Scenario-Based Design and Validation of REST Web Service Compositions

Irum Rauf(✉), Faezeh Siavashi, Dragos Truscan, and Ivan Porres

Department of Information Technologies, Åbo Akademi University, Turku, Finland
{irum.rauf,faezeh.siavashi,dragos.truscan,ivan.porres}@abo.fi

Abstract. We present an approach to design and validate RESTful composite web services based on user scenarios. 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 specify user scenarios via UML Sequence Diagrams. The service specifications are transformed 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 for computing coverage level during test generation. We validate our approach with a case study of a holiday booking web service.

Keywords: REST · Web service composition · Model-based testing · UPPAAL · TRON

1 Introduction

REST (REpresentational State Transfer) web services are built on the principles of the REST architectural style [12] which aims at producing scalable and extensible web services. The REST interface offers a CRUD interface (create, retrieve, update and delete) to its users via a set of standard HTTP methods. In additions, REST offers stateless behavior that facilitates scalability.

Different web services published over the internet can be composed into new composite web services which fulfill new service goals using the functionality of partner web services. Automated systems, for example hotel reservation systems, are often built as stateful composite services that require a certain sequence of method invocations that must be followed in order to fulfill service goals. Creating such composite services with advanced scenarios and REST features requires rigorous development approaches that are capable of creating web services that can be trusted for their behavior.

With the rise in use of REST web services in different domains offering complex and timed scenarios, there is an increasing need for validation approaches to effectively and efficiently detect faults in the specifications and implementations of such services.

V. Monfort and K.-H. Krempels (Eds.): WEBIST 2014, LNBIP 226, pp. 145–160, 2015.
DOI: 10.1007/978-3-319-27030-2_10

In this article, we present a scenario-based validation and verification approach that can help the service developer in improving the quality of service specifications and implementations. The approach supports the creation of timed and stateful behavior with the confidence that the service fulfills its advertised functionality. The Web Service Composition (WSC) is specified using the Unified Modeling Language (UML) starting from the requirements of the WSC. A code skeleton of the WSC is automatically generated and manually completed by the developer. In order to perform validation and verification of the composition, the UML specifications are transformed into UPPAAL timed automata (UPTA). We use the UPPAAL tool set [23] to simulate the specifications and to verify their properties via model-checking. We also use them to automatically generate tests in order to validate the implementation.

Requirements traceability is an important component of our approach. The requirements of the composition are included in the UML specifications and then propagated to UPTA. They are used for both verifying the reachability of those model elements implementing them and for reasoning about the coverage level of the tests generated. Upon detecting failures, the traced requirements are used to trace back errors either in the models or in the implementation.

We exemplify and validate our approach with a relatively complex example of a holiday booking composite REST web service extracted from an industrial application. The example shows how stateful and timed web services offering complex scenarios and involving other web services can be constructed efficiently using our approach.

The paper is organized as follows: Sect. 2 presents our approach and the tool support is discussed in Sect. 3. The case study is presented in Sect. 4, followed by the evaluation of the approach in Sect. 5. The related work is discussed in Sect. 6 and conclusions are drawn in Sect. 7.

2 Our Approach

Our scenario-driven approach to verify and test the composite REST web service is shown in Fig. 1. We start by inferring service requirements in tabular format from specification document and the corresponding user scenarios from the specification document of the REST WSC. Each user scenario is detailed by one or several UML sequence diagrams. In addition, we build several perspectives of the WSC such as a resource, a behavioral and a domain model using UML class and state machine diagrams. This is an extension of our previous work, in which we designed behavioral interfaces for web services that were RESTful by construction [26]. We transform the service design models to UPTA, which are simulated and model-checked by reasoning the properties such as deadlock, liveness, reachability, and safety. If inconsistencies are found, the UML-based service design models are updated. These design models are used to implement the service in the Python-based Django web framework [16] using our partial code generation tool [26] which generates code skeletons with pre- and post-conditions for every service method. The skeleton is manually completed

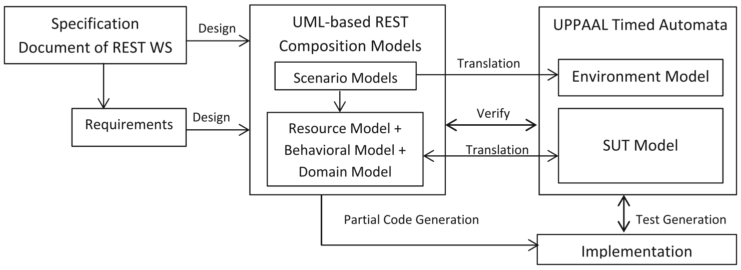

Fig. 1. Scenario-based V&V approach for REST CWS.

by the service designer. The verified UPTA specifications are used for online model-based conformance testing of the implementation.

