

A Model-Based Testing Method for Dynamic Aspect-Oriented Software

Maria Laura Pires Souza and Fábio Fagundes Silveira^(✉) 

Federal University of São Paulo – UNIFESP, São José dos Campos, Brazil
malaura.s1@gmail.com, fsilveira@unifesp.br

Abstract. Aspect-oriented programming (AOP) is used to implement crosscutting concerns such as persistence and safety in program units called aspects. To ensure that these concerns behave as specified and do not introduce faults into the application, rigorous software testing practices should be applied. Even though there are statements in the literature that the adoption of AOP takes a software to get better quality, it does not provide correctness by itself. Therefore, the test remains an important activity to ensure aspects are correctly integrated into the main system. Additionally, in a dynamic environment: new aspects may be incompatible with aspects already woven; and aspects to be removed can hold the system to an inconsistent state. Available approaches in the literature do not directly investigate the problem of testing dynamic aspects within the context of a target application. This paper presents a method to apply tests in dynamic aspects that verify the interactions between aspects and classes, as well as among aspects. Aiming to support the method, we also introduce a model to represent the dynamic behavior of aspects and a new strategy to derive testing cases. To evaluate the effectiveness of the test cases generated by the method, mutation operators were applied to the model and simulated with a model checker. Results showed that the approach is capable of detecting faults in dynamic aspects interactions into a target application.

Keywords: Dynamic aspect-oriented · Model-based testing · Mutation testing

1 Introduction

Aspect-oriented programming (AOP) consists in a programming technique that supports the implementation of cross-cutting concerns, such as persistence, security, and logging in units called aspects, where such concerns are implemented. To ensure that these additional concerns behave as specified and do not introduce defects in the application, rigorous and different test levels should be made in the application. Besides, these tests should be extended to the interaction between these aspects and also to the classes of the target application. The division of a system known as separation of concerns refers to the ability to identify,

encapsulate, and manipulate concrete portions of software that are relevant to a particular concern [11].

Some difficulties can be verified when applying tests in aspects compositions [2, 13] in a dynamic environment: loaded aspects may be incompatible with aspects already read and/or running; and removed aspects can lead the system to an inconsistent state.

Available approaches in the literature do not directly investigate the problem of testing dynamic aspects within the context of a target application. So, in order to find solutions to some of the problems mentioned above, this paper presents a testing method for dynamic AO applications. The method, called MESOADI, contemplates the verification of interactions between dynamic aspects, aiming to improve the behavioral tests between classes and aspects, and in composites of dynamic aspects, seeking to reduce the resources involved, such as effort and test time.

This paper is organized as follows: Sect. 2 presents AO definition, especially about dynamic aspects and model-based testing definition. Section 3 briefly summarizes related works on testing dynamic systems. Next, Sect. 4 describes the proposed method with their elements and a case study description. Section 5 highlights the main results and discussion are described. Finally, Sect. 6 concludes the paper and points out future works.

2 Dynamic AOP and Model-Based Testing

Aspect-oriented (AO) development was proposed due to the difficulties encountered in the treatment of code spreading and interlacing during maintenance and software development. Its purpose is to separate levels of concerns during development. The most common examples of crosscutting concerns, which are separated by AO, are those relating to non-functional requirements such as persistence, logging, authentication, security, fault tolerance, among others.

AOP introduces new concepts to the software development process. Among them is the aspect, which is the encapsulation unit of a crosscutting concern [2]. The aspects have structural and/or behavioral attributes that can be applied to various parts of the system. The process of insertion of the aspects, known as weaving, is responsible for injecting the aspects into the modules in which they should act. Weaver is the tool that combines object-oriented (OO) code with the AO code for the operation of the final system.

Regarding dynamic aspect-oriented programming (DAOP), it is possible to apply and remove aspects to a system at runtime, without the need to restart it, which is very useful when designing real world applications. Dynamic aspects are a necessary mechanism, especially if one aspect implements a crosscutting concerns at one point and the requirement for functionality changes dynamically, on the fly. One of the advantages of DAOP is that it removes AOP overhead when aspects are not required [1]. Also, it allows dynamic configuration of aspect behavior and aspect reconfiguration depending on the state of the base system.

Testing plays a critical role in any software development project. However, it is often overlooked as an expensive activity and hampered by the wide variety of programming languages, operating systems and hardware platforms that constantly evolve [12]. This creates serious problems in the software production phase, leading to high costs, poor business reputation and even killing human beings.

Model-Based Testing (MBT) consists of a technique for automatically generating a set of test cases (TC) by using models extracted from the software requirements [3]. Before any software testing activity, one has to validate the model to be sure that it will not cause any errors in the software or vice versa, so this model also needs to be tested. This way, there are rules for modeling software, as it is necessary for everyone involved in software development to have a standard model and know how to interpret them. There are several techniques for specifying systems that are used to add more stringency to the MBT, such as Finite State Machines (FSMs), which was used in this paper, Statecharts, and Petri nets.

The strategies TCs generation aim to verify if an implementation is correct with its specification, through the execution of activities of test and validation in systems described by models [8]. Although the strategies have a common goal, the difference between them is the cost, the size of the set and the effectiveness in finding defects in the system. For this paper, an extension of the Binder's *Round-Trip Path* strategy (RTP) was used. The RTP strategy traverses the FSM graph through an algorithm and generates a tree called *State Transition Tree* (STT) corresponding to that path, where the initial state is the root node.

