An Integrated Approach to Design and Validate REST Web Service Compositions

Rauf, Irum; Siavashi, Faezeh; Truscan, Dragos; Porres Paltor, Ivan

Published: 01/01/2013

Document Version
Final published version

Document License
Publisher rights policy

Irum Rauf | Faezeh Siavashi | Dragos Truscan | Ivan Porres

An Integrated Approach for Designing and Validating REST Web Service Composition

Turku Centre *for* Computer Science

TUCS Technical Report
No 1097, December 2013

An Integrated Approach for Designing and Validating REST Web Service Composition

Irum Rauf
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
irum.rauf@abo.fi

Faezeh Siavashi
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
faezeh.siavashi@abo.fi

Dragos Truscan
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
dragos.truscan@abo.fi

Ivan Porres
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
ivan.porres@abo.fi

TUCS Technical Report
No 1097, December 2013

Abstract

We present an integrated approach to design and validate RESTful composite web services. We use the Unified Modeling Language (UML) to specify the requirements, behavior and published resources of each web service. In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We show how to transform service specifications into UPPAAL timed automata for verification and test generation. The service requirements are propagated to the UPPAAL timed automata during the transformation. Their reachability is verified in UPPAAL and they are used as test goals during test generation. We validate our approach with a case study of a holiday booking web service.

Keywords: RESTful, Validation, UPPAAL

1 Introduction

The REST architectural style was introduced by Roy Fielding in 2000 [10] as an effort to simplify the development of scalable web services. Although REST is traditionally used to develop simple CRUD (create, retrieve, update and delete) services, it is possible to create and compose REST web services that offer complex services and stateful behavior. Also, REST web service compositions may also offer time critical behavior. With the use of web services in businesses and critical applications, there is an increasing need for design approaches to develop web service compositions that support complex scenarios and timed behavior while complying with the REST architectural style and for testing and validation approaches to detect faults in specifications and implementations of such services effectively and efficiently.

A web service composition is developed to provide a new service that fulfills specific business goals. It relies on the functionality of different web services termed as partner web services. Thus, it is important to ensure that the services taking part in the composition process deliver the right functionality collectively. This is not a trivial task. A REST composite web service (CWS) can do more than simply offering a CRUD interface. It may switch to different service states during its lifecycle to fulfill a complex scenario. The service can offer different behavior at different times depending on its current state.

In this paper, we present an integrated approach to design and validate RESTful composite web services. We use Unified Modeling Language (UML) [31] to model our service specifications via an extension of our previous work in [26]. We use UML, since it is widely used by the software industry and it targets design requirements independently of the implementation details. In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style.

The UML-based service design models represent the system graphically and are comprehensible for a human user, however, in order to validate and verify their dynamic properties we suggest a set of reversible mechanized steps for translating UML specifications into UPPAAL timed automata (UPTA) [22]. We show how different properties of the composition such as reachability, liveness and safety can be verified in the UPPAAL model-checker tool. If problems are detected, they are addressed in UPPAAL and the models are transformed back to UML. Performing the verification of the WSC in a model-checking tool allows us to increase the quality of the specifications before proceeding to the implementation of the WSC. In addition, once the verification and implementation of the WSC is completed successfully, we validate the implementation of the service composition by model-based conformance testing using the UPPAAL TRON tool [22]. The composite service design models are updated based on the results of the verification process and a skeleton of the implementation is generated automatically. The implementation targets the Django web development framework. [15].

Requirements traceability is an important component of our approach. The requirements of the composition are included in the UML specifications and then propagated to *UPPAAL Timed Automata* (UPTA). They are used for both verifying the reachability of those model elements implementing them and for reasoning about their coverage after the tests are executed. Upon detecting failures tests the traced requirements can be used to identify the error in either models or in the specifications.

We exemplify and validate our integrated approach with a relatively complex case study of a holiday booking system web service.

We use the Unified Modeling Language (UML) [31] to model the structural and behavior properties of the behavioral interface of a stateful REST composite web services, based on our previous work to design behavioral interface of REST web services [26]. A behavioral interface specifies not only the allowed methods for a web service but also the sequence of method invocations and the conditions under which these methods should be invoked and their expected results. We use UML to model our service specifications since UML is industrially well-accepted and it targets design requirements independently of the implementation details. These service design models represent the system graphically and are comprehendable for a human user, however, in order to reason the design specifications and to make them understandable for verification tools, these design models are transformed to an intermediate format. We transform our composite web service design models into UPPAAL timed automata (UPTA).

The paper is organized as follow: Section II gives an overview of our approach and the tool support for the approach is discussed in section III. The case study is presented in section IV and the validation of the approach is presented in section V. The related work is discussed in section VI and the section VII concludes the paper.

2 Our Approach

An overview of our integrated design and validation approach is given in Figure 1. The left side of the figure shows our previous work, whereas the right side shows the contribution of this paper and how it is connected to the previous work.

In our previous work (Figure 1–right), we designed behavioral interfaces for web services that were RESTful by construction [26] . These design models were implemented in Django webframework using our partial code generation tool [29] that generated code skeletons with pre- and postconditions for every service method. The design models were also analyzed for their consistency [28].

In this work (Figure 1–right), we show how to design the behavioral interface of a service composition and provide both static and dynamic validation of a composite REST web service. Previously, we modeled composite REST web service with UML behavioral state machines using entry actions for invoking partner

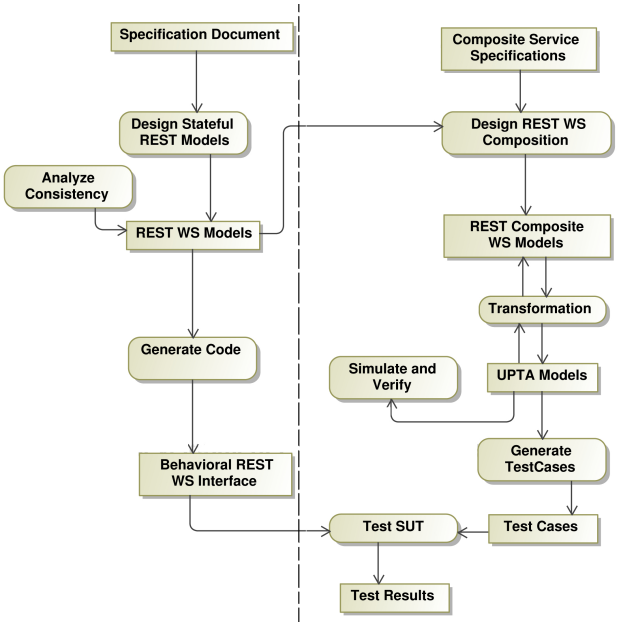

Figure 1: Flow Diagram of V&V Approach for REST CWS

services [27]. In this work, we use UML protocol state machines since we consider that UML protocol state machines are useful to describe the interface and intended behavior of a service, while omitting implementation details. We show method invocations as effects on the transition instead of entry actions of the state. In addition, we use a domain model to represent interfaces between composite web service and the partner services. The behavioral models of composite web service are transformed to UPTA, simulated and verified. Once the models are verified, and implemented, we generate test cases from UPPAAL models and test the implementation for its behavior. In addition, we added requirements from specification document to our behavioral model.

