state. Similarly, an OCL expression $booking.id- > size() = 1$ implies the response for invoking GET on booking resource was 200, meaning the resource exists. The state invariant: $booking.id- > size()=1 \quad and \quad booking.cancel- > size() = 0$ specify that initially a booking exists but it has not been canceled. We say that the state invariant in behavioral model for the REST API is defined as a predicate over its resources.

The behavioral model also provides information about interface method contracts. In our approach, a contract binds the service user to pose a valid request and constrains its provider to provide the correct behavior by asserting pre- and post-conditions of methods. For example, in order to invoke PUT on *Refund* resource, the invariant of its source states, i.e. $booking.id- > size() = 1 \quad and \quad booking.cancel- > size() = 1 \quad and \quad booking.payment- > size() = 1$ should be true, and its guard conditions, if any. Similary, for the method to be successful, its post-condition should be true,i.e., the state invariant of the target state of $PUT(refund)$ - $canceledConfirmed$, should be true and the state invariant of its source state should be false.

# 5 Formal Development and Verification of HB Service

Design and verification of REST services with advanced scenarios can be significantly facilitated by the use of formal model-based techniques. It allows the developers to build a system in a rigorous way and verify that the system specification meets the requirements. Unlike testing, formal techniques based on theorem proving allow us to ensure full coverage of possible system behaviours. In this section, we show our approach to use Event-B method and the associated Rodin platform to formally model and reason about correctness of design model of RESTful service. As a demonstration of the approach we use HB RESTful service.

We focus on analysing consistency of service design models with REST constraints. Each service within the HB service has a state invariant. As depicted in the behavioural model in Figure 3, for each state we define (local) invariants as predicates over resources. For a state to be active, its state invariant should be true, otherwise it should be false. When the client makes a service request, it is mapped to a transition to the behavioural model that has the method as a trigger. The transition is fired from a source state to target state. If the state invariant of source state is inconsistent, a service can never exists in this state and it would be impossible for implementation

of the interface to decide which transition to take as a result of a service request.

In this paper, we treat the state invariants which let the behavioural model behave against the UML superstructure specifications for state chart diagrams as inconsistent state invariants. In general, such inconsistency might cause whole system become unsatisfiable. Thus while modelling in Event-B, we will gradually unfold the system architecture and on every refinement step we formulate the consistency properties and prove that they are preserved within a system execution. Our development is supported by Event Refinement Structure (ERS) [10] approach that keeps the handling of refinement steps more simple and allows to get more automated proving. The ERS approach provides a tree-like graphical representation of the events, with an explicit representation of the events ordering and the refinement relationships.

## 5.1 Overview of the Development of HB service

In this section we highlight the main parts of the performed development and discuss verification results.

**Refinement Strategy.** The overall refinement structure of the HBS model is presented on Fig. 4. Our Event-B model consists of an abstract and six refinement levels as follows:

- HBS_M0: Models the creation of a booking resource and the possibility of its handling.
- HBS_M1: Refines the handling phase, introducing the activation and deactivation of a booking.
- HBS_M2: Introduces various outcomes of active booking processing (confirmed and unconfirmed case).
- HBS_M3: Models various outcomes of payment procedure (for both confirmed or unconfirmed case).
- HBS_M4: Refines payment model according to the system requirements.
- HBS_M5: Introduces the possibility to cancel a booking (for both confirmed and unconfirmed case) and delete this booking.
- HBS_M6: Refines cancellation model of a confirmed booking and introduces booking refund procedure.

In our modelling, we adopt such a granular refinement sequence because it is very important to show from the beginning that there are cases when

a booking handling deviates from its "normal" path and does not require the same handling process. The refinement strategy is also influenced by the ERS restriction that a combinator can only be applied to simple events and no combinator can be the direct parent of another combinator [9]. Such restriction promotes for smaller changes at each refinement level. Also, we take advantage of an idea of superstates to unfold the system behaviour. Namely, we start with global superstate and then atomise it to its corresponding substates according to UML behavioural diagram.