3 Related Works

Several researches on AO systems and dynamically adaptive systems show up important works in these areas. Zhang and Cheng [15] separated the adaptive behavior and specifications of non-adaptive behaviors into dynamic programs. For this, a process was introduced for the construction of adaptive models, automatically generating adaptive programs of the models, besides checking and validating the models. For the authors, the main tasks for the adaptation of a point are to identify the states that are suitable for adaptation and to define adaptive transitions from these states. Zhao et al. [16] propose a definition for non-adaptive program and adaptive program through finite state machines. For them, adaptation is an action that changes the behavior of a state in one FSM to a state into another FSM.

Fuentes and Sánchez [7] published a paper with the objective of presenting an extension of the Unified Modeling Language (UML) for the construction of aspect-oriented models. Silveira et al. [13] proposed the METEORA, a state-based testing method for AO programs, which provides classaspect and, more specifically, aspectaspect faults detecting capabilities. Ferrari et al. [6] describe an approach based on mutation testing for AspectJ programs. Lindström et al. [10] propose the use of AOM (Aspect-Oriented Models) mutation to test crosscutting concerns.




4 The MESOADI Method

The MESOADI method aims to apply state-based tests in dynamic aspects, through a model of behavioral representation of interactions between dynamic aspects. The model, called MEADI, consists of states, pseudo-states, and transitions and are based on models proposed by Zhang and Cheng [15] and Zhao et al. [16].

MEADI is composed of several FSMs with transitions between them. Each time an aspect is added or removed, the model represents an adaptation to the time and system at runtime, switching to another FSM, which represents the new behavior. This adaptation corresponds to an action that changes the behavior of a state in one FSM to a state in another FSM.

States that are added or removed by aspects have $\langle\langle aspect \rangle\rangle$ stereotypes and are yellow in color, while states added by classes are white in color. The change from one FSM to another happens through special transitions called *priority transitions*. They have colors in these transitions that indicate whether the aspect has been added (red) or removed (blue). In addition, the MEADI has an element that indicates when a pointcut is encountered and the type of advice that acts on it. Table 1 shows the advice representation and the possible priority transitions for a joinpoint in a transition t and state S , where t comes from. From the table, we see that the priority transition always shows the advice contained in the aspect that was added or removed and the transition in which it is applied. The state S appears in the priority transition because there are times when the transition is executed and when it goes to the other FSM needs to return to the state before the transition in which the aspect affects. An example is when a security aspect is added with advice before for password verification before moving on to the next state. The added aspect leads to the state that checks the password and if the password is incorrect, it must go back to the previous state and not continue to the next state (as would happen if the password were correct).

Table 1. Advices and priority transitions representations.

Advice	Symbol	Added Aspect	Removed Aspect
before		$b\{t, S\}$	$-b\{t\}$
around		$ar\{t, S\}$	$-ar\{t\}$
after		$a\{t, S\}$	$-a\{t\}$

When the priority transitions are found, they take priority over the other transitions, and the FSM moves to the state they point, already on another machine. This occurs in specific states, called *quiescent states* [15], only where the aspect is added or removed. The reason for the adaptations to occur only in the quiescent states is because before reaching these states, the program has no change in its behavior. Even if an aspect is added or removed at any time, the

behavior of the program will change when this aspect is sensitized, causing the program to find the priority transition, which will lead to the transition to the FSM that models the new behavior.

The great limitation of the FSMs as a visual formalism for the description of complex systems is due to the problem of the explosion of the number of states and transitions that occur as the system becomes more complex [9]. To deal with this problem, the application under testing is modeled and tested by submodules.

4.1 Case Study: An Intelligent Transportation System

MESOADI was applied to two different case studies. The one chosen to be reported here refers to an intelligent transportation system [14]. The system, called ITS, is made up of a set of context-sensitive vehicles, and it was adapted to the dynamic OA context by Fuentes and Sanches [7].

In the ITS, vehicles navigate autonomously from a given origin to a predetermined destination. Each vehicle travels along a “virtual circuit”, which has to be previously calculated with the aid of a GPS for a given target point [7]. This circuit can be any outdoor arena, and the vehicle can travel inside it. The vehicle receives the information from the GPS periodically, the time interval being dependent on the vehicle speed.

When driven autonomously, the vehicle needs to build a real-time perception of its surrounding environment so that, if an error occurs, it makes decisions about its next move. Before a trip, vehicles are notified with the information and guidelines of the “virtual circuit”.

As the vehicle receives information from your GPS periodically, an error-handling module (an aspect) should monitor that the response time of the GPS is never exceeded, and react when this constraint is violated. If this constraint is violated, an error handling strategy must be applied. One possible solution is the temporary use of GPS data from a nearby vehicle. However, this is only possible if the vehicle is in circulation on a highway where the neighboring vehicles are going in the same direction and with an almost constant speed. If the vehicle is in the city, where the behavior of the vehicles is less predictable, information about other vehicles is out of use, and the human driver would be forced to manually control the vehicle until the GPS recovers.