Requirements Traceability. Service requirements are inferred from the specification document and they serve as service goals. A service should be checked for its service goals to validate that the service does what it is required to do. By addressing the service requirements at the design phase and propagating them to the verification and validation stages, we provide a mechanism by which a service implementation can be validated for its goals and the unfulfilled requirements can be traced back to the design phase to find faults in the design.

Requirements Table. Service requirements are generally domain-specific since they are inferred from the specifications. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the UML state machine (SM) as *comments* on the transitions and are propagated to UPTA such that the links between requirements and the model elements are 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 with the purpose of verifying them during simulation. Each requirement label is translated into a boolean variable (initialized to *False*) and attached to the corresponding edge in UPTA.

Scenario Models. The behavioral requirements are elicited as scenario models using UML sequence diagrams. These scenario models are translated to environment model in UPTA since these scenarios define different conditions under which the composite service can be invoked.

We require that our testing approach must validate that the service requirements are met by IUT, and the service works correctly in different scenarios, in order to build confidence of the developer that the system is doing what it is required to do. Thus, the coverage level of scenarios and requirements is monitored during test generation and execution. Once the test report is available, we can check which requirements have been validated and which have failed. The main strength of using both the requirements table and scenario models in our approach is that the former helps in tracing the unfulfilled requirements to the design models and locating the faults in the design of the service. On the other hand, the later helps in determining if the service works fine in different scenarios and identify under which conditions the service shows a faulty behavior.

REST Composition Models. The web service compositions that we build exhibit RESTful features such as addressability, connectivity, statelessness and uniform interface. Thus, we model several perspectives of a service composition:

Scenario Models. Some of the behavioral requirements of the service are elicited into scenario models using UML sequence diagrams. These scenario models provide details of the interaction between composite service and its partners and also insights on how a certain scenario is realized. This information facilitates the development of the composition and they are also used later on to validate the service implementation.

Resource Model. The concept of resource is central to the structure of REST web service. It represents a piece of information [28]. We represent the static structure of REST web service with resource model which is modeled with a UML class diagram. Each class defines a *resource*. The direction of the associations specify navigability (connectivity) direction between resources, while their role names give the relative URI of resources (addressability). The *collection resources* without the incoming transitions are termed as *root* such that every *resource* defined in the resource model should be reachable via the *root* and the graph formed should be connected (connectivity).

Behavioral Model. The behavioral model represents the dynamic structure of the service using UML state machines. Each state represents the service state and the transition triggers are restricted to the side-effect methods of HTTP protocol, i.e., PUT, POST and DELETE (uniform interface). The statelessness feature of the REST interface is preserved while building stateful REST web service by defining state invariants as boolean predicates over the states of different resources. The state of a resource is given by its representation retrieved by invoking a HTTP GET method on it. We are thus able to define service states as predicates over the resources without maintaining any hidden session or state information (statelessness). The state invariants in the SM are written as Object Constraint 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 UML *comments* on the transitions that satisfy them.

Domain Model. The domain model of the composite service is represented with a UML 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.

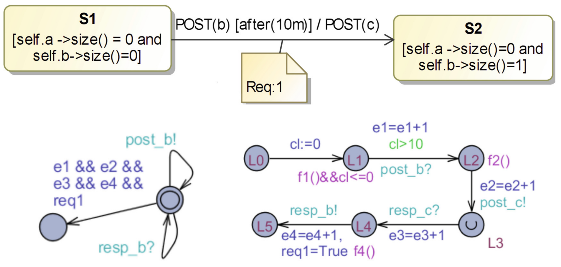
S1
[self.a ->size() = 0 and self.b->size()=0]
POST(b) [after(10m)] / POST(c)
Req:1
S2
[self.a ->size()=0 and self.b->size()=1]
post_b!
e1 && e2 && e3 && e4 && req1
resp_b?
L0
cl:=0
L1
e1=e1+1
cl>10
post_b?
f1()&&cl<=0
L2
f2()
e2=e2+1
post_c!
L3
resp_c?
e3=e3+1
L4
resp_b!
e4=e4+1, req1=True
f4()
L5

Fig. 2. Example of state model (top), corresponding environment model (bottom left), and flattened TA (bottom right).

Transformation. In order to make the models amenable for simulation and model checking we employ a set of mechanized steps for translating UML-based service specifications into UPPAAL timed automata (UPTA) [23].