In the following subsection, we highlight some important and interesting steps in the development of the HB service demonstrating the application of the ERS combinators and their corresponding Event-B specifications.

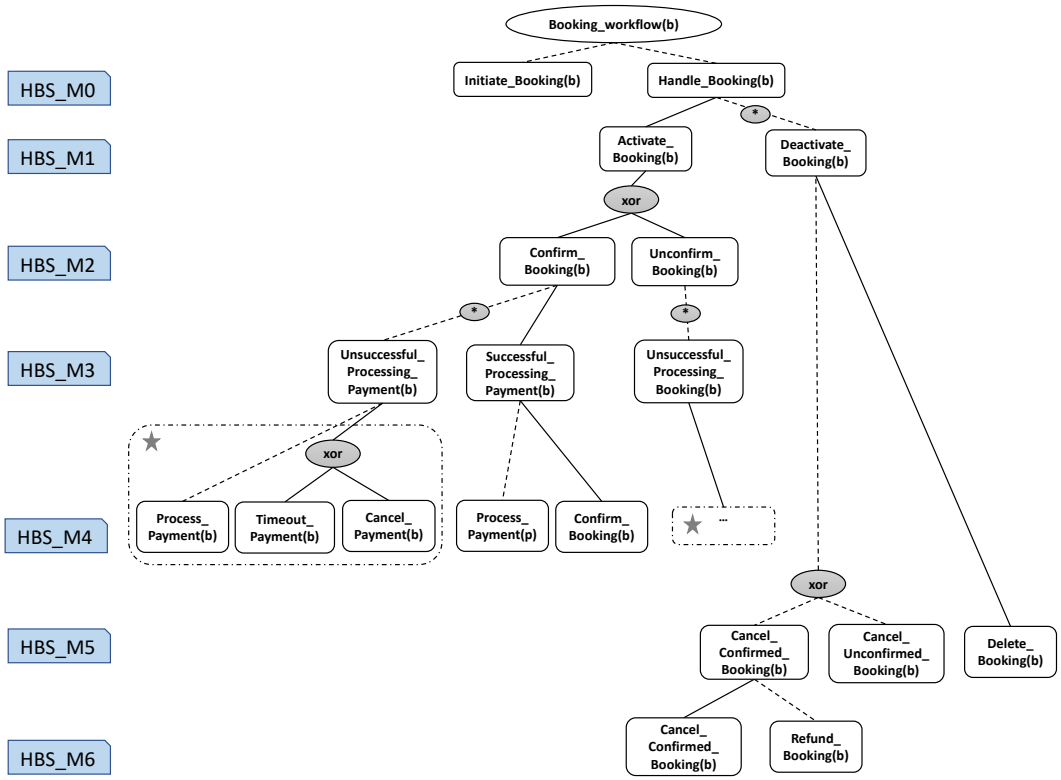


Figure 4: Overall ERS diagram of the booking workflow

## 5.2 Description of the Development

In a HB service model we apply multiple instances modelling to enable possibly simultaneous handling to different bookings. Multiple instance modelling is indicated by the addition of the parameter $b$, representing booking, to the root of the diagram (see Fig. 4). The root of the tree, *Booking_workflow(b)*, represents the name of the flow-diagram and the parameter $b$ indicates mul-

tiple instances. Multiple instances means different instances of a workflow may be executed in an interleaved manner.

At the abstract level, we distinguish between two cases of a booking processing, initiation of a booking and its further handling. The corresponding leaves *Initiate_Booking* and *Handle_Booking* will be transformed into events in the Event-B machine. The ordering of the leaf events is from *left-to-right*, so *Initiate_Booking* can execute first, followed by *Handle_Booking*. To describe the control-flow of the events using Event-B, variables with the same name of the leaf events are generated. We refer to these variables as control variables. Since we have multiple instances modelling the type of the control variables is a set. These control variables are used in the Event-B model to specify the control-flow of the events using invariants and guards. The initial Event-B model HBS_M0is presented in Fig.5.

