Testing adaptation policies for software components

Frédéric Dadeau, Jean Philippe Gros, Olga Kouchnarenko

Noname manuscript No.
(will be inserted by the editor)

Testing Adaptation Policies for Software Components

Frédéric Dadeau · Jean-Philippe Gros · Olga Kouchnarenko

the date of receipt and acceptance should be inserted later

Abstract Self-adaptive systems have to implement adaptation policies described by sets of rules, that express how the components are reconfigured within the system, the priority of a given reconfiguration to happen, when a given (sequence of) event(s) occurs and when specific conditions on the system state are satisfied. However, when this priority is given by a fuzzy value (e.g. high, medium, low) depending on external and internal events, it has to be implemented inside the software with particular implementation choices made.

This paper is dedicated to the validation of adaptation policies, using a model-based testing approach, and a verdict establishment that is based on both the runtime verification of temporal properties, and the detection of inconsistencies between the adaptation policy and the reconfigurations implemented in the self-adaptive system. We propose a means to establish a test verdict based on the respect of the adaptation policy by the implementation, along with coverage measures of the rules. This provides an interesting feedback on the adaptation policy rules, allowing to detect reconfigurations that should not have occurred, high-priority reconfigurations that are never triggered, or low-priority reconfigurations that are too frequently executed, potential inconsistencies in the rules, or wrong interpretation of priorities. The test verdict is made based on the analysis of the execution traces of the system, which is stimulated using a usage model that describes the probabilities of external events to occur. An experiment, performed on a Vehicular Ad-hoc Network of autonomous vehicles, illustrates the interest of the approach.

1 Context and Motivations

Recent years have seen the increase of cyber-physical systems which include a large scale of examples such as medical devices and systems, aerospace systems, transportation vehicles and intelligent highways with their associated problems such as security, safety and validity as mentionned in [28]. These systems are made of individual components

FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS
15B avenue des Montboucons, 25030 Besançon, Cedex, France
Email: `firstname.lastname@femto-st.fr`

that communicate together and react to changes in their execution environment by reconfiguration operations. Thus, the components can be activated or deactivated on-the-fly (i.e., during the system's execution) in order to adapt systems to the evolution of their execution context, measured by sensors. For example, a connected car may choose to rely on a WiFi signal, rather than on a GPS connection to save battery.

Dynamic reconfigurations modify the architecture of self-adaptive [9] component-based systems. To happen in suitable circumstance, reconfiguration operations are used in adaptation policies. Each policy is composed of reconfigurations and of rules that specify priorities of reconfigurations, that are guarded by specific (sequences of) events that may either exploit a state property, or involve temporal logic properties. Additionally, in order to ensure the system consistency throughout the successive reconfigurations that may occur, temporal properties can also be designed to be checked at run-time, and to be enforced by the system.

As mentionned in [17], during the development of the self-adaptive system, this formal model of an adaptation policy has to be implemented and, on this occasion, choices can be arbitrarily made. Indeed, the reconfigurations are triggered by sequences of external events that may occur, and guarded by a condition on the internal state of the system. Finally, the priorities on the reconfigurations can be expressed using fuzzy values (e.g. high, medium, low or with the notion of emergency), for which different interpretations can be made during the development. Thus, the conformance of the actual system reconfigurations w.r.t. the adaptation policy has to be established, to ensure that the system adheres to the defined reconfiguration priorities.

Usually, Model-Based Testing (MBT) approaches rely on the use of a behavioral model that describes how the system behaves and from which tests can be computed to ensure that the implementation conforms to the model [36]. Adaptive systems represent a challenge in the sense that, depending on the actual implementation of the adaption policy, a large set of system implementations is admissible, and it is irrelevant to settle a behavioral model (amongst all the possible ones) and require the implementation to stick to it. Similarily, it is not realistic to define a single model that captures all possible implementations of such a system. Thus, the validation procedure of such systems should rely on the analysis of relevant execution traces. It is thus mandatory to dispose of (*i*) a way to generate relevant test cases (and a way to determine the relevance of a test case, e.g. using coverage criteria), and (*ii*) a means to evaluate if the observed execution of the system conforms to its admissible behavior.

To address these issues we propose to employ a model-based testing approach that relies on the use of a probabilistic model that represents the environment in which the adaptive system is executed. We propose to establish the test verdict, resulting from the execution of the test cases, by relying on both the runtime verification of temporal properties that have to satisfied by the system, and the observation of the adaptation rules that are triggered w.r.t. the rules that were enabled. In addition, we propose to measure the coverage of the rules of the adaptation policy, in order to identify which rules have been triggered during the test, and at which frequency, so as to compare them to the fuzzy values they intent to represent. This approach aims to detect different situations that reflect a possibly-incorrect implementation of the adaptation policy, such as: (*i*) a high-priority rule that is never triggered whilst being applicable, (*ii*) a low-priority rule that is systematically applied while other rules with greater priority were applicable, and (*iii*) never-occurring rules which originate from inconsistent guards or unreachable triggering events.

Among this diversity of examples of adaptive systems, we have chosen the Vehicular Adhoc Network [20] case study which is a representative example of such systems. Besides, this example presents the difficulty of having multiple instances of vehicles that may appear or disappear dynamically, contrary to other systems in which the existing components are set initially and their number does not evolve. Our approach is experimented on this case study, for which we aim to evaluate: (i) the relevance of the coverage criteria that we have proposed, regarding the coverage of properties or the coverage of the reconfiguration rules, and (ii) the ability to produce tests that can detect errors.
This article describes three contributions.

1. We propose original coverage criteria, tailored to adaptive systems, based on the coverage of temporal properties, written using patterns, and adaptation policy rules.
2. We introduce a test verdict establishment that combines the run-time verification of temporal properties and the analysis of the reconfiguration traces w.r.t. the adaptation policy.
3. We present the evaluation of a random test generation process for validating adaptive component software based on a usage model of the component system.

This article is organized as follows. Section 2 describes the background of this proposal, namely self-adapting software component systems, and the associated formalism for describing adaptation policies and the temporal properties language. Then, the model-based testing process is described in Sect. 3, along with the definition of the coverage criteria that we propose. Section 4 presents the test generation approach that we develop, based on a usage model of the system's environment. Then, Section 5 explains how the test verdict is established, based on the execution traces that were previously generated, and describes the coverage measure that we perform on the adaptation policy rules. An experiment on a network of autonomous vehicles case study is reported in Sect. 6. Section 7 presents related works on testing adaptive systems. Finally, Section 8 concludes and presents future directions of this work.

2 Background

The section presents the framework of the self-adaptive component systems, for which we introduce a running example and provide basic formal definitions.

2.1 Component Systems on a Running Example

Let us start with a running example of a Cycab vehicle location controller. Cycab is an autonomous vehicle developed by Inria, whose first version has been used to compose an autonomous platoon of vehicles. We focus here on the localization part, the vehicle has a WiFi and a GPS signal receptors. The position is more accurate when both WiFi and GPS signals are activated but it consumes more battery. A component system is designed to reconfigure the activation of WiFi and GPS signals depending on the level of battery and the area of the Cycab. For example, if the Cycab is inside a WiFi area, the GPS component may be deactivated depending on the actual level of battery.

The developed Cycab component system is summarized in Fig. 1. The two components WiFi and GPS collect the location data. The Merger component retrieves these data and merges them with a trust coefficient. The Location component requests the merged data to the Merger component by the intermediate of the Controller component.

Let us now introduce our notion of component systems in relation with this example. In this paper, we follow the definition in [22] inspired by [33]. Components are compositions of entities that can be assembled to create an application. Components are independent and can be implemented independently as such. A same component can be instantiated several times. The component-based systems under consideration are hierarchical, with two types of components. Primitive components are basic components providing data or services, while composite components contain other components. Only primitive components can have some attributes used as configuration variables. In our example, the Location component is a composite component as it contains the Merger, Controller, GPS and WiFi components, which are in their turn primitive components.

Required and provided interfaces are interaction points between components. A provided interface is an interface that the component realizes, whereas a required interface is an interface that the component needs to be able to run. To facilitate the readability, an interface displayed on the left (resp. on the right) of a component is a provided interface (resp. required interface). In our example, the GPS component provides a gpsPosition interface used by the Merger component with the required interface getGpsPosition. Let $V = \{v_1, \ldots, v_n\}$ be a set of variables taking values from their respective domains $D_1, \ldots, D_n$[1]. The variables values and status of components can pass through the interfaces if the provided interface has access to them.

Bindings (or client-server links) and delegations link component interfaces. In the Cycab example displayed in Fig. 1, components Merger and GPS are in a (well-known) client-server relationship via their respective interfaces gpsPosition and getGpsPosition. Linking provided (resp. required) interfaces to provided (resp. required) interfaces puts them in a delegation relationship. In Fig. 1, Location and Controller components both have a required interface, securePosition for Location and position for Controller. This way Location component delegates his required interface to Controller component.

In the rest of the paper the state of a component system is called a configuration. As in [11], a configuration is a set of above-mentioned architectural elements (components, interfaces, and variables) together with their types and relations to structure and to link them. Let $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ be a set of configurations. We introduce a set CP of configuration propositions on the components and the relations between them. In

[1] Notably $\mathbb{B}_4 = \{\top, \top^p, \bot^p, \bot\}$ and $\mathbb{B}_2 = \{\top, \bot\}$ where $\top$ (resp. $\bot$) stands for *true* (resp. *false*) and $\top^p$ (resp. $\bot^p$) stands for potentially *true* (resp. potentially *false*).

