An Integrated Approach for Designing and Validating REST Web Service Compositions

Irum Rauf, Faezeh Siavashi, Dragos Truscan and Ivan Porres
Åbo Akademi University, Dept. of Information Technologies, Turku, Finland

Keywords: REST, Web Service Composition, Model-based Testing, UPPAAL, TRON.

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 for computing coverage level during test generation. We validate our approach with a case study of a holiday booking web service.

1 INTRODUCTION

Web services have machine-readable interfaces that automate the task of communicating information between machines and reduce time and human efforts. They are being increasingly used in the industry in order to automate different tasks and offer services to a bigger audience. REST architectural style aims at producing scalable and extensible web services using technologies that play well with the existing tools and infrastructure of the internet. This has encouraged notable enterprizes to use REST web services to meet their needs. REST interfaces offer a CRUD interface (create, retrieve, update and delete) to its users via a set of standard HTTP methods. They offer stateless behavior that facilitates scalability and requires that no hidden session or state information be carried between method calls.

Different web services published over the internet can be composed to form a composite web service such that the composed web service fulfills 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. Designing and developing such stateful composite services with REST features is not a trivial task and requires rigorous approaches that are capable of creating reliable web services.

Thus, with the use of web services in businesses and critical applications, there is an increasing need for a) design approaches to develop web service compositions that support complex scenarios and timed behavior while complying with the REST architectural style and b) validation approaches to effectively and efficiently detect faults in specifications and implementations of such services.

In this article, we present a design and validation approach that facilitates the service developer to create reliable, timed and stateful composite REST web services. As a first contribution, we introduce an approach in which we use the Unified Modeling Language (UML) (UML, 2009) to model our service specifications via an extension of our previous work in (Porres and Rauf, 2011). We use UML since it has emerged as a standard modeling language at industrial level (Budgen et al., 2011) and has sophisticated tools due to large user base. This can make adoption of the approach easier in the industry.

The service design models and their implementations should be validated for their correct behavior in order to build trust on the service functionality. We have used the model checking approach for this purpose. Model checking is a way to exhaustively and automatically check if a finite-state model of a program satisfies its specifications (Clarke et al., 1994). The UML-based service design models represent the system graphically and are comprehensible for a human user. In order to make the models amenable

104 Rauf I., Siavashi F., Truscan D. and Porres I..
An Integrated Approach for Designing and Validating REST Web Service Compositions.
DOI: 10.5220/0004949601040115
In *Proceedings of the 10th International Conference on Web Information Systems and Technologies* (WEBIST-2014), pages 104-115
ISBN: 978-989-758-023-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

for model checking we suggest, as a second contribution, a set of reversible mechanized steps for translating UML-based service specifications into UPPAAL timed automata (UPTA) (Larsen et al., 2009). UPPAAL is a commonly used model checking tool for verifying real time systems through modeling and simulation (Larsen et al., 1997). We verify basic properties of our design models such as reachability, liveness and safety using the UPPAAL model-checker tool. UPTA are updated (if needed) based on the verification results and transformed back to UML. From the UML models, a skeleton of the composite service is generated automatically in the Django web development framework (Holovaty and Kaplan-Moss, 2009) using our partial code generation tool.

In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We need to check if the service implementation is functioning correctly along with partner services and if the service goals and timed constraints are being fulfilled. Thus, as a third contribution, we show how we validate the implementation of the RESTful web service composition with a model-based testing (MBT) approach using the UPPAAL TRON tool (Larsen et al., 2009). By using MBT, test cases can be automatically generated with an increased probability of test coverage and with an ease of test case maintenance.

Requirements traceability is an important component of our integrated approach. The requirements of the composition are included in the UML specifications and then propagated to UPTA specifications. 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, traced requirements can be used to localize the errors either in the models or in the specifications.

We exemplify and validate our integrated approach with a relatively complex case study of a holiday booking composite REST web service from industrial context. The case study shows how stateful and timed web services offering complex scenarios and involving other web services can be constructed using our approach.

