On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services

Faezeh Siavashi[1], Dragos Truscan[1] and Jüri Vain[2]
[1]*Faculty of Science and Engineering, Åbo Akademi University, Vattenborgsvägen 3, 20500, Turku, Finland*
[2]*Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, Tallinn, Estonia*

Keywords: Web Service Composition, Specification Mutation, Robustness Testing, Model-based Testing, UPPAAL, TRON.

Abstract: We present a model-based mutation technique for testing the robustness of Web service compositions. Specifications of a Web service composition is modeled by UPPAAL Timed Automata and the conformance between the model and the implementation is validated by online model-based testing with the UPPAAL TRON tool. By applying a set of well-defined mutation operators, we generated model mutations. We validate all generate mutants and exclude the invalid ones. The remaining mutants are used for online robustness testing providing invalid test inputs and revealing vulnerabilities of the implementation under test. We experimented our method on a Booking System web service composition. The results show that from a total of 1346 generated mutants, 393 are found suitable for online model-based testing. After running the tests, 40 of the mutants revealed 3 new errors in the implementation. The experiment shows that our approach of mutating specifications is effective in detecting errors that were not revealing in the conventional conformance testing methods.

1 INTRODUCTION

Recently, the popularity of web services has increased in the industry. Web services are software applications that support machine-to-machine interactions over the Internet. They are accessible via ubiquitous protocols while expressing a well-defined interface. This advantage opens the door to new business opportunities by making it easy to communicate with partner services and by covering a wider range of users. Web Service Composition (WSC) is the combination of different services to satisfy a new service. Examples of using WSC can be seen in many web applications that enhance their services by using utilities that are offered by famous companies such as Google, Amazon, and Facebook (Sheng et al., 2014).

One principle characteristics of a WSC is its distributed resources, where other services or client web applications access to information by message protocols. This kind of systems should be robust against erroneous inputs. In this context, testing WSCs plays an important role. Not only the expected behavior of the implementation under test (IUT) should be tested, but also the IUT should not contain any unexpected behavior. The functionality of the system can be checked by running test cases derived from the specification while finding unexpected behaviors of the system can be done by *robustness testing*, which executes invalid inputs and detects the vulnerabilities or unexpected behavior of the IUT.

Defining test inputs by modeling the specifications is preferred over the manually written test scripts since the machine can verify the correctness of the models and automatically generate the test inputs. Moreover, it supports more extensive and systematically constructed sets of test cases.

One way to create invalid test inputs is using *mutation testing*, where a set of well-defined mutation operators systematically create syntactic changes to the specifications and produce mutants. This concept was primarily applied for mutating the source code of a system, however, it has also been applied to different modeling languages as well (Budd and Gopal, 1985). Mutants generate invalid scenarios as test cases, which are executed against the IUT. If the IUT respects the mutation without raising an exception, it means that its behavior is inconsistent with its specification (i.e, the IUT accepts an unspecified sequence of inputs).

In this paper, we propose an approach for robustness testing of WSCs using UPPAAL Timed Automata (UTA). The conformance between the model and the

Siavashi, F., Truscan, D. and Vain, J.
On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services.
DOI: 10.5220/0005970800150026
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 15-26
ISBN: 978-989-758-194-6

IUT is first checked via UPPAAL TRON, an online testing tool which supports both test generation and test execution. In online testing, only one test input is generated and executed on the IUT at a time, and based on the test output the next test input will be selected.

As a first contribution, we introduce a testing method, which derives mutants from the specification and executes them via online testing. We use a selection of the mutation operators that are defined in the literature and slightly change them to generate mutants that are suitable for our work.

As a second contribution, in our methodology, we add verification properties to mutated model segments to ensure reachability of the mutated elements at runtime. This step is supported by a mutation generator tool, which implements selected mutation operators and performs early verification of each mutant. If a mutant does not pass the verification properties, it cannot be used for online testing, hence, we eliminate them. Furthermore, to ensure that the mutated part will be executed during the testing process, we monitor whether the mutated elements are reached during test execution.

As a third contribution, we empirically evaluate which existing mutation operators for UPPAAL timed automata are applicable to online testing. We define two formulas to measure the efficiency of mutation operators as well as their rates of fault detection.

The remainder of this paper is organized as follows: In Section 2, we briefly review the background studies. We present the steps of our methodology on specification mutation testing in Section 3 and selection criteria for valid mutants. The experiment is presented in Section 4. The results are discussed and possible improvements are suggested in Section 6, and the threats of validity of the proposed method are discussed in Section 7. We review the literature for related work in Section 8. Finally, we conclude our study and present future work in Section 9.

2 Background

We first review UPPAAL tool set, and introduce the conformance testing with UPPAAL TRON and the concept of specification mutation testing.

2.1 UPPAAL Timed Automata (UTA)

UPPAAL is a model-checker tool for modeling, simulation, and verification of real-time systems using an extended version of timed automata called UPPAAL timed automata (UTA) (Beharmann et al., 2004). A timed automaton is a finite state machine with locations, actions, and clocks.

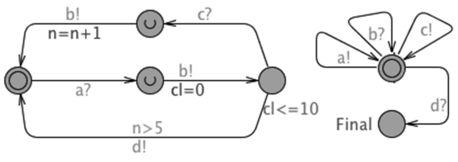


Figure 1: Example of an UTA model.

