Selective Test Generation Approach for Testing Dynamic Behavioral Adaptations

Mariam Lahami[1(✉)], Moez Krichen[1,2], Hajer Barhoumi[1], and Mohamed Jmaiel[1,3]

[1] ReDCAD Research Laboratory, National School of Engineering of Sfax, University of Sfax, B.P. 1173, Sfax, Tunisia
{mariam.lahami,hajer.barhoumi}@redcad.org
[2] Faculty of Computer Science and Information Technology, Al-Baha University, Al Bahah, Kingdom of Saudi Arabia
moez.krichen@redcad.org
[3] Research Center for Computer Science, Multimedia and Digital Data Processing of Sfax, B.P. 275, 3021 Sfax, Sakiet Ezzit, Tunisia
mohamed.jmaiel@enis.rnu.tn

Abstract. This paper presents a model-based black-box testing approach for dynamically adaptive systems. Behavioral models of such systems are formally specified using timed automata. With the aim of obtaining the new test suite and avoiding its regeneration in a cost effective manner, we propose a selective test generation approach. The latter comprises essentially three modules: (1) a model differencing module that detects similarities and differences between the initial and the evolved behavioral models, (2) an old test classification module that identifies reusable and retestable tests from the old test suite, and finally (3) a test generation module that generates new tests covering new behaviors and adapts old tests that failed during animation. To show its efficiency, the proposed technique is illustrated through the Toast application and compared to the classical Regenerate All and Retest All approaches.

1 Introduction

Due to increasingly rapid changes in the context, goals, and user requirements of recent critical software systems, there is a demand to perform automatically validation tasks at runtime to ensure firstly that existing functionalities have not been affected by dynamic changes. Secondly, it is essential to verify that new requirements are fulfilled by the new version of the system.

One of the emerging and promising techniques for testing a software is the Model Based Testing (MBT) approach. Instead of writing hundred of test cases manually, the test designer defines an abstract model of the System Under Test (SUT) and then MBT tool generates automatically a set of test cases from the model. MBT methods have recently gained increased attention because maintaining and adapting test cases can be facilitated and also automated [17].

Another technique widely used for testing an evolving software system is the Regression Testing. Most research and tools perform usually white box regression

K. El-Fakih et al. (Eds.): ICTSS 2015, LNCS 9447, pp. 224–239, 2015.
DOI: 10.1007/978-3-319-25945-1_14

testing at design time [7,16]. Up to our best knowledge, there is a trend to merge these two techniques to build approaches and tools for black/gray box regression testing, known as specification-based regression approaches [4,15]. The majority of those potential solutions use UML diagrams to model SUT behaviors and deal mainly with selecting test cases from existing test suites specified before modifications. This is not sufficient because new tests have to be generated and thus stored test suites need to be updated [5,6].

Following this direction, we provide a selective test generation approach called TestGenApp, that reduces the cost of adapting and maintaining test suites covering either modified or newly added behaviors at runtime. Our proposal ensures that tests cases continue to be consistent and fault revealing even if the SUT evolves dynamically. It avoids the regeneration of the full test suite by covering only affected parts of the behavioral model. The latter was specified using UPPAAL Timed Automata formalism. The well-establish tool UPPAAL Cover is customized to generate effectively new abstract tests and to adapt failed ones. To do so, we reuse its expressive approach to specify coverage criteria, called Observer Automata. To show its efficiency, the proposed technique is illustrated through the Toast dynamic application [13] and compared to the classical Regenerate All and Retest All approaches.

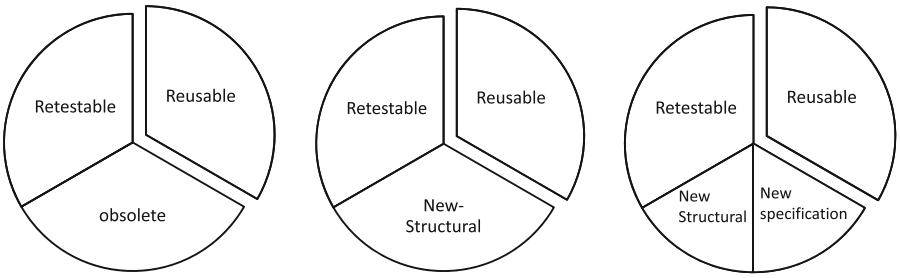
The rest of this paper is organized as follows. Section 2 provides background material for understanding the research problem. Subsequently, the selective test generation approach is outlined in Sect. 3. Afterward, its application to the Toast application is highlighted in Sect. 4. Section 5 draws comparison with related work in the context of selective regression testing. Finally, in Sect. 6, we conclude with a summary of paper contributions, and we identify potential areas of future research.