The transformation from UML design models to UPTA has been discussed in [27]. It takes as input the resource model, domain model, and behavioral model and generates two artifacts in UPTA: the SUT model specifying the behavior of the service and of its partner services (a generic example is presented in Fig. 2) and an *environment model* which simulates the behavior of the service user. Two kinds of environment models are generated automatically: a canonical model which allows to simulate freely all possible behaviors of the SUT and a model used for testing different user scenarios.

The transformation of the user scenarios from sequence diagrams to UPTA environment models is applicable to Sequence Diagrams(SD) with a restricted set of elements. The following generic steps are used by the transformation:

- Each SD has may have several lifelines, which are grouped into two groups: SUT and environment. The messages exchanged between the two groups will provide the testing interface.
- For each input message to the SUT group, we define an edge to a new location in UPTA. The edge is labeled by the name of the message and it has associated a sending channel(!).
- For each output message from the SUT group, we define a new edge to a new location with a receiving channel (?).
- For SD fragments (i.e., alt, loop, opt), based on the number of conditions in the fragment, we define several edges from a location and use the conditions as guards on the edges.
- Timing constraint and duration constraint are transformed into location invariants and edge guards in UPTA.
- Tracking variables are added to each scenario trace in UPTA. If a scenario has more than one exit points (alternatives) several variables are added. A tracking variable is an updated tuple *(sd_no = false, sd_no = true)* on the first edge in the scenario trace and respectively on the last edge in the trace.
- UPTA traces stemmed from different SDs are included in one single UPTA environment.

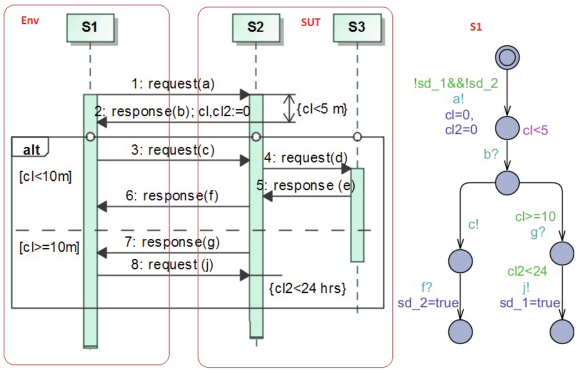

Fig. 3. Example of SD of three services (left) and the UPTA model of S1 as environment (right).

The resulting UPTA environment will have channel synchronizations matching the SUT model obtained in the first transformation.

Figure 3 shows an example of a SD with three lifelines (right) and its transformed environment model (left). Assuming *S1* as the environment, the UPTA environment model contains only emitting/receiving messages to/from *S1*. Response(b) should be received within 5 min ($cl < 5m$) and request(j) can be sent before 24 h ($cl2 < 24$ hrs).These timing constraints are modeled as location invariants and guards in UPTA. For modeling before and after a deadline ($cl < 10$ and $cl >= 10$) in sequence diagram, we used **alt**, which is transformed into two different locations with their corresponding edges (c! and g?) in UPTA. The timing constraints in **alt** are translated as location invariant and edge guard in the model.

Verification. We use the UPPAAL model-checker [23] to verify basic properties of our design models such as reachability, liveness, and safety. In addition, we check whether the service user scenarios are satisfied. This allows one to eliminate design errors that can be otherwise expensive to detect and correct at later stages of the development cycle. If problems are found, updates are manually fixed in the UML design models.

Test Generation. A skeleton of the composite service is generated automatically in the Django web development framework [16] using our partial code generation tool. The implementation is manually completed by the service developer. In order to validate that the implementation of the composite service is functioning correctly along with its partner services and if the service goals and timed constraints are being fulfilled, we generate tests from the UPTA models and execute them online (on-the-fly) against the implementation. During the test execution we monitor how different test coverage criteria are fulfilled, how the requirements are covered, and whether the user scenarios are validated.

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 meta-model specifications and prevent the developer from doing basic modeling mistakes.

Code Generation. The code-skeleton of the updated service design models of REST composite web service can be generated using our tool presented in [26]. The tool generates code skeleton for design models in Django that is a high level Python web framework [16]. The generated code also has behavioral information such that *pre* and *post* conditions for each method are included and the developer just has to write the implementation of the operations.

UML→UPTA Transformation. A Python script is used to automate the transformation.

Test Generation. We generate tests using UPPAAL TRON, an extension of UPPAAL for online model-based black-box conformance testing [24]. A test adapter is used by UPPAAL TRON to expose the observable I/O communication between the test environment model and the SUT model. Our adapter implements the communication with the SUT by converting abstract test inputs into HTTP request messages and HTTP response messages into abstract test outputs. UPPAAL TRON 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 Implementation Under Test (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.