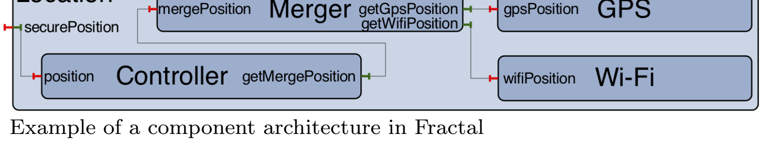

Fig. 1 Example of a component architecture in Fractal

particular, configuration propositions are used to define consistent configurations. An *interpretation* function $l : \mathcal{C} \to CP$ gives the largest conjunction of $cp \in CP$ evaluated to true on $c \in \mathcal{C}$ which is used to characterize the current state in the most precise way.

Reconfigurations make the component-based architecture evolve dynamically. They are combinations of primitive operations such as instantiation/destruction of components; setting components on/off; binding/unbinding of component interfaces; starting/stopping components; setting variable values of components. The normal running of different components also changes the architecture, e.g., by modifying variable values or stopping components. Let $\mathcal{R}_{run} = \mathcal{R} \cup \Theta \cup \{run\}$ be a set of evolution operations, where $\mathcal{R}$ is a finite set of reconfiguration operations, Θ is the set of operations triggered by external events, and *run* is the name of a generic action used to represent all the running operations of the component-based system. In this definition we assume, similarly to [21], that external events are captured by the system and processed immediately by triggering an internal method from the Θ set.

Definition 1 (Reconfiguration model) The operational semantics of component-based systems with reconfigurations is defined by the labelled transition system $S = \langle C, C^0, \mathcal{R}_{run}, \to, l\rangle$ where $\mathcal{C}$ is a set of configurations, $\mathcal{C}^0 \subseteq \mathcal{C}$ is a set of initial configurations, $\mathcal{R}_{run}$ is the set of evolution operations (including internal actions that do not perform a reconfiguration), $\to \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ is the reconfiguration relation, and $l : \mathcal{C} \to CP$ is a total interpretation function.

Let us note $c \overset{ope}{\to} c'$ for the transition $(c, ope, c') \in \to$, also called a *step*.

Example 1 (Reconfigurations on the running example) On the Cycab example, one of the possible reconfigurations consists in adding the GPS component to the current configuration. The addgps reconfiguration operation is the sequencing of the following primitive operations:

$$\mathsf{add}(gps);\ \mathsf{bind}(gpsPosition, getGpsPosition);\ \mathsf{start}(gps)$$

in which add is the operation that sets a component on (here the *gps* component in parameter), bind is the operation that binds together the two interfaces *gpsPosition* and *getGpsPosition* in parameter, and start is the operation that starts the component in parameter.

Definition 2 (Reconfiguration path) Given a reconfiguration model S, a reconfiguration path (or a path for short) σ of S is a sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall\, i \geq 0,\ \exists\ ope_i \in \mathcal{R}_{run}.\ (c_i, ope_i, c_{i+1}) \in \to$. The trace of σ, written $tr(\sigma)$, is the word $ope_0\, ope_1 \ldots ope_i \ldots$ composed of observed operations $ope_0, ope_1, \ldots, ope_i, \ldots$.

We write c_i or $\sigma(i)$ to denote the i-th configuration of σ. The notation σ_i denotes the suffix path $\sigma(i), \sigma(i{+}1), \ldots$, and σ_i^j the segment path $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1), \sigma(j)$. Let Σ denote the set of paths, and Σ^f ($\subseteq \Sigma$) the set of finite paths. A configuration c' is reachable from c when there is a path $\sigma = c_0, c_1, \ldots, c_n$ in Σ^f s.t. $c = c_0$ and $c' = c_n$. An execution is a path σ in Σ s.t. $\sigma(0) \in \mathcal{C}^0$ in which $\mathcal{C}^0$ is the set of initial configurations.

2.2 Temporal Properties

In this section, we briefly recall the FTPL[2] logic patterns introduced in [11]. In addition to configuration properties (*cp*) in *CP* mentioned above, the proposed logic contains external events (*ext*), as well as events from reconfiguration operations, temporal properties (*temp*) together with trace properties (*trace*) embedded into temporal properties. Let $Prop_{FTPL}$ denote the set of the FTPL formulae obeying the FTPL grammar in Fig. 2.

```
<FTPL>     ::=  <temp> | <events> | cp
<temp>     ::=  after <events> <temp>
             |  before <events> <trace>
             |  <trace> until <events>
<trace>    ::=  always cp
             |  eventually cp
             |  <trace> ∧ <trace>
             |  <trace> ∨ <trace>
<events>   ::=  <event>,<events> | <event>
<event>    ::=  ope normal
             |  ope exceptional
             |  ope terminates
             |  ext
```

Fig. 2 FTPL syntax

The FTPL semantics from [22] is summarized below. It is basic for events and configuration propositions, and runtime-oriented for other properties. External events (like events in [21]) occur instantaneously and can be seen as invocations of methods performed by (external) sensors when a change is detected in their environment. For each external event *ext* that may occur on a given execution path σ, we define *a*) a guard cp_{ext}, which is a first-order logic formula over the parameters specified in the invocation of the method *ext*, and *b*) an assertion $eval_\sigma$, valued in $\mathbb{B}_2$. Intuitively, if, at or before the *i*-th and after the $i-1$-th state (or, if $i=0$, at the first state) of an execution path σ, there is at least one occurrence of *ext* s.t. $cp_{ext} = \top$ then $eval_\sigma(cp_{ext}, i) = \top$, otherwise $eval_\sigma(cp_{ext}, i) = \bot$.

Definition 3 (FTPL semantics) Let $\sigma \in \Sigma$. The FTPL semantics $\Sigma \times Prop_{FTPL} \to \mathbb{B}_2$ is defined by induction on the form of the formulae as follows:

For configuration properties:		
$\sigma(i) \models cp$	*if*	$l(\sigma(i)) \Rightarrow cp$
For the event(s):		
$\sigma(i) \models ope$ **normal**	*if*	$i > 0 \land \sigma(i-1) \neq \sigma(i) \land \sigma(i-1) \overset{ope}{\to} \sigma(i) \in \to$
$\sigma(i) \models ope$ **exceptional**	*if*	$i > 0 \land \sigma(i-1) = \sigma(i) \land \sigma(i-1) \overset{ope}{\to} \sigma(i) \in \to$
$\sigma(i) \models ope$ **terminates**	*if*	$\sigma(i) \models ope$ **normal** $\lor\ \sigma(i) \models ope$ **exceptional**
$\sigma(i) \models ext$	*if*	$eval_\sigma(cp_{ext}, i) = \top$
$\sigma(i) \models event, events$	*if*	$\sigma(i) \models event \lor \sigma(i) \models events$
For the trace properties:		
$\sigma \models$ **always** cp	*if*	$\forall\, i.(i \geqslant 0 \Rightarrow \sigma(i) \models cp)$
$\sigma \models$ **eventually** cp	*if*	$\exists\, i.(i \geqslant 0 \land \sigma(i) \models cp)$
$\sigma \models trace_1 \land trace_2$	*if*	$\sigma \models trace_1 \land \sigma \models trace_2$
$\sigma \models trace_1 \lor trace_2$	*if*	$\sigma \models trace_1 \lor \sigma \models trace_2$
For the temporal properties:		
$\sigma \models$ **after** $event\ temp$	*if*	$\forall\, i.(i \geqslant 0 \land \sigma(i) \models event \Rightarrow \sigma_i \models temp)$
$\sigma \models$ **before** $event\ trace$	*if*	$\forall\, i.(i > 0 \land \sigma(i) \models event \Rightarrow \sigma_0^{i-1} \models trace)$
$\sigma \models trace$ **until** $event$	*if*	$\exists\, i.(i > 0 \land \sigma(i) \models event \land \sigma_0^{i-1} \models trace)$

A reconfiguration model S satisfies a property $\phi \in Prop_{FTPL}$, denoted $S \models \phi$, *if* $\forall\, \sigma.(\sigma \in \Sigma(S) \land \sigma(0) \in \mathcal{C}^0 \Rightarrow \sigma \models \phi)$.

[2] FTPL stands for TPL (Temporal Pattern Language) prefixed by 'F' to denote its relation to Fractal-like components and to first-order consistency constraints over them.

2.3 Adaptation Policies

Adaptation policies are defined by: 1. architectural reconfiguration operations to specify the possible modifications of the architecture; and 2. adaptation rules to link the properties concerning the component-based system and the need[3] to activate a reconfiguration.

In our approach, reconfigurations in adaptation policies are guarded by specific (sequences of) events that may either exploit a configuration proposition, or involve temporal logic properties. We adapt definitions in [6,22] to fit in with our component-based system model semantics, when aiming to test adaptation policies applications.

Definition 4 (Adaptation policy) Let S be a reconfiguration model, and *Ftype* a set of fuzzy types. Given $\sigma(i) \in \mathcal{C}$, an adaptation policy for $\sigma(i)$ is defined as $A = \langle R_N, R_R \rangle$, where:

- $R_N \subseteq \mathcal{R}$ is a finite (non-empty) set of reconfiguration names,
- $R_R = \{\langle F, B, G, I \rangle\}$ is a finite (non-empty) set of adaptation rules, where
  - $F \in \mathit{Ftype}$ is a fuzzy type,
  - $B \subseteq \{\phi_\sigma(i) = \mathit{value} \mid \phi \in \mathit{Prop}_{FTPL} \wedge \mathit{value} \in \mathbb{B}_2\}$ is a set of properties in Prop_{FTPL} evaluated in $\mathbb{B}_2$ on $\sigma(i)$,
  - $G \subseteq \{cp_\sigma(i) = \mathit{value} \mid cp \in CP \wedge \mathit{value} \in \mathbb{B}_2\}$ is a set of configuration propositions in CP evaluated in $\mathbb{B}_2$ on $\sigma(i)$,
  - $I \subseteq R_N \times F$ is a relation between reconfigurations and fuzzy values.