2 Background

This section provides background material on testing evolvable systems, formal modeling as timed automata, coverage criteria and automata observers.

2.1 Testing Evolvable Systems

One of the well-known technique used to check the correctness of software after modification is *Regression Testing*. As quoted from [8], it "attempts to validate modified software and ensure that no new errors are introduced into previously tested code". This technique guarantees that the modified software is still working according to its specification and it is maintaining its level of reliability. It is commonly applied during development phase and not at runtime. Leung et al. [12] present two types of regression testing. In the progressive regression testing, the SUT specification can be modified by reflecting some enhancements or some new requirements added in the SUT. In the corrective regression testing, only the SUT code is modified by altering some instructions in the program whereas



(a) Old test suite (b) New test suite in corrective regression testing (c) New test suite in progressive regression testing

Fig. 1. Test classification

the specification does not change. For the above defined types, Leung et al. illustrate the same test case classification of the old test suite into three categories (see Fig. 1a):

- Reusable tests: valid tests that cover the unmodified parts of the SUT.
- Retestable tests: still valid tests that cover modified parts of the SUT.
- Obsolete tests: invalid tests that cover deleted parts of the SUT.

After modifications, new tests can be classified into two classes (see Fig. 1b and c):

- New specification tests: include new test cases generated from the modified parts of the specification.
- New structural tests: include structural-based test cases that test altered program instructions.

Contrary to regression testing, *runtime testing* is emerging as a novel solution for the validation of dynamically evolvable systems. It is defined in Brenner et al. [3] as an online testing method that is carried out on the final execution environment of a system when the system or a part of it is operational. It can be performed either at deployment-time or at service-time. The deployment-time testing serves to validate and verify the assembled system in its runtime environment while it is deployed for the first time. For systems whose architecture remains constant after initial installation, there is obviously no need to retest the system when it has been placed in- service. On the contrary, if the execution environment or the system behavior or its architecture has been changed, service-time testing will be a necessity to verify and validate the new system in the new situation.

In this work, we merge findings on both specification-based regression testing and runtime testing with the aim of conceiving a runtime model-based testing approach. The latter is applied in a cost effective manner whenever the SUT behavior evolves dynamically.

2.2 Formal Modeling Using UPPAAL

UPPAAL is a well-established model-checking tool charged with verifying a given model w.r.t. a formally expressed requirement specification. It uses a popular and widespread formalism for specifying critical and real-time systems, called Timed Automata (TA). Indeed, a system is modeled as a network of timed automata, called processes. A timed automaton is an extended finite-state machine equipped with a set of clock-variables that track the progress of time and that can guard when transitions are allowed. In this work, we adopt the particular UPPAAL style of timed automata. As they have proven their expressiveness and convenience, behavioral models of both initial system and the evolved system are expressed using the formal notation bellow:

Definition of Timed Automaton

Let $\mathcal{C}$ be a set of valued variables called clocks, and $\mathcal{A} = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ with $\mathcal{I}$ a set of input actions, $\mathcal{O}$ a set of output actions (denoted a? and a!), and the non-synchronizing action (denoted τ). Let $\mathcal{G}(\mathcal{C})$ denote the set of guards on clocks being conjunctions of constraints of the form $c \bowtie n$, and let $\mathcal{U}(\mathcal{C})$ denote the set of updates of clocks corresponding to sequences of statements of the form $c := n$, where $c \in \mathcal{C}$, $n \in \mathbb{N}$, and $\bowtie \in \{\leqslant, \leq, =, \geq, \geqslant\}$. A timed automaton over $(\mathcal{A}, \mathcal{C})$ is a quadruple (L, l_0, I, E), where:

- L is a set of locations, $l_0 \in L$ is an initial location.
- $I : L \longmapsto \mathcal{G}(\mathcal{C})$ a function that assigns to each location an invariant.
- E is a set of edges such that $E \subseteq L \times \mathcal{G}(\mathcal{C}) \times \mathcal{A}_\tau \times \mathcal{U}(\mathcal{C}) \times L$

We shall write $l \xrightarrow{g,\alpha,u} l'$ when $\langle l, g, \alpha, u, l' \rangle \in E$.

Due to space limitations, the semantics of TA as well as the semantics of a network of TA have not been presented in this paper. Moreover, it is worthy to note that UPPAAL modeling language extends timed automata with additional features such as integer variables, urgent locations, and committed locations, etc. For more details, readers can refer to [2].