The paper is organized as follows: Section II gives an overview of our approach, while tool support is discussed in section III. The case study is presented in section IV and the evaluation 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–left), we designed behavioral interfaces for web services that were RESTful by construction (Porres and Rauf, 2011). These design models were implemented in the Django web framework using our partial code generation tool (Rauf and Porres, 2011) which generated code skeletons with pre- and post-conditions for every service method. The design models were also analyzed for their consistency (Rauf et al., 2012).

In the current work, we focus on designing, verifying and testing composite REST web service (Figure 1–right) as follows:
Design: First of all we extend our design approach to create composite REST web services with UML. Our approach takes as input the behavioral interface specifications of the partner REST web services and the business requirements. With these inputs, we construct models for composite web service using our design approach.
Verification: We then provide verification of the design models by reasoning on the basic properties of models like deadlock, liveness, reachability and safety with the UPPAAL model-checker. To achieve this, we transform the service design models to UPTA, which are simulated and verified. Based on verification results, the UML-based service design models are updated.
Transformation: Transformation step generates two types of automata from service design models. One of the types corresponds to service design models and the other type represents the environment model. The environment model simulates the behavior of service user to invoke the interface service methods in order to facilitate test generation.
Code Generation: The code skeletons are generated from UML service design models to Python-based Django web framework using our code generation tool (Rauf and Porres, 2011) that are completed manually by the developer.
Testing: For model-based testing of the service implementation, we have used online conformance testing tool UPPAL-TRON that validates the service implementation against its UPTA specification models at runtime.
Evaluation: In the end, we have evaluated our validation approach for its efficiency using mutation testing and benchmarked.

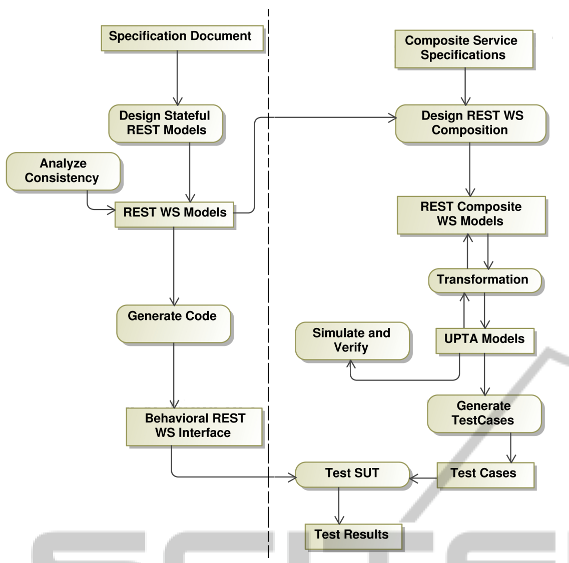


Figure 1: Activity Diagram of Design and Validation Approach for REST CWS.

2.1 REST Composition 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 (Porres and Rauf, 2011). For the details of the design approach, readers are referred to (Porres and Rauf, 2011).

The concept of resource is central to the structure of REST web service. It represents a piece of information (Richardson and Ruby, 2008). We respresent the static structure of REST web service with resource model which is modeled with a UML class diagram. Each class represents a *resource definition*. We have used the term *resource definition* to define a resource entity such that its instances are called resources. This is analogue to the relationship between a *class* and its *objects* in the object-oriented paradigm.

The direction of the association between *resource definitions* gives the navigability direction between them while their role names give the relative URI of resources (addressability). The *collection resource definiton* without the incoming transitions is termed as *root* such that every *resource definition* in the 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 it is modeled by a UML State Machine (SM). 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 the REST interface is preserved 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 the resources without maintaining any hidden session or state information (statelessness). The state invariants in the SM are written as Object Constrain Language (OCL) expressions. OCL is commonly used to define constraints in UML models, including state invariants (Birgit Demuth, 2009).