2.1 Design Models

We require that a composite REST web service interface should exhibit the REST interface features, i.e. addressability, connectivity, statelessness and uniform interface. We have modeled our composite REST web service interfaces with a *resource model*, a *behavioral model* and a *domain model* that exhibit these features.

This work extends our previous work on creating behavioral interface specifications of individual REST web services [26].For the details of the design approach, readers are referred to [26].

A resource model represents the static structure of the service and is modeled with a UML class diagram. Each class represents a *resource definition*. 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.

The direction of association between *resource definitions* gives the navigability direction between them and their role names give the relative URI of its resources (addressability). The *collection resource definiton* without the incoming transitions is termed as *root* with the requirement that every *resource definition* in resource model should be reachable via *root* and the graph formed should be connected (connectivity).

A behavioral model represents the dynamic structure of the service and is modeled by a UML state machine. Each state represents the service state and the trigger methods of transitions are restricted to the side-effect methods of HTTP, i.e., PUT, POST and DELETE (uniform interface). The statelessness feature of REST interface is not compromised while building stateful REST web service by defining state invariants as boolean expression of states of different resources. The state of a resource is given by its representation retrieved by invoking a GET on it. We are thus able to define service states as predicates over its resources without maintaining any hidden session or state information (statelessness). The state invariants in behavioral model are written as Object Constrain Language (OCL) expressions. OCL is commonly used to define constraints in UML models, including state invariants [5].

For modeling a service composition, the models are required to represent method invocations on the partner services. The service invocations to partner services are modeled as effects on the transitions. The composite web service requirements, inferred from the specification document, are added as comments on transitions that satisfy them. These requirements should be met by the implementation of the service in order to fulfill the service goals.

The *domain model* of the composite service is represented with a class diagram. It represents interfaces between the composite service and its partner services. The required and provided interface methods between the composite and its partner services are modeled with required and provided interfaces in the domain model, respectively.

2.2 Verification

Verification of models is a process of determining whether the models are designed correctly and represents the developer's conceptual descriptions and specifications. Model checking is one of the ways to exhaustively check the models automatically. The service design models of composite REST web service should be verified for their basic properties in order to build confidence of the developer on the models before implementing them. By verifying the models before implementing them we can get rid of design errors that can be expensive to detect and correct at later stage of the development cycle.

UPPAAL model checker is used for modeling, simulation and verification of

real-time systems [21]. It consists of set of timed automata, clocks, channels that synchronize the systems (automata), variables and additional elements. A real-time system is modeled as a closed network of timed automata. Each automaton in the network is specified via a template. These templates can be instantiated as processes. A template in UPTA is composed of locations, edges, clocks and variables representing all properties of the system. Synchronization between different processes can be provided using channels. Two edges in different automata can synchronize if one is emitting (denoted as *channel_name!*) and the other is receiving (denoted as *channel_name?*) on the same channel. *Guards* are the conditions on transitions that specify that this transition is fired only if its corresponding guard is satisfiable. Similarly, the conditions associated to states, called *invariant*, specify that the system can be in this state only if its invariant is satisfiable.

The query language used in UPPAAL is a simplified version of TCTL [3] that consists of path formulae and state formulae. Path formulae can be classified into reachability, safety and liveness properties and the deadlock is expressed using state formula. The syntax of TCTL used for UPPAAL is defined as follows:

- A $\Box$ φ - for all automata's paths, the property φ is always satisfiable.
- A $\Diamond$ φ- for all automata's paths, the property φ is eventually satisfiable.
- E $\Box$ φ - there is atleast a path in the automata such that property φ is always satisfiable.
- E$\Diamond$ φ - there is atleast a path in the automata such that property φ is eventually satisfiable.
- $\varphi \rightsquigarrow \phi$ - when φ holds, ϕ must hold.

Using these properties, UPPAAL can verify deadlock, reachability, safety and liveness properties in UPTA of service design models. If there is a state in the model that has no outgoing transition, then the model is said to be in a deadlock. Reachability properties validate the basic behavior of the model by checking whether a certain property is possible in the model with the given paths. The safety property checks that something bad will never happen and the liveness property is verified to determine that something will eventually happen.

However, before using UPPAAL model checker to verify these properties we need to give our service design models in UML formal foundations that are understandable by the verification tool. This has to be done in an automated manner to avoid extra efforts from the service developer. In section 3, we give our tool support for this and explain in detail the transformation from UML to UPTA.

2.3 Requirements Traceability

Service requirements can be inferred from the specification document and they serve as service goals. A service should be checked for it service goals in order

to validate that the service does what it is required to do. By catering service service requirements at the design phase and propagating them to the validation stage we provide a mechanism by which a service requirement can be validated for its goals and the unfulfilled requirements can be traced back to the design phase to find faults in the design. Service requirements are generally domain specific since they are inferred from the specification document. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the behavioral model as comments on the transitions and are propagated in UPTA such that the links between requirements and the model elements is preserved. These requirements are included in all the models and traced throughout the process, i.e., at UML, UPTA and test level, respectively.

The requirements are formulated as reachability properties in UPTA. Each requirement label is translated into a boolean variable (initialized to False) and attached to the corresponding edge in the UPTA. This mapping is explained in more detail in the Section 3 on the UML to UPTA transformation.

We require that our testing approach must validate that these requirements are met by IUT, in order to build confidence of the developer that the system is doing what it is required to do. Thus, they are used as test target during test generation. Once the test report is available, we can check which requirements have been validated and which have failed.

2.4 Model-Based Test Generation

Model-based testing (MBT) is a method that provides an abstract model of a system under test (SUT) and preforms automatic test case generation based on the specifications of the SUT [32]. In MBT, modeling the environment of a system is important since the environment generates test cases from whole or some parts of the model to satisfy the test criteria. Environment models help in automation of testing in three ways: the automation of test case generation from a simulated environment, the selection of test cases, and the evaluation of their test results. Our UML to UPTA transformation tool generates UPTA of behavioral model and also creates environment model in UPTA.

We provide automatic test generation using UPPAAL TRON, which is an extension of UPPAAL for online model-based black-box conformance testing [22]. The environment model in UPTA is used to generate different test suites. Currently, we provide the following untimed coverage criteria with our environment model: *edge coverage*, *edge-pair coverage* and *requirements coverage*. During test generation, the environment model randomly selects test cases and communicates to the test adapter.

A test adapter is used by UPPAAL TRON to expose the observable I/O communication between the test environment model and the SUT model, as shown in Figure 2. Our adapter implements the communication with the SUT by convert-

ing abstract test inputs into HTTP request messages and HTTP response messages into abstract test outputs. The TRON tool generates tests via symbolic execution of the specifications using randomized choice of inputs. Based on the timed sequence of input actions from the simulation, the adapter preforms input actions to IUT and waits for the response. Output from IUT is monitored and generated as output actions for the simulation. The conformance testing is achieved by comparing outputs of IUT to the behavior of the simulation.