It is worthy to note that several restrictions have to be fulfilled in this work by each timed automaton in the SUT behavioral specification. They have to be deterministic input enabled output urgent timed automata (DIEOU-TA). For short these restrictions means that: (i) Two transitions with the same label lead to the same state, (ii) no delay can be done when an input is enabled, (iii) when an output is enabled, no input, output, or delay is permitted [10]. Moreover, we assume that the test specification is given as a closed network of TA that can be partitioned into one network of TA modeling the SUT behavior, and one modeling its environment behavior (ENV). Note here that the tester replaces the environment and controls the SUT via a distinguished set of observable input and output actions.

2.3 Automata Observer for Specifying Coverage Criteria

A coverage criterion is a specification of items such as locations and edges to be traversed or visited by the timed automaton. An example of a coverage criterion

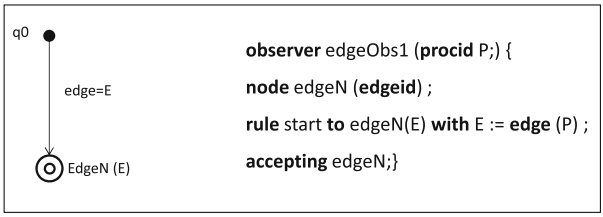


Fig. 2. Edge coverage observer presented in both textual and graphical notations.

is *edge coverage* which means that a test case should traverse all edges of a given timed automaton. An item to be traversed or visited is called a coverage item and can be modeled by an observer. The latter observes the execution of a timed automaton and accepts when the coverage item is covered by the trace.

As depicted in Fig. 2, an observer automaton can be presented either in graphical or textual notations. It is made of locations and edges. Locations are labeled with a name and optional variables, and edges are labeled with predicates. Two special types of locations are identified: the initial location denoted with a black filled circle, and the accepting location denoted with a double circle. In addition, an observer can only have one initial location but can reach several accepting locations. Formally, an observer automaton is a quadruple $(\mathcal{Q}, q_0, \mathcal{Q}_f, \mathcal{B})$ where:

- $\mathcal{Q}$ is a finite set of observer locations.
- q_0 is the initial observer location.
- $\mathcal{Q}_f \subset \mathcal{Q}$ is a set of accepting observer locations.
- $\mathcal{B}$ is a set of edges such $q \xrightarrow{b} q'$ where b is a predicate based on attributes of timed automata as variables, edges, locations, etc.

We believe that the use of the observer language simplifies the expression of coverage criteria as it can be used to specify and combine the most popular ones such as "all edges", "all locations", "all-definition use-pairs"[9], etc. In our context, this formalism is used to express our own coverage criteria: covering new items and adapted ones.

3 Selective Test Generation Approach

The proposed selective test generation approach, called TestGenApp, is built upon three modules as outlined in Fig. 3. Each one is introduced in the following subsections.

3.1 Model Differencing Module

Model differencing technique is used to detect similarities and differences between the original model $\mathcal{M}$ and the evolved $\mathcal{M}'$ taken as inputs. It generates as output a colored $\mathcal{M}_{diff}$ that highlights changed and unchanged elements. Added locations and transitions are marked in Red, modified locations

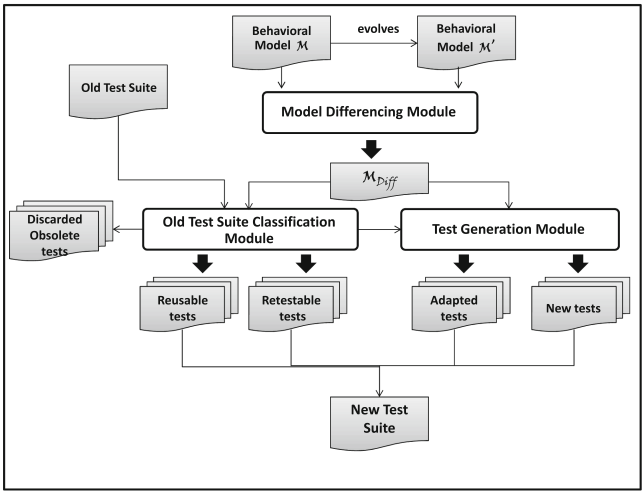


Fig. 3. TestGenApp: selective test generation approach.

and transitions are marked in Yellow, finally unchanged locations and transitions are marked in Green. The colored model includes a new variable called *col* initially equal to 1. This variable is updated in response to the performed modification. If target and source locations or transition labels (guard, update, synchronization) are newly added and then colored in Red, the *col* variable associated with this transition is multiplied by zero (i.e., $col := col * 0$). Similarly, if transition labels, source or target location are colored in Yellow, the *col* variable is multiplied by two (i.e., $col := col * 2$). Otherwise, transition labels, target and source location are unchanged and so the *col* variable is multiplied by one (i.e., $col := col * 1$).