In UPPAAL, a system is designed as a network of several such timed-automata called processes working in parallel. A process can be executed individually or in sync with another process. Synchronization of two processes is possible by using input/output actions (denoted as ”!” for emitting and ”?” for receiving synchronizations, respectively). The processes consist of locations and edges. The state of the system can be shown by the locations of all processes, their clock values, and their variable values. The edges between locations represent state transitions including clock resets. UPPAAL is extended further with global and local to some process variables that can be of type *integer*, *boolean*, and *clock*.

Transitions can be constrained by predicates (over the clocks or variables) known as *guards*, which defines when the corresponding edge is enabled. The state transitions are specified on edges as variable updates. A location can be restricted over the clock invariants, which specify how long the system can stay in that location. If there is more than one enabled edge at a time, then one of them will be randomly selected. This means that UPPAAL supports non-deterministic modeling, which gives more freedom to represent behaviors, especially in systems with random discrete events (Hessel et al., 2008).

An example of a UTA model is shown in Figure 1. The model consists of two automata modeling the behavior of a system under test and of its environment. The communication between the system and its environment is modeled using channel synchronizations and shared variables.

The UPPAAL model-checker uses a simplified version of TCTL (Alur et al., 1990), which enables to exhaustively verify the models w.r.t their specifications. The query language consists of state formulae and path formulae. State formulae (φ) is an expression that describes the properties of an individual state while path formulae can be used to specify which properties (like reachability, safety, and liveness) hold over a given path (Beharmann et al., 2004).

If there is a state in the model that has no enabled outgoing transitions, then the model is said to be in a deadlock. *A* $\square$ *not deadlock* query, can be used to verify that for all paths in the model, there is no deadlock state.

The safety property checks that "something bad will never happen". In UPPAAL it can be expressed in the form $A \square \varphi$ (φ should be true in all reachable states) and $E \square \varphi$ (there should exist a maximal path such that φ is always true).

The liveness property determines that "something will eventually happen" and it is shown by $A \lozenge \varphi$ (φ is eventually satisfied) and $\varphi \leadsto \phi$ (whenever φ is satisfied, then eventually ϕ will be satisfied).

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 reachability can be expressed in the form of $E \lozenge \varphi$ (there is a path from the initial state, such that φ is eventually satisfied along that path).

2.2 Online Model-based Testing

There are two distinct approaches in testing: offline and online testing. In offline testing, the complete test scenarios and test oracle are created before the test execution, whereas online testing is a combination of test generation and execution: only one test input at a time is generated and executed and the next test input depends on the current test output (Larsen et al., 2005b). This continues until the test termination criteria are satisfied or an error occurs. Usually, the test stimulus is selected randomly from the enabled test inputs. In online testing, the state-explosion problem is reduced because only a portion of the state space is needed to be calculated and stored at each time. Also, the non-determinism of systems can be simulated on-the-fly by random selection of the tests.

In this study, we use the online Model-Based Testing (MBT) UPPAAL TRON, which is an input/output conformance testing tool for testing real-time systems based on the *rtioco* conformance relation (Larsen et al., 2005a). An UTA model typically consists of two partitions: a system partition and an environment partition. The abstract test inputs generated from the environment are translated into executable test inputs by using an *adapter*, which is an interface between TRON and the IUT. The outputs of the IUT also translated to model-level test outputs. Thus, the I/O conformance of the model and of the IUT is observed by TRON.

The result of online testing with TRON can be *passed*, *failed* or *inconclusive*. An inconclusive test result means that the environment model cannot be updated since the IUT output is unexpected or it has a delay in providing test output.

2.3 Specification Mutation Analysis

Specification mutation analysis is used to design tests to evaluate the correctness and consistency of the specification and the program (Budd and Gopal, 1985). When the mutation analysis is applied to the specification a set of *mutation operators* create slightly altered versions (mutants) of the specification. The tests will be generated from the mutated specification and used to assess whether the IUT is accepting the faulty tests.

In the literature (Belli et al., 2016) the following types of mutants are defined:

Killed: A mutant is said to be *killed* if tests generated from it fail against the implementation, under the precondition that the tests generated from the original model have passed.

Alive: A mutant is called *alive* if the IUT passes all test cases generated by the mutant. Alive mutants can be divided into two types:

Equivalent: An alive mutant is semantically equivalent if it manifests the same behavior as the original model, whereas they are syntactically different.

Non-equivalent: An alive mutant is known as *non-equivalent* if it does not have the same behavior as the original model, however, the differences cannot be detected during testing. These mutants indicate that the implementation is too permissive and is not able to detect the invalid inputs.

Our goal of using mutation for testing is to find the non-equivalent alive mutants since they show that there might be some inconsistencies between the specification and the implementation. Differing between non-equivalent alive mutants from equivalent mutants is done manually.

3 METHODOLOGY

An overview of our method is given in Figure 2. It is divided into five phases.

Design and Conformance Testing is based on our previous work on design and validation of WSCs (Rauf et al., 2014), where we presented an approach to design web services and their behavioral interfaces in UML. We transformed the design models from UML to UTA for verification and testing the implementation of a WSC.

The participating web services and the user behavior are modeled as distinct timed automata. The user behavior supports non-deterministic choices, as well as timing criteria.