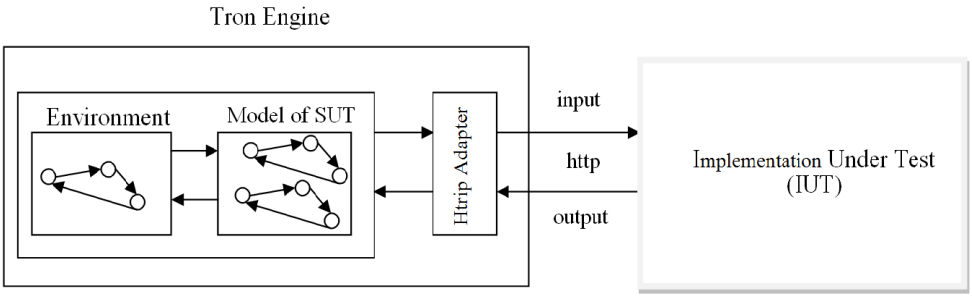

Figure 2: UPPAAL TRON test setup

3 Tool Support

Modeling in UML. The design models are modeled using MagicDraw [2]. Static validation of models is done via OCL using the validation engine of Magic Draw. We rely on predefined validation suites for UML contained in MagicDraw for the basic validation of the model. These validation suites contain rules that check that the designed UML model conforms to UML metamodel specifications and prevent the developer from creating incorrect models.

Code generation. The code-skeleton of the updated service design models of REST composite web service can be generated using our tool presented in [29]. The tool generates code skeleton for design models in Django that is a high level python web framework [15]. The generated code also has behavioral information such that the pre and post conditions for each method are included and the developer just has to write the implementation of the operations.

UML→UPTA transformation. The transformation from UML design models to UPTA is an extension of our approach presented in [24]. The extension of transformation generates several artifacts: UPTA model, test environment model and a skeleton for test adapter depicting the testable interfaces of the composite web service.

Resource Model: The resource model is represented as a template. The name of the resource model is mapped to template name. The resource definitions in the resource model are specified as variables with 1 or 0 value, specifying if a resource exists or not, respectively.

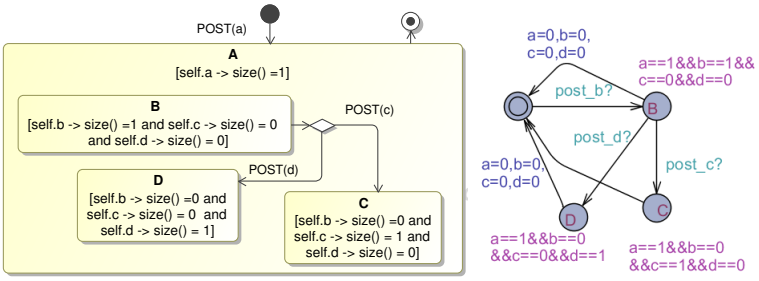

Figure 3: (Left) Composite State in Behavioral Model. (Right) Flattened locations in UPTA

Resource Attributes The attributes of resource definitions are inspected and for each integer attribute, an integer variable is declared in the UPPAAL model. Similarly, the boolean attributes are declared as integer array of 0 and 1.

Domain Model The domain model shows set of operations offered and required by composite and its partner web services. The corresponding communication between templates in UPPAAL is represented by channel synchronizations. Two edges in different automata in UPPAAL can synchronize if one is emitting and the other is receiving on the same channel. The operations in an interface are thus translated into a binary synchronization channel in UPPAAL. The template of the service that realizes the interfaces acts as the receiving automaton and the sending automaton is specified by the template of the service that uses the interface. In addition, a list of interface entries is created in the test adapter according to the interfaces between IUT and its environment.

Behavioral Model: The behavioral model of REST web service is encoded by timed automaton that are represented by templates. These templates can be instantiated as processes. This is similar to the behavior of objects in object oriented paradigm where state machine of a class is instantiated for objects of the class. Figure 3 shows an example of transformation from the behavioral model to UPTA.

States: A state in behavioral model is mapped to a location in UPTA. Similarly, its state invariant is mapped to corresponding location invariant. The subclauses of state invariants are translated to the variables corresponding to the respective resource definition. For example, in Figure 3, $self.a- > size() = 1$ is translated as $a = 1$ and $self.b- > size() = 0$ as $b = 0$. The initial state corresponds to the initial location. The final states are translated by having an edge from the corresponding location to initial location and updating all the variables to their initial values, as shown in Figure 3. The choice state in behavioral model is replaced by two edges in UPTA that are originating from the same source location to different target locations.

State Hierarchy: The behavioral model may contain composite states for better representation of specifications. UPTA, however, does not support the notion of

location hierarchy. We flatten the composite states to several simple states by including the state invariant of super states in the contained states that are then mapped to the respective locations in UPTA. For example, in Figure 3, the top figure contains a behavioral model with a composite state that is flattened to UPTA model shown at the bottom. States B, C and D in the behavioral model correspond to the locations B, C and D of UPTA, respectively. Note that all the locations contain the state invariant of superstate A in the behavioral model.

Transitions: A transition in the behavioral model is mapped to an edge in UPTA and guards on the transition are mapped to guards on the corresponding edge in UPTA. In Figure 4, we shows how the transitions in behavioral model (top) are translated to UPTA (bottom right). The locations $L1$ and $L5$ corresponds to states $S1$ and $S2$ of behavioral model, respectively. The state invariants are translated to location invariant and represented as functions for the purpose of diagram clarity. The transition between S1 and S2 is triggered by POST(b) after 10 minutes as specified in the guard.

Trigger Methods: The trigger methods from the behavioral model is translated to receiving channels in UPTA. This receiving channel is in sync either with the automaton of partner service or with the environment model.

Time Events: The time events in behavioral diagram are replaced by clocks in UPTA. The clock is reset in the incoming edge to the location (L1) and is also included in the location invariant. Thus, the guard *after(10m)* is translated to $cl > 10$ in corresponding UPTA edge.

Effects: The effect on the transition, i.e., $POST(c)$ show invocation to the partner service. The communication between two web services is established by using a unique channel in a transition. For instance, emitting a transition from a web service to the other one can be replaced by producing a channel in an UPPAAL process, and the other process is the receiver of the channel. The effect of transition that invokes a remote service is represented with two edges and an urgent location (marked with U in the circle) in between, i.e., edges *e2* and *e3* and urgent location *L3*. An urgent location in UPTA does not allow any delays [21]. Thus, the first edge (e1) is synchronized with the environment model and the second edge (e2) has sending channel synchronized to the receiving channel of partner's automaton. The third edge (e3) is synchronized to receive acknowledgment response from the partner(as we model asynchronous service) and the sending channel on the fourth edge (e4) is synchronized with the environment to indicate the completion of transition.

Requirements: The requirements on the transition is translated into a boolean variable (initialized to *False*) and attached to the corresponding edge in the UPTA assigning a *True* value. This is shown in Figure 4 with *Req1= True* on edge e4. This implies that whenever this edge would be traversed, this requirement will be met. These can be formulated as reachability properties to attain requirement coverage and tracked during test generation.

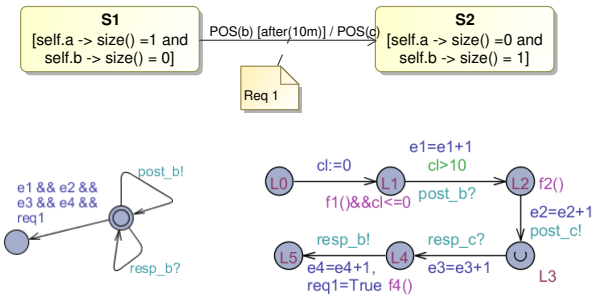

Figure 4: a)Example of Behavioral Model (top) Corresponding Environment Model (bottom left) and flattened UPTA (bottom right)

Environment Model: The environment model in UPTA has sending channels that are received by the composite web service automaton as inputs to trigger the process. This is similar to interface method calls in behavioral model by the service user. All the interface methods of the service specified in the behavioral model are mapped to the sending channels in the environment model and the response of successful transition is received from the composite web service via receiving channel. This is shown by the transformation of an excerpt of behavioral model to UPTA environment model in Figure 4. The environment model initiate the automaton (bottom right) by sending channel $post\_b!$ and the process completes when the channel $resp\_b?$ is received.