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. 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

Model verification is a process of determining whether the models are designed correctly and represent 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 service designer on the models before implementing them. This allows one to eliminate 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 (Larsen et al., 1997). It consists of set of timed automata (TA), clocks, channels that synchronize the systems (automata), variables and additional elements. A real-time system is modeled as a closed network of TA. Each automaton in the network is specified via a *template*, which can be instantiated as *process*. 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 that enable a transition when they are satisfied. Similarly, the conditions associated to locations, called *invariant*, specify that the system can stay in the location if and only if the invariant is satisfiable.

The query language used in UPPAAL is a simplified version of TCTL (Alur et al., 1990) that consists of state formulae and path formulae. State formulae (φ) is an expression that describes an individual state, while path formulae can be classified into reachability, safety and liveness properties. Deadlock is expressed using state formulae. The syntax of TCTL path formulae that are used in UPPAAL is defined as follows:

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

If there is a location 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 present our tool support for this and explain in detail the transformation from UML to UPTA.

2.3 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 (Utting and Legeard, 2007). 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 both the SM of SUT and the environment model.

We provide automatic test generation using UPPAAL TRON, which is an extension of UPPAAL for online model-based black-box conformance testing (Larsen et al., 2009). 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 converting 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 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.

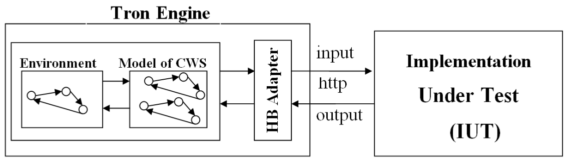


Figure 2: UPPAAL TRON test setup.

2.4 Requirements Traceability

Service requirements can be inferred from the specification document and they serve as service goals. A service should be checked for its service goals in order to validate that the service does what it is required to do. By catering to the 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 specifications. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the 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 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, their coverage level 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.

3 TOOL SUPPORT

Modeling in UML. The design models are modeled using MagicDraw (Mag, 2013). 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 (Rauf and Porres, 2011). The tool generates code skeleton for design models in Django that is a high level Python web framework (Holovaty and Kaplan-Moss, 2009). 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 (Nobakht and Truscan, 2013). 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. In UPTA the resource model is represented as a template. The *resource definitions* in the resource model are specified as variables with 1 or 0

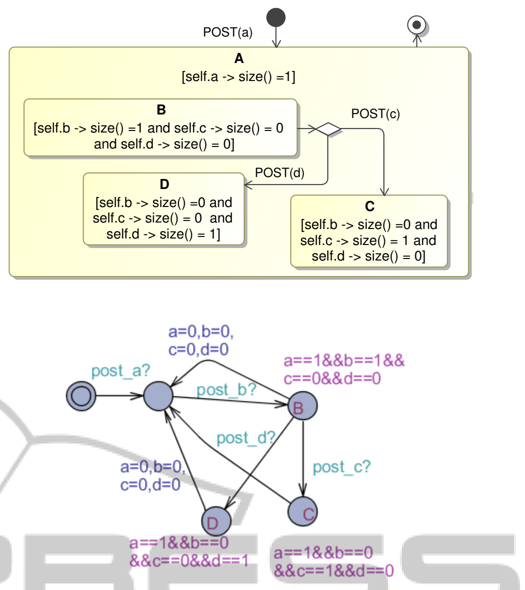

POST(a)
A
[self.a -> size() =1]
B
[self.b -> size() =1 and self.c -> size() = 0 and self.d -> size() = 0]
POST(c)
POST(d)
D
[self.b -> size() =0 and self.c -> size() = 0 and self.d -> size() = 1]
C
[self.b -> size() =0 and self.c -> size() = 1 and self.d -> size() = 0]
a=0,b=0, c=0,d=0
post_a?
post_b?
a==1&&b==1&& c==0&&d==0
B
post_d?
a=0,b=0, c=0,d=0
post_c?
D
C
a==1&&b==0 &&c==0&&d==1
a==1&&b==0 &&c==1&&d==0

Figure 3: (Left) Composite State in UML State Machine. (Right) Flattened locations in timed automaton.

value, specifying if a resource exists or not, respectively. 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 arrays of 0 and 1.

Domain Model. The domain model shows set of operations offered and required by the composite web service 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.