Let $AP = \{A, A', \ldots\}$ denote a finite set of adaptation policies for $\sigma(i)$. Let us denote $B_{\sigma(i)}$ (resp. $G_{\sigma(i)}$) the conjunction of the properties evaluations in B (resp. guards evaluations in G) on $\sigma(i)$.

In practice (see e.g. [6,24]), an adaptation policy can be described as a set of rules of the syntactic form:

```
when trigger b
   if guard g
   then utility of reconfiguration R is priority p
```

in which: ***trigger*** $b = \wedge_{b_j \in B} b_j$ is a condition on the value of a FTPL property, that designates the piece of the system's execution during which the reconfiguration may be performed; ***guard*** $g = \wedge_{g_l \in G} g_l$ is a condition on the value of a first-order logic predicate on the current configuration of the system, which expresses the conditions under which the reconfiguration may occur; ***reconfiguration*** $R \in R_N$ is the name of the reconfiguration that may occur, with a priority level designated by ***priority*** p.

Example 2 (Adaptation policy rules) On the Cycab example, the following rule

```
when (after start, exit (TRUE until entry)) = TRUE
   if (gps in Components and wifi in Components) = TRUE
   then utility of removegps is high
```

specifies that, after entering a Wi-Fi zone (in which the GPS signal is not available), if both the GPS and WiFi components are active, then it is relevant to remove the GPS. This rule is based on the occurrence of a given external event.
The second rule

[3] As in [6,10], fuzzy values (e.g. in {`low`, `medium`, `high`}) are used to express this need.

```
when (power < 33) = TRUE
   if (gps in Components and wifi in Components) = TRUE
   then utility of removegps is high
```

specifies that when the battery of the Cycab is below a given threshold it is interesting to remove the GPS if both GPS and WiFi components are set on. Contrary to the previous rule, this latter is not triggered by an external event but rather by the constant evolution of the battery level over time.

The adaptation policy represents a model of how the system should behave depending on the events raised by the environment and the current configuration. This behavior has to be implemented when the system is built. Nevertheless, it is possible that the developer makes arbitrary choices on the priority of the rules, or incorrectly implements the detection of the applicable reconfigurations. This may cause the system to reconfigure when it should not, or to skip relevant reconfigurations that would prevent certain properties from being violated. The testing process described in the following sections aims to validate that the adaptation policy is correctly implemented by detecting such errors.

3 Model-Based Testing Process for Component Systems

In this section, we first present the Model-Based Testing approach (MBT) that we propose to validate component systems under adaptation policies. After having introduced the general approach, we present the dedicated coverage criteria that we propose, based on the two available artifacts: temporal properties in FTPL, and adaptation policies.

3.1 Overall approach

The overall approach can be seen as a model-based testing approach [3], as depicted in Fig. 3 In MBT, the model is used to generate test cases and/or establish the test verdict that concludes on the conformance of the system under test (1), namely the implementation under adaptation, w.r.t. the model.

In this approach, several models are considered. As a first novel contribution, we propose to use an usage model (2) that describes the behaviour of the system, by formalizing possible sequences of events using a probabilistic automaton. This latter is used to generate the test cases as a set of *traces*, namely of sequences of external events. The temporal properties (3) are verified at runtime during the system's execution to ensure that this latter satisfies the properties, contributing to the test verdict establishment. This step is made by employing the monitoring technique proposed in a previous work [23].

Finally, it is mandatory to check that the adaptation policy rules (4) have been faithfully implemented, and especially, that the priorities have been respected, to ensure the conformance. To achieve this last part, we propose, as a novel contribution, several conditions that can be used to establish the test verdict related to the compliance of the system to the adaptation policy. We propose to evaluate, at each step of the execution, which reconfigurations were eligible, and compare them to the actual reconfigurations that are observed on the system. Our last contribution is the computation of metrics

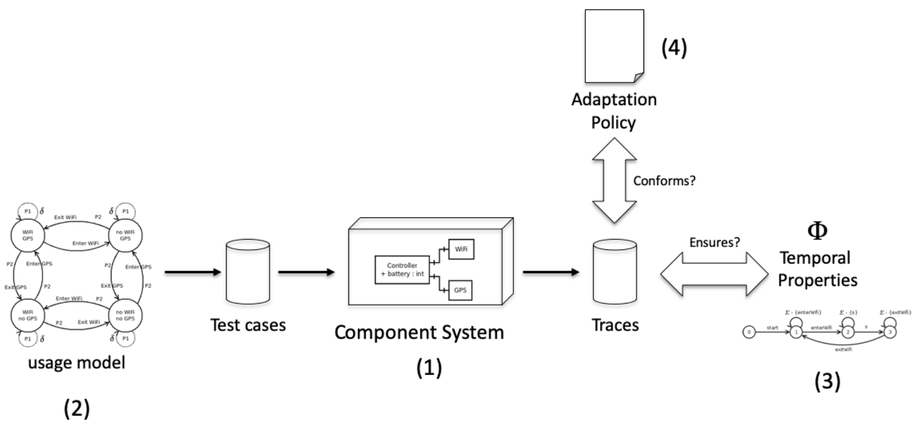

Fig. 3 MBT approach for validating adaptation policies

regarding the occurrences of reconfigurations and the triggering of associated rules which provides an interesting feedback to the developer.

In the rest of this section, we first define the test cases and their executions. We then define the coverage criteria that we define, based on the reconfiguration paths and the considered artifacts: the FTPL properties and the adaptation policy.

3.2 Test Cases and Reconfiguration Paths

In order to analyze the execution results of the test cases that we produce–to either establish a verdict or provide a test coverage measure, it is mandatory to be able to observe what happens on the implementation when the tests are run.

In our approach, a test case is defined as a finite sequence of external events that are sent to the adaptive system. As we consider time-sensible systems, we add a special event denoted by δ which represents the fact that no action is performed on the system for a given period of time. δ defines a controllable quiescence. In the case of real-time systems, it can be suffixed by the number of time units the tester waits. For example, δ^{2s} indicates that the absence of external events lasts 2 seconds before considering the next test step. In the case of a discrete event system, the quiescence corresponds to one step. Similarly, a suffix can be specified to additionally indicate the number of steps (e.g. δ^2 for 2 steps).

Definition 5 (Test case) Let Θ be the set of external events to which the system reacts, a test case tc is defined as a sequence of events which can be either (i) an external event $ev \in \Theta$, or (ii) a controllable quiescence δ.
A non-empty set TS of test cases is called a *test suite.*

When executed, a test case is mapped onto a reconfiguration path of the model, in Def. 2. Notice that the controllable quiescence δ does not appear in the reconfiguration paths of the model.Thus, the controllable quiescence notion in the considered paradigm (real-time or discrete event systems) and reconfiguration path notion can be seen as independent.

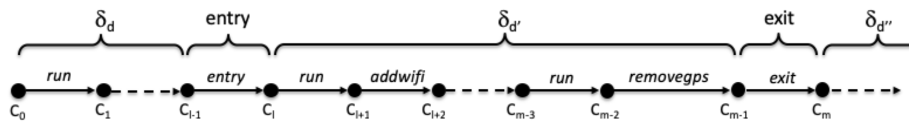

Fig. 4 A reconfiguration path (below) and a test case (above)

In practice, a test case can be seen as a log trace which records the activity w.r.t. the various interactions between the system and its environment. Thus, the detection of an external event by the adaptive system may lead to several cases. This event may update the internal state of the system, which will be mapped to a *run* in the path. Such an update may possibly be followed by a reconfiguration that is triggered by the implementation. This is mainly the case of δ events which let the system evolve when no external events occur. Otherwise, the external event may directly trigger a reconfiguration in response.

Let tc be a test case, as in Def. 5. We denote by $exec(tc)$ the reconfiguration path that corresponds to the execution of test case tc on the system under test.

Example 3 (From a test case to a reconfiguration path) Figure 4 illustrates an execution of the test case and a reconfiguration path, i.e., a sequence of configurations with reconfiguration operations/external events linking them. In this example, the test case starts by a quiescence moment when the battery level decreases. This quiescence is mapped to several *run*-operations in the reconfiguration path. When the environment generates an *entry* event, as soon as it is observed by the system, it fires the reconfiguration that adds the WiFi component to the system. Later on, during a subsequent δ period the battery level drops under a given threshold, which triggers another reconfiguration that removes the GPS component to save energy. Notice that these reconfigurations correspond to the implementation of the adaptation policy rules that are associated with the component system.

We now define the coverage criteria based on these definitions.

3.3 Coverage Criteria for Components under Adaptation Policies

In our testing process, two model artifacts can be used to define the testing objectives: the FTPL properties and the adaptation policy rules. We propose to define dedicated coverage criteria for the following purposes. First, these criteria can be used as a means to evaluate a test suite, by measuring how much of the considered artifact the test suite covers. Second, these criteria provide a test objective, namely a stopping criterion for the test generation that can be used to decide when to stop the test generation phase.

Notice that the following assumes the ability to observe the successive configurations of the system's execution.

3.3.1 FTPL Property Coverage using Automata

FTPL properties are used to describe safety or liveness properties that can be checked at run-time during the system's execution. As explained in Sect. 2.3, FTPL properties are expressed using a pattern-like scheme. Thus, from a textual property expression,

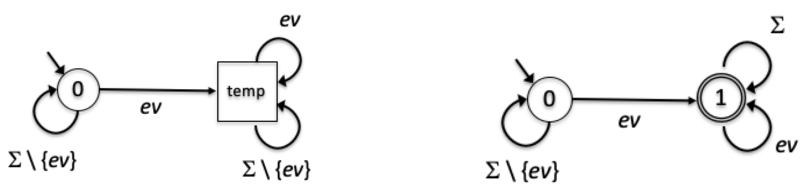

Fig. 5 Automata for **after** *ev temp* (left), and **before** *ev trace* or *trace* **until** *ev* (right)

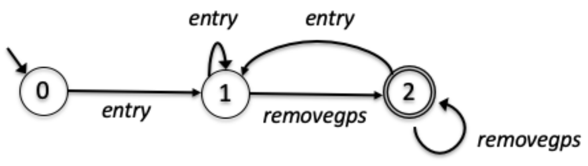

Fig. 6 Automaton for the **after-before** patterns combination

it is possible to build a test property automaton, labelled with the events involved in the FTPL property, like in [34].