A UML class diagram is shown in Fig. 1a to describe the ITS classes. This diagram shows two classes: (1) the `Context` class that store information about the current context of the vehicle, as speed mode (fast or slow) and type of path (city or highway); and (2) the `Coordinator` class which ensures that the foregoing elements cooperate properly so that the car is driven safely. This last class implements the behavior of the vehicle on or off, the driving mode is chosen (manual - when the driver drives the vehicle - or automatic - when the vehicle sails autonomously) and, in addition, there is verification of the next position it should follow if it is in automatic mode using GPS. Figure 1b shows a class diagram for the aspects of error handling and the change of context. Class names contain the `<<aspect>>` stereotype indicating that class is one aspect.

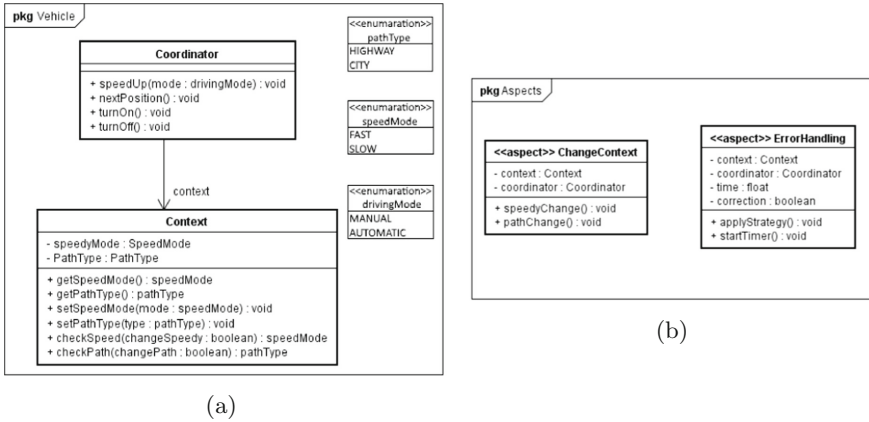


Fig. 1. Class diagram of (a) Coordinator and Context classes; and (b) ErrorHandling and ChangeContext aspects.

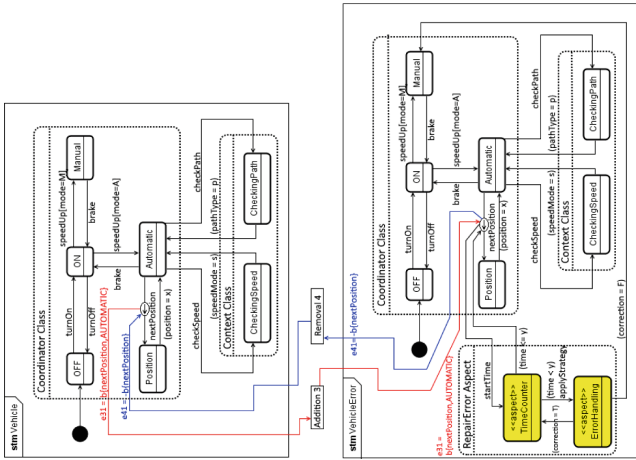


Fig. 2. MEADI with the ErrorHandling aspect added and removed from the application.

Using the MESOADI, Fig. 2 presents the MEADI with the ErrorHandling aspect that contains the FSM Vehicle and the FSM VehicleError. The aspect checks the length of time the vehicle has to receive information from your GPS. If an error occurs that is not resolved with respect to time, it needs to assume the manual mode. The modeling will not consider the various situations that may cause the error, only that an error has occurred that may or may not be corrected, because in this way the main objective, which is to show the iteration with the dynamic aspect, will be reached. This aspect is added to the system whenever the vehicle is navigating intelligently, so when the vehicle is navigating manually, the aspect can be removed. When the priority transition is found in the FSM Vehicle, one

aspect is added, and this transition is followed up to the FSM `Vehicle`. The same happens when a priority transition is found in the FSM `VehicleError`, it will follow to FSM `Vehicle`, where the aspect does not exist.

The initial state of the FSM `Vehicle` is called `OFF` because the vehicle is off. When the vehicle is switched on, the FSM moves to the `ON` state, where the vehicle is stationary, waiting for the driving mode to be activated to accelerate the vehicle. This driving mode can be manual, leading to the `MANUAL` state, or automatic, leading to `AUTOMATIC` state. In this last state, the vehicle starts driving without a driver, so you need to check the speed mode (`CHEKINGSPEED` state) and the type of the path (`CHEKINGPATH`) to adapt to the environment. In addition, a “virtual circuit” is plotted and the `POSITION` state contains the precise information about the next point at which the vehicle will be driven. When the priority transition is found in FSM `Vehicle`, one aspect is added, and this transition is followed up to FSM `VehicleError`. The added aspect is responsible for verifying that the GPS data is received at the correct time and adds two new states to the model: `TIMECOUNTER` and `ERRORHANDLING`.

This aspect has a before advice with pointcut in the transition `nextPosition` because it does a time count before checking the next position. This time count is done in the state `TIMECOUNTER`. If the time count ends and the GPS has not yet returned any position, that is the time is longer than expected, a transition to apply a strategy to this error leads for the `ERRORHANDLING` state, where a treatment strategy error message is applied. If the strategy is efficient and corrects the error, the system continues in automatic mode. Otherwise, the system switches to manual mode.