A python script is currently used to create the environment model, encoding test targets for edge coverage, from a given UPTA model by analyzing the channels observable from the environment. The original idea has been discussed in [14]. This will be merged in the final version of the transformation.

Test coverage information. In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in [19]) is used to automatically add *counter* variables for each edge of a given automaton in a UPTA model and a corresponding update of the given variable on the corresponding edge. Whenever the edge is visited during simulation or execution, the variable is incremented, allowing thus to track which edges have been visited and how many times. This enables one to track coverage level wrt. e.g., edge coverage or edge pair coverage. This script will also be included in the final version of the UML→UPTA transformation.

4 Case Study

Our case study is a Holiday Booking (HB) REST composite web service (CWS) that is built on inspiration from the *housetrip.com* service. It is a holiday rental online booking site, where you can search and book an apartment in the country

of your destination. We have built it as a REST composite web service.

The user of the service searches for a room in a hotel from the list of available hotels at HB before travel. He books the room (if it is available) and that booking is reserved by HB with the hotel for 24 hours. The user must pay for the booking within 24 hours. If the user does not pay within this time then the booking is canceled. If the booking is paid, then the HB service invokes a credit card verification service and waits for the payment confirmation. When the payment is confirmed, HB invokes the hotel service to confirm the booking of the room. If the hotel does not respond within 1 day or it does not confirm at all, the booking is canceled and the user is refunded. If the hotel service confirms, then a booking is made with the hotel. The payment is not released to the hotel until the user checks in. When the user checks in and is satisfied, HB releases the money to the hotel and booking is marked as paid by the hotel.

Design Models. The design of composite REST web service and its partner services is modeled with resource, behavioral and domain models. These models for HB REST composite web service are shown in Figure 5, Figure 6 and Figure 7, respectively. The behavioral model of Payment and Hotel partner services are shown in Figure 8.

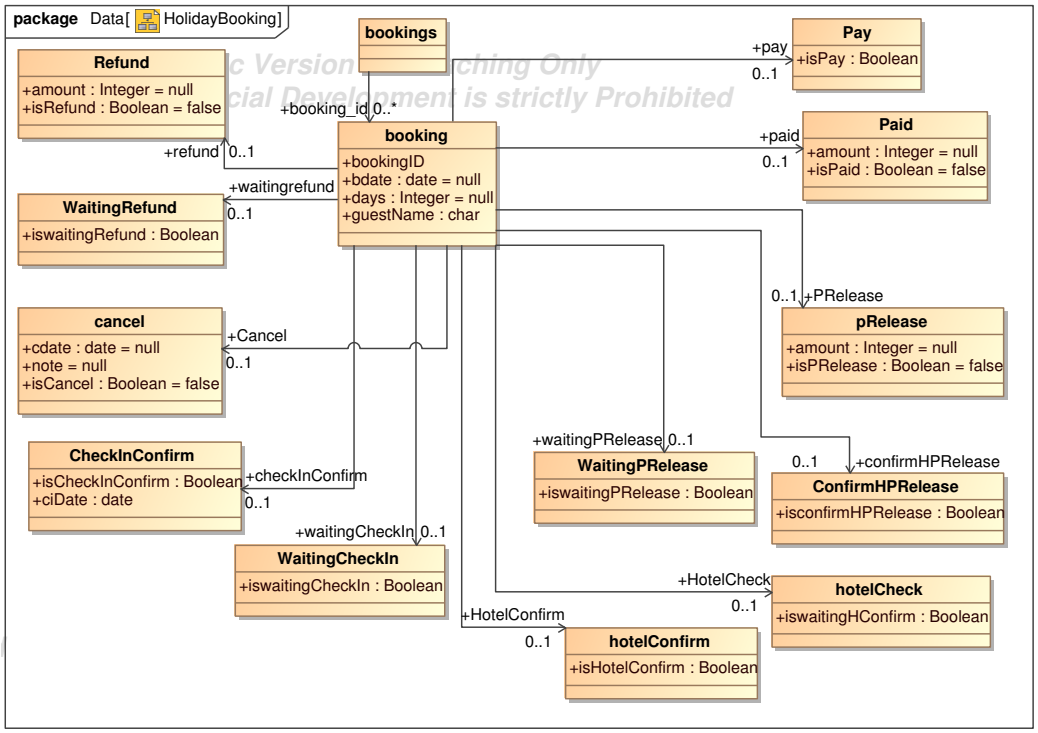

Figure 5: Resource Diagram of Holiday Booking Composite REST Web Service

Requirements Traceability. We have inferred functional and temporal requirements from specification document for our case study. Table 1 shows the requirements for HB RESTful composite web service. These requirements should