Several kinds of changes are taken into account in this work. First of all, we support clock modification, location addition/removal and transition addition/removal. Moreover, state modification is considered by changing state invariant or changing the incoming and the outgoing transitions. Finally, transition modification is also handled by changing guard, synchronization and update fields. It is worthy to note that we support elementary as well as complex modifications.

3.2 Old Test Suite Classification Module

Based on the test classification proposed by Leung et al. [12], the old test suite issued from the original model $\mathcal{M}$ is divided into reusable, retestable and obsolete tests. Tests that traverse inchanged parts (locations and transitions marked as green) on the $\mathcal{M}_{diff}$ model are classified as *reusable tests*. Tests that cover at least a removed state or removed transition are classified as *obsolete tests* and will be automatically discarded from the new test suite.

We extend Leung et al. work by animating the remaining set of tests. First, we obtain valid tests called *retestabe tests* that cover the same coverage items in the new model and traverse the same paths with possibly updated clock values. Second, some failed tests are also detected. Generally, they cannot be animated on the new model because they may traverse altered paths. Thus, they need to be adapted by regenerating them. The Algorithm 1 is used in this test classification step. To do so, a new variable called $ColY$ is added to the $\mathcal{M}_{diff}$ model and it is used to mark the new reachable transition with the failed test. It will be used later to generate the adapted test.

Algorithm 1. Test Classification Algorithm

Input: Old test traces $\mathcal{TR}$
A network of Timed Automata TA_{diff} highlighting unchanged and changed elements.
Output: Reusable tests T_{Ru}.
Retestable tests T_{Rt}.

BEGIN
for each trace in $\mathcal{TR}$ **do**
  $coveredItemsList$= get_CoveredItems(trace)
  if (VerifColorGreen($coveredItemsList$)= true) **then**
    $trace \in T_{Ru}$
  else
    if (isAnimated($trace$,TA_{diff})=true) **then**
      $trace \in T_{Rt}$
    else
      $trace$ needs to be adapted
      the new reachable $transition.setUpdate(ColY := 1)$
    end if
  end if
end for
END

3.3 Test Generation Module

The used test generation technique is based on model checking. The main idea is to formulate the test generation problem as a reachability problem that can be solved with the model checker tool UPPAAL [2]. However, instead of using model annotations and reachability properties to express coverage criteria, the observer language is used.

In this direction, we reuse the finding of Hessel et al. [9] by exploiting its extension of UPPAAL namely UPPAAL CO$\surd$ER[1]. This tool takes as inputs a model, an observer and a configuration file. The model is specified as a network of

[1] http://user.it.uu.se/~hessel/CoVer/index.php.

```
observer obs (procid P;varid col;varid colY;) {
node edgeN (edgeid, varid) ;
node edgeA(edgeid,varid) ;
rule start to edgeA(K,colY) with K:=edge(P),eval(colY)==1;
rule start to edgeN(E, col) with E:=edge(P),eval(col)==0 ;
accepting edgeN,edgeA;}
```

Fig. 4. Covering new and adapted tests with automata observers

UPPAAL timed automata (.xml) that comprises a SUT part and an environment part. The observer (.obs) expresses the coverage criterion that guides the model exploration during test case generation. The configuration file (.cfg) describes mainly the interactions between the system part and the environment part in terms of input/output signals. It may also specify the variables that should be passed as parameters in these signals. As output, it produces a test suite containing a set of timed traces (.xml).

Our test generation module is built upon these well-elaborated tools. The key idea here is to use UPPAAL CO$\surd$ER and its generic and formal specification language for coverage criteria to generate new tests and adapted ones.

As depicted in Fig. 4, we express our own observer that covers only new behaviors and adapts modified ones. Observer parameters are denoted with capital letters. The parameters can refer in the model to variables, edges, locations, variable valuations, etc. In this example, we use the E parameter for an edge. The observer collects all different edges from the parameter process P. $edge(E)$ is a predicate which evaluates to true if the observer monitors edge E of the timed automaton. The evaluation of the *col* variable to zero indicates that the current edge is marked as new edge. The rule *rule start to edgeN(E, col) with E:=edge(P),eval(col)==0;* formalizes this new edge coverage criterion. A test sequence satisfies this coverage criterion if when executed on the model it traverses at least on edge where the *col* variable is updated to zero. Similarly, if the variable *colY* is evaluated to one as outlined in the rule *rule start to edgeA(K,colY) with K:=edge(P),eval(colY)==1;* this means that the current edge is marked as modified edge. Thus, the monitored edge can be taken as accepted covered item by the new generated test sequence.