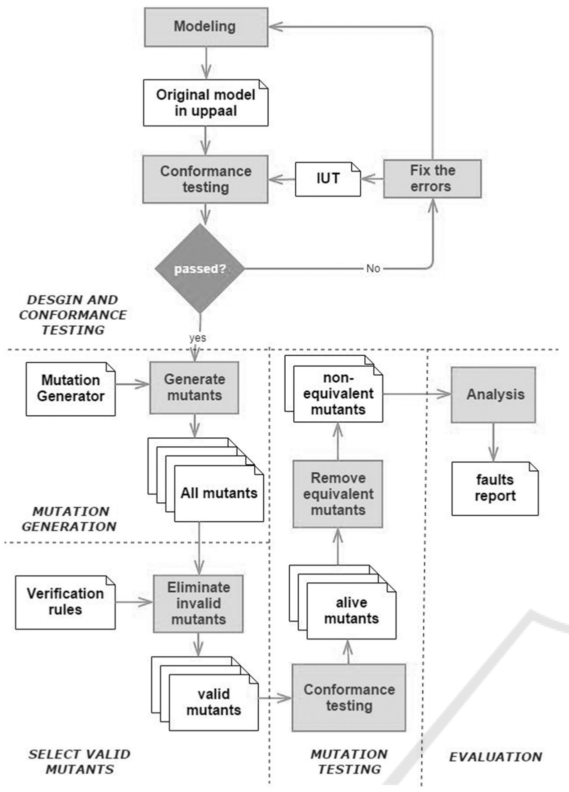


Figure 2: Our approach of Specification Mutation Testing.

The model is verified according to the criteria and timing constraints that are given in the requirements of the WSC. The verification is done using TCTL. For instance, we ensure that the model is deadlock-free and all states of the system are reachable meaning that the model can reach all test goals. These verification rules ensure that the model is usable for online testing.

With TRON, an online testing session is established and the conformance of the implementation is checked. External errors in IUT or in the model are fixed.

Mutation Generation: Mutation operators for TA have been formally defined and presented by two studies in the literature (Aboutrab et al., 2012; Aichering et al., 2013) and are shown in Table 1.

By summarizing Table 1, the following mutations can be applied to the different elements of TA.

- *Guard*: A guard over clock variables can be mutated in three ways: by widening, restricting, or shifting the time value. If the guard contains other variables than the clock variables, it can be mutated by negating the predicate.
- *Invariant*: An invariant can be changed by shifting it to a greater or smaller value. E.g., add/subtract value 1 to/from the value of the invariant.
- *Action*: Name of I/O actions can be changed to other defined actions. Also, changing their source and target locations will manipulate the behavior of the model and so can be used as a mutant.
- *Location*: A location can be made a sink location, which means that it accepts all other actions. It simulates a trap condition, where all actions in the process are accepted in the same location. Removing a location and adding a new location are other mutations that can be applied in TA.

Table 1: Mutation operators of timed-automata.

(Aboutrab et al., 2012)	(Aichering et al., 2013)
Restricting Timing Constraints (RTC)	Change guard
Widening Timing Constraints (WTC)	
Shifting Timing constraints (STC)	
-	Change invariant
Resetting a Clock (RC)	Invert reset
Not-Resetting a Clock (NRC)	
Exchanging Input Actions (EIA)	Change action
Exchanging Output Action (EOA)	
Transferring Destination Locations (TDL)	Change Target
-	Change source
-	Negate guard
-	Sink location

We have restrict some of the operators in such a way that they are suitable for online testing with TRON. As we mentioned earlier, the IUT and its environment (user, or other systems) are specified in separate automata and they communicate by synchronization of input/output transitions (actions). All transitions between the IUT and its environment are observable by TRON. Based on the type of the input or output, TRON controls which action can be executed at a time. The mutation operators for transitions without synchronizations (or internal transitions) will not be observed by TRON. Therefore, we restrict the mutation operators to only be applied to observable synchronizations.

Additionally, we adapt the mutation operators to be used for testing web services. For instance, for each HTTP request message to a web service, we have a corresponding HTTP response message and they are modeled as a pair of input/output actions. The requests are defined as input actions coming from the user (or the environment). One mutation option would be to change the name of the input actions, which mutates the sequence of the HTTP request messages. However, defining mutation for the HTTP response messages (i.e, output actions) cannot help in mutation analysis since the IUT generates them and we can only observe them. For instance, for a booking request, the WSC either accepts or rejects it and both of these responses cannot be mutated in the model-level. Therefore, we limit the mutation operators to change the name of input actions only.

Finally, we do not change the direction of the synchronizations (i.e, "?" to "!") since, in our modeling approach, the requests from the users are modeled as input actions ("?"). Changing the inputs into output actions means that the requests should be changed into responses and it would not allow test generation at all.

Below we present a list of operators that we selected from Table 1 for our methodology.

1. **Change Name of Input Action (CNI)** replaces the name of an input action (denoted by "?") with the name of other actions. Thus, the expected sequence of the inputs to the implementation will be different.
2. **Change Target (CT)** changes the target of an action to other location. This operator can break the flow of test inputs and violate the state of the IUT. Both input and output actions can be mutated by this operator.
3. **Change Source (CS)** changes the source location of an action to other locations. Similar to CT, this operator gives a different I/O sequence.
4. **Change Guard (CG)** changes the clock constants in guards by a random value. It is effective for mutating the condition of enabling an action.
5. **Negate Guard (NG)** negates guards, which may result in omitting some paths of the test model.
6. **Change Invariant (CI)** shifts the values of invariant conditions to a different range, extending or restricting the constraints of the model. It can cause actions fire earlier (or later) that the expected time.
7. **Invert Reset (IR)** deletes the resetting of the clock and moves it to one action before or after. It means that the resetting is flipped one clock earlier or later.