The `ChangeContext` aspect is responsible for detecting the message that may result in changes to the contextual information, requiring an appropriate update of the context. It notices changes in speed mode and path type. The aspect contains a pointcuts in the `checkSpeed` transition and `checkPath` transition. When the speed mode check transition is called, if the speed value exceeds a certain constant value, the vehicle is considered to be in the fast mode. Otherwise, it is in the slow mode. If it goes from slow to fast or vice versa, the `ChangeContext` aspect will make this change. The same occurs when the vehicle is in the city and goes to a highway or vice versa.

Figure 3 shows the MEADI that contain the states added by this aspect. In this example, the FSM `Vehicle` starts the system without any aspect and when a priority transition is found (`e11` or `e12`), it is followed, because the `ChangeContext` aspect has been added. At this point, the transition takes the system to the FSM `ChangeContextVehicle`, where two states are added, `CHANGESPEED` and `CHANGEPATH`. The advice is of the type `around`, because these states are substitutions of the states `CHEKINGSPEED` and `CHEKINGPATH`, respectively, given the condition that the speed or path has been changed. When the aspect is removed from the system, the transition `e21` or `e22` will be found, going to the FSM `Vehicle`.

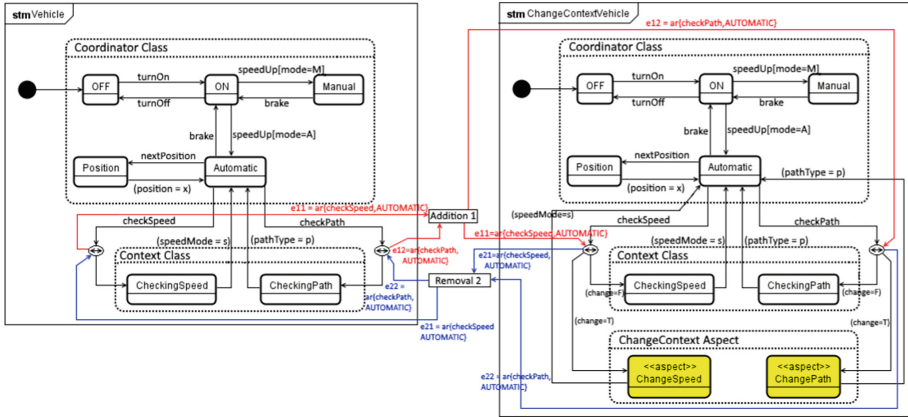


Fig. 3. MEAD with the ChangeContext aspect added and removed from the application.

4.2 Dynamic Combined Reacheability Tree

A state transition tree (STT) is the result of applying the RTP strategy to an FSM, and each path is a set of arcs from which a TC is derived. However, strategies developed for the OO paradigm, in general, can not be directly applied to the AO paradigm, taking into account the different specificities that exist for the treatment of cross-cutting concerns [13]. Furthermore, in a dynamic context, these strategies do not address the addition and removal of states and transitions at runtime.

In this work, STTs were modified and named Dynamic Combined Reachability Tree (DCRT) so that they can be applied to dynamic AO systems. DCRTs have priority transitions that take them to the other DCRTs. Each time a priority transition is found, it leads to the root of another tree, from which the TCs are derived. That is, the number of existing trees for the model is equal to the number of adaptations that the program may suffer, both when an aspect is added or removed. When there are static aspects or aspects already combined in the application, the states that represent them will be in the built DCRTs and, therefore, the number of trees will remain the same.

When an aspect is added, a pseudo-state will be the root of the new DCRT. This is due to the fact that a DCRT or TCs can not start with a transition and then a root state. In this case, it is necessary to have an initial state as root, which will be the pseudo-state. For the removed aspects, as the priority transition only indicates that it has been removed, it does not appear in the DCRT and therefore does not need to have a pseudo-state as root. The pseudo-states have the same priority transition nomenclature, but with the e uppercase, for example, for an e11 transition, the pseudo-state will be E11.

In DCRTs, states are represented by their names in capital letters, transitions by their names in lowercase letters, conditions are in square brackets and the returns of functions in parentheses, also in lowercase letters. The details of the construction of these DCRTs are presented below.

4.3 A New Strategy to Derive Test Sequences – $RTP_{MESOADI}$

In order to construct the DCRT from the dynamic modeling of the aspects, an adaptation of the original *Round-Trip Path* (RTP) method [3] was performed, named $RTP_{MESOADI}$. For each change in the MEADI, a new DCRT is constructed. The main change in the RTP relates to its stopping criterion, established as follows:

- When an aspect adds a state that is applied before, around, or after a transition, the transition t that precedes it and indicates the type of advice is saved. When the state appears again:
 - If the transition that precedes it is equal to t , then the DCRT repeats the state and stop the path;
 - Otherwise, the tree passes through the state and continues its generation obeying the original RTP stop criterion [3], that is, stop the path when a state is repeated or when it is a final state.