The computation of the automaton associated with the property is made by composing the basic automata for the **after**, **before** and **until** patterns depicted in Fig. 5. In this figure, the squared state represents a hierarchical state where one of the two automata can be inserted. Double-circled state represents final states which show the end of the property scope as expressed in its textual version (cf. [34] for more detail).

To remove hierarchical states, this automaton is flattened in the following way:

– the event labelling the reflexive transition on the hierarchical state is removed from the outgoing transitions of the inner automaton states;
– the reflexive transition on the hierarchical state is copied and duplicated on each state of the inner automaton;
– all transitions reaching the hierarchical state are connected to the initial state of the inner automaton.

Notice that the automaton for property φ focuses only on the *relevant* events regarding φ, and the other events are not represented on the automaton transitions.

Definition 6 (FTPL property automaton) Let φ be an FTPL property on a model S, and let Ev_φ be the set of events occurring in φ. The automaton associated with φ, denoted by $\mathcal{A}_\varphi$, is defined as a quadruplet $\langle Q, q_0, Q_f, T\rangle$ where Q is a set of states, $q_0 \in Q$ is the initial state, $Q_f \subset Q$ is the set of final states, and $T \in Q \times Ev_\varphi \times Q$ is the transition function.

Example 4 (An FTPL Property Automaton) The automaton for property **after** entry **before** removegps **eventually** power < 33 is given in Fig. 6.

In order to measure the coverage of a property automaton, each step in the reconfiguration path is mapped to a state of the automaton.

Definition 7 (Mapping ***state*****)** Let $\mathcal{A}_\varphi$ be a property φ automaton, and let σ be a reconfiguration path. The *state* mapping associating a state in $\mathcal{A}_\varphi$ with each configuration of σ, is recursively defined by:

$$state(\sigma(0)) = q_0$$

$$state(\sigma(i)) = \begin{cases} q_i \ \mathit{if} \ \exists\, ev \in \Theta.\sigma(i-1) \overset{ev}{\rightarrow} \sigma(i) \wedge state(\sigma(i-1)) \overset{ev}{\rightarrow} q_i \in T_{\mathcal{A}_\varphi} \\ q_{i-1} \ \mathit{otherwise} \end{cases}$$

We now define the FTPL property coverage criteria for testing, by adapting [8].

Definition 8 (Covered transitions in $\mathcal{A}_\varphi$) Let *tc* be a test case, $\sigma = exec(tc)$ an evolution path linked to *tc* execution, and let φ be an FTPL property. The set of *covered* transitions in $\mathcal{A}_\varphi = \langle Q, q_0, Q_f, T\rangle$, denoted $covered(tc, \varphi)$, is defined as

$$\{q \overset{ev}{\rightarrow} q' \in T \mid \exists\, j \geq 0.\ (state(\sigma(j)) = q \wedge state(\sigma(j+1)) = q' \wedge \sigma(j) \overset{ev}{\rightarrow} \sigma(j+1) \wedge \exists\, k \geq j.state(\sigma(k)) \in Q_f)\}$$

Notice that this definition requires, for a transition to be considered as covered, that the execution reaches afterwards a state mapping to a final state of the automaton. As in a final state a decision can be made on the validity of the property, it is thus mandatory that a test case reaches such a state. Otherwise, the property cannot be evaluated as it is impossible to conclude on its validity.

Example 5 (Property coverage on a Test Case execution) Let us consider property **after** *entry* **before** *removegps* **eventually** *power* < 33 whose automaton is depicted in Fig. 6. For the test case and an associated reconfiguration path in Fig. 4, let assume that the incomplete parts of the path contain only *run* operations. Then the transitions of the property that are covered by this test case are $0 \overset{entry}{\rightarrow} 1$ and $1 \overset{removegps}{\rightarrow} 2$.

Definition 9 (Property coverage) An FTPL property φ is said to be *covered* by a test suite *TS*, if each transition of property φ automaton is covered by at least one test case in *TS*:

$$\bigcup_{tc \in TS} covered(tc, \varphi) = T_{\mathcal{A}_\varphi}$$

3.3.2 Adaptation Policy Coverage

Adaptation policies are defined as a set of rules, that apply to a given component, as presented in Sect. 2.3.

Definition 10 (Adaptation rule coverage) Let *tc* be a test case, $\sigma = exec(tc)$ an evolution path linked to *tc* execution, and $A = \langle R_N, R_R\rangle$ an adaptation policy. A rule $r = \langle F, B, G, I\rangle \in R_R$ is said to be *covered* by *tc*, denoted *tc covers r*, if the following holds:

$$\forall\, ope.ope \in dom(I) \Rightarrow \exists\, c, c' \in \sigma.c \overset{ope}{\rightarrow} c' \wedge B_c \wedge G_c$$

Intuitively, this definition corresponds to the activation, at a given step of the reconfiguration path, of the reconfiguration operation that is described in the adaptation rule, from a configuration in which rule's trigger and guard are both true. Notice that this coverage criterion requires a rule to be executed at least once, regardless of its utility.

We now lift this coverage notion to the set of rules. The coverage of the adaptation policy is defined as the activation of all the rules contained in the adaptation policy. This coverage is evaluated on a set of reconfiguration paths obtained by the execution of the test cases.

Definition 11 (Adaptation Policy Coverage) Let R_R be the set of reconfiguration rules in a given adaptation policy A, and let TS be a test suite. The adaptation policy A is said to be *covered* if each adaptation rule $r \in R_R$ is covered by a test case tc: $\forall\, r.r \in R_R \Rightarrow \exists\, tc \in TS.\, tc\ covers\ r$

These two test criteria provide test objectives tailored to temporal properties in FTPL and adaptive systems, by focusing on their adaptation policy. We now present two mandatory aspects of the testing process, namely the test generation, that is in charge of computing test cases, and the test verdict, that is in charge of establishing if the execution of a test case succeeds or fails.

4 Test Generation with a Usage Model

This section describes the test generation process that we propose. It aims to produce test cases as sequences of external events that exercise the system, as defined in Def 5.

Adaptive systems react to external events, (presumably) accordingly to an adaptation policy that provides guidelines to detect when to execute reconfigurations on the system. However, many different, but correct, implementations of the adaptation policy are possible. Thus describing a model of these implementations is a complex task that would require to make many design choices on the behavior of the system.

In order to avoid the description of a complete model, from which test cases can be computed, we propose to consider a usage model of the system under test [39]. Thus, in our case, the model does not represent the systems behavior, but rather describes how is the environment in which the system is executed. As a consequence such models are usually smaller than models describing a whole system.

This kind of models mainly aims to specify the various events that may occur in the environment. We focus on controllable events that can be sent to the system under test and to which it reacts. As we consider time-based systems, in addition to the external events of the system, we explicitly define the controllable quiescence (as defined in Sect. 3.3), as an absence of external event for a given time period, during which the system updates its sensors and possibly its internal state. The most common way to design such a model is to rely on probabilistic automata as follows.

Definition 12 (Usage model probabilistic automaton) A usage model is defined as a deterministic probabilistic automaton $\mathcal{A}_p = \langle Q, q_0, A_\delta, F, P\rangle$, where Q is a set of states, $q_0 \in Q$ is the initial state, A_δ is the set of controllable events, made of the set of external actions augmented by Δ which designates the set of controllable quiescence (the absence of external events), F is a transition relation $F \in Q \times A_\Delta \nrightarrow Q$, and P is the probability of a transition $P : Q \times A_\Delta \to [0;\ 1]$ such that $\forall\, p \in Q \Rightarrow \Sigma_{a \in A_\Delta} P(p, a) = 1$[4].

A difference between these automata and those describing temporal properties is twofold. First, property automata may display internal events, namely reconfiguration operations, that are not visible from the environment point of view. Second, environment automata display the controllable quiescence that is not present in the property automata. This definition is illustrated on our running example.

[4] We assume that if an event does not label an outgoing transition of the current state, its probability is 0.

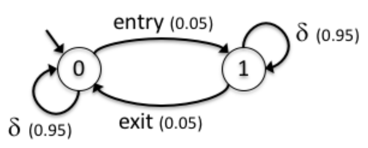

Fig. 7 Usage model of the Cycab example

Example 6 (Usage Model of the CyCab) The Cycab example displays two external events: *entry* and *exit*, which respectively denote the entry and exit of a Wi-Fi zone. Assuming that these events only occur with a (user-defined) 0.05 probability, the usage model for this example is given by the graphical representation in Fig. 7, in which number in parentheses designates the probability associated with the transition. Notice that we assume that the time is discrete, and thus, δ represents a quiescence of one time unit.

In this setting we define a test case as a sequence of transitions obtained by traversing the usage model automaton of the system. A test case for a given usage model $\mathcal{A}_\delta = \langle Q, q_0, A_\delta, F, P\rangle$ is a trace of a path in automaton $\mathcal{A}_\delta$, namely, a finite sequence of events $e_0 e_1 ... e_{n-1}$ in which, at each step, the triggered event e_i has a probability $P(q_i, e_i) > 0$. This characterizes a path in $\mathcal{A}_\delta$ that starts from the initial state and follows existing transitions with non-zero probabilities.