4 Application to the Toast Case Study

In this section, we describe the application of the presented technique on a case study in the telematics and fleet management domain called Toast [13]. For this aim, our prototype has been implemented in Java language. We have used the UPPAAL model checker, version 4.1.18 for modeling the SUT specification with timed automata and for checking that the developed models are deadlock free. Regarding the test generation, UPPAAL CO$\surd$ER version 1.4 is adopted.

4.1 Toast Description

Toast is an OSGi[2] application used to demonstrate a wide range of EclipseRT technologies. It provides means to manage and to interact with devices installable in a vehicle. For the sake of simplicity, the studied scenario covers the case of emergency notification. The vehicle comprises two devices: an Airbag and a GPS. If the airbag deploys, an Emergency Monitor is notified. In this case, the monitor queries the GPS for the vehicle heading, latitude, longitude and speed (see Fig. 5). The three components in the Toast initial configuration can be modeled by the three UPPAAL timed automata as shown in Fig. 6. In the beginning, timing constraints are omitted and we focus mainly on synchronization of inputs and outputs signals between components.

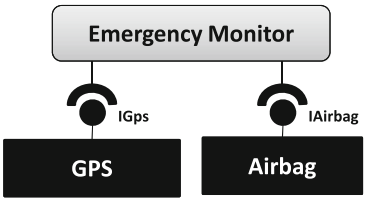


Fig. 5. Initial toast architecture.

When the airbag is deployed (i.e., this internal action is modeled with an empty label from location _0 to location _1), the Airbag component sends a message (via the action em) to the Emergency Monitor and waits for an acknowledge (an emAck action). In case of a negative replay (modeled by action emNoAck), the Airbag sends the emergency message again. Afterwards, the Emergency Monitor interacts with the GPS to get vehicle's latitude, longitude, heading and speed. To allow the system to come back to its initial state, the Airbag is undeployed (this internal action is modeled with an empty label from location _5 to location _6). Similarly, it sends a corresponding message to the Emergency Monitor. Notice that the Emergency Monitor depends on both GPS and Airbag but GPS and Airbag are independent of one another. Also, it can only communicate with the GPS component.

4.2 Dynamic Toast Evolution

Starting from the basic configuration introduced in the subsection below, new components and features can be installed at run-time during system operation. To prove the feasibility of our approach and its efficiency in reducing the test generation cost, five cases of behavioral adaptations are studied as illustrated in Table 1.

[2] Open Services Gateway initiative.

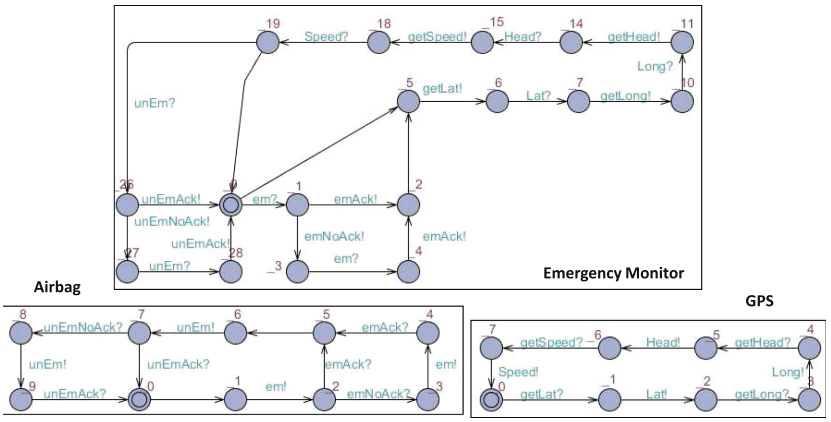


Fig. 6. Behavioral models of the initial configuration.

Table 1. Several studied toast evolutions.

Case study evolutions	Kind of the evolution	Templates	States	Transitions
Case 0:Initial toast configuration	———	GPS	8	8
		Airbag	10	12
		Emergency	17	21
Case 1:Case 0 with updated GPS behavior	Complex (updating transition labels)	GPS	8	8
		Airbag	10	12
		Emergency	17	21
Case 2: Case 0 with errors in data transmission	Complex (adding states and transitions)	GPS	16	20
		Airbag	10	12
		Emergency	25	33
Case 3: Case 0 with timing constraints	Complex (adding transitions)	GPS	8	12
		Airbag	10	12
		Emergency	17	25
Case 4: Case 2 with Back End Server	Complex (adding templates)	GPS	16	20
		Airbag	10	12
		Emergency	29	38
		Back End	4	5
Case 5: Case 4 with timing constraints	Complex (adding and updating transitions)	GPS	16	28
		Airbag	10	12
		Emergency	29	46
		Back End	4	5