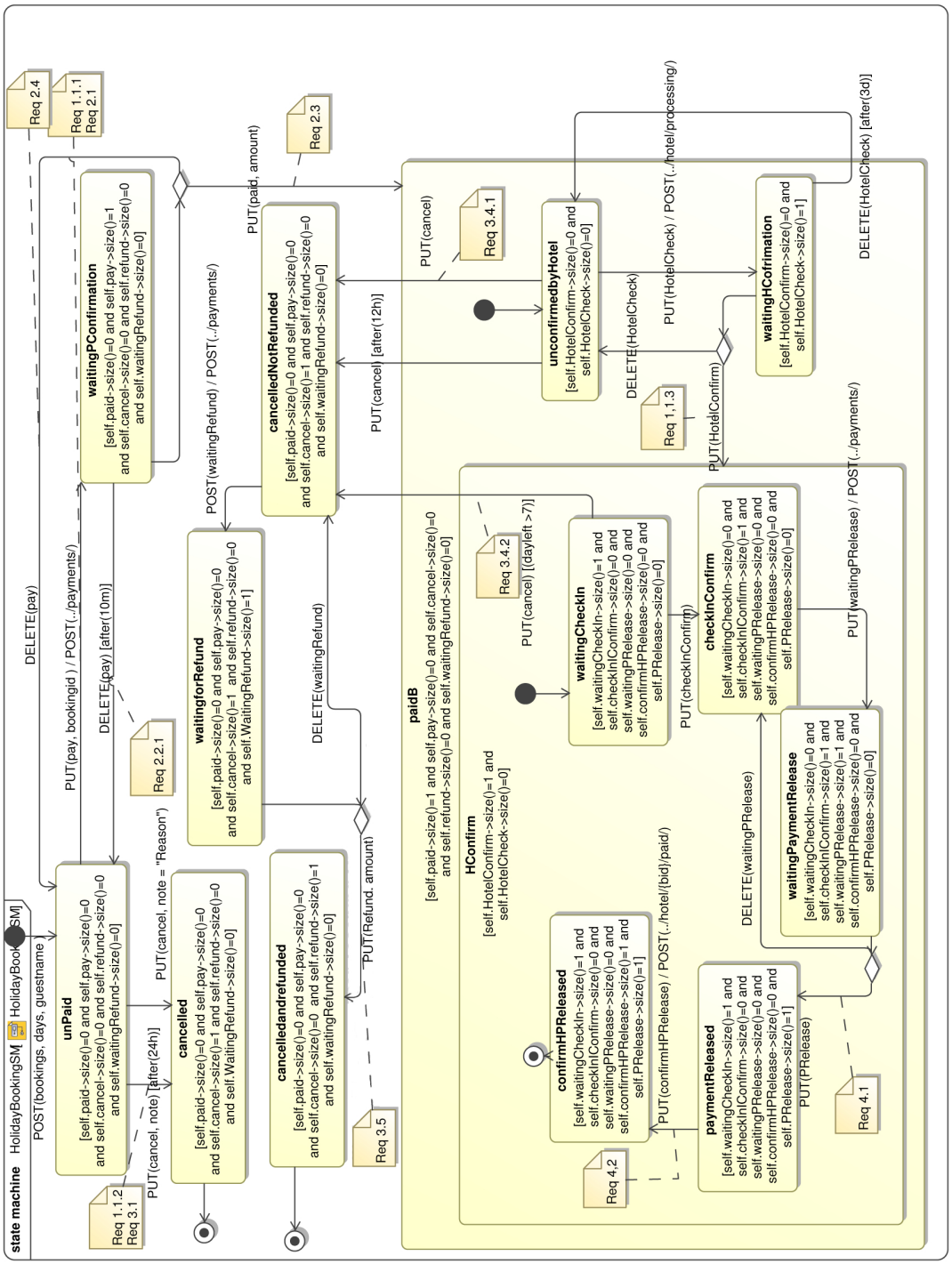

Figure 6: Behavioral Diagram of Holiday Booking Composite REST Web Service

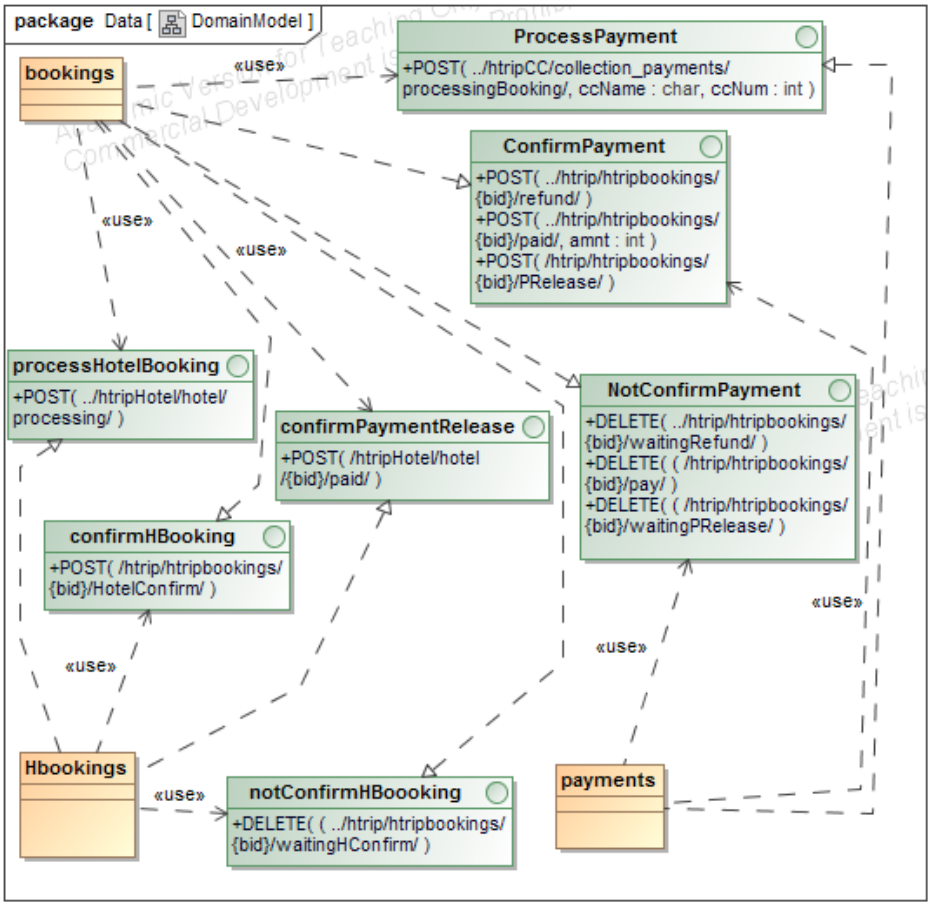

Figure 7: Domain Model for Holiday Booking Composite REST Web Service

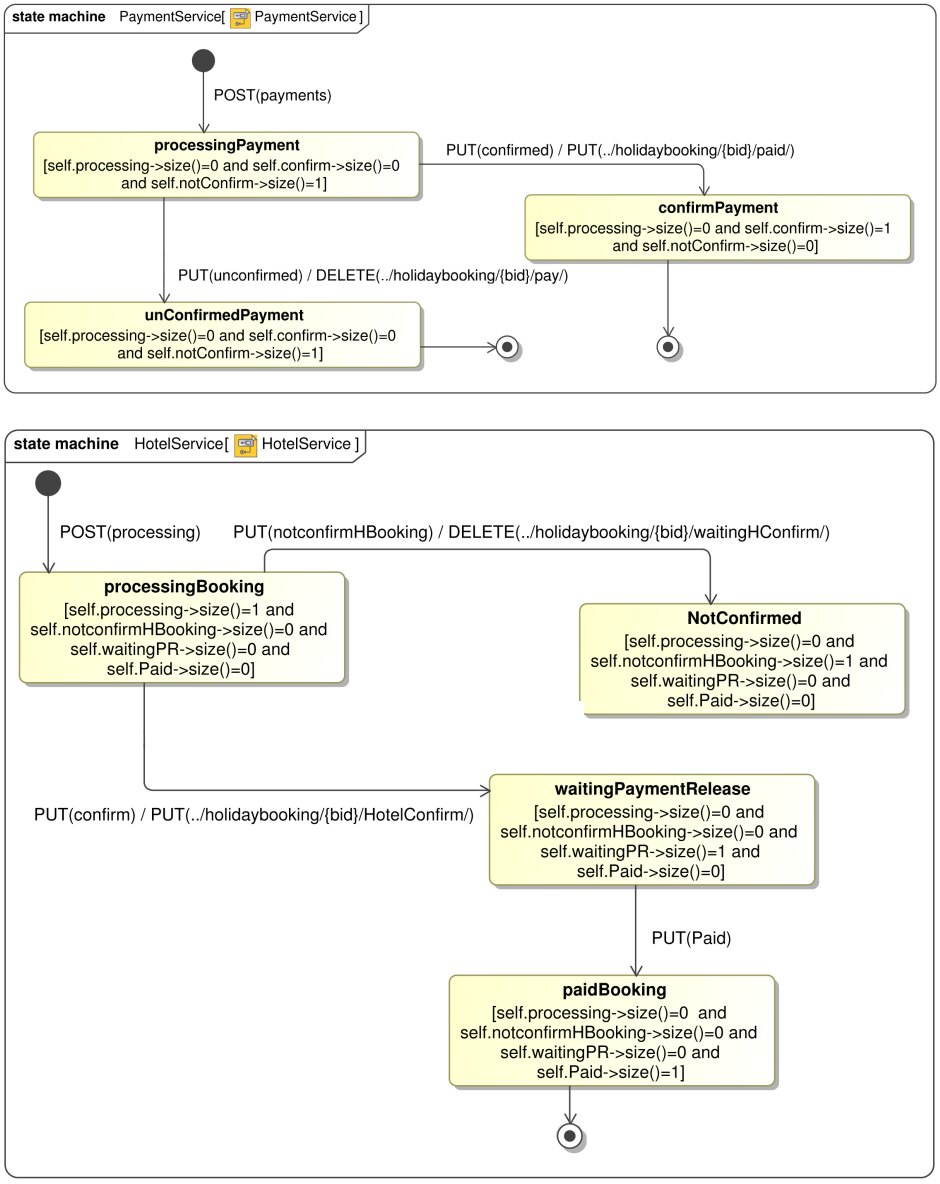

Figure 8: Behavioral Model for (Up) Payment REST Web Service (Down) Hotel REST Web Service