Behavioral Model. The SM of the REST web service is encoded to TA that are represented by templates, which are instantiated as processes. Figure 3 shows an example of transformation from the SM to TA.

States. A state is mapped to a location in UPTA, and a state invariant is mapped to corresponding location invariant. The subclauses of the state invariant are translated to 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 the SM is replaced by two edges in the TA model that are originating from the same source location to different target locations.

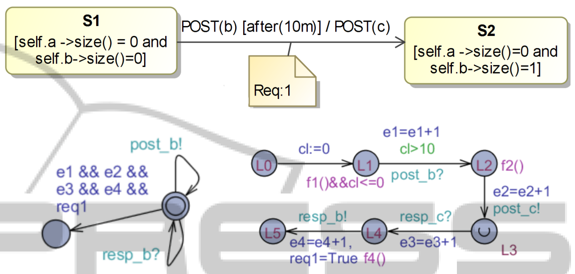
State Hierarchy. The SM may contain composite states for better representation of specifications. UPTA, however, does not support the notion of location hierarchy. We flatten the composite states into several simple states by including the state invariant of super states in the contained states that are then mapped to the respective locations in UPTA. For example, in Figure 3, the top figure contains a SM with a composite state that is flattened to UPTA model shown at the bottom. States B, C and D in the SM 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 SM.

Transitions. A transition in the SM 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 the SM (top) are translated to UPTA (bottom right). The locations $L1$ and $L5$ correspond to states $S1$ and $S2$ of SM, respectively, and locations $L0$, $L2$, $L3$, and $L4$ are the extra locations created during the transformation process as explained below. The state invariants are translated to location invariants and represented as boolean functions for the purpose of diagram clarity. The transition between states S1 and S2 is triggered by POST(b) after 10 minutes as specified in the guard. In UPTA, this is represented as guard over the clock variable *cl*.

Trigger Methods. The trigger methods from the SM are translated in to receiving channels in UPTA. This receiving channel is in sync either with the automaton of the 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 $cl > 10$ on the corresponding UPTA edge.

Effects. The effect on the transition, i.e., $POST(c)$ shows invocation to the partner service. The communication between two web services is established by using a unique channel synchronization. For instance, emitting a request from a web service to the other one can be replaced by synchronizing a channel in an UPPAAL process, and the other process is the receiver of the synchronization. The effect of the 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 (Larsen et al., 1997). Thus, the first edge (e1) is synchronized with the environment model and the second edge (e2) synchronizes with the partner 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.


S1
[self.a ->size() = 0 and self.b->size()=0]
POST(b) [after(10m)] / POST(c)
S2
[self.a ->size()=0 and self.b->size()=1]
Req:1
post_b!
e1 && e2 && e3 && e4 && req1
resp_b?
L0
cl:=0
L1
f1()&&cl<=0
e1=e1+1
cl>10
post_b?
L2
f2()
e2=e2+1
post_c!
L3
resp_c?
e3=e3+1
L4
resp_b!
e4=e4+1, req1=True
f4()
L5
Figure 4: Example of SM (top) Corresponding Environment Model (bottom left) and Flattened TA (bottom right).

Requirements. The requirement on the transitions are translated into a boolean variable (initialized to *False*) and attached to the corresponding edge which updates it to *True*. 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. This can be formulated as reachability properties to attain requirement coverage and tracked during test generation and execution.

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 the SM. All the interface methods of the service specified in the state machine 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 channels. This is also shown in Figure 4: the environment model initiates 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, from a given UPTA model by analyzing the channels observable from the environment. The original idea has been discussed in (Hessel et al., 2008). This will be merged in the final version of the UML→UPTA transformation script.

Test Coverage Information. In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in (Koskinen et al., 2013)) 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 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$\rightarrow$UPTA transformation script.

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 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 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, 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 and information here. The detailed case study is available at (Rauf et al., 2013)

Design Models. The design of HB composite REST web service is modeled with resource, behavioral and domain models. The state machine of HB composite service is shown in Figure 5.