In order to compute the test cases, a Markov random walk [30] is performed on the probabilistic model. The considered test generation algorithm, sketched in Fig. 1 is a classical walk through the automaton by firing the outgoing transitions of the current exploration state. It results in a sequence of events that provides a means to stimulate the system under test in a coherent way w.r.t. the actual environment. Notice that this algorithm is parameterized by a bound in the size of the test case.

```
1 output ← []
2 $q = q_0$
3 $i \leftarrow 0$
4 while i < n do
5     choose an outgoing transition $a \in A_\delta$ from $q$ according to its probability
6     output ← concat(output, [$a$])
7     $i \leftarrow i + 1$
8 done return output
```

Algorithm 1: Test generation algorithm

Example 7 (Test case for the Cycab example) Figure 8 displays an example of a test case for the Cycab component system. This test case displays frequent occurrences of the quiescence (δ) which represents a period of time during which the systems state evolves without any request from the environment; concretely its battery decreases.

This algorithm can be employed to generate test suite, that aims to reach specific coverage criteria, such as those given in Sec. 3.3. With such test cases, the system is stimulated by means of external events, that may either trigger the evolution of the system's internal state, or trigger a reconfiguration of the system, as presented in Sec. 3.

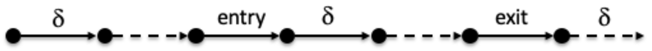

Fig. 8 A test case for the Cycab example

When the test case is executed on the system, it produces a reconfiguration trace that can be analyzed to decide if the system complies with the various specifications, namely, the adaptation policies and the temporal properties that are formalized. The test execution phase with verdict establishment and coverage measurements are now described.

5 Test Verdict w.r.t. Adaptation Policy

The test verdict that we intend to establish is based on two formalized artifacts. First, it refers to temporal properties that can be verified at run-time. Second, it is based on the adaptation policy that defines conditions under which the system may execute reconfigurations. This section presents how these two formal artifacts can be used to establish the test verdict.

5.1 Test Verdict based on Runtime Properties Checking

This part relates to a previous work [23] on the verification of FTPL properties. The semantics of the property language is provided in Sec. 2.

A test case, when executed, provides a finite reconfiguration trace, against which FTPL properties can be verified. However, when the test case does not cover the property, it is impossible to decide if this latter is satisfied or not. For example, consider an **after** e_1 **before** e_2 **eventually** cp property. If the trace stops after having observed e_1 but before observing e_2 it is impossible to conclude on the eventual occurrence of cp in this interval. In this case, the property should be declared as "potentially false" ($\perp^p$ using the semantics in [23]), and the test case which produced the trace is inconclusive regarding this property.

Hence, our test verdict establishment process relies on a prefix of the reconfiguration trace that stops if no subsequent configuration is associated with the final state for the property automaton.

Example 8 The part of the reconfiguration trace in Fig. 4 that is used to evaluate the FTPL property of Example 4 ends at configuration c_{m+1} since no subsequent final states of the automaton (in Fig. 6) can be reached afterwards.

We now define the test verdict for the execution of test case w.r.t. a given property.

Definition 13 (Test verdict w.r.t. an FTPL property) The verdict of the execution of a test case tc that produces, when executed, reconfiguration trace $tr(\sigma)$, w.r.t property φ is defined by :

$$verdict(tc, \varphi) = \begin{cases} \textit{pass if } \exists\, i.state(\sigma(i)) \in Q_{f_{A_\varphi}} \wedge \sigma_0^i \models \varphi \wedge \forall\, k > i \wedge state(\sigma(k)) \notin Q_{f_{A_\varphi}} \\ \textit{fail if } \exists\, i.state(\sigma(i)) \in Q_{f_{A_\varphi}} \wedge \sigma_0^i \not\models \varphi \\ \textit{inconclusive if } \forall\, i.state(\sigma(i)) \notin Q_{f_{A_\varphi}} \end{cases}$$

Notice that each test case of a given test suite has to be evaluated for each property of interest that is designed on the system.

5.2 Test Verdict based on the Adaptation Policy

This section defines how a second test verdict can be established, this time based on the information contained in the adaptation policy $A = \langle R_N, R_R \rangle$. Intuitively, this consists in detecting the occurrences of unexpected reconfigurations, based on the reconfiguration rules that are described in the adaptation policy.

For each configuration $\sigma(i)$ of σ related to the execution of a test case, we compute three sets of reconfiguration operations that we define as follows:

- $\mathit{trig}_{\sigma(i)}$ is the set of adaptation rules that can be triggered in $\sigma(i)$:

$$\mathit{trig}_{\sigma(i)} = \{r \in R_R \mid B^r_{\sigma(i)} = \mathit{true}\}$$

- $\mathit{elig}_{\sigma(i)}$ is the set of adaptation rules that can be activated:

$$\mathit{elig}_{\sigma(i)} = \{r \in R_R \mid B^r_{\sigma(i)} = \mathit{true} \wedge G_{\sigma(i)} = \mathit{true}\}$$

- $\mathit{actual}_{\sigma(i)} \in \mathcal{R}_{run}$ which designates the operation that was executed in $\sigma(i)$.

These sets help us identify the set of reconfigurations that could be executed in a given configuration during the system's execution. In addition, let $R_{N\,\mathit{trig}_{\sigma(i)}}$ (resp. $R_{N\,\mathit{elig}_{\sigma(i)}}$) denote the set of names of reconfiguration operations in R_N that are concerned in the utility part I for the rules in $\mathit{trig}_{\sigma(i)}$ (resp. $\mathit{elig}_{\sigma(i)}$).

We now explain how to use these sets to establish the test verdict and produce coverage metrics on the adaptation policy. To this end, we define two illegal behaviors that relate to the triggering of reconfigurations while other or none were expected.

Definition 14 (Wrong reconfiguration) A wrong reconfiguration occurs at a given step $\sigma(i) \overset{ope}{\rightarrow} \sigma(i+1)$ of a reconfiguration path σ if the actual reconfiguration operation that is executed does not belong to the set of eligible reconfigurations names:

$$\mathit{actual}_{\sigma(i)} \in \mathcal{R} \wedge \mathit{actual}_{\sigma(i)} \notin R_{N\,\mathit{elig}_{\sigma(i)}}$$

Notice that this non-conformance occurs only when a reconfiguration is actually performed ($\mathit{actual}_{\sigma(i)} \in \mathcal{R}$). Indeed, it is possible that the implementation chooses to deliberately ignore a potential reconfiguration. This option is not necessarily considered as an error, and can be caught by the coverage measures that we aim to produce additionally.

Definition 15 (Unexpected reconfiguration) An unexpected reconfiguration occurs if the system under adaptation performs a reconfiguration operation while no reconfiguration rule is eligible:

$$\mathit{actual}_{\sigma(i)} \in \mathcal{R} \wedge \mathit{elig}_{\sigma(i)} = \varnothing$$

This second non-conformance is a particular case of a wrong reconfiguration, in which $\mathit{elig} = \varnothing$.

The test verdict is established by determining if a wrong or an unexpected reconfiguration occurs. In this case, the test *fails*, otherwise it *passes*.

Definition 16 (Test verdict w.r.t. an adaptation policy) Let *tc* be a test case, and A an adaptation policy. The test verdict w.r.t. A is established by the *verdict* function defined by:

$$verdict(tc, A) = \begin{cases} fail \text{ if } \exists\, \sigma(i) \in exec(tc).\ actual_{\sigma(i)} \neq run \wedge actual_{\sigma(i)} \notin R_{N\,elig_{\sigma(i)}} \\ pass \text{ otherwise} \end{cases}$$

We aim to establish whether the implementation under adaptation complies with the specified adaptation policy. Therefore, we propose to detect divergences between the implementation and the adaptation policy, when wrong reconfigurations are triggered. In addition, we also propose to perform a coverage measure of the occurrences of reconfigurations to provide statistics on the rules of the adaptation policy.

5.3 Adaptation Policy Coverage Measure

In addition to a test verdict, we propose to address the question of the coverage of the different rules by a given test (or more generally, by a test suite). The goal of this measure is to provide a feedback on the "completeness" of the considered test(s), in order to know if the (parts of the) rules have been covered, and, if so, how frequently. In addition, this mechanism provides a means to detect errors in the design of the adaptation policy (e.g. if a rule is too restrictive that it is never triggered), or possible errors in their implementation, especially when it comes to comply with the fuzzy values that describe the rules' utilities.

In practice, this kind of coverage measure is meant to be performed only if the tests passed. We first define the notion of rule coverage, and we then define additional measures that can provide a useful feedback on the adaptation policy.

5.3.1 Coverage of a Rule

The aim of the rule coverage is to evaluate if the different rules of the adaptation policy have been activated by the considered test cases. To this end we define a function that counts the number of executions of a rule for a given reconfiguration path.

Definition 17 (Number of executions of a rule) Let σ be a reconfiguration path, the number of executions of a rule $r \in R_R$ on σ is given by:

$$\#actual^r(\sigma) = \sum\nolimits_i f_a^r(\sigma(i))$$

where $f_a^r(\sigma(i))$ is the characteristic function of predicate $actual_{\sigma(i)} \in dom(I_r)$ (which equals to 1 if the predicate holds, and 0 otherwise).

This information can be used to evaluate if a given rule is covered by a test case, or, more generally, by a test suite. However, if the rule is never covered, multiple causes can be identified. First, the test cases do not reach a configuration where the reconfiguration is applicable. In this case, the test suite has to be refined to try to cover the rule. Second, some parts of the rules might be incorrectly written and present a too restrictive (or even unreachable/invalid) trigger or guard. For such cases, we propose to measure the number of times these different parts of the rule are satisfied.

5.3.2 Eligibility of a Rule

First, we define the number of triggerings of a rule, which is, for a given adaptation rule, the number of configurations in which its triggering property became true.