For the MEADI of Fig. 3, six DCRTs are derived using the $RTP_{MESOADI}$. The roots of these DCRTs are: initial state of the FSMs (1) Vehicle; (2) ChangeContextVehicle; quiescent states (3) e11; (4) e12; (5) e21; and (6) e22. For example, when the ChangeContext aspect is added, if the e11 priority transition from FSM

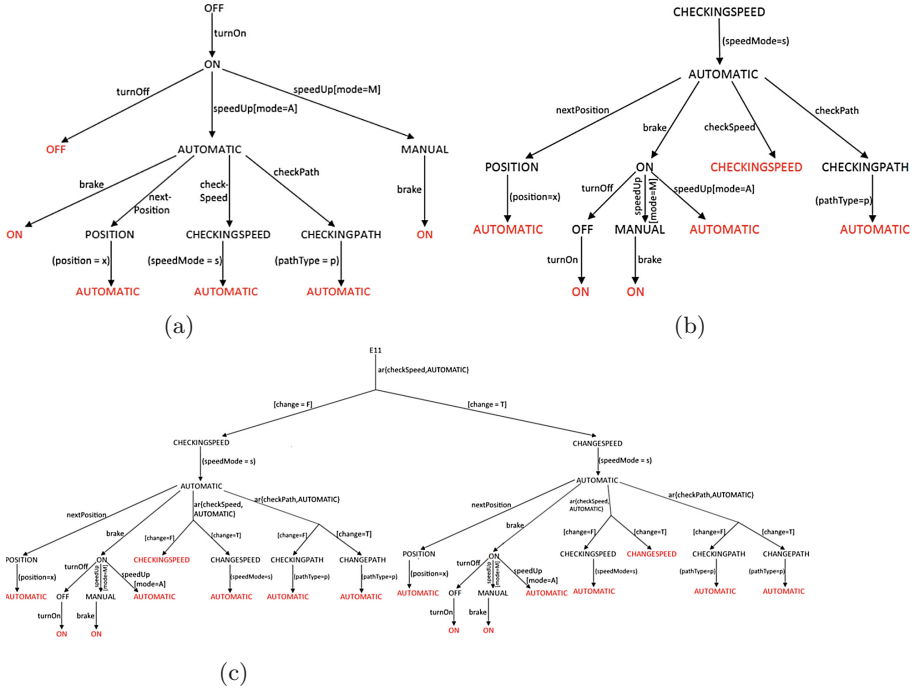


Fig. 4. DCRTs: (a) initial state of the FSM Vehicle; (b) quiescent state e21; and (c) quiescent state e11. (The red nodes are leaf nodes. (Color figure online))

Vehicle to the FSM `ChangeContextVehicle` is found, a pseudo-state `E11` is the root of a new DCRT. The same happens with the `e12` priority transition. The `e21` and `e22` priority transitions also lead to the creation of two new DCRTs, but the root of these DCRTs corresponds to the states that these transitions point to. Figure 4 shows the DCRTs with the root trees in an initial state `Vehicle`, a quiescent state with the addition of the aspect (`e11`) and a quiescent state with the aspect removal (`e21`).

The same occurs for the MEADI of Fig. 2 that four DCRTs are derived using the algorithm. The roots of these DCRTs are: initial state of the FSMs (1) `Vehicle`; (2) `VehicleError`; quiescent states (3) `e31`; and (4) `e41`.

With the TC derivation strategy used (RTP_{MESOADI}), DCRTs were created where each path corresponds to an abstract TC. For TCs to have coverage of 100%, all paths of all DCRTs must be traversed.

5 Results and Discussion

To evaluate MESOADI and its elements, we measured the effectiveness of testing sequences generated by this method through the use of mutation testing. In this section, Mutation Operators (MOs) are described for aspect-oriented models to be evaluated through fault-based testing. Then, we report the translation of the ITS to timed automata in a model-checker called UPPAAL. So, the MOs are applied to the automata for their simulation with the TCs derived from the RTP_{MESOADI} . Finally, the mutation scores are obtained.

MOs used here were based on and adapted from [10]. The work developed a mutation-based search for static AOP models. Therefore, for each of the proposed MOs, adaptations occurred in relation to the cited work. The MOs focus on the aspects and elements of AO (pointcuts and advice) because they are syntactic structures present only in AO models. Table 2 shows the MOs and their descriptions related to pointcuts and advices, respectively.

5.1 Timed Automata and Model-Checker

MESOADI was also analyzed using timed automata through the use of a model-checking tool called UPPAAL¹. Timed automata are FSMs that are extended with clocks. The main focus in the use of UPPAAL in this work is the use of a model checker algorithm to simulate the result of the application of the TCs derived from the algorithm RTP_{MESOADI} , not the verification of time or performance.

Timed automata are finite state machines with timing constraints associated with their edges and states, and are intended to model the behavior of real-time systems [5]. In a timed automata clocks are represented by a finite set of real-valued variables \mathcal{C} and events are represented by a finite alphabet Σ .

A network of timed automata $\mathcal{A}_1 || \dots || \mathcal{A}_n$ over (Σ, \mathcal{C}) is the parallel composition of n timed automata over (Σ, \mathcal{C}) , where components are required to

¹ Available at: <http://www.uppaal.org/>.

Table 2. Mutant operators to pointcuts and advices.