**Machine** HBS_M0
**Variables** *Initiate_Booking*, *Handle_Booking*, *bookingState*, ...
**Invariants** $Initiate\_Booking \subseteq BOOKINGS \ \wedge Handle\_Booking \subseteq Initiate\_Booking \wedge$
$(\forall b.\ b \in Initiate\_Booking \Rightarrow bookingState(b) = 1) \wedge ...$
**Events**...
Initiate_Booking $\widehat{=}$
**any** $b$
**where** $b \in BOOKINGS \wedge b \notin Initiate\_Booking$
**then** $Initiate\_Booking := Initiate\_Booking \cup \{b\}$
$bookingState(b) := 1$
**end**
Handle_Booking$\widehat{=}$
**any** $b$
**where** $b \in Initiate\_Booking \wedge b \notin Handle\_Booking$
**then** $Handle\_Booking := Handle\_Booking \cup \{b\}$
**end**
**end**

Figure 5: Event-B model at the Abstract Level of the Booking Workflow

In ERS, the dashed lines indicate that the events are newly added events, and they are not refining events [10]. Therefore, the abstract level of Event-B, first row, can only have dashed lines. ERS also allows the addition of different combinators between events, represented within an oval shape. In Fig. 4, we used two different combinators, the *xor*-combinator and the *-constructor (loop-constructor).

In the first refinement HBS_M1, the event *Handle_Booking* is refined by *Activate_Booking* event. Here we also use the loop constructor to model zero or more executions of a leaf *Deactivate_Booking*.

Further, in the second refinement HBS_M2, the *xor*- combinator indicates the exclusive choice between events, in this case either *Confirm_Booking* or *Unconfirm_Booking* can execute, but not both. The abstract event *Active_Booking* is refined by two of them.

In the third refinement HBS_M3, the event *Confirm_Booking* is decomposed into the sequence of events *Unsuccessfull_Processing_Payment* (as a result of applying the *- combinator.) followed by the execution of the event

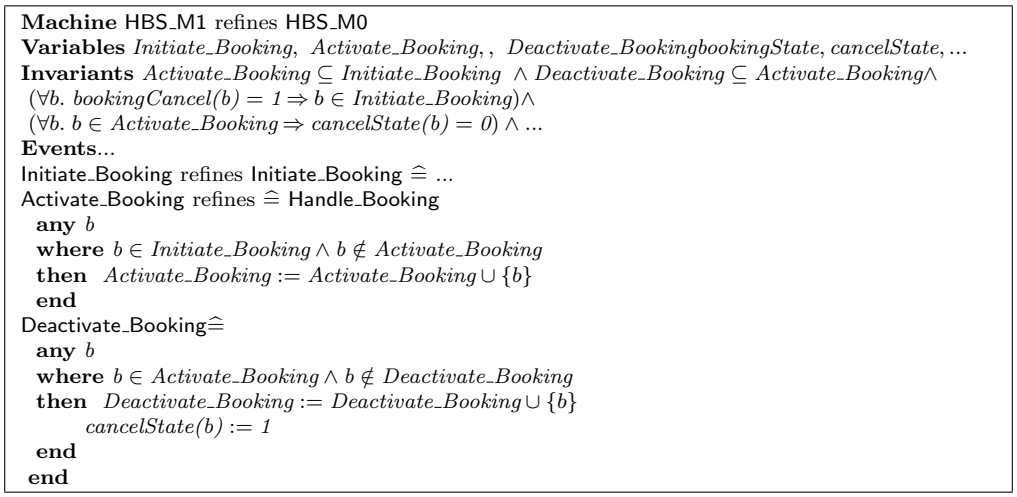


Figure 6: Event-B model at the First Refinement Level of the Booking Workflow

*Successfull_Processing_Payment*. Here again the loop *-constructor is used to model zero or more executions of a leaf. The event following the loop event (*Successfull_Processing_Payment*) can execute immediately after execution of the event preceding the loop event (*Initiate_Booking*). At this step, the event *Unconfirm_Booking* is refined as well.