Definition 18 (Number of triggerings of a rule) Let σ be a reconfiguration path, the number of triggerings of a rule $r \in \mathcal{R}$ on σ is given by:

$$\#trig^r(\sigma) = \sum_i f_t^r(\sigma(i))$$

where $f_t^r(\sigma(i))$ is the characteristic function of predicate

$$r \in trig_{\sigma(i)} \wedge r \notin trig_{\sigma(i-1)} \wedge actual_{\sigma(i-1)} \notin dom(I_r)$$

Second, for a given path σ, we define the number of eligibilities of a rule, which is, for a given rule, the number of configurations of σ in which the rule became eligible.

Definition 19 (Number of eligibilities of a rule) Let σ be a reconfiguration path, the number of eligibilities of a rule $r \in \mathcal{R}$ is given by:

$$\#elig^r(\sigma) = \sum_i f_e^r(\sigma(i))$$

where $f_e^r(\sigma(i))$ is the characteristic function of predicate

$$r \in elig_{\sigma(i)} \wedge r \notin elig_{\sigma(i-1)} \wedge actual_{\sigma(i-1)} \notin dom(I_r)$$

These measures, reported for each rule, help to evaluate which part of the rule has not been satisfied during the test cases execution. However, in some cases, the rule may have been eligible, but the implementation deliberately ignores it. For such cases, in addition, we compute the frequency of the daptation rules.

5.3.3 Frequency of a Rule

The frequency of a rule activation is measured as the number of times this rule was activated on the number of configurations in which this rule became eligible.

Definition 20 (Frequency of a rule) We define the frequency of a rule $r \in R_R$ for a given test case tc as follows:

$$freq^r(tc) = \frac{\#actual^r(exec(tc))}{\#elig^r(exec(tc))}$$

This measure is useful to evaluate whether a given rule is frequently activated or not. Once measured, the frequency can be compared against the fuzzy value associated with the rule in order to detect a potential inconsistency in the adaptation policy implementation. Notably, it allows detecting a *high*-utility rule that is less frequently applied than a *low*-utility rule.

We now present experiments that we have conducted to evaluate the relevance of the developed MBT approach.

6 Experimentation

This section reports on our experiments that aim to assess the contributions presented in this article. We start by presenting a case study, and then we describe the performed experiments.

6.1 Case Study: an Ad-hoc Network of Autonomous Vehicles

We consider the Vehicular Adhoc Network (VANet) case study which represents an example of Cyber-Physical Systems [20]. In this system, vehicles are either organized in platoons, namely groups of vehicles, or they are in solo mode. Each platoon has a leader which heads the convoy. A vehicle in solo mode can ask to join another solo vehicle to create a new platoon, or simply join an existing platoon. The vehicles' autonomy (e.g. depending on available energy supports) decreases over time. The vehicles can leave the platoon in order to exit the road, either because they reach their destination, or because they need to refill their energy supports. In addition, the leader of the platoon may change over time, either because it needs to leave the platoon or because another vehicle is a more appropriate candidate to head the convoy (e.g. because it has either a greater autonomy or a farther destination).

6.1.1 System under Test

The VANet system can be seen as a component-based system, in which vehicles are connected and linked together to constitute platoons, and connected to a main component, which represents the road, as depicted in Fig. 9. In this figure, there are two platoons (delimited by dotted line) and three solo vehicles. The leaders of the platoons are the vehicles that are directly connected to the road.

We have designed and developed a reference implementation of the VANet as a Java simulator that implements the main functions of the case study. Eventhough this system is a simplification of a real system, in which the distances between vehicles

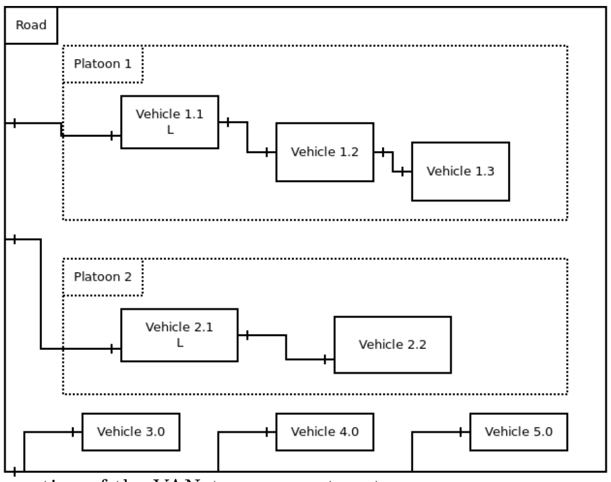

Fig. 9 A configuration of the VANet component system

have been abstracted, it still represents a challenging case study, notably due to the dynamic aspect of the system, in which the number of vehicles is not set and evolves with time. Thus, the state space of the system is infinite. To simulate a real time, we consider clock ticks that represent the time evolution. The considered entities are the road and the vehicles. Platoons are abstract entities that only exist to cluster vehicles. Platoons data model includes the number of vehicles that it contains, and a reference to the current leader. Vehicles are characterized by their battery level, their distance to their destination, and a state which indicates the status of the vehicle (solo, platooned or stopped to refill its battery).

In the implementation, the external events and reconfiguration operations are mapped to methods in specific objects. We consider the following external events that represent the control points of the system:

1. *CreateVehicle* adds a new vehicle on the road.
2. *RequestJoin* that represents a vehicle is joining a platoon.
3. *ForceQuitPlatoon* occurs when a vehicle is forced to leave the platoon.

These events are used in the usage model that we consider. A probability is associated with each of them.

The following reconfigurations can be performed by the system. In the context of a platoon:

1. *CreatePlatoon* adds a new platoon on the road.
2. *DeletePlatoon* deletes a platoon from the road.

In the context of a vehicle:

1. *ElectVehicle* promotes the vehicle to be the leader of the platoon.
2. *RevokeVehicle* replaces the leading vehicle with another vehicle of the platoon.
3. *JoinPlatoon* adds the vehicle to the platoon, in response to a *RequestJoin* that is successfully performed.
4. *QuitPlatoon* occurs when the vehicle leaves the platoon.

When executed, this system produces logs providing the necessary information to infer the successive configurations along with the different operations that are triggered. These logs can be analyzed to establish the test verdict and perform measures on the adaptation policy as described before.

6.1.2 FTPL Properties

For the experiments the following FTPL properties are considered. The first two properties are expressed in the context of a platoon:

$\varphi 1$: **after** *CreatePlatoon* **before** *DeletePlatoon* **always** $\mathit{Leader} \neq \text{null}$
specifies that when a platoon exists it must always have a leader vehicle.

$\varphi 2$: **after** *CreatePlatoon* **before** *DeletePlatoon* **always** $\mathit{VehicleNumber} > 2$
specifies that a platoon must always contain at least two vehicles.

The following three properties are expressed in the context of a vehicle:

$\varphi 3$: **always** $\mathit{Battery} > 0$ and $\mathit{Distance} > 0$
specifies that a vehicle must always have a strictly positive remaining distance and a battery level.

```
when (after ElectLeader) = TRUE
  if (battery < 20) = TRUE then
       utility of RevokeVehicle is high

when (battery < 5) = TRUE
   if (state = platooned) = TRUE then
        utility of QuitPlatoon is high
```

Fig. 10 Two reconfiguration rules of the VANet adaptation policy

$\varphi 4$: **after** *ElectVehicle* **before** *RevokeVehicle* **always** $Battery > 15$
specifies that when a vehicle is leader, it must always have a battery level strictly greater than 15%.

$\varphi 5$: **after** *JoinPlatoon* **before** *QuitPlatoon* **always** $state \neq stopped$
specifies that when a vehicle is inside a platoon, it must not refill its battery.

6.1.3 Adaptation Policy

The adaptation policies we have designed, describe the behavior of the vehicles for the system where the considered reconfigurations consist, for a vehicle, in creating or joining a platoon, replacing platoon's leader, or leaving the platoon.

To save space, we provide in Fig. 10 an excerpt of the VANet adaptation policy. The first rule specifies that after a vehicle has been elected as a leader, it is necessary to consider replacing it when it runs low on energy. The second rule specifies that when the vehicle's battery reaches a given threshold, the vehicle has to leave the platoon.

Our adaptation policies rules aim to guide the behavior of the VANet depending on the attributes of the vehicles. We consider that the higher the utility of a reconfiguration is, the higher the chances to trigger the corresponding reconfiguration operation is. For example, when two vehicles want to leave the platoon at the same moment, the reconfiguration with higher priority is chosen. Recall that the utility is determined by several criteria, like the battery level or the distance from the target destination.

An example of execution of this system under adaptation is depicted in Fig. 11. This figure displays one graphic per vehicle showing:

1. The remaining distance (in dashed plot) and battery level (in lined plot) (in %);
2. The length of the platoon in which the vehicle is. If the vehicle is in solo mode, its platoon length is one. Bold dots symbolize the fact that the vehicle is leader of the platoon;
3. The stations that appear as big dots on the X-axis;
4. The external events that are displayed with vertical lines. The names of the events are provided.

The vehicles behaviors shown in the graphics are guided by adaptation policies. Let consider vehicle A in Fig. 11. When its battery level drops below 20% at tick 80, this vehicle may leave the platoon to refill its battery at the next station. As vehicle A was leader before leaving the platoon, a new leader has to be elected. We can see that vehicle C becomes leader. A few steps after, vehicle C runs low on energy, and also leaves the platoon. As there is no vehicle available to become the new leader, the platoon is deleted.