be fulfilled by the IUT in order to satisfy the service goals. These requirements are added as comments on Figure 6.

Verification: The design models of Holiday Booking composite REST web service are translated to UPTA shown in Figure 9.

The query language used in UPPAAL is a simplified version of TCTL [3] that consists of path formulae and state formulae. The path formulae quantify paths and traces of the model and state formulae describe individual locations with regular logical operators. Path formulae can be classified into reachability, safety and liveness properties and the deadlock is expressed using state formula.

The verification properties are specialized for our holiday booking case study and some of them are mentioned below.

Deadlock Freedom: The Holiday Booking Service, the hotel service and the payment service models are all deadlock free. This means that the composite service is never stuck in a certain state forever. (i.e., $A[]\ not\ deadlock$)

Reachability: All the states in the Holiday Booking service are reachable. This means that the HB service model receives and sends messages to its partner services smoothly and the model is validated for its basic behavior,
(i.e., $E \Diamond\ CompService.p$).

Safety: Some of the safety properties in our model are: a) Payment should be released iff user has checked in, i.e., ($E\Box CompService.c2\ imply\ CompService.g2$), b) If the payment is released by the holiday booking CWS then the hotel service booking is paid, i.e., ($E\ \Box\ \ CompService.p\ \ \text{imply}\ \ CompService.j$).

Liveness: Some of the liveness properties in our model are: a) When the payment is not paid within 24 hours, the booking is canceled (i.e., $CompService.b\ and\ compService.cl\ >\ 24\ \ \rightsquigarrow\ \ CompService.b1$), b) If the Hotel Service does not confirm in 3 days then the booking is considered not confirmed (i.e., $CompService.l\ and\ CompService.cl > 3\ \rightsquigarrow\ CompService.m$). the booking is considered not confirmed (i.e., $CompService.l\ and\ CompService.cl > 3 \rightarrow CompService.m$).

Test setup The testing process includes the TRON engine, an adapter, the IUT and the model of system. The TRON engine establishes TCP/IP connection on a local port and via that the adapter starts communications. The adapter works as an interface, which translates the inputs and outputs of the model to proper HTTP requests to/from the IUT. The UML to UPTA transformation also generates (optionally) a skeleton of the test adapter, depicting what interfaces should be implemented between TRON and IUT. Once the adapter is fully implemented, it can be reused for different versions of the models as long as there is no new I/O messaging being done. The IUT in this case study is a web service composition of three web services: HolidayBooking, Hotel and Payment Service.

Table 1: Requirements Table of Holiday Booking REST CWS

Req	Sub-Requirements
1- Booking	1.1 - A booking should be paid 1.1.1 - A booking should be paid within 24 hours of the booking. 1.1.2 - If a booking is not paid within 24 hours of the booking, then it is canceled by the system 1.1.3 - A confirmed paid booking, waits for user check in
2- Payment	2.1 - When user pays for the booking, partner service should be invoked to process the payment. 2.2 - The HB CWS should wait for response from the payment service 2.2.1 -If the payment service does not respond in 10 minutes, it is considered not working and the booking is marked unpaid 2.3 - If the partner service confirms the payment, the booking should be marked paid 2.4 - If the partner service unconfirms the payment, then the booking should be considered unpaid.
3- Cancel	3.1 - A booking is canceled if it is not paid for 24 hours 3.2 - A paid booking that is not confirmed by the hotel is marked unconfirmed 3.3 - A paid booking that is unconfirmed by the hotel is canceled after 12 hours. 3.4 - A paid booking can be canceled by the user 3.4.1 - A paid booking can be canceled by the user if it is not waiting for payment confirmation or hotel confirmation. 3.4.2 - User can cancel paid booking only before 7 days of checkin. 3.5 . A canceled booking must be refunded.
4- Payment Release	4.1 - If the user checks in then the payment must be released to the hotel. 4.2 - When the payment is released to the hotel, HB CWS must notify the hotel about release of the payment

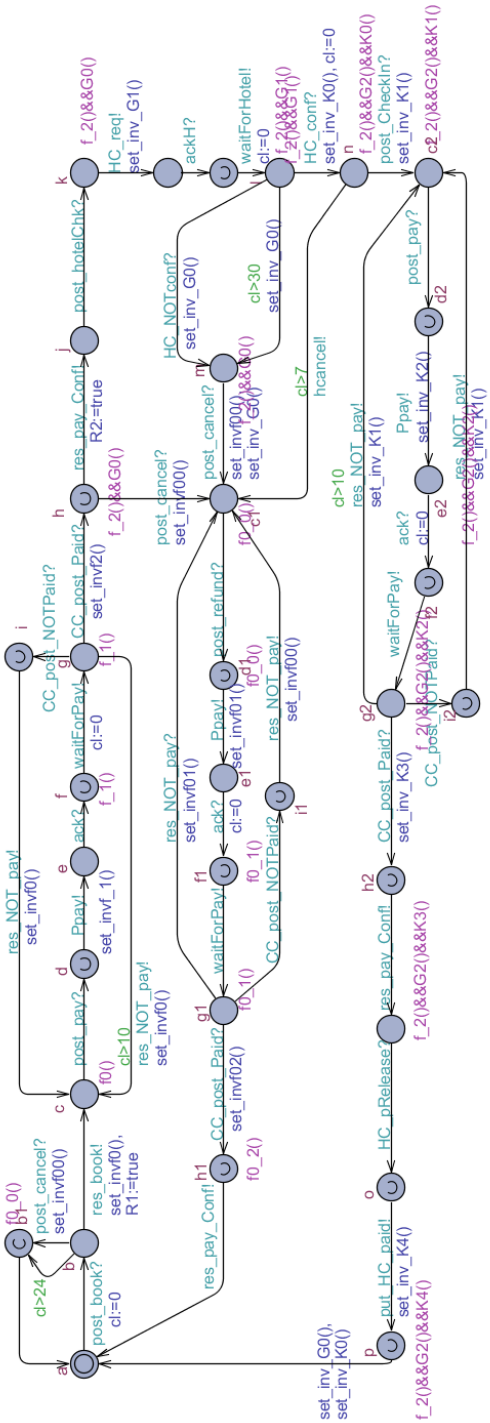

Figure 9: UPPAAL automata of Holiday Booking Composite REST Web Service

Composition Model We specified a TRON adapter which communicates between IUT and the composition model shown in Figure 9. The adapter identifies the emitting and receiving channels as well as defines the corresponding HTTP request functions. During test execution, the composition model waits until a channel call comes from the environment model, then the adapter translates the incoming channel to a specific HTTP request and send it to IUT. The response from IUT will be checked in the adapter and forwarded to the composition model as a response channel.