Further, in the HBS_M4 refinement, the event *Unsuccessfull_Processing_Payment* is decomposed into the sequence of *Process_Payment*, and *Timeout_Payment* or *Cancel_Payment* events, as a result of applying the *xor*- combinator. This design refinement pattern for introducing payment outcomes is applied also to refine the event *Unsuccessfull_Processing_Payment* of the *Unconfirm_Booking* branch. Moreover, *Successfull_Processing_Payment* is decomposed into the sequence of events *Process_Payment* followed by *Confirm_Booking*, where *Successful_Processing* is the directly refining event *Activate_Booking* and *Unsuccessful_Processing* is a newly added event.

Further, in the last refinement steps, we unfold the system architecture by modelling different cases of booking cancellation and refine *Deactivate_Booking* and *Cancel_Confirmed_Booking* events, correspondingly.

Generating the Event-B elements from the ERS diagram is based on the generator framework which is part of the generic Diagram Extension framework [7]. Each rule transforming an ERS element to an Event-B element implements the rule and defines the methods enabled, and fire. The ERS plug-in provides a graphical environment for the ERS approach, and also supports the validation of the ERS diagrams. Applying the tool in modelling a complex case study resulted in having more consistent and systemic models, faster modelling and the graphical front-end made understanding and validating the model easier [9].

### 5.3 Formal Verification of HB service

During the formal development of HB service, on every refinement step, we identify the main system-level properties (e.g., consistency between the states and corresponding superstate) and demonstrate how to formally specify and verify them as a part of the refinement process. Naturally, formal modelling allows us to identify situations, where the desired properties can be violated (in the case, if the behavioural model contains any design errors). For instance, we formulate and prove the following consistency properties:

$$\forall\, b.\ b \in \mathit{Active\_Booking} \Rightarrow \mathit{bookingState}(b) = 1 \ \wedge\ \mathit{cancelState}(b) = 0,$$
$$\forall\, b.\ b \in \mathit{Deactivate\_Booking} \Rightarrow \mathit{bookingState}(b) = 1 \ \wedge\ \mathit{cancelState}(b) = 1,$$
$$\forall\, b.\ b \in \mathit{Unconfirm\_Booking} \Rightarrow \mathit{bookingState}(b) = 1 \wedge \mathit{cancelState}(b) = 0 \ \wedge\ \mathit{confirmationState(b)} = 0$$
$$\forall\, b.\ b \in \mathit{Successful\_Payment} \Rightarrow \mathit{paymentState}(b) = 1 \ \wedge\ \mathit{paymentProcessingState}(b) = 1.$$

In general, the incorrect design errors in specification might prevent the system in achieving its goals and jeopardise such essential property as safety. Then while deriving a design models, we should ensure that the system behaves safely, i.e., safety properties are not violated while its execution. Development in Event-B allows us to formulate and verify a number of the safety properties associated with HB service. They are defined and proved as system invariants. For example,

$$\forall\, b.\ b \in \mathit{Cancel\_Confirmed\_Booking} \Rightarrow \mathit{refundState}(b) = 1 \ \wedge\ \mathit{paymentConfirmationState}(b) = 1.$$

This property says that if the booking has been refunded, it should been paid before (i.e., only paid bookings will be refunded).

The HB service is an example of non-terminating system. Thus we need to guarantee that there is always an enabled event in our resulting Event-B specification, i.e. that the system is *deadlock free*. As a part of our verification process, we formulate and prove that the system never reaches the state from that it cannot perform a transition. For deadlock-freeness, we build the disjunction of the guards of all Event-B events other than INITIALISATION and we prove it is a theorem. Thus, we can assure that one event at least can always been fired.