Requirements Traceability. We have inferred functional and temporal requirements from specification document for our case study. Table 1 shows the requirements for *Booking* and *Payment Release*. These requirements should be fulfilled by the IUT in order to satisfy the service goals. They are added as comments to the model in Figure 5.

Verification. The design models of Holiday Booking (HB) composite REST web service are translated to UPTA with the help of transformation tool. Here, we only show an excerpt of UPTA in Figure 6. The detailed model and the specifications of the partner web services are available in (Rauf et al., 2013).

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

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- ...	2.1 - ...
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

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

Deadlock Freeness. The HB Service, the hotel service and the payment service models are all deadlock free. This means that the composite service is never reach to a state that cannot preform a transition (i.e., $A[]\ not\ deadlock$). Note that the following queries are made for complete model and only some of them can be traced in Figure 6.

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 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 $CardService.c2$), 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 hours, 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.

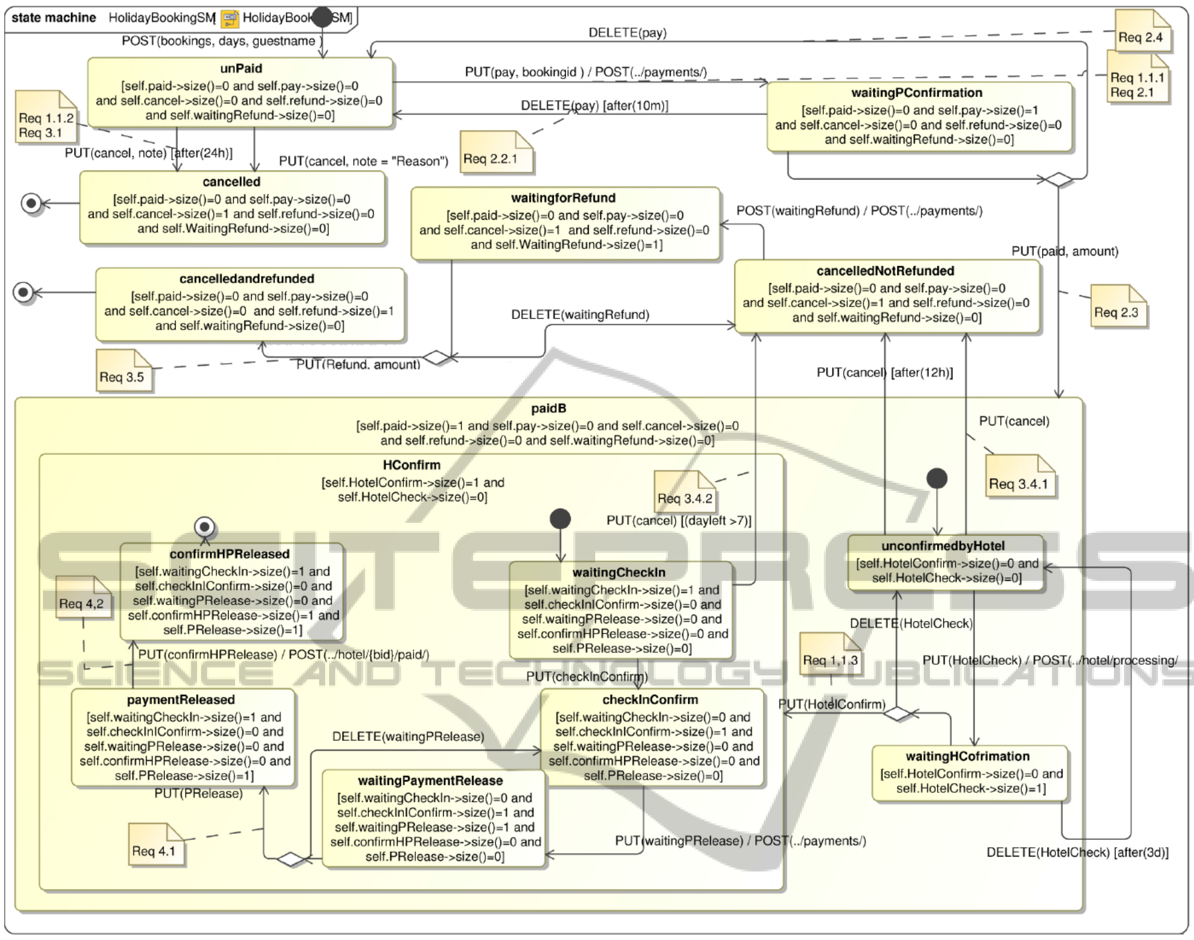


Figure 5: UML State Machine of Holiday Booking Composite REST Web Service.