Test Environment The environment model specifies the user actions, such as booking, canceling a reservation, requesting for the payment, paying, refunding and checking in. These are created from the observable channel synchronizations of the composite web service. The automaton in Figure 10 shows the environment model satisfying edge and requirements coverage. While in Figure 10 they are encoded in the guard as a `verdict()` boolean function in the form: $r_1 \&\& \ldots r_n \&\& e_1 \ldots \&\& e_m$ where r_i and e_j are variables corresponding to requirements and, respectively, to edges of the composite web service in Figure 9. Whenever all the verdict function evaluates to TRUE environment model can go to the final location.

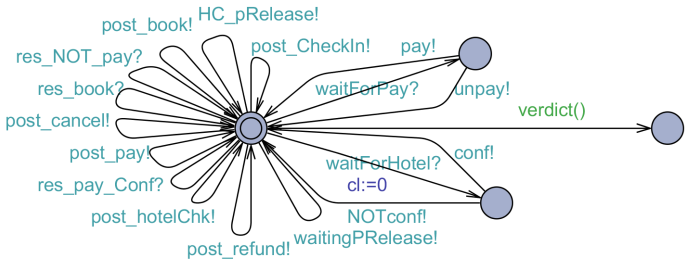

Figure 10: Environment model

5 Validation of Approach

Our Holiday booking composite REST web service had 14 states and 25 transitions. These were translated to 34 locations and 46 edges for the corresponding UPPAAL timed automaton. Similarly, the payment service had 3 states and 4 transitions and corresponding locations and edges were 5 and 6 in UPTA, respectively. The hotel service had 4 states and 5 transitions that were translated to 7 locations and 9 edges. In addition, the environment model created had 2 states and 13 edges. The ratio of translating states and transitions from behavioral model of Holiday Booking service to UPTA was higher than that of partner services since the composite service had more method invocations to partner service resulting in greater

number of locations and edges per transition.

One issue with using formal tools like UPPAAL for verification and test generation, is the scalability of the approach, due to the state space explosion. In contrast to offline test generation, where the entire state space has to be computed, in online test generation only the symbolic states following the current symbolic states have to computed. This reduces drastically the number of symbolic states making the test generation less prone to space explosion and thus more scalable. For instance, the number of explored symbolic states when generating, with the `verifyta` tool, traces satisfying complete edge coverage (i.e., $\&e_1 \ldots \&\&e_m$, where e_j are tracking variables corresponding to all m edges of the HBS models) was 583. In the same time, the maximum number of symbolic states reported by Tron during a test session achieving complete edge coverage was 489.

For benchmarking the verification process, we have used the `verifyta` command line utility of UPPAAL for verification of the specified 5 properties. We have used the `memtime` tool to measure the time and memory needed for verification. The result showed in average 0.20 seconds and 54996 KB of memory being used. Although the memory utilization depends heavily on the symbolic state space, it shows that the current size models leave room for scalability of the approach. A known limitation of UPPAAL is that the maximum memory size it can use is close to 4GB due to its 32-bit architecture.

The complexity of the models resulted for the holiday booking CWS in Figure 9, allowed us to verify all specified properties in UPPAAL and to generate testing using TRON. If the models would become too complex and they exhibit a state explosion, they could either be optimized at UPPAAL level, or they can be split into several parts, via slicing or aspect oriented approaches, each focusing on a different concern of the system.

In order to evaluate the efficiency of our approach, we compared the specification coverage with the code coverage yielded by a given test run. Since we had access to the source code of the IUT, we used the *coverage* tool for Python [1] to report the code coverage for each test session. The Table 2 lists results of several measurements:

Table 2: Correspondence between code coverage and edge coverage

Run	Edge Coverage	Code Coverage
1	64 %	55%
2	80%	67%
3	97%	96%

Table 2 shows that in the third run, we were able to achieve more than 95% of the code/edge coverage. Although many of the errors were caused by wrong modeling, testing revealed some errors of the implementation: in holiday booking service, there was an error in sending *cancel* request and the POST header variable in *refund* request was incorrect so the variable could not be saved in the

database. In hotel service, the confirmation was sent by the wrong method, so it was rejected by the composite service. There were also some minor mistakes in the card service.

In order to evaluate the fault detection capabilities of our approach, we have manually created 30 mutated versions of the original HBS program code. Each mutation had one fault seeded in the code, for instance replacing POST with DELETE, introducing delays in the response via sleep(), change of logical conditions, etc. We assumed that the original version of the program is the correct one. For each mutated version, we have run the test session 10 times, every time achieving more than 95% of edge coverage. Out of the 30 mutated programs, 26 mutants are killed. For the remaining ones, we have analyzed if the seeded fault was part of the code covered by the test run, and if not the mutant was replaced with a new one and the process repeated.

6 Related Work

There is already a large body of work on using model checking techniques for validation and verification of web service compositions. Overviews of such works can be found in [30] and [6]. Mostly authors have used web service specific specification languages as their starting point and converted the specifications to an intermediate language that is accepted by model checking tools. Then, by taking advantage of the model checking tool capabilities they performed simulation, verification or test generation via model-checking. Most of these works use the selected model-checking tool only for simulation and verification; only a handful generate abstract tests from the verification conditions, and in most cases it is not clear how the abstract test cases (i.e., the counterexample traces) are transformed into executable ones and executed. In the following, we will revisit those works which are most similar to ours.

We can distinguish roughly two approach categories: those that target the PROMELA language [25] which is the input language for the SPIN model-checker [16], and those that target the UPPAAL timed automata which is the input language for the UPPAAL model-checker [4].

In the first category, the vast majority of approaches have used BPEL or OWL-S[23] for the specification of the web service composition. For instance, Garcia [12] generates test cases using test case specifications created from counterexamples that are obtained from model checking. The transition coverage criterion is used to identify transitions in BPEL specification that define the test requirements for producing test cases. These transitions are mapped to the model and expressed in terms of LTL property expressions. Test cases are generated using the test case specifications created from counter examples obtained from model-checking. Transition coverage is obtained by repeatedly executing the tool with each previously identified transition.

Fu. et al. [11] provide an elaborated framework for analyzing, designing and verifying web service compositions. Their work provides both bottom-up and top-down approach to analyze the interaction between web services. In top-down approach, the desired conversation of a web service is specified as guarded automaton where guards of the automaton are XPath queries with LTL properties. The guarded automata are converted to PROMELA and used as input to SPIN model checker. The bottom-approach translates BPEL to guarded automaton and is used in similar fashion with SPIN model checker after translating guarded automaton to PROMELA. The web service conversations are analyzed for synchronization in order to verify their compatibility.

One distinct approach is given by Huang et al. [17]. They automatically translate OWL-S specification of composite web service into a C-like specification language and Planning Domain Definition Language (PDDL) through a proposed integrated process. These can be processed with the BLAST model checker which can generate positive and negative test cases during model checking of a particular formula and test the web service using the test cases. They propose an extension to the OWL-S specifications and the PDDL to support their approach and use a modified version of the BLAST tool. Abstract, both positive and negative test cases are generated by formulating verification conditions for manually specified properties.