Figure 3 shows the generated mutants of a model and sample mutants using the above operators. In our approach, we only apply first order mutation. That is, a mutated model contains only one mutated segment based on a single operator.

Select Valid Mutants: In our approach, we enforce that every time a mutated model is generated, we create a corresponding reachability rule to check whether it is a valid mutant for online testing or not.

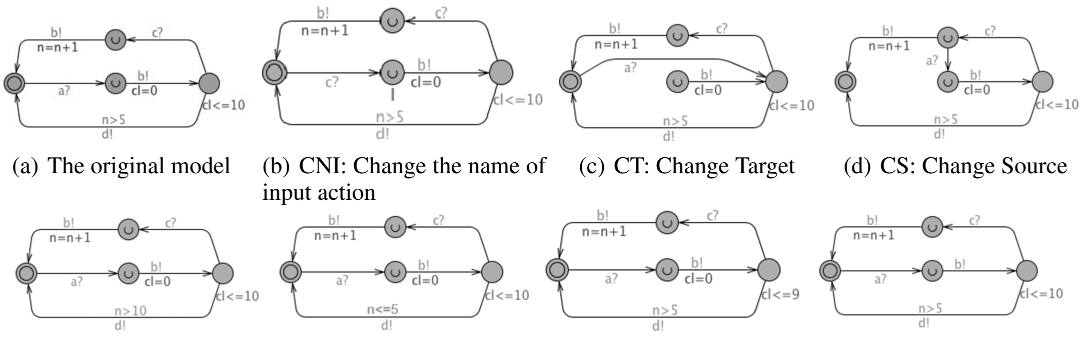
In UPPAAL, the reachability property is defined for locations, thus, when an action is mutated, we define the reachability property for the target location of that action. For instance, in Figure 3(b), the input action $a?$ is mutated into $c?$, hence, the reachability for this mutation should be defined for its target location (i.e, l). For example, in Figure 3(b), we have $E \diamond l$, which verifies that the mutation can be executed. An alternative to the reachability rule would be to define a *trap variable* (Gargantini and Heitmeyer, 1999) and set its initial value to false. For the mutated action, then, the variable will be updated to true, and so the reachability can be achieved by checking if the variable eventually will be set to true ($E \diamond trap == true$). One can use trap variables to ensure that the mutation part of the model will be reached during the test execution as well. In the case that the minimum repetitive execution of mutation is needed the boolean trap variable should be replaced by an integer counter variable *count* and the reachability condition with $E \diamond count >= const$. Those models that pass the verification process are considered as *valid mutants* and can be executed against the IUT.

Having verification rules offers two main advantages. First, it reduces the number of mutants used for testing by eliminating false negatives which cause semantic and syntactic errors. Secondly, it avoids having traps in the model, which may increase the size of the state space.

Mutation Testing: Each valid mutant model is executed in a testing session with UPPAAL TRON. The verdict of an online testing session with TRON can be *passed*, *failed*, or *inconclusive*. In TRON, an inconclusive verdict indicates that either the observed output from the IUT is not valid, or there is an unacceptable delay in sending inputs. We consider that the mutants that generate inconclusive test cases, exhibit different behavior than the original model and thus they are considered as *killed*. If the IUT passes the test, then two different scenarios are possible: either the mutant is an equivalent model to the original one (i.e, equivalent mutant), or not equivalent, but there is a defect in the implementation that allows mutated inputs (i.e, non-equivalent mutant). We defer automatic equivalence detection for future work. When executing the mutants we assume implicitly that these test runs are exhaustive w.r.t. the mutation, i.e. all mutations injected are also covered by these test runs.

Evaluation: The last phase of our methodology is to evaluate the result by reasoning about the unexpected behaviors that the IUT shows during testing. The non-equivalent mutants generate different invalid test inputs, thus, these test inputs are manually evaluated to find the correlations between them and the actual faulty behaviors.

Tool Support: We implemented the selected mutation operators as a tool in order to generate the mutants automatically. The tool uses UPPAAL TA XML



(a) The original model (b) CNI: Change the name of input action (c) CT: Change Target (d) CS: Change Source

(e) CG: Change Guard (f) NG: Negate Guard (g) CI: Change Invariant (h) IR: Invert Reset

Figure 3: A model with examples of mutants generated by the selected mutation operators.

format as input. From a given model, the tool generates mutants based with the selected mutation operators. In addition, it adds reachability and deadlock-freeness rules to the mutants and verifies them with the *verifyta* tool, which is a command-line verification tool for UPPAAL models.

4 EXPERIMENT

We exemplify our approach using the case study presented in (Rauf et al., 2014). In this section, first, we review the case study, and then we apply the specification mutation method.

4.1 Case Study

For evaluation, we used a WSC that is implemented in REpresentational State Transfer (REST) (Richardson and Ruby, 2008) architectural style. The composition of web services is based on a central service which orchestrates other services. This service synchronizes the execution of different methods on the web services participating in the composition and satisfies the specifications. The central web service (i.e, the composition service) can invoke other services while exhibits timed behaviors in a RESTful architecture.