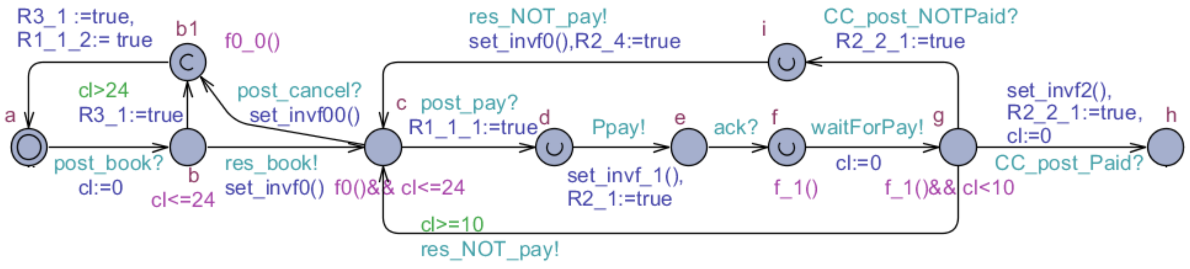


Figure 6: Excerpt of UPTA model of Holiday Booking Composite REST Web Service.

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 7 shows the environment model satisfying edge and requirements coverage. In Figure 7 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 6. Whenever the verdict function evaluates to TRUE environment model can go to the final location.

Test Setup. Similar to Figure 2, the test setup comprises the TRON engine, the adapter, and the IUT. The IUT is a web service composition of three web services: Holiday Booking, Hotel and Payment Services. The test adapter composed of a set of test cases which satisfy the test requirements that are listed in Table1.

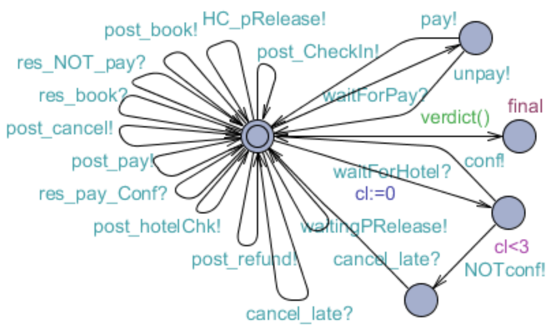


Figure 7: Environment model.

5 EVALUATION

The UML state machines of the HB composite REST web service had 14 states and 25 transitions. These were translated into a UPTA model with 34 locations and 46 edges. Similarly, the state machines of Payment service had 3 states and 4 transitions which transformed in to a 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.

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 974. In the contrast, the maximum number of symbolic states reported by TRON during a test session achieving complete edge coverage was 12 (see Figure11).

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. Figure 11 plots the evolution of the number of symbolic states for 10000 model time units (20 seconds).

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 (pyt, 2013) 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	100%	78%

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.

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. This resulted into a mutation score of 93.3%.

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

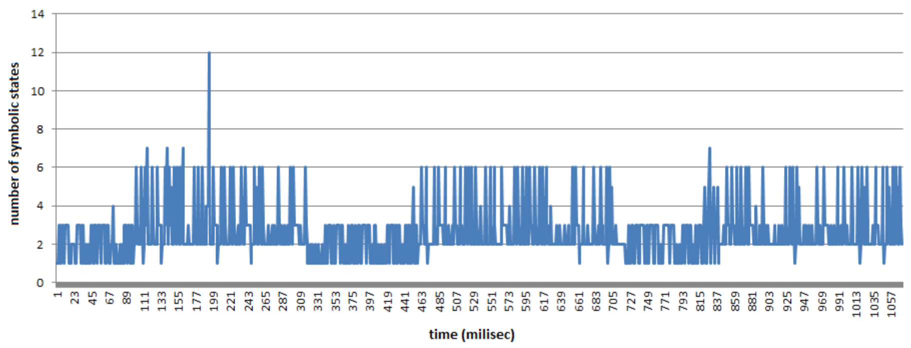


Figure 8: Evolution of symbolic states.