Mutant operator – pointcuts	Description
<i>Pointcut weakening (PCW)</i>	Adds the pointcuts that should be in the FSM, but also adds new pointcuts in transitions that are not affected by the aspect
<i>Incorrect pointcuts (IPC)</i>	Add in the FSMs only the pointcuts that are not contained in the original FSMs
<i>Pointcut strengthening (PCS)</i>	Does not selecting one or more pointcuts, selecting only a subset of correct pointcuts
Mutant operator – advices	Description
<i>Advice on incorrect pointcut (AIP)</i>	Associating existing advice with pointcuts that are also existent but incorrect
<i>Advice replacement at pointcut (ARP)</i>	Change the types of advices in the FSMs

synchronize on delay transitions and discrete transitions are required to be synchronized on complementary actions [10]. An action $a?$ is complementary to $a!$.

UPPAAL is a tool for validation (through graphic simulation) and verification (through automatic model checking) of systems in real time. The idea is to model a system using timed automata, to simulate them, and then to check their properties. The simulation step is to run the system interactively to see if it works as planned. In the verification step, the accessibility properties are checked, that is, whether a particular state is accessible or not. This is called model-checking and is basically an exhaustive search that covers all possible dynamic behaviors of the system. Only the graphic simulation step is used for the analysis of MESOADI. In UPPAAL, the double-circle state indicates that it is the initial state. The system consists of a network of processes that are made up of a set of locations. The transitions between locations define how it behaves.

Through synchronizations, it is possible to invoke or activate one or more transitions using a previously defined synchronization channel. The use of the “!” tag can be seen as a send and the “?” tag as a reception. When a process is in a state from which there is a transition with $c!$ synchronization, the only form of this transition is activated is if there is another process in another transition marked with $c?$ or contrariwise.

The ITS was manually translated into automata for UPPAAL. Because the tool does not support AOP, some adaptations were necessary. The states that have been added by aspects are represented by the yellow color for a better understanding of the model. When an advice is found, the automata were modeled to behave as expected, bearing in mind whether the advice is before, after or around. All transitions in the model will be represented by synchronizations. The mutation operators that were used for the analysis do not affect variables, so

even the transitions that in the original model exist variables will be represented by synchronizations.

The MEADI contains several FSMs, so for the simulations all the automata corresponding to each of these FSMs, and all the automata that have a quiescent state as the initial state, are modeled. In ITS there are nine automata with initial states in: (1) FSM Vehicle; (2) FSM ChangeContextVehicle; (3) FSM VehicleError; quiescent states (4) e11; (5) e12; (6) e13; (7) e21; (8) e22; and (9) e23. Automata 1, 2 and 3 will have the names of the FSMs to which they correspond and the automata 4, 5, 6, 7, 8 and 9 will have the name of their quiescent states, changing the e for E. Automats that do not contain states added by aspects will not be tested, because MOs only affect aspects. Thus, the MOs will be applied in five automata (ChangeContextVehicle, VehicleError, E11, E12 and E31).

The mutants generated from the original models are classified as: (1) killed mutants – when the mutant shows a behavior different from the original model; (2) equivalent mutants – when for each possible entry of the original model, the mutant version will show the same behavior (such mutants can not be distinguished from the original model by any test); and (3) stillborn mutants – when they are syntactically illegal and therefore are not accepted by the model verifier (for this reason, the latter are not considered in the analysis).

For the application of TCs, another process, called “TC”, is created next to each of the automata. This process contains the TC using the synchronizations for the simulation in the automata. Figure 5 shows TC derived from the DCRT of Fig. 4a, which uses the FSM Vehicle of the MEADI, being applied to the original automaton no UPPAAL. In the lower right window, the vertical arrows show the transitions and the horizontal arrows show the synchronization between automata. The last state of the TC is called “TC_passed” (in this case it would be the “ON” state), because every time the simulator is deadlocked (shown in the upper left window), if the last state of the “TC” process is this, having applied a MO, it means that the mutant remains alive. If the simulator is blocked and has not yet reached this state, it means that the TC has failed. Therefore, the mutant is considered killed. However, if for any possible simulation, the result with mutant or no mutant is the same, the mutants are called equivalent mutants. Thus, all mutants that remain alive need to be analyzed in order to determine whether or not they are equivalent.

In the case of stillborn mutants, the tool points to a syntax error and the templates are not sent to the simulation.

The analysis performed with the obtained results refers to the metrics of confidence or adequacy of the test cases generated by the MESOADI, being used as reference the mutation scores. The automata used show all possible scenarios for the modeling of ITS with MESOADI.