The WSC offers a Hotel Booking System (HBS), including a Card service, a Hotel service, and a Booking service. This case study is specified, implemented and verified in our previous work in details in (Rauf et al., 2014). The Card service deals with payments and refunds for booking requests, whereas the Hotel service keeps track of the details of booking records such as name, the number of days and type of room, also giving access to the hotel manager for accepting or declining the booking requests. The Booking service is responsible for communications with customers, the Hotel, and the Card services. From the specification of HBS, we define the following scenarios:

Booking: A customer can search for a room in a hotel by accessing the booking service. He books the room (if it is available) and that booking is reserved by the Booking service for 24 hours.

Payment: If the user does not pay within 24 hours then the booking will be automatically canceled. If the booking is paid, then the Booking service invokes Card service and waits for the payment confirmation.

Hotel Confirmation: When the payment is confirmed, Booking service invokes the Hotel service to confirm the booking of the room. The Hotel service can confirm and assign a room for the customer, or it can reject the request.

Refund: If the Hotel service does not respond within 1 day, rejects the request, or does not confirm at all, the booking is canceled and the user is refunded.

Check-in: If the Hotel service confirms, then a booking is made with the hotel. The user now can check in to the hotel.

Hotel Payment Release: The payment is not released to the hotel until the user checks in. When the user checks in, the Booking service releases the money to the hotel and the booking is marked by the hotel as paid.

4.2 Model

From the above descriptions, we have specified the system as a UTA model which consists of four automata: three for the web services and one for the environment. Figure 4 shows the models of the case study and the interactions between the services and the environment. In this experiment, we mutate only the Booking service that is larger and handles the communications among other services and users. The Booking service model consists of 33 locations, 39 actions, 4 guards, and 4 clock invariants.

Table 2: Result of mutation testing.

Name	Generated	Valid	Killed	Alive
CNI	180	28	24	3
CT	567	314	242	72
CS	567	38	6	32
CG	12	6	6	0
NG	4	1	1	0
CI	12	4	4	0
IR	4	2	2	0
Total	1346	393	285	107

After verifying the model, we developed an adapter for translating the model-level inputs into HTTP requests which are sent to the IUT, and then, we generated tests using UPPAAL TRON. The use of online MBT proved beneficial as our implementation under test exhibits non-deterministic behavior. For instance, in the scenario of Hotel Confirmation, there are three possible cases from the hotel: confirmation, rejection, or no response. Any of these choices are given the same chance to be executed with non-deterministic modeling.

4.3 Generating Valid Mutants

Table 2 shows the numbers of mutants generated from each mutation operators. Since the Booking service represents the composition of different web services as well as communicating to the user, it is a good candidate to be mutated. The mutation generator provided 1346 mutants, from which 393 of mutants were valid (i.e, passed the verification rules). The total time for generation and validation of all mutants took 258 seconds in a 4 cores machine running the Ubuntu 14.4 Server operating system. As the numbers show, having verification in the early stage of testing would help in removing non-relevant mutants and hence the total time of the test execution will be considerably reduced.

As it can be seen in Table 2, a majority of 314 valid mutants are generated by the CT operator, in contrast with 38 valid mutants provided by the CS and 28 from the CNI. The other mutation operators have a small share of valid mutants.

4.4 Mutation Testing

We set the test session for executing tests 3 minutes for each mutant model covering all actions in the model ensuring that the mutated element was also covered at runtime. It roughly took 7 hours to complete running all valid mutants. The time was sufficient for covering all valid mutations of interest. Therefore, it was postulated that if no failure is detected during this time, and the test is passed, then the mutant is alive.

5 RESULTS

We check whether the alive mutants were able to show any fault in the behavior of the web services and which of the mutation operators generates more effective mutations in online testing.

We also present two formulas for the efficiency of mutation operators showing how many of the alive mutants address faults. We need, therefore, to separate the equivalent mutants from the alive mutants. The analysis is based on the reasoning why the mutated inputs could not be detected by the IUT.

Automatically detecting all equivalent mutants is an impossible task since they are undecidable (i.e, there is no possible solution to confirm that a mutant has equivalent behavior to its original program). Although there are several approaches to the detection of equivalent mutants, it still requires human effort. We manually distinguished the equivalent mutants by checking whether the mutants change the sequence of the test scenarios and how it affects the functionality of the IUT. It is done by checking if all the test scenarios can be covered by the mutants and where is the location of the mutation in the model.

It is worth noting that not all of the non-equivalent mutants cause violations in the functionality of the IUT. For example, in the model of Booking service, changing the target location of the action *post_hotelChk* to the location *a* does not cause an invalid test scenario. Despite the fact that such mutant does not cover all test scenarios, it will pass the test. The reasoning behind this is that from the initial location, *a*, any booking requests will be considered as a new booking request and will be a new booking record. Therefore, such non-equivalent mutants do not violate the functionality of the Booking service.

Since in the robustness testing the goal is to detect unexpected behaviors of the IUT, having more alive mutants indicates that the corresponding operators are more effective. Hence, we define the following formulas for analyzing the mutation operators:

Mutation Efficiency: For each mutation operator, we calculate how many mutants are alive. We calculate the efficiency of each mutation operator in generating alive mutants:

$$ME_i = \frac{A_i}{V_i}\ , \qquad (1)$$

where A is the number of alive mutants, V is the number of valid mutants of operation i.