Sometimes, external events cause a reconfiguration, as for the *RequestJoin* event at tick 200, after which vehicles B and C are put together to create a new platoon. This

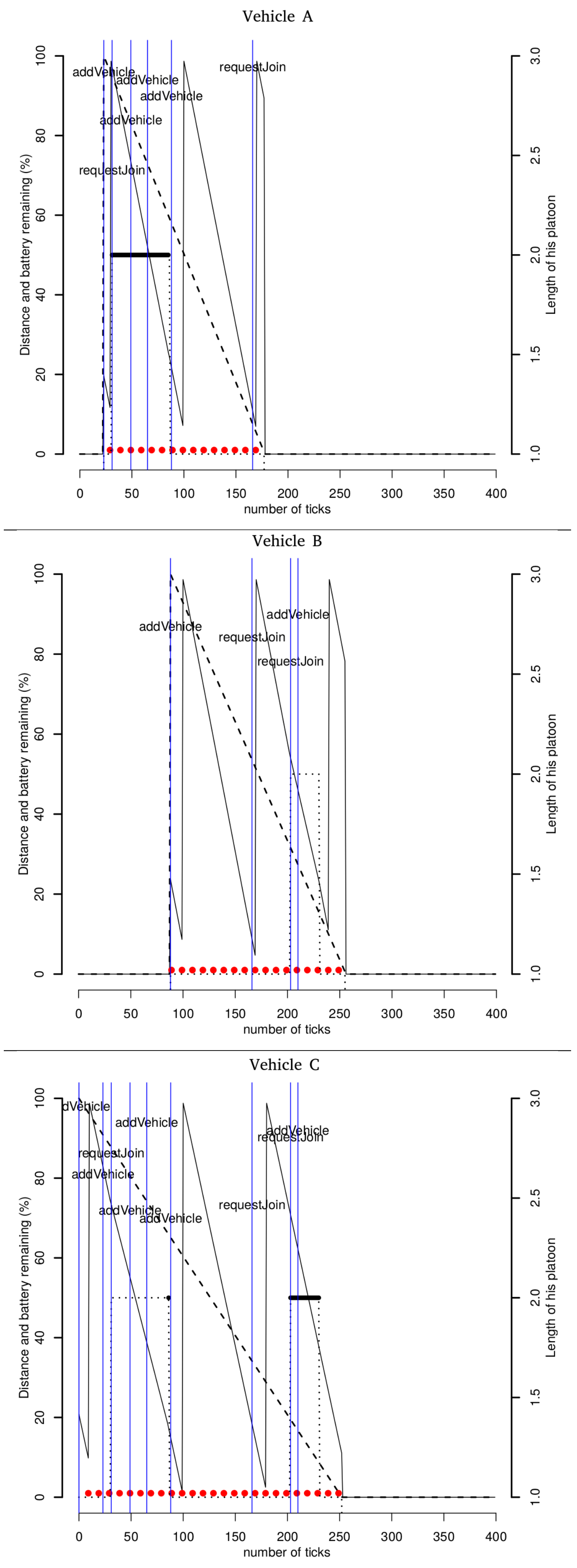

Fig. 11 An extract of the execution of the case study

event corresponds, in the real world, to a situation in which a vehicle is getting close to another vehicle and request a merging.

6.1.4 Usage Model

The probabilistic model of the environment is implemented in the ModelJUnit framework [25], a library that makes it possible to design and exploit Finite State Machines (FSM) in Java. Contrarily to other tools working on explicitly given automata, ModelJUnit offers the possibility to encode the FSM states using variables and abstract them by defining a state identification function. Thus, the environment model of the VANet, which highly depends on dynamic creation of vehicle instances, does not have to be explicitly defined. In addition, ModelJUnit can be used to generate test cases offline, or online by being directly connected to the system under test. For our experiments, we use this latter possibility to generate the test traces, so as to be able to decide if a vehicle is solo or platooned. Indeed, as the only external event consists, for a solo vehicle, to request a join on a platoon, it is mandatory to know if the vehicle is already platooned or not before sending such a request. As there is no control on this event, we need to examine the current state of the execution to know if the request is relevant or not.

For the needs of the experiments, ModelJUnit has been extended to support probabilities on the transitions, and a new algorithm for performing a random walk described in Sect. 4 has been implemented. Each generated test case is recorded so as to be able to replay it afterwards. The objective of the experiments is to validate the detection capabilities offered by our approach on the adaptation policies.

6.2 Experiments with the Case Study

We describe here the experimental procedure. We start by the research questions, and explain our approach, before discussing the results and the threats to validity.

6.2.1 Research Questions

Our experiment aims to address the following research questions.

RQ1. To what extent are property-based coverage criteria relevant for detecting errors?
This question relates to the capability of our approach to characterize test cases that will detect errors. This includes the ability to both generate relevant test sequences and observe the current configuration to detect violations.

RQ2. To what extent are rule coverage criteria complementary to property-based criteria?
This question relates to the second coverage criteria, and aims to evaluate the usefulness of considering the rules of the adaptation policy.

RQ3. To what extend are probabilistic models appropriate to generate test cases that satisfy these criteria?
This question relates to the technique we propose to generate test cases. It is important to assess the capability of our models to be able to generate test cases that satisfy the coverage criteria that we defined.

6.2.2 Experimental Procedure

In order to address the three research questions, we set up an experimental process that we applied to the case study of the VANet.

First of all, we designed a test model of the VANet environment describing the occurrences of interactions that may occur between vehicles. Then we used our test generator to produce random test suites, and we report, for different length of test cases, and average size of test suites that it is necessary to generate in order to satisfy the considered coverage criteria. We expect that, first, the test generation process will be able to generate test suites that satisfy the coverage criteria, and, second, that it does not require extensive test suites to reach this coverage. This addresses RQ3, related to the capability of a probabilistic random approach to fulfill the coverage criteria that we proposed.

Second, we designed a set of mutants in order to evaluate the capability of the test cases to detect faults, by killing the mutants. Mutants are a variant of the original system, in which a fault has been introduced. The fault must be realistic (Competent Programmer Hypothesis), and represent a fault or an implementation choice that a programmer might do during the development of the system. The mutants are produced manually, so as to avoid the issue designing blindly equivalent mutants. The mutations we consider rely on a simple fault model, which consists in changing, in the code of the application, the conditions that are used to decide whether or not a given reconfiguration should occur as some event or changes in the internal state happens. The considered mutation will thus either strengthen or weaken the conditions under which the implementation decides that a given reconfiguration is performed.

The test cases are run on the mutants and we use the two verdict establishment definitions (based on properties and based on the adaptation policy) to decide if each test succeeds or fails. We report, when a mutant is killed, which one of them was used to detect the fault. This aims to address RQ1 and RQ2, related to the relevance of these coverage criteria in terms of fault detection capabilities. We expect that some mutants will be killed by each test verdict establishment that we have proposed.

Third and finally, as the design of the usage model is a complex task that is error prone, especially when it comes to the definition of the probabilities in the system, we study the impact of different choices of probabilities in this process. Thus, we consider two additional usage models, from which we repeat the above-mentioned process: one model in which the probability of quiescence is increased, and the other in which the probability of quiescence is decreased, w.r.t. the original model we designed. We then compare how test suite size and test cases length evolve, and we evaluate if, in these cases, the detection capabilities are impacted or not, by running these new tests on the set of mutants we designed previously. This also addresses RQ3.

6.2.3 Results

Random test generation. As we are dealing with a random approach, the results that we present are an average of 15 generations of one test case of 4000 steps. The results are shown in Table 1 which displays the average, minimum and maximum number of steps that were necessary to achieve the targeted coverage (i.e., 100% coverage of property or rules).

We can see that for the considered length, a 100% coverage of rules and properties can be achieved in a reasonable number of steps and the test generation is not time-

	#steps (avg)	#steps (min)	#steps (max)
100% property coverage	1486	307	3475
100% rules coverage	1269	322	2991

Table 1 Necessary number of steps to reach 100% coverage

consuming (each test generation takes less than 30 seconds). For this first experiment, we considered a single long test. We also considered computing several smaller tests. Table 2 shows the results of these test generations. We present, for each test suite, its characteristics in terms of number of tests (column #tests) and test lengths (column |test|). We then report on the average coverage of properties and rules.

The results show that the tests need a given number of steps to be able to trigger all the reconfiguration rules, and to reach configurations on which the temporal properties can be evaluated.

Mutant detection. We designed a set of mutants that correspond to: (*i*) a strengthening of the conditions used in the code to trigger the reconfigurations (mutants M1, M4) which will lead to reconfigurations that are not triggered when they should, (*ii*) a weakening of the reconfigurations conditions (mutants M12, M13) which will lead to the unexpected triggering of reconfigurations, (*iii*) a different implementation in the ways events are handled (mutants M2, M3), (*iv*) a change in the way reconfigurations are prioritized (mutants M5, M7, M10, M11), and (*v*) a functional error in the implementation of the specification (mutant M6, M8, M9).

We tested these mutants on a single test case of 4000 steps with the defined FTPL properties. The same test was executed on all the mutants. The execution of the test stops as soon as a violation is detected. We report in Table 3 the results of the execution and indicate for each detected mutant which property was violated and at which step of the test. Mutants M1-M9 are detected by the verdict given in Def. 13 which observes a violation of the properties we defined. Mutants M12 and M13 are detected using the test verdict given in Def. 16 which detects unexpected reconfigurations (at steps 398 and 3026 of the test case respectively). Finally, mutants M10 and M11 are not killed by any of these two verdict assignment techniques, but they can be spotted by the frequency analysis presented given in Def. 20.