The presented case study contains two aspects that can be added or removed, where one of them contains two pointcuts. The MOs were applied at these pointcuts and their advices and all the possibilities modeled. In the examples that will be presented, the transitions, states, and synchronizations represented by the red color show the elements added to the model. The states, transitions,

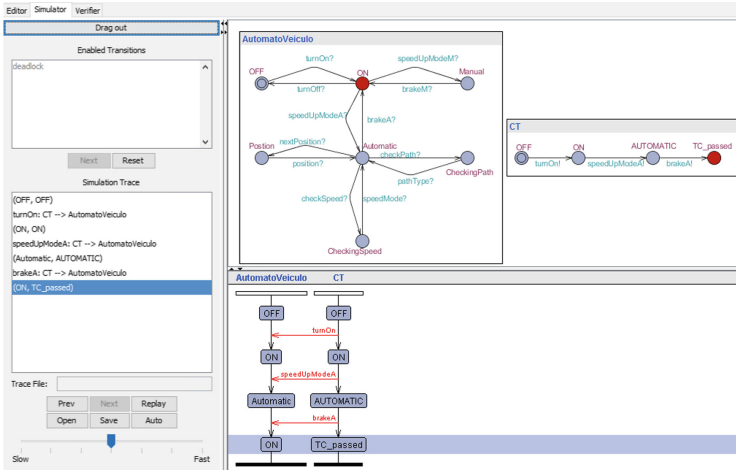


Fig. 5. Application of a TC in the automaton Vehicle

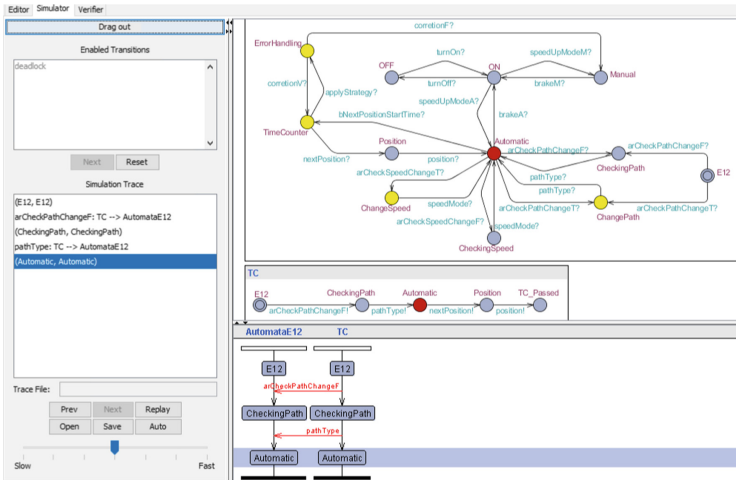


Fig. 6. Application of a TC to the automaton E12 mutated with the MO PCW

and synchronizations represented by gray color, and which contains two crossed traces, show the elements removed. The following are the MOs analyzed from this case study.

1. *PCW*: 7 mutants were generated for this operator, all killed by some TC. Figure 6 shows an example of this MO being applied to the E12 automaton and compared to a TC derived from $RTP_{MESOADI}$. This automaton should contain only the pointcuts of the *ChangeContext* aspect, however, the *PCW* operator also adds another pointcut of the *ErrorHandling* aspect in the “nextPosition”

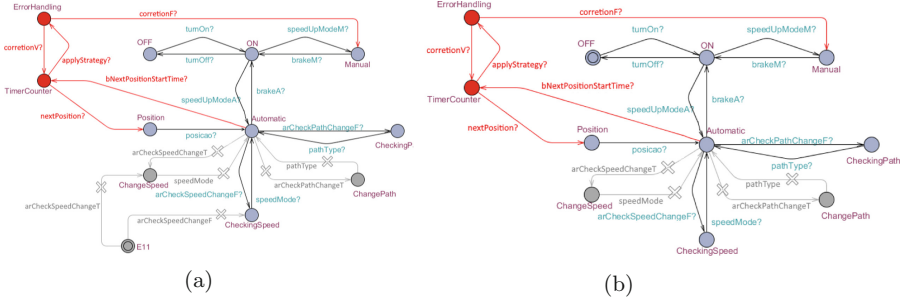


Fig. 7. *PCI* operator applied to automata (a) E11 and (b) ChangeContextVehicle.

transition. This mutant was killed because the simulator was blocked before the end of the TC scan.

2. *PCI*: 6 mutants were generated by this operator. Only 2 of them could be used since 4 mutants contained syntactic errors. The two mutants analyzed were killed by TCs. Figure 7a shows the two pointcuts of the ChangeContext aspect being deleted from the automaton and the pointcut of the ErrorHandling aspect being added. That is, the operator adds in the automaton E11 only a pointcut that is not contained in the original automaton. In the case of the E11 automaton, the initial state is removed and, for this reason, the mutant is then considered to be stillborn. Figure 7b shows how the automated ChangeContextVehicle was after the application of the *PCI* operator.
3. *PCS*: 6 mutants were generated by this operator. Of this total, 4 of them were used and killed, and 2 were characterized as stillborns, for the same reason as above.
4. *API*: 10 mutants were generated for this operator. For all of them, at least one TC derived from the $RTP_{MESOADI}$ killed the mutants.
5. *ARP*: 10 mutants were generated by this operator. All of them were killed by the TCs applied.

5.2 Effectiveness Level of the Generated Test Cases

DeMillo [4] provides an objective measure for the adequacy of the TCs of the *P* program by defining a mutation score, which consists in evaluating the suitability of *T* in relation to the test. This score ranges from 0 to 1 and the higher, the greater the effectiveness of the test suite generated.

The generation of the mutants was done carefully and manually. None of the mutants generated by the *PCW*, *API*, and *ARP* operators were considered to be stillborn. That is, all were used in the evaluation. As no equivalent mutant was generated by these operators, the obtained mutation scores have resulted in 1. The mutation operators *PCI* and *PCS* were the only ones that generated stillborn mutants.