can be found in (Rusli et al., 2011) and (Bozkurt and other, 2010). 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 (Part and Peschke, 2003) which is the input language for the SPIN model-checker (Holzmann, 1997), and those that target the UPPAAL timed automata which is the input language for the UPPAAL model-checker (Behrmann et al.,).

In the first category, the vast majority of approaches have used BPEL or OWL-S(Martin et al., 2004) for the specification of the web service composition. For instance, Garcia (García-Fanjul et al., 2006) 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. Transition coverage is obtained by repeatedly executing the tool with each previously identified transition.

Fu. et al. (Fu et al., 2005) provide 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 that are converted to PROMELA and used as input to SPIN model-checker. The bottom-up approach translates BPEL to guarded automaton and then used 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. (Huang et al., 2005). They automatically translate OWL-S specification of composite web service into a C-like specification language and PDDL through an 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.

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 (Cambronero et al., 2011), Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL 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 (Cambronero et al., 2012). In (Dıaz et al., 2007), 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 (Ibrahim and Al Ani, 2013) 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 (Guermouche and Godart, 2009), 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. 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 (Zhang et al., 2011) suggest the use of the temporal logic XYZ/ADL language (Zhu and Tang, 2003) for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) which are verified in UPPAAL.

In (Lallali et al., 2008), 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$\rightarrow$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

(2013). Code coverage measurement for Python – coverage, v. 3.6. Online at https://pypi.python.org/pypi/coverage. retrieved: 20.08.2013.

(2013). Nomagic MagicDraw webpage at http://www.nomagic.com/products/magicdraw/.

Alur, R. et al. (1990). 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.

Behrmann, G. 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.

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

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

Budgen, D., Burn, A. J., Brereton, O. P., Kitchenham, B. A., and Pretorius, R. (2011). Empirical evidence about the uml: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392.

Cambronero, M. et al. (2012). Wst: a tool supporting timed composite web services model transformation. *Simulation*, 88(3):349–364.

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

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542.

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

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

García-Fanjul, J. et al. (2006). 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.

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

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

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

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

Huang, H. et al. (2005). 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.

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

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

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

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

Larsen, K. G. et al. (2009). UPPAAL Tron user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*.

Martin, D. et al. (2004). OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04.

Nobakht, M. and Truscan, D. (2013). An Approach for Validation, Verification, and Model-Based Testing of UML-Based Real-Time Systems. In Lavazza, L., Oberhauser, R., Martin, A., Hassine, J., Gebhart, M., and Jntti, M., editors, *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 79–85. IARIA.

Part, I. and Peschke, M. (2003). Design and validation of computer protocols.

Porres, I. and Rauf, I. (2011). Modeling behavioral RESTful web service interfaces in UML. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1598–1605. ACM.

Rauf, I. et al. (2012). Analyzing Consistency of Behavioral REST Web Service Interfaces. In Silva, J. and Tiezzi, F., editors, *The 8th International Workshop on Automated Specification and Verification of Web Systems*, Electronic Proceedings in Theoretical Computer Science, page 115. EPTCS.

Rauf, I. and Porres, I. (2011). Beyond CRUD. pages 117–135.

Rauf, I., Siavashi, F., Truscan, D., and Porres, I. (2013). An Integrated Approach to Design and Validate REST Web Service Compositions. Technical Report 1097.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly.

Rusli, H. M. et al. (2011). Testing Web services composition: a mapping study. *Communications of the IBIMA*, 2007:34–48.

UML, O. (2009). 2.2 Superstructure Specification. *OMG ed*. http://www.omg.org/spec/UML/2.2/.

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

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

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