Test Coverage Information. In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in [20]) is used to automatically add *tracking* variables (also referred to as *traps* in the UPPAAL community) 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 the 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 integrated in the final version of the UML→UPTA transformation script. W.r.t scenario-coverage each scenario will have its own tracking variable, changing value when the scenario is considered fulfilled (see for instance variables *Sc1* and *Sc2* in Fig. 8 (left)).

4 Case Study

Our case study is a Holiday Booking (HB) composite REST web service that is built on inspiration from the *housetrip.com* service, with the purpose of having

Table 1. Requirements of Holiday Booking CWS (excerpt).

Req	Sub-Requirements
2- Payment	2.1 - When user pays for the booking, partner service should be invoked to process the payment
	2.2 - If the partner service confirms the payment, the booking should be marked paid
	...
3- Cancel	3.1 - A paid booking can be canceled by the user
	3.2 . A canceled booking must be refunded
	...

a case study similar in complexity to real services. This service is a holiday rental online booking site, where one can search and book an apartment in the destination country.

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 h. The user must pay for the booking within 24 h. 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, HB releases the money to the hotel and the booking is marked by the hotel as paid. Due to space limitation, we only show some of the models in here while complete details are available at [26].

Requirements. We have inferred functional and temporal requirements from specification document for our case study. In total we specified 4 main requirements with their sub-requirements. Some of these requirements are accompanied by scenario models. For brevity, Table 1 shows only two of these requirements, *Payment* and *Cancel*. The scenario models in Figs. 4 and 5 detail how their corresponding user scenarios are fulfilled by the composite service.

Design Models. The design of HB composite REST web service is modeled with resource, behavioral and domain models. Due to space reasons only an excerpt of the state machine of HB composite service is shown (Fig. 6). Service requirements are traced to the state machine by including them (and their sub-requirements) as comments linked to transitions.

UML→UPTA Transformation. The timed automaton corresponding the HB service from Fig. 6 is given in Fig. 7. The detailed model and the specifications of the partner web services are available in [26].

Figure 8 shows the two types of environment models produced by the transformation: one modeling the user scenarios in Figs. 4 and 5, and a canonical

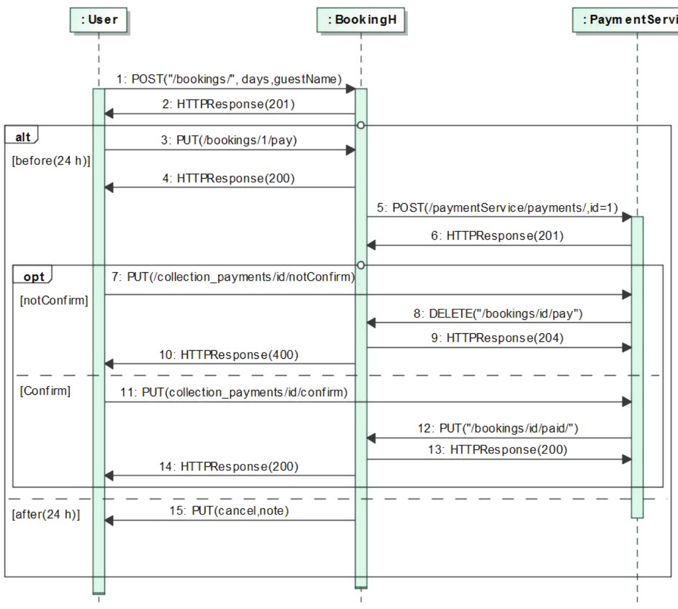

Fig. 4. Scenario model for user payment and invoking payment service.

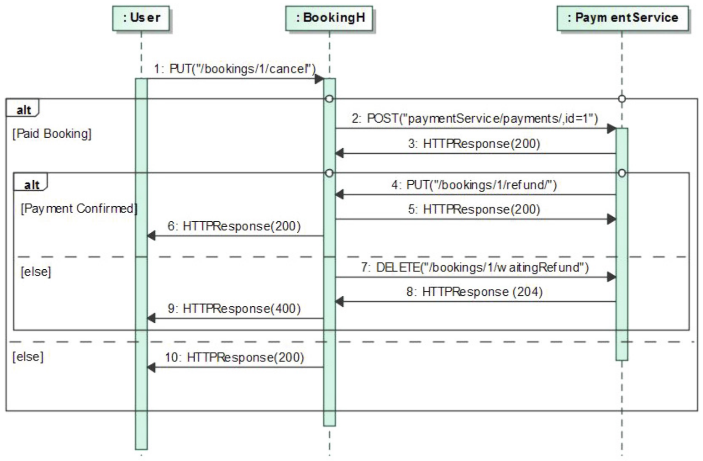

Fig. 5. Scenario model to cancel booking.

model. Each scenario has associated a tracking variable (e.g., *Sc1*) which helps in performing the verification and monitoring test coverage.