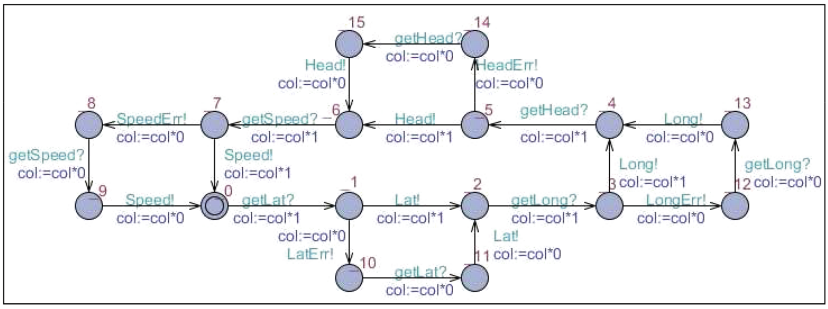


Fig. 7. The GPS_{diff} model in case 2.

Case 1: The initial Toast architecture is maintained whereas GPS behavior is changed with sending bearing[3] measure instead of heading measure to the Emergency Monitor. In the new version of the GPS model, getHead? and Head! transition labels are replaced with getBearing? and Bearing! transition labels. Such modification is propagated to the Emergency Monitor model, as well. Notice that for each modified transition the *col* variable is updated with $col := col * 2$.

Case 2: The initial GPS behavior is improved with taking into account errors during data emission to the Emergency Monitor. To handle such problem, the new version of the GPS component sends vehicle information again in case of error occurrence. Such modification introduces new states and transitions both in GPS and Emergency Monitor models.

Due to space constraints, Fig. 7 depicts only the generated GPS model by the model differencing module. It points out that for each new transition the *col* variable is updated with $col := col * 0$ and for each unchanged transition this variable is updated with $col := col * 1$.

Case 3: The initial GPS behavioral model is enhanced with timing constraints. In fact, we assume that the new GPS version sends each requested vehicle information in lapse of time that does not exceed *Tprocessing* which is equal to 4 time units. In this case, different transitions are updated and others new transitions are added to both Emergency Monitor and GPS automata. Figure 8 illustrates modifications made on the GPS component. As mentioned before, the *col* variable is updated in response to modifications made on the SUT model.

Case 4: The Toast architecture is evolved while including the new Back End component [1]. The latter is responsible with collecting information from the

[3] Bearing is the direction from the vehicle location to the destination point given in degree from the north whereas the heading is a direction toward which a vehicle is (or should be) moving.

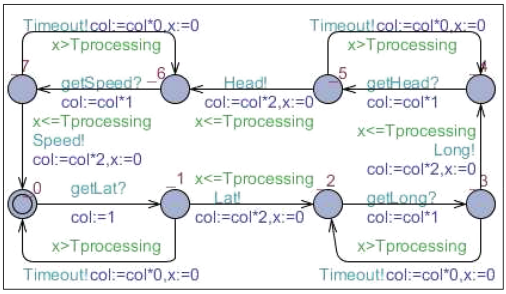
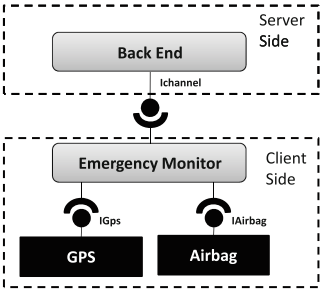
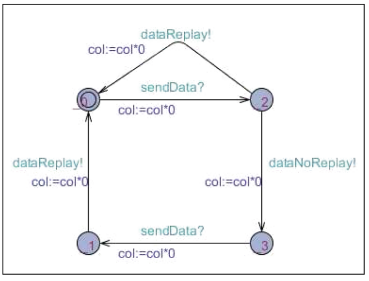


Fig. 8. The GPS_{diff} model in case 3.

Emergency Monitor. It is a server running entirely on a separate computer and it listens for the client to report emergencies. The new architecture is outlined in Fig. 9a. The overall toast behavior is changed and a new template for the Back End component is introduced as shown in Fig. 9b.



(a) The evolved toast architecture.



(b) Back End behavioral model.

Fig. 9. The addition of Back End template in the toast specification (Case 4).

The main idea is to establish the correctness of the evolved Toast application in a cost effective manner. This can be done by avoiding the regeneration of the full test suite and applying our TestGenApp technique as discussed in the next subsection.