## 6 Related Work

The formal modeling and verification of web services and their compositions has been studied by number of researchers in the past. In [32], Beek et.

al present an overview of different approches that specify web service compositions using formal methods and XML-based standards to formalize the specification of web services and their compositions.

For instance, the problem of inadequate support for development of cloud services has been identified by Boer et al. [6]. They propose to explicitly represent the notion of resources into the abstract behaviour specification language. They aim at integrating reasoning about correctness with simulation. In our case, Event-B allows us to formally specify and verify system-level properties, while in [6] the stress is put on creating executable specifications and analysis of corresponding traces.

In [3], the authors present a formal approach based on refinement using the Event-B method that defines a compensation mechanism to repair failed services at runtime. So far, we do not consider any fault tolerance aspects in our service executions. However, to extend our work we can rely on the continuous-time probabilistic extension of the Event-B framework presented in [31].

In our previous work [30], we use Event-B framework to reason about reconfigurable service-oriented systems. We define the design rules for stepwise development of a dynamically reconfigurable system in Event-B and demonstrate how to formally assess the chosen reconfiguration strategy as well as evaluate whether the incorporated fault tolerance mechanism fulfils the reliability and performance objectives.

In the area of model checking, there is vast majority of approaches the use BPEL or OWL-S[18] for the specification of the web service composition. The work of Huang et al. [14] automatically translate OWL-S specification of composite web service into a C-like specification language PDDL which can be processed with the BLAST model checker.

There is also an extensive body of knowlege that uses UPPAAL model-checker[5] in verifying properties for web services, e.g., works in [8] and [16]. These works mainly use BPEL specifications as a reference specification, translate them to UPPAAL timed automata(UPTA) and use UPPAAL model checker to verify and validate web serivces and their compositions with different strategies.

These and other works [32] that focus on BPEL4WS processes, OWL-S and UPTA are dependent on specific execution languages for SOAP based services.

In our previous work [21], we verify the inconsistency issues in the structural specification of REST interfaces using OWL2. The approach used ontology reasoners to reason only about the structural specifications of services. In [22], we verified and validated service compositions using UPPAAL and UPPAAL-Tron [17]. Compared to [22], our work in this paper addresses the state-explosion problem with UPPAAL, detect inconsisteny problems in behavioral interface speciations, verify different model checking properties and

develop the system with different perspective using step-wise refinenment of Event-B.

In [29], Snook and Butler present a UML profile that can be used to support model refinement through specialization of UML entities. Our modeling differs in that we use standard UML without the need of a profile that represents REST constraints such that REST interfaces can be generated directly from UML models. In addition, our work uses OCL to define state invariants based on resource states, which is not the case in UML-B.

# 7 Conclusion

Formal techniques are successfully applied in development and verification of complex dependable systems. REST APIs are being increasingly used in industry and have usage in a wide range of applications. Developing REST services that offer more than CRUD behavior with a large number of resources has become an interesting design challenge. In this paper, we explore the usage of formal application of Event-B on the REST architectural style. Our approach successfully addresses inconsistency design issue, model checking of service specifications and the state-explosion problem that may arise due to a large number of resources. In future, we plan to apply our approach to larger case studies and consider fault tolerance aspects in web service executions.

# References


[1] Status Code Definitions. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Accessed: 05.03.2018.

[2] J.-R. Abrial. *Modeling in Event-B.* Cambridge University Press, 2010.

[3] Guillaume Babin, , et al. Web service compensation at runtime: Formal modeling and verification using the event-b refinement and proof based formal method. *IEEE Trans. Services Computing*, 10(1):107–120, 2017.

[4] Ralph-Johan Back and Kaisa Sere. From action systems to modular systems. In *International Symposium of Formal Methods Europe*, pages 1–25. Springer, 1994.