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

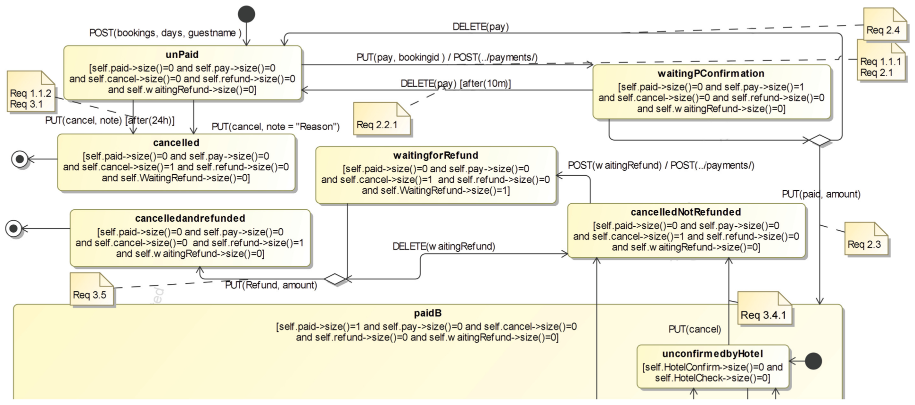
POST(bookings, days, guestname)
unPaid
[self.paid->size()=0 and self.pay->size()=0 and self.cancel->size()=0 and self.refund->size()=0 and self.waitingRefund->size()=0]
DELETE(pay)
PUT(pay, bookingid) / POST(../payments/)
DELETE(pay) [after(10m)]
waitingPConfirmation
[self.paid->size()=0 and self.pay->size()=1 and self.cancel->size()=0 and self.refund->size()=0 and self.waitingRefund->size()=0]
Req 2.4
Req 1.1.1
Req 2.1
Req 1.1.2
Req 3.1
Req 2.2.1
PUT(cancel, note) [after(24h)]
PUT(cancel, note = "Reason")
cancelled
[self.paid->size()=0 and self.pay->size()=0 and self.cancel->size()=1 and self.refund->size()=0 and self.WaitingRefund->size()=0]
waitingforRefund
[self.paid->size()=0 and self.pay->size()=0 and self.cancel->size()=1 and self.refund->size()=0 and self.WaitingRefund->size()=1]
POST(waitingRefund) / POST(../payments/)
PUT(paid, amount)
cancelledandrefunded
[self.paid->size()=0 and self.pay->size()=0 and self.cancel->size()=0 and self.refund->size()=1 and self.waitingRefund->size()=0]
cancelledNotRefunded
[self.paid->size()=0 and self.pay->size()=0 and self.cancel->size()=1 and self.refund->size()=0 and self.waitingRefund->size()=0]
Req 2.3
DELETE(waitingRefund)
PUT(Refund, amount)
Req 3.5
Req 3.4.1
paidB
[self.paid->size()=1 and self.pay->size()=0 and self.cancel->size()=0 and self.refund->size()=0 and self.waitingRefund->size()=0]
PUT(cancel)
unconfirmedbyHotel
[self.HotelConfirm->size()=0 and self.HotelCheck->size()=0]

Fig. 6. Excerpt of UML state machine of holiday booking composite REST web service.

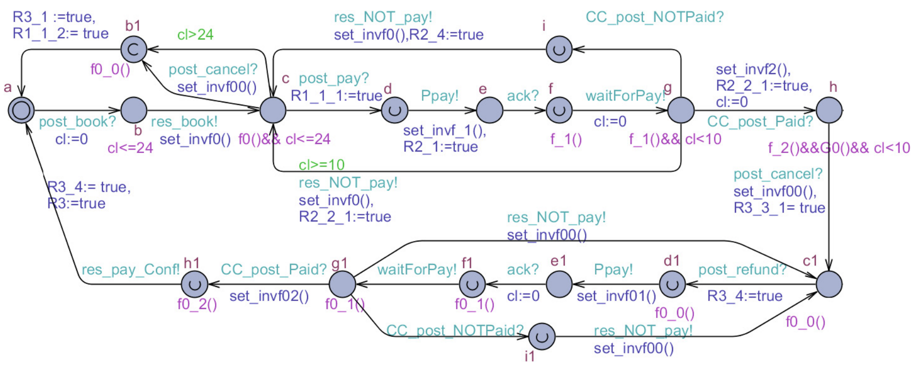

Fig. 7. Excerpt of UPTA model of holiday booking composite REST web service.