4.3 Selective Test Generation with TestGenApp

As mentioned before, UPPAAL CO$\sqrt{}$ER is used to generate diagnostic traces. From each diagnostic trace, a test sequence that is an alternating sequence of concrete delay actions and observable actions, may be obtained simply by projecting the trace to the environment model, while removing invisible transitions, and summing adjacent delay actions [9]. The same tool is used to generate test

Table 2. Comparison between Regenerate All, Retest All and TestGenApp strategies.

Case study evolutions	Regenerate All	Retest All		TestGenApp			
		Old	New	Reusable	New	Retestable	Adapted
From Case 0 to Case 1	2 traces	2	0	1	0	0	1
From Case 0 to Case 2	6 traces	2	4	2	4	0	0
From Case 0 to Case 3	6 traces	2	4	0	4	2	0
From Case 2 to Case 4	6 traces	6	2	4	2	0	0
From Case 3 to Case 5	11 traces	6	5	6	5	0	0
From Case 4 to Case 5	11 traces	6	8	0	8	3	0

cases when dynamic evolutions take place. An example of the obtained test sequence for case 0 is highlighted in the following:

em! emAck? getLat? Lat! getLong? Long! getHead? Head! getSpeed? Speed! unEm! unNoEmAck? unEm!

Compared to the Regenerate All technique, our proposal reduces the number of generated traces as shown in Table 2. For instance, instead of generating the full test suite (6 traces here) when the Toast evolves from Case 2 to Case 4, only 2 traces are newly generated and 4 traces are still valid and cover unimpacted parts of the model by the evolution. Similarly, the evolution from Case 0 to Case 2 requires the generation of 4 tests and the selection of 2 retestable tests from the old test suite.

Concerning the Retest All strategy, it consists in re-executing all tests from the old test suite and generates tests for uncovered behaviors. The main limitation of this approach is that it possibly re-executes obsolete tests. As outlined in Table 2, when the Toast evolves from Case 0 to Case 1, the two old traces are re-executed by this approach whereas one of them is failed and requires to be adapted.

To conclude, our proposal reduces the cost of test generation and gives an important information about the obtained tests and which parts of the SUT they cover.

5 Related Work

There has been a spate of interest in how to reestablish confidence in modified software systems. As one of the key method to improve software quality and dependability, selective regression testing has been widely used. Three kinds of approaches are identified in this research area: code-based regression testing [7,16], model based regression testing [4–6,15] and software architecture based regression testing [14]. In the first class, Granja et al. [7] discuss two techniques of code-based regression testing. The first one deals with identifying program modifications and selecting attributes required for regression testing. Based on dataflow analysis, the second technique uses the obtained required elements to select retestable tests. According to the metrics defined in Rothermel et al. [16],

authors show that their proposal has a good precision, a high generality but requires further work to attain inclusiveness and efficiency. Contrarily to these approaches, our work deals with model based regression testing. Such method has the main advantage to handle test selection and test generation on a higher abstraction level.

In the second class, we introduce the work of Brian et al. [4] that present a UML-based regression test selection strategy. By supporting changes in actions of sequence diagrams and in variables, operations, relationships and classes, the proposed change analysis approach automatically classifies tests on obsolete, reusable and retestable tests. In the same direction, the approach cited in [6] deals with minimizing the impact of test case evolution by avoiding the regeneration of full test suites and focusing only on generating the new or the updated ones. A point in favor this work is the improvement of test classification based on code analysis proposed by [12]. In fact, authors enable more precise test status definition based on model dependence analysis. Notably, retestable tests are animated on the model and can be classified as updated, adapted, unimpacted, re-executed, outdated or failed tests. Conversely to our approach, these approaches are based on UML as a semi formal description language to model system behaviors (namely class, object and statechart diagrams). Analyzing such various diagrams to identify modification impact can be seen as a tedious task. Such problem has been resolved by Pilskalns et al. [15]. They present a regression testing approach based on an integrated model called Object Method Directed Acyclic Graph (OMDAG) built from class diagrams, sequence diagrams and OCL expressions. They consider when a path in the OMDAG changes it affects one or more test cases associated to the path and they classify changes as NEWSET, MODSET and DELSET. Based on Extended Finite State Machine (EFSM) models, [5] proposes a safe regression technique relying on dependence analysis, as well. It supports three types of elementary modifications of the machine (addition, deletion, modification of a transition). Similarly, our approach takes into account such kinds of transition modifications and improves them by considering also addition, deletion and modification of a state. Another point in favor our approach is the support of either elementary or composite modifications.