[5] Gerd Behrmann et al. UPPAAL 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[6] Frank S. Boer et al. Formal modeling of resource management for cloud architectures: An industrial case study. In *Service-Oriented and Cloud Computing*, LNCS 7592, pages 91–106. Springer, 2012.

[7] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods, 7th International Conference, IFM 2009*, pages 20–38. Springer Heidelberg, 2009.

[8] M Emilia Cambronero et al. Validation and verification of Web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[9] Dana Dghaym et al. Extending ERS for modelling dynamic workflows in event-b. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017*, pages 20–29, 2017.

[10] Asieh Salehi Fathabadi et al. Language and tool support for event refinement structures in event-b. *Formal Asp. Comput.*, 27(3):499–523, 2015.

[11] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, 2000.

[12] A. Holovaty and J. Kaplan-Moss. The django book. *Online version of The Django Book*, 2010. http://docs.djangoproject.com/en/1.2/.

[13] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right.* Apress, 2009.

[14] Hai Huang et al. Automated model checking and testing for composite web services. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 300–307. IEEE, 2005.

[15] Dongwoo Kim et al. Model-based api-call constraint checking for automotive control software. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 217–224. IEEE, 2016.

[16] Mounir Lallali et al. Automatic timed test case generation for web services composition. In *IEEE Sixth European Conference on Web Services (ECOWS)*, pages 53–62. IEEE, 2008.

[17] Kim G Larsen et al. UPPAAL TRON user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*, 2009.

[18] David Martin et al. OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[19] OMG. *OCL, OMG Available Specification, Version 2.0*, 2006.

[20] Cesare Pautasso et al. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[21] Irum Rauf et al. Analyzing consistency of behavioral rest web service interfaces. In Josep Silva and Francesco Tiezzi, editors, *The 8th International Workshop on Automated Specification and Verification of Web Systems*, Electronic Proceedings in Theoretical Computer Science, pages 1–15. EPTCS, 2012.

[22] Irum Rauf et al. Scenario-based design and validation of rest web service compositions. In *International Conference on Web Information Systems and Technologies*, pages 145–160. Springer, 2014.

[23] Irum Rauf and Ivan Porres. Beyond crud. In *REST: From Research to Practice*, pages 117–135. Springer, 2011.

[24] Irum Rauf and Ivan Porres. Designing level 3 behavioral restful web service interfaces. *ACM SIGAPP Applied Computing Review*, 11(3):19–31, 2011.

[25] Irum Rauf and Elena Troubitsyna. Towards a model-driven security assurance of open source components. In *International Workshop on Software Engineering for Resilient Systems*, pages 65–80. Springer, 2017.

[26] Irum Rauf and Elena Troubitsyna. Generating Cloud Monitors from Models to Secure Clouds. *Accepted for publication in IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

[27] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, 2008.

[28] Rodin. Event-B platform: online at `http://www.event-b.org/`.

[29] Colin Snook and Michael Butler. Uml-b: Formal modeling and design aided by uml. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[30] A. Tarasyuk et al. Formal development and assessment of a reconfigurable on-board satellite system. In *Computer Safety, Reliability, and Security - 31st International Conference, SAFECOMP 2012*, volume 7612, pages 210–222. Springer, 2012.

[31] A. Tarasyuk et al. Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In *IFM 2012*, pages 237–252. Springer, 2012.

[32] Maurice Ter Beek et al. Web service composition approaches: From industrial standards to formal methods. In *Internet and Web Applications and Services, 2007. ICIW'07. Second International Conference on*, pages 15–15. IEEE, 2007.

[33] OMG Uml. 2.0 superstructure specification. *OMG, Needham*, 2004.

[34] Maja Vukovic. Internet programmable iot: On the role of apis in iot: The internet of things (ubiquity symposium). *Ubiquity*, 2015(November):3, 2015.

[35] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 2010.

# Appendix

## Event-B Development

---

Turku
Centre *for*
Computer
Science

Vesilinnantie 3, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-3688-4
ISSN 1239-1891