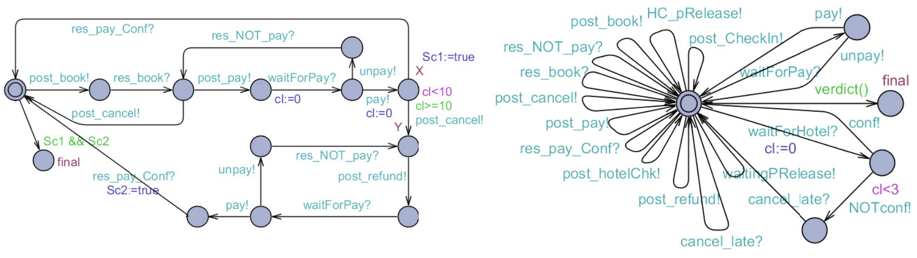

Fig. 8. Excerpt of Scenario-based environment (left) and canonical environment (right).

Deadlock Freeness. The HB Service, the hotel service and the payment service models are all deadlock free. This means that the composite service never reaches a state that cannot preform a transition (i.e., $A\,\square$ *not deadlock*). Note that the

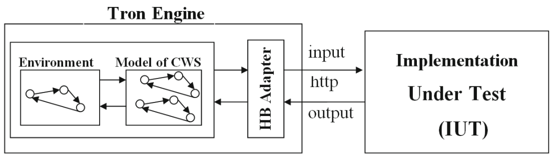

Fig. 9. UPPAAL TRON test setup.

following queries are made for complete model and only some of them can be traced in Fig. 7.

Reachability. All the locations in the HB service are reachable. This means that the model receives and sends messages to the partner services smoothly and the model is validated for its basic behavior (i.e., $E\lozenge\, CompService.r$), where r is the last location in the TA model and indicates that all processes for a certain booking is completed.

Safety. Some of the safety properties in our model are: (a) Payment should be released iff the user has checked in, i.e., ($E\,\Box\, CompService.h2$ imply $Card\ Service.c2$), where *c2* is the location after check-in and *h2* is the location after payment release, (b) If the payment is released by the HB service then the Hotel service is paid, i.e., ($E\,\Box\, CompService.h2$ imply $HotelService.p$), where p is the location in Hotel service model for hotel payment.

Liveness. Some of the liveness properties in the model are: (a) When the payment is not paid within 24 h, the booking is canceled (i.e., $CompService.c$ and $compService.cl > 24 \leadsto CompService.b1$), where c indicates waiting for the payment, *cl* indicates clock of the model and *b1* indicates the booking request is going to cancel due to the delay, (b) If the Hotel Service does not confirm within 3 days then the booking is considered not confirmed (i.e., $CompService.o$ and $CompService.cl > 3 \leadsto CompService.n$), where o is the location for waiting for the hotel response and n is the location for canceling. For the scenario environment, we identified a boolean variable for each scenario. Initially, all variables are false, and at the end of each scenario the corresponding variable will be set to true. The verification rule shows that all scenarios are reachable ($E\lozenge\, SDEnv.Sc1$ and $SDEnv.Sc2$ and $SDEnv.Sc3$), where SDEnv indicates the environment model, and Sc1, Sc2 and Sc3 are the variables. Timing constraints in scenario environment is verified by checking if the user is waiting for the service payment confirmation more than 10 h (i.e., in location X), then she can cancel the reservation (i.e., $SDEnv.X$ and $SDEnv.cl > 10 \leadsto SDEnv.Y$), Y and X are locations.

Testing. The test setup comprises the TRON engine, the test adapter, and the IUT. The IUT is a web service composition of three web services: Holiday Booking, Hotel and Payment Services, whereas the environment model is one of the models in Fig. 8. Whenever all the tracking variables monitored by the environment models are *true*, e.g., scenario 1 and 2 are fulfilled or all edges of the SUT model are covered, the environment transitions to the final state. This approach is used as a stopping criterion for testing (Fig. 9).

Table 2. Correspondence between code coverage and edge coverage.

Run	Edge Coverage	Code Coverage
1	64 %	55 %
2	80 %	67 %
3	100 %	78 %

5 Evaluation

The UML state machines of the HB composite REST web service had 14 states and 25 transitions. These were translated into an UPTA model with 34 locations and 46 edges. Similarly, the state machines of the Payment service had 3 states and 4 transitions which were translated into an UPTA model with 5 locations and 6 edges. The Hotel service had 4 states and 5 transitions that were translated into 7 locations and 9 edges. In addition, the environment model created had 4 locations and 13 edges.

Similarly, the two user scenarios discussed in this article (Figs. 4 and 5) comprised of 15 and respectively 11 messages which were transformed into the automaton in Fig. 8-left with 13 locations and 17 edges.