With the simulation of the application of the TCs to the automata temporized with MOs in UPPAAL, it was observed that the TCs generated by $RTP_{MESOADI}$ to the ITS presented a high degree of adequability (effectiveness). Table 3 summarizes the mutation scores obtained by the mutation operator used.

Table 3. Mutation scores for the ITS using the MESOADI.

Mutation operator	Generated mutants	Killed mutants	Equivalent mutants	Stillborn mutants	Mutation score
<i>PCW</i>	7	7	0	0	1.00
<i>PCI</i>	6	2	0	4	1.00
<i>PCS</i>	6	4	0	2	1.00
<i>API</i>	10	10	0	0	1.00
<i>ARP</i>	10	10	0	0	1.00
Total	39	33	0	6	

The high mutation score obtained can be explained by the fact that it is a small system and that it generated a low number of mutants. For this reason, no equivalent mutants were generated. To obtain a more accurate result and to generate equivalent mutants, the MESOADI and the generation of the mutants need to be automated for the application in a more complex system and empirically validated in a future work.

6 Conclusion and Future Work

The test activity of dynamic aspects is by no means a trivial task. In addition, despite its importance in real-world applications, the development and testing of dynamic aspects are still under-explored areas. In dynamic aspect-orientated, with the addition or removal of runtime aspects, the loaded aspects may be incompatible with aspects already read and/or running, and removed aspects may lead the system to an inconsistent state. This work described the difficulties encountered for modeling and applying the test activity in this type of system.

The MESOADI constitutes a proposal for the application of state-based tests for dynamic aspects. The behavior represented by MEADI is described through several Finite State Machines (FSMs), which have transitions (adaptations) between them, allowing to represent how the system can adapt as the context change occurs, in this case, adding or removing aspects dynamically. For the derivation of the test cases (TCs) several Dynamic Combined Reachability Tree (DCRT) are constructed, by means of the $RTP_{MESOADI}$, where each one represents an adaptation suffered by the test application in a dynamic way. From the constructed trees, we obtain the TCs, derived from the sequences of transitions of these trees. The results obtained by the application of MESOADI showed that

the generated test sequences presented a high degree of suitability considering the mutation scores obtained in the evaluation.

Future work includes the development of a tool to support the MESOADI and carrying out an experimental study to evaluate in a more rigorous way the proposed approach.

Acknowledgments. The authors would like to thank CNPq (grant 455080/2014-3) and FAPESP for financial support.

References

1. Alam, F.E., Evermann, J., Fiech, A.: Modeling for dynamic aspect-oriented development. In: Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering, pp. 143–147. ACM (2009)
2. Alexander, R.T., Bieman, J.M., Andrews, A.A.: Towards the systematic testing of aspect-oriented programs. Rapport technique, Colorado State University (2004)
3. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, Boston (2001)
4. DeMillo, R.A.: Mutation analysis as a tool for software quality assurance. Technical report, DTIC Document (1980)
5. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed automata patterns. *IEEE Trans. Softw. Eng.* **34**(6), 844–859 (2008)
6. Ferrari, F.C., Rashid, A., Maldonado, J.C.: Towards the practical mutation testing of AspectJ programs. *Sci. Comput. Program.* **78**(9), 1639–1662 (2013)
7. Fuentes, L., Sánchez, P.: Dynamic weaving of aspect-oriented executable UML models. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI. LNCS, vol. 5560, pp. 1–38. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-03764-1_1](https://doi.org/10.1007/978-3-642-03764-1_1)
8. Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. *IEEE Trans. Softw. Eng.* **17**(6), 591–603 (1991)
9. Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**(3), 231–274 (1987)
10. Lindström, B., Offutt, J., Sundmark, D., Andler, S.F., Pettersson, P.: Using mutation to design tests for aspect-oriented models. *Inf. Softw. Technol.* **81**, 112–130 (2016)
11. Moreira, R.M., Paiva, A.C., Aguiar, A.: Testing aspect-oriented programs. In: 2010 5th Iberian Conference on Information Systems and Technologies (CISTI), pp. 1–6. IEEE (2010)
12. Myers, G.J., Sandler, C.: The Art of Software Testing. Wiley, Hoboken (2004)
13. Silveira, F.F., da Cunha, A.M., Lisbôa, M.L.: A state-based testing method for detecting aspect composition faults. In: Murgante, B., et al. (eds.) ICCSA 2014. LNCS, vol. 8583, pp. 418–433. Springer, Cham (2014). doi:[10.1007/978-3-319-09156-3_30](https://doi.org/10.1007/978-3-319-09156-3_30)
14. Sivaharan, T., Blair, G.S., Friday, A., Wu, M., Duran-Limon, H., Okanda, P., Sørensen, C.F., EU FET: Cooperating sentient vehicles for next generation automobiles. In: ACM/USENIX MobiSys 2004 International Workshop on Applications of Mobile Embedded Systems (WAMES 2004 Online Proceedings) (2004)

15. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 371–380. ACM, New York (2006)
16. Zhao, Y., Li, Z., Shen, H., Ma, D.: Development of global specification for dynamically adaptive software. *Computing* **95**(9), 785–816 (2013)