These works focus on BPEL processes and OWL-S, this makes them dependent on specific execution languages for SOAP based services whereas our work is not dependent on implementation and supports REST architectural style. In addition, their work does not support requirement traceability and is not clear how tests are generated and executed. Furthermore, the works that use the PROMELA language for specification do not address real-time properties, due to the limited support for time in PROMELA.

In the second category, researchers have targeted timed automata specifications. In [8], Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL (Web Services Choreography Description Language) into a network of timed automata and then use UPPAAL tool for validation and verification of the described system. They also capture capture requirements by extending KAOS goal model and implement them. The work is supported by WST tool that provides model transformation of timed composite web services [7]. It takes UML sequence diagram and translates it to WS-CDL and then to UPPAAL for simulation and verification. In [9], Diaz et. al also provide a translation from WS-BPEL to UPPAAL timed automata. Time properties are specified in WS-BPEL and translated to UPPAAL. However, requirements are not traced explicitly, while verification and testing are not discussed.

Ibrahim and Al-Ani [18] transform BPEL specification to UPAAAL. The specification includes safety and security non-functional properties which are later formulated into guards in the UPPAAL model which could be similar to our verification of requirements. They do not consider neither real-time properties nor

test generation.

In [13], Nawal and Godart deal with checking the compatibility of web service choreography supporting asynchronous timed communications using model checking based approach. They use model checker UPPAAL and present compatibility checking distinguishing between full and partial compatibility and full incompatibility of web services. They propose a set of required abstractions for timed asynchronous communicating services that allow the use model checker UPPAAL. Our work is somewhat similar to their work as we support time critical stateful REST webs service compositions using UPPAAL, however, in addition to verification we use UPPAAL with TRON to validate the implementation of web services.

Zhang [33] suggest the use of the temporal logic XYZ/ADL language [34] for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) which are verified in UPPAAL.

In [20], uses BPEL specifications as a reference specification and transform them to an Intermediate Format(IF) based on timed automata and then propose an algorithm to generate test cases. Similar to our approach, tests are generated via simulation in a custom tool, where the exploration is guided by test purposes. One noticeable difference is that time properties are added manually to the IF specification, while we specify them at UML level.

These works provide approaches to verify and validate the service specifications by checking the properties of interest using UPPAAL tool, however our work, in addition to model checking the properties also performs conformance testing of the service composition via online model-based testing with the TRON tool and provides requirement traceability for non-deterministic systems.

7 Conclusion

We have presented an integrated approach to design and validate RESTful composite web services. In our approach, a service can invoke other services and exhibit complex and timed behavior, while still complying with the REST architectural style. We showed how to model the service composition in UML, including time properties. We modeled communicating web services and explicitly define the service invocations and receiving service calls.

We use model checking approach with UPPAAL model checker to verify and validate our design models. From the verified specification, we generate tests using an online model-based testing tool. The use of online MBT proved beneficial as our system under test exhibits non-deterministic behavior due to concurrency and real-time domain.

With the help of requirements traceability mechanism we traced requirements to UML models and, via the UML→UPTA transformation to timed automata

models. Their reachability is verified in UPPAAL and they are used as test goals during test generation. Linking requirements to generated tests allowed us to quickly see which requirements have been validated and which have not. In addition, it allows us to identify from which parts of the specification/implementation the source the detected error has originated.

We exemplified our approach with a relatively complex case study of a holiday booking web service and we provided preliminary evaluation results.

References

[1] Code coverage measurement for Python – coverage, v. 3.6. Online at https://pypi.python.org/pypi/coverage. retrieved: 20.08.2013.

[2] Nomagic MagicDraw webpage at http://www.nomagic.com/products/magicdraw/, August 2013.

[3] R. Alur et al. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[4] G. Behrmann et al. Uppaal 4.0. In *QEST '06 Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 125 – 126. IEEE Computer Society Washington, DC, USA.

[5] C. W. Birgit Demuth. Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice,*, pages 81–89, 2009.

[6] M. Bozkurt and other. Testing web services: A survey. *Department of Computer Science, Kings College London, Tech. Rep. TR-10-01*, 2010.

[7] M. Cambronero and n. p. y. p. others journal=Simulation, volume=88. Wst: a tool supporting timed composite web services model transformation.

[8] M. E. 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] G. Dıaz et al. Model checking techniques applied to the design of web services. *CLEI Electronic Journal*, 10(2), 2007.

[10] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[11] X. Fu et al. Synchronizability of conversations among web services. *Software Engineering, IEEE Transactions on*, 31(12):1042–1055, 2005.

[12] J. García-Fanjul et al. Generating test cases specifications for bpel compositions of web services using spin. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, page 83, 2006.

[13] N. Guermouche and C. Godart. Timed model checking based approach for web services analysis. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 213–221. IEEE, 2009.

[14] A. Hessel et al. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. Springer-Verlag, 2008.

[15] A. Holovaty and J. Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[16] G. J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

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

[18] N. Ibrahim and I. Al Ani. Beyond functional verification of web services compositions. *Journal of Emerging Trends in Computing and Information Sciences*, 4, Special Issue:25–30, 2013.

[19] M. Koskinen et al. Combining Model-based Testing and Continuous Integration. In *Proceddings of the International Conference on Software Engineering Advances (ICSEA 2013)*. IARIA, October 2013. TO APPEAR.

[20] M. Lallali et al. Automatic timed test case generation for web services composition. In *on Web Services, 2008. ECOWS'08. IEEE Sixth European Conference*, pages 53–62. IEEE, 2008.

[21] K. G. Larsen et al. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[22] K. G. Larsen et al. Uppaal tron user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*, 2009.

[23] D. Martin et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[24] M. Nobakht and D. Truscan. Tool support for transforming uml-based specifications to uppaal timed automata. Technical Report 1087, 2013.

[25] I. Part and M. Peschke. Design and validation of computer protocols. 2003.

[26] I. Porres and I. Rauf. Modeling behavioral restful web service interfaces in uml. pages 1598–1605, 2011.

[27] I. Rauf et al. Modeling a composite restful web service with uml. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 253–260. ACM, 2010.

[28] I. Rauf et al. Analyzing consistency of behavioral rest web service interfaces. In J. Silva and F. Tiezzi, editors, *The 8th International Workshop on Automated Specification and Verification of Web Systems*, Electronic Proceedings in Theoretical Computer Science, page 115. EPTCS, 2012.

[29] I. Rauf and I. Porres. Beyond crud. pages 117–135, 2011.

[30] H. M. Rusli et al. Testing web services composition: a mapping study. *Communications of the IBIMA*, 2007:34–48, 2011.

[31] O. UML. 2.2 Superstructure Specification. *OMG ed*, 2009. `http://www.omg.org/spec/UML/2.2/`.

[32] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[33] G. Zhang et al. Model checking for asynchronous web service composition based on xyz/adl. In *Web Information Systems and Mining*, pages 428–435. Springer, 2011.

[34] X.-Y. Zhu and Z.-S. Tang. A temporal logic-based software architecture description language xyz/adl. *Journal of Software*, 14(4):713–720, 2003.

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-2995-4
ISSN 1239-1891