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., $e_1 \& \ldots \& e_j \& \ldots \& e_m$, where e_j are tracking variables corresponding to all m edges of the HBS models) was 974. In the contrast, the maximum number of symbolic states reported by TRON during a test session achieving complete edge coverage was 12.

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 2 s 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.

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. Table 2 lists results of several measurements.

Although many of the errors were caused by modeling mistakes, testing revealed some errors in the implementation as well. For instance, in the HB service, there was an error in sending *cancel* request and another error found in the POST header in *refund* request. Also in the Hotel service, the confirmation was sent by the wrong method, so it was rejected by Holiday Booking service. Similar errors were detected by applying Scenario-based environment model.

In order to evaluate the fault detection capabilities of our approach, we have manually created 30 mutated versions of the original HB service program code. Each mutation had one fault seeded in the code, for instance replacing POST with DELETE, removing one line of the source code, change of logical conditions, etc. The faults were always seeded in those parts of the code that is covered when achieving 100 % edge coverage of the model. We assumed that the original version of the composite web service is the correct one, as we were able to run the 100 test sessions in TRON against it. For each mutated version of the composite web service, we set the TRON to execute 100 test sessions against it. When a fault was discovered, the mutant was considered as *killed*. If the mutated statement has been covered by the test runs but no failure was detected, we mark it as *alive*. Out of the 30 mutated programs, 28 mutants were killed and 2 were alive, using the canonical test environment in Fig. 8-left. This resulted into a mutation score of 93.3 %.

6 Related Work

A large body of work on using model checking techniques for validation and verification of web service compositions has been done and overviews of works can be found in [7,29]. Mostly authors have used web service specific specification languages as starting point and converted specifications to models using model checking tools. Then, 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. We can distinguish roughly two verification approaches: those that target the PROMELA language [25] which is the input language for the SPIN model-checker [17], and those that target the UPPAAL timed automata as modeling tool [4]. In the following, we will revisit those works which are most similar to ours.

Garcia [14] uses counterexamples to specify and generate test cases in model checking tool. The transitions in BPEL define the test requirements. The transitions are mapped to the model expressed in LTL properties. Fu et al. [13] provide a framework for both bottom-up and top-down approach analyzing web service compositions. In top-down, the conversation of a web service is specified as guarded automaton converted to PROMELA modeled in SPIN model-checker. The bottom-up approach translates BPEL to guarded automaton and used SPIN tool after translating guarded automaton to PROMELA. The synchronization of web service conversations are analyzed in order to verify the compatibility.

Huang et al. [18] present a work that automatically translate OWL-S specification of composite web service into a C-like specification language and PDDL. These can be processed with the BLAST model-checker which can generate positive and negative test cases of a particular formula.

These works focus on BPEL processes and OWL-S which make them dependent on specific execution languages for SOAP based services whereas our work is not dependent on implementation and supports REST architectural style. Besides, they do not support requirement traceability and is not clear how tests are generated and executed. Furthermore, the PROMELA language cannot address real-time properties, due to the limited support for time in PROMELA. Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL into a network of timed automata using UPPAAL tool [8]. They model the requirements by extending KAOS goal model. The work is supported by WST tool that provides model transformation of timed composite web services [9]. Diaz et al. also provide a translation from WS-BPEL to UPPAAL timed automata [10]. 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 [19] specify safety and security non-functional properties in BPEL and later formulated into guards in the UPPAAL model. They do not consider neither real-time properties nor test generation. In [15], Nawal and Godart use UPPAAL to check compatibility of web service choreography supporting asynchronous timed communications. They distinguished between full and partial compatibility and full incompatibility of web services. 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 the web services.

Zhang [30] suggest the use of the temporal logic XYZ/ADL language [31] for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) which are verified in UPPAAL. In [21], uses BPEL 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. The time properties are added manually to the IF specification, while we specify them at UML level.

Biswal et al. present a test generation approach using UML activity diagram to define scenarios [6]. Arnold et al. provide a framework that supports automatic test generation from scenarios and also transforms them to test cases that can run on actual IUT [3]. Enoiu et al. presented an approach to generate test suites for PLC software using UPPAAL [11]. Larsen et al. presented an approach in which scenario-based requirements are translated to timed automata, reducing the problem of model consistency and verification effort [22]. These works provide approaches to verify and validate the service specifications by checking the properties of interest using UPPAAL. However, in our work, in addition to model checking the properties we also perform conformance testing of the service composition via online scenario-based testing with the TRON tool and we provide requirement traceability for non-deterministic systems.