In the third class, Muccini et al. [14] propose an effective approach called SARTE: SA-based regression testing. The authors apply regression testing at both code and architecture levels whenever the system architecture or its implementation evolve. First , they test the conformance of a modified implementation to a given software architecture. Second, they test the implementation conformance to the new software architecture. SA specifications are modeled using Labeled Transition Systems (LTS). Similar to our proposal, authors utilize a SAdiff algorithm which compares the behavioral models of both architectures and returns differences between them. This technique was used to identify tests to rerun covering the affected paths. This work differs from our in two major ways. First, our approach deals with model-based black box regression testing. Thus, we assume that code is not available and we consider that any modification done at the code source level

is reflected at/in the behavioral model of the SUT. Second, our focus is mainly on dynamic behavioral adaptations, structural modifications are studied in previous work [11].

6 Conclusion

This paper described a selective test generation approach for critical and dynamically adaptive systems formally modeled as Timed Automta. First of all, a model differencing technique was applied to detect similarities and differences between initial and evolved behavioral models. Moreover, we presented a simple and flexible technique for specifying coverage criteria using observer automata with parameters. This technique was adopted to generate in a cost effective manner new tests and adapt modified ones. The use of UPPAAL as a well-established model checker and its extension for test generation UPPAAL CO$\sqrt{}$ER makes our approach more consistent and sound.

Its application to the Toast architecture shows the efficiency of TestGenApp in reducing the cost of test generation especially when model scale increases. The comparison of our solution with the Regenerate All and the Retest All strategies highlighted such efficiency.

As future work, we investigate efforts in improving our methodology and applying it to more complex and real systems. Also, we aim to conceive several transformations rules for the mapping of the obtained abstract test suites to executable TTCN-3[4] test suites.

References

1. De Angelis, F., Di Berardini, M.R., Muccini, H., Polini, A.: CASSANDRA: an online failure prediction strategy for dynamically evolving systems. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 107–122. Springer, Heidelberg (2014)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Inf. Syst. Front. **9**(2–3), 151–162 (2007)
4. Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Inf. Softw. Technol. **51**(1), 16–30 (2009). http://dx.doi.org/10.1016/j.infsof.2008.09.010
5. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: Proceedings of the 3rd International Workshop on Advances in Model-based Testing, A-MOST 2007, pp. 54–62. ACM, New York (2007). http://doi.acm.org/10.1145/1291535.1291541

[4] Testing and Test Control Notation.

6. Fourneret, E., Bouquet, F., Dadeau, F., Debricon, S.: Selective test generation method for evolving critical systems. In: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011, pp. 125–134. IEEE Computer Society, Washington (2011). http://dx.doi.org/10.1109/ICSTW.2011.95
7. Granja, I., Jino, M.: Techniques for regression testing: selecting test case sets tailored to possibly modified functionalities. In: Proceedings of the Third European Conference on Software Maintenance and Reengineering, CSMR 1999, p. 2. IEEE Computer Society, Washington (1999). http://dl.acm.org/citation.cfm?id=794202.795237
8. Harrold, M.J.: Testing: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ICSE 2000, pp. 61–72. ACM, New York (2000). http://doi.acm.org/10.1145/336512.336532
9. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008)
10. Hessel, A., Larsen, K.G., Nielsen, B., Pettersson, P., Skou, A.: Time-optimal real-time test case generation using Uppaal. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 114–130. Springer, Heidelberg (2004)
11. Lahami, M., Krichen, M., Jmaiel, M.: Runtime testing framework for improving quality in dynamic service-based systems. In: Bianculli, D., Guinea, S., Hallé, S., Polini, A. (eds.) Proceedings of the 2nd International Workshop on Quality Assurance for Service-based Applications, QASBA 2013, in conjunction with ISSTA 2013, July 15, 2013, pp. 17–24. ACM, Lugano (2013). http://doi.acm.org/10.1145/2489300.2489335
12. Leung, H., White, L.: Insights into regression testing [software testing]. In: 1989 Proceedings Conference on Software Maintenance, pp. 60–69 (1989)
13. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley, Upper Saddle River (2010)
14. Muccini, H., Dias, M.S., Richardson, D.J.: Software architecture-based regression testing. J. Syst. Softw. **79**(10), 1379–1396 (2006). http://dx.doi.org/10.1016/j.jss.2006.02.059
15. Pilskalns, O., Uyan, G., Andrews, A.: Regression testing uml designs. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance, ICSM 2006, pp. 254–264. IEEE Computer Society, Washington (2006). http://dx.doi.org/10.1109/ICSM.2006.53
16. Rothermel, G., Harrold, M.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. **22**(8), 529–551 (1996)
17. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)