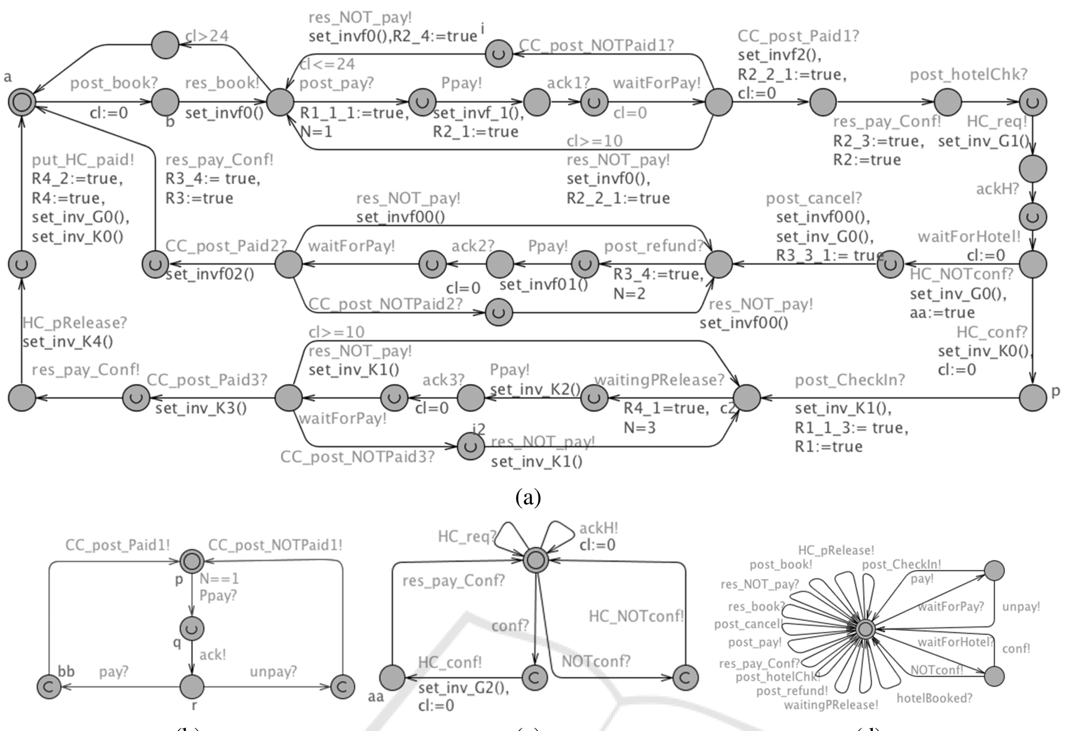


Figure 4: The model of HBS: (a) Booking service, (b) Card service, (c) Hotel service and (d) Environment.

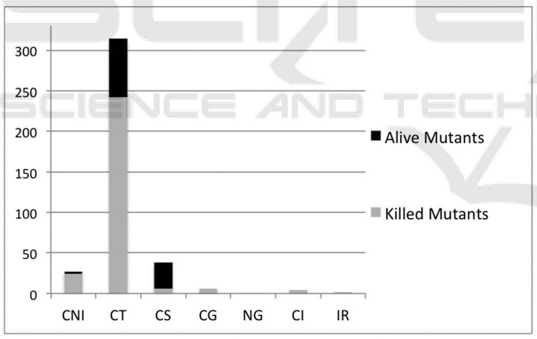


Figure 5: The proportion of alive and killed mutants for each mutation operator.

Mutation Fault Detection: After analyzing alive mutants and removing the equivalent mutants, we check which non-equivalent mutant corresponds to a fault. For each mutant that was able to show a fault in the IUT, we score the corresponding operator. For each operator, we measure the mutation fault detection with following formula:

$$MFD_i = \frac{NE_i}{T_i - E_i}\ , \qquad (2)$$

where *NE* is the number of non-equivalent mutants that reveal hidden faults, *T* is the number of total mutants and *E* is the number of equivalent mutants.

The primary result shows that the total alive mutations belong to three operators: CT, CS, and CNI with 72, 32 and 3 mutants respectively. The CT operator is a good candidate for mutation testing since it generates the highest number of alive mutants. It can be debated that changing the order of the test inputs may cause changing the state of the IUT and hence, the IUT may reach to an unknown state (i.e, unexpected condition). Moreover, as it can be observed from Figure 5 that the proportions for the total number of alive and killed mutants for each individual operator show that the mutation operators CT and CNI were the most suitable operators for our case study.

By using Formula 1, we calculated the efficiency of the operators CT, CS and CNI, which result in 22.9%, 84.2% and 10.7%, respectively (Table 3). This means that the CS operator is more effective in successfully generating alive mutations.

Table 3: Mutation efficiency and Mutation fault detection of the mutation operators.

	mutation efficiency	fault detection
CT	22.9%	62.5%
CS	84.2%	8,3%
CNI	10.7%	0

Analysis of the result shows that there are some faults in the implementation of the case study that were not detected during the conformance testing. We found the following problems in the behavior of the implementation:

- Ten different mutants revealed the same fault in the *Hotel Confirmation* scenario. For example, one faulty scenario is: from a single booking, it is possible to send the confirmation request more than once. Nine of these mutants were generated by the CT operator and one by the CS operator.
- Seventeen mutants showed that there is a fault in the *payment* scenario of the IUT. After payment confirmation from Card service, a new payment for the same booking can be made. Also, for a single booking, there could be several payments. Seven of these mutants are generated by the CT operator and the rest 10 are from the CS operator.
- Thirteen mutants made faulty changes in the *refund* scenario, which could not be detected in the original testing. Four of them belong to the CT and 9 are from the CS.