7 Conclusions

We have presented a scenario-based approach to verify 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 w.r.t user scenarios. From the verified specification, we generate tests using an online model-based testing tool. The use of online model-based testing 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 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 (2013). https://pypi.python.org/pypi/coverage (Retrieved: 20 August 2013)
2. Nomagic MagicDraw, August 2013. webpage at http://www.nomagic.com/products/magicdraw/
3. Arnold, D., Corriveau, J.P., Shi, W.: A scenario-driven approach to model-based testing (2010)
4. Behrmann, G., et al.: Uppaal 4.0. In: QEST 2006 Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, pp. 125–126. IEEE Computer Society, Washington, DC (2006)
5. Birgit Demuth, C.W.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, pp. 81–89 (2009)
6. Biswal, B., Nanda, P., Mohapatra, D.: A novel approach for scenario-based test case generation. In: International Conference on Information Technology, ICIT 2008, pp. 244–247, December 2008
7. Bozkurt, M., et al.: Testing web services: a survey. Department of Computer Science, Kings College London, Technical report TR-10-01 (2010)
8. Cambronero, M.E., et al.: Validation and verification of web services choreographies by using timed automata. J. Logic Algebraic Program. **80**(1), 25–49 (2011)
9. Cambronero, M., et al.: WST: a tool supporting timed composite web services model transformation. Simulation **88**(3), 349–364 (2012)
10. Dıaz, G., et al.: Model checking techniques applied to the design of web services. CLEI Electron. J. 10(2) (2007)

11. Enoiu, E.P., Sundmark, D., Pettersson, P.: Model-based test suite generation for function block diagrams using the uppaal model checker. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013, pp. 158–167. IEEE Computer Society, Washington, DC (2013). http://dx.doi.org/10.1109/ICSTW.2013.27
12. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California (2000)
13. Fu, X., et al.: Synchronizability of conversations among web services. IEEE Trans. Softw. Eng. **31**(12), 1042–1055 (2005)
14. García-Fanjul, J., 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), p. 83 (2006)
15. Guermouche, N., Godart, C.: Timed model checking based approach for web services analysis. In: IEEE International Conference on Web Services, ICWS 2009, pp. 213–221. IEEE (2009)
16. Holovaty, A., Kaplan-Moss, J.: The definitive guide to Django: web development done right. Apress (2009)
17. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
18. Huang, H., et al.: Automated model checking and testing for composite web services. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2005, pp. 300–307. IEEE (2005)
19. Ibrahim, N., Al Ani, I.: Beyond functional verification of web services compositions. J. Emerg. Trends Comput. Inf. Sci. **4**, 25–30 (2013). Special Issue
20. Koskinen, M., et al.: Combining model-based testing and continuous integration. In: Proceedings of the International Conference on Software Engineering Advances (ICSEA 2013). IARIA, October 2013 (to appear)
21. Lallali, M., et al.: Automatic timed test case generation for web services composition. In: IEEE Sixth European Conference on Web Services, ECOWS 2008, pp. 53–62. IEEE (2008)
22. Larsen, K., Li, S., Nielsen, B., Pusinskas, S.: Scenario-based analysis and synthesis of real-time systems using uppaal. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 447–452, March 2010
23. Larsen, K.G., et al.: UPPAAL in a nutshell. Int. J. Softw. Tools Tech. Transf. (STTT) **1**(1), 134–152 (1997)
24. Larsen, K.G., et al.: CISS, BRICS. UPPAAL Tron user manual. Aalborg University, Aalborg (2009)
25. Part, I., Peschke, M.: Design and validation of computer protocols (2003)
26. Rauf, I.: Design and Validation of Stateful Composite RESTful Web Services. Ph.D. thesis (2014)
27. Rauf, I., et al.: An integrated approach for designing and validating rest web service compositions. In: Monfort, V., Krempels, K.H. (eds.) 10th International Conference on Web Information Systems and Technologies, vol. 1, p. 104–115. SCITEPRESS Digital Library (2014)
28. Richardson, L., Ruby, S.: RESTful web services. O'Reilly (2008)
29. Rusli, H.M., et al.: Testing web services composition: a mapping study. Commun. IBIMA **2007**, 34–48 (2011)
30. Zhang, G., Shi, H., Rong, M., Di, H.: Model checking for asynchronous web service composition based on XYZ/ADL. In: Gong, Z., Luo, X., Chen, J., Lei, J., Wang, F.L. (eds.) WISM 2011, Part II. LNCS, vol. 6988, pp. 428–435. Springer, Heidelberg (2011)
31. Zhu, X.Y., Tang, Z.S.: A temporal logic-based software architecture description language xyz/adl. J. Softw. **14**(4), 713–720 (2003)