From 40 different mutants, 3 hidden faults are revealed in the implementation.

Half of the mutants that revealed faults were from the CT operator and half were from the CS. We used Formula 2 to measure fault detection capability of each mutation operator. The result of the calculation is shown in Table 3 as well, showing that CT gets the best score in revealing faults.

Table 3 illustrates information on how the mutation operators are able to show some faults in the case study. The result in the first column shows how many alive mutants have remained after the mutation testing without having further information about the equivalent mutants.

Here, it seems that CS is a better operator than the others. However, after removing the equivalent mutants and calculating the fault detection ability of each operator, CT provides a better percentage. The second column in the table shows the result. All of the alive mutants generated by CNI were found equivalent and hence CNI is ranked 0 in fault detection.

6 DISCUSSION

Some improvements can reduce the test execution time while increasing the probability of finding faults. For instance, both CS and CT were able to reveal all three faults and since both of them have generated large numbers of mutants, selecting one of them can considerably reduce test generation and execution times. The result of mutation testing indicates that an intelligent choice of the mutation operators can attain high mutation efficiency scores while reducing the time of testing.

Another improvement could be done in the process of fault detection. Redundant work is done on detecting the same faults. This extra effort can be reduced by categorizing the alive mutants in such a way that all mutations of a certain location or action in the model will be in a category. As soon as any of the mutants in a category detects a fault, then the rest of the mutants on that group can be eliminated from the fault detection analysis. The idea behind this is that the locations and actions in a model represent actual states of the system under test and if there is a state which contains a fault, then any mutant from that state may be able to reveal that fault. However, more experiments are needed to show the correctness of this mutation reduction technique.

More extensive studies are needed in order to investigate how the specification mutation can be applied in larger case studies preferably industrial-sized web services. Besides, more experiment on larger scales would be helpful in finding whether there is any correlation between certain mutation operators and the real faults in design and implementation of web services.

It should be noted that the presented approach for robustness testing does not specifically designed for composite of web services, but any individual service can also be tested. We selected the WSC since it includes more communications and timing behaviors.

The main downside of model-based mutation testing comes from MBT: the process of design models from the specification, verifying them and writing the test adapter (to translate model-level test inputs into acceptable test script for the IUT and vice versa) is time consuming. We have reduced the design and verification time by reusing the same models from the previous research. The mutation testing does not add any overhead into MBT. The mutation generator tool automatically generates correct and valid mutations and thus, it reduces the mutant generation time.

7 THREATS TO VALIDITY

There are three main threats related to our study. One is related to the mutation operators. Despite the fact that we have followed the systematically and formally defined mutation operators and implemented them in our study, there might be some more effective mutation operators or combinations of operators that we have missed. We argue that the current number of mutation operators provides a large number of mutants which can provide faulty test inputs which are close to the accepted inputs.

Another threat is that although the test model is designed and validated very carefully and the IUT is well-tested, there might be some mistakes in designing the test model. However, the probability of such mistakes is low since we have applied conformance testing and fixed the bugs prior to mutation analysis.

Judgmental errors may have happened during the classification between equivalent and non-equivalent mutants. For comparing the mutation models and the original one, we checked the alive mutants and applied formal verification rules.

8 RELATED WORK

A comprehensive analysis is done on all available mutation testing method presenting the current state of the art in this field and the open challenges (Jia and Harman, 2011) .

Lee and Offutt (Lee and Offutt, 2001) introduced an Interaction specification Model which formalize the interactions among Web components. They defined a set of mutation operators for XML data model in order to mutate the inputs of the Web components. Li and Miller (Li et al., 2009) presented mutation testing methods using XML schema to create invalid inputs. Mutation testing is extended to XML-based specification languages for Web services. Lee et al. presented an ontology based mutation operators on OWL-S, which is an XML-based language for specifying semantics on Web services(Lee et al., 2008). They mutate semantics of the specifications of their case study such as data mutation, condition mutation, etc. Wang and Huang presented a mutation testing approach based on OWL-S to validate the requirements of Web services (Wang and Huang, 2008). Also, Dominguez et al. presented a mutation generator tool for WS-BPEL.

We discuss those that are similar to our approach. Work has been done on using model checking techniques for validation and verification of WSC. There are two studies that review the literature on testing Web services (Rusli et al., 2011), (Bozkurt and other, 2010). Starting from specification languages for modeling Web services, researchers perform simulation, verification and test generation using model checking tools. Most of the works use model checking for specification and verification and only a group of them use the models for the test generation as well. We discuss those that are similar to our approach. Using TA models for mutation testing has been mostly studied on a real-time and embedded system. In (Aboutrab et al., 2012) and (Aichering et al., 2013) mutation operators for TA are presented. Aboutrab et al. proposed a set of mutation operators for timed automata to empirically compare priority-based testing with other testing approaches (Aboutrab et al., 2012). However, in their approach, the generation of mutations is done manually.

Aichernig at al. presented model-based mutation testing real-time system using UPPAAL (Aichering et al., 2013). The mutation operators that are defined in their work more detailed and some of them are implemented as a mutation on bounded model-checking and incremental SMT solving. They showed that using mutations for timed automata has potential on debugging and revealing the unexpected behavior of the IUT.

We applied/modified the mutation operators of TA presented by these studies for testing the robustness of WSC. Similar to (Aichering et al., 2013), we applied mutations on non-deterministic models, however, in their work, they use only the UTA model of the IUT and do not consider the environment. In our approach, however, each mutant is a closed model communicating with its environment as well as other systems. We check deadlockfreeness and reachability in order to reduce the number of invalid mutants. Also, we use different verification and test generation processes.

There are some works that target UTA as the specification language for Web services. In most of the works, the authors transformed the specification that is defined in their selected languages into UTA and then they investigated their research. For instance, in (Rauf et al., 2014), the specification of a WSC is defined initially in the form of UML and then transformed into UTA for an online testing purpose. In (Cambronero et al., 2011), Cambronero et al. verify web services by the UPPAAL tool for validation and verification of their described system that is transformed from WS-CDL into a network of TA. In (Dıaz et al., 2007), Diaz et al. also provide a translation from WS-BPEL to UTA. Time properties are specified in WS-BPEL and translated to UTA. However, requirements are not traced explicitly, while verification and testing are not discussed.

9 CONCLUSIONS AND FUTURE WORK

Due to the increasing popularity of combining different Web services as a new Web service, robustness of such systems gained attention in the recent years. We have presented a model-based mutation testing approach for Web service compositions using the UP-

PAAL TA.

Our method starts with the design model that is specified as UPPAAL TA, verified UPPAAL TRON applied for conformance testing thereafter.

We used our mutation generator tool which implements a set of mutation operators applied on the test model for the purpose of online testing. In order to reduce the number of trivial invalid models and also increase the efficiency of testing, we defined a set of verification rules for each mutant. We verified whether the generated mutants are deadlockfree and if the mutation part of each mutant is reachable. If both of these criteria are satisfied, then we select the mutant as a valid mutant. We used UPPAAL TRON for executing all of the mutation models against the system under test.

We presented our approach with an experimental study on Hotel Booking System as a case study. The Web services are implemented in REST architectural style and with timing constraints. Our hotel booking case study has been designed and validated with UPPAAL test model and also the testing evaluated with a series of mutation in the source code of the case study.

The results showed that from a total 1346 generated mutants, 393 were found to be valid mutants that were usable for testing. After running the test, 40 of the mutants were found to identify 3 hidden faults in the implementation of the IUT. The experiment indicates that our approach of specification mutation testing was effective to reveal inconsistency between the specification and the implementation under test.

The primary results of this study showed that our method in robustness testing a valid approach in improving the quality of web service implementations, by detecting faults not detected by the traditional MBT process.

Our experiments also showed that some of the existing mutation operators for time automata are more efficient than the others at finding faults.

There are some research directions that certainly improve the current approach. The next work will be running more experiments, on different case studies in different application domains. More experiments help us to gain more information about mutation operators and correlations between the type of the case study and the common faults.

Another improvement will be to investigate how to detect equivalent mutants. Automation of this process of the approach reduces the errors and increases the scalability of the target applications.

Moreover, we plan to apply mutation selection and mutation reduction techniques to increase the probability of fault detection. Defining new mutation operators, categorizing the mutants, etc., will be investigated in our future work.

ACKNOWLEDGMENTS

We would like to thank prof. Andreas Zeller from University of Saarland, Germany for his valuable comments and anonymous reviewers for their useful suggestions.

REFERENCES

Aboutrab, M. et al. (2012). Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669.

Aichering, B. et al. (2013). Time for MutantsModel-Based Mutation Testing with Timed Automata. In *Tests and Proofs*, pages 20–38. Springer.

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.

Beharmann, G. et al. (2004). A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer.

Belli, F. et al. (2016). Model-based mutation testingapproach and case studies. *Science of Computer Programming*, 120:25 – 48.

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

Budd, T. A. and Gopal, A. S. (1985). Program testing by specification mutation. *Computer Languages*, 10(1):63 – 73.

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.

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

Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In *Software EngineeringESEC/FSE99*, pages 146–162. Springer.

Hessel, A. et al. (2008). Testing Real-time Systems Using UPPAAL. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing*, pages 77–117. Springer-Verlag, Berlin, Heidelberg.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678.

Larsen, K. et al. (2005a). Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306. ACM.

Larsen, K., Mikucionis, M., and Nielsen, B. (2005b). Online testing of real-time systems using uppaal. In Grabowski, J. and Nielsen, B., editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg.

Lee, S. et al. (2008). Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 149–156.

Lee, S. C. and Offutt, J. (2001). Generating test cases for XML-based Web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209.

Li, J.-h., Dai, G.-x., and Li, H.-h. (2009). Mutation analysis for testing finite state machines. In *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*, volume 1, pages 620–624. IEEE.

Rauf, I. et al. (2014). An Integrated Approach for Designing and Validating REST Web Service Compositions. In Monfort, V. and Krempels, K.-H., editors, *10th International Conference on Web Information Systems and Technologies*, volume 1, page 104115. SCITEPRESS Digital Library.

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.

Sheng, Q. et al. (2014). Web services composition: A decades overview . *Information Sciences*, 280:218 – 238.

Wang, R. and Huang, N. (2008). Requirement Model-Based Mutation Testing for Web Service. In *Next Generation Web Services Practices, 2008. NWESP '08. 4th International Conference on*, pages 71–76.