Indeed, the modification introduced in the third category of mutants changed the behavior of the system w.r.t. the initial implementation but this did not lead to an observable defect. However, we observed in these cases that the frequency of rules triggering was modified w.r.t. the original implementation, and we were somehow "suspicious" w.r.t. the utility of the rule described in the adaptation policy. Namely, in

#test	|test|	% Prop.	% Rules
5	2000	100%	100%
10	1000	100%	100%
10	400	100%	100%
20	200	100%	87.5%
10	200	98.5%	87.5%
10	100	96.7%	62.5%
500	100	100%	87.5%

Table 2 Test generation capabilities for different test suite sizes

	M1	M2	M3	M4	M5	M6	M7	M8	M9
$\varphi 1$									#108
$\varphi 2$						#10			
$\varphi 3$	#268	#114	#23				#429		
$\varphi 4$				#73	#53				
$\varphi 5$								#338	

Table 3 Mutants detection

Implementation	Property cov. (avg)	% rules cov. (avg)
Reference	100%	100%
Variant 1	99.34%	92.5%
Variant 2	100%	86.1%
Variant 3	100%	100%

Table 4 Test generation results for the different variants

M10 mutant a low-utility rule was systematically triggered, and in M11 mutant, a medium-utility rule was not triggered.

We consider this mutant detection as a warning more than an error, and if the programmer has a doubt in the results he can verify his program. This illustrates that the additional measures on the rules coverage that we proposed can be useful to indicate a potential error in the system.

Coverage of the adaptation policy. The task of designing an appropriate model is tedious and error-prone, especially when it comes to defining the probabilities of the different events. In order to evaluate the risks of defining wrong probabilities on the usage model, we designed 3 variants of the usage model, which correspond to a modification for the external events of the probabilities of their occurrences.

In variant 1, we decreased the probabilities of *ForceQuitPlatoon*, *CreateVehicle* and *JoinPlatoon*, in consequence we increased the *Tick* probability. In variant 2, we increased the probabilities of the external events *ForceQuitPlatoon*, *CreateVehicle* and *JoinPlatoon*, and we decreased the *Tick* event probability. Finally, in Variant 3, we increased only the probability of events *CreateVehicle* and *JoinPlatoon* and we decreased the probability of *ForceQuitPlatoon*. The variants illustrate different choices that can be made, notably when defining the balance between the quiescence and the occurrence of events.

The results that we obtained are provided in Table 4. We ran the test generation process of a single test of 4000 steps. We measured the coverage we obtained w.r.t. the properties and the rules, and we compared them to the reference usage model, used for the rest of the experiments.

We can see that the increase of probability occurrence of some critical operations, such as *ForceQuitPlatoon*, may lead to decreasing the occurrence of some rules. In this case, the system forces a vehicle to leave the platoon more frequently, which prevents some of the configurations used in other rules to be reached.
It is thus mandatory, when designing the probabilities of the external events, to carefully choose the probabilities associated with events that deactivate components.

6.2.4 Threats to Validity

One of the threats to validity relates to the definition of the mutants, as one may argue that the mutants were designed so as to be detected by our approach. However, they were designed in a systematic manner (by weakening or strengthening conditions in code decisions), so this threat is very limited.

A second threat to validity would be the definition of the usage model. To mitigate this, we experimented with several models that present different probabilities on their transitions. Even if we did design models that would differ in the transitions themselves, we assume that the transitions, describing the sequencing of events, are less error-prone than the definition of the probabilities. Even if a different model would maybe prevent the coverage criteria to be fulfilled, it could be seen as an incorrect model that would not correctly describe how the environment in which the system is executed behaves.

A third threat to validity relates to the use of a simulation for the experiment which may be considered shallow compared to a real case testing approach. As mentioned above, we stay confident that our approach is representative enough of adaptive systems. Furthermore, testing directly on real systems is not only time-consuming, but may be impossible for very dangerous scenarii [16] as it highly depends on external variable factors. Following [31], we believe that first series of validation can be performed under simulations, and they are sufficient for illustrating the ability for finding errors when using this kind of approaches.

A last threat to validity relates to the fact that, in this experiment, only one case study has been used, with several mutants. Although we consider our description of VANet (network) is representative of adaptive component systems, it would obviously be better if we could have a diversity of subjects to consolidate the results of experiments. However, let us emphasize that adaptive component systems are very complex and need time to be designed. This is why random test generation and mutation testing fit in well with our needs and are part of the described experimental procedure. Notice that similar results have been obtained on the running example of the Cycab vehicle location controller used in this article. The results for this simple example are not described in this section.

7 Related Work

Self-adaptation is a very active research field with challenges and open questions in various domains. The roadmap [9] emphasises an important challenge consisting in bridging the gap between the design and the implementation of self-adaptive systems. Validation by testing is needed for obtaining incremental assessments for more confidence in self-adaptation [32,40]. The recent work [26] provides a literature review in which different testing approaches are classified in a decision tree according to several criteria such as technique type, usage of mutation analysis, and evaluation dimension. Our approach relates to the black-box testing technique type, with mutation analysis in effectiveness and scalability dimensions.

Fuzz testing, or fuzzing [35], is a software testing technique that aims at discovering weaknesses by inputting massive amounts of data (often random and/or invalid). Behavioural fuzzing sends (invalid) sequences of valid data. These sequences can either be generated from a model, like in [29], or by re-engineering the result of a previous run of the system, namely its log files. By using specifics of the reconfiguration model in [22] to generate the data to be injected, the work in [38] allows the tester to focus on specific parts of the sequence of configurations that enables adaptation policies to be tested. Robustness testing [18] describes a class of approaches that evaluate the degree to which a system or a component can function correctly in the presence of invalid inputs or in stressful environmental conditions. Our approach complements these works, as we mainly focus on establishing a test verdict and measuring a model coverage rather than generating test cases.

Differently from general run-time testing techniques, where the adaptive software systems are considered together with the simulations of their environment, see e.g. [14, 15], our approach is based on a usage model for representing the behavior of the environment rather than the system itself [19]. Moreover, like design-time testing approaches, see [4] for some examples, it focuses on specific behaviour depending on external events to perform reconfiguration operations guided by adaptation policies with temporal properties. In [27], the authors use resource prediction to manage evolution of utility on self-adaptive systems. In our approach, in addition to its main function, the utility is used to help detect potential inconsistencies in the adaptation rules. The evaluation framework in [37] aims at evaluating quality-driven self-adaptive software systems. That framework is based on a set of adaptation properties mapped to software quality attributes. Thus, corresponding software quality metrics can be used to assess adaptation properties. Our approach is different, as the adaptation policies with temporal patterns are not directly linked to quality metrics. However, our MBT approach provides means for calculating such metrics, via the proposed coverage measure inspired from temporal logics properties coverage described in [7].

Our purpose is close to that in [13] where the authors aim to deal with self-organization by testing separately self-adaptation mechanisms. Like in that work, in our approach is it possible to examine particular situations, namely where reconfiguration operations are performed to comply with adaptation policies depending on temporal properties and external events. In [12], the authors present an online test case generation procedure with a test case selection strategy. Unlike [13, 12], our work is focused on testing systems under adaptation policies allowing us to detect specification defects and implementation errors.

8 Conclusion and Future Work

This paper has presented a model-based testing approach that aims to establish whether a self-adaptive component system faithfully implements an adaptation policy that is defined independently. Testing new adaptation policies is complicated and time consuming, especially for large systems that would require tailored settings to test specific policies. Our approach relies on a usage model, that is used to generate test cases as sequences of external events driving the systems execution. Two verdicts have been proposed; the firts one is based on the use of temporal properties written in FTPL, and the second one is based on detecting if a reconfiguration is legitimate or not. This approach has been experimented on a case study of the platoons of vehicles, which the paper reports on. We have also experimented our proposals on smaller case studies, namely Inria's CyCab autonomous vehicle [1], which is simpler than the VANet example, and for which we obtained similar results.

Unlike for other model-based testing approaches, usage models have to be manually designed, which can be seen as a strong requirement to apply this technique. However, in the case of usage models, they are not used to fully describe the behavior of the system under test, but rather correspond to additional artifacts, that require a minimal effort of modelling to be employed. This test generation approach can be declined in an offline process, when the test cases are first generated to be executed on the system under test afterwards. Nevertheless, it is also possible to consider that, for each generated step from the automaton, this step is immediately executed on the system, and the different verification and measures are done on-the-fly. The declination to an online testing approach is straightforward. Finally, notice that, even though the experimentation has been made on a dedicated implementation, it could also be performed on an integrated framework such as Fractal [5] or BIP [2] which provide similar observation capabilities. Such a development is part of future work.

Future work directions also include the definition of different techniques to improve the coverage of the adaptation rules. One option is to design dedicated test generation algorithms that target the triggers and guards of the adaptation policy rules, so as to generate test cases that aim to provoke the firing of the reconfigurations in a more active manner. Another future work is to adapt the approach to a wider class of self-adaptive Cyber-Physical Systems, which would require to take into account the real-time aspects of such systems. In particular, we plan to evaluate our approach on the realistic simulator VIPS [16] which would make it possible to confirm the results on the same case study of the platoons of vehicles, including potential external perturbations. In this context, the use of an online testing process could be done.

Finally, during the experiments, we noticed that the measures that we proposed at the end of Sec. 5 could be useful w.r.t. the validation of the correct implementation of the adaptation policy. We plan to investigate this aspect further, with the objective to be able to detect inconsistencies between the fuzzy values provided in the adaptation policy rules and the actual frequences of the reconfigurations that are observed.

References

1. G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. Le cycab de l'INRIA Rhône-Alpes. Rapport technique RT-0229, INRIA, april 1999.
2. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of the Fourth IEEE Int. Conf. on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
3. B. Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.
4. R. V. Binder, B. Legeard, and A. Kramer. Model-based testing: Where does it stand? *ACM Queue*, 13(1):40–48, 2014.
5. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.
6. F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Expression qualitative de politiques d'adaptation pour Fractal. In Y. Aït Ameur, editor, *2ème Conf. sur les Architectures Logicielles (CAL 2008), 3-7 Mars 2008, Montréal, Québec, Canada*, volume RNTI-L-2 of *Revue des Nouvelles Technologies de l'Information*, page 119. Cépaduès-Éditions, 2008.
7. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking*. *Formal Methods in System Design*, 28(3):189–212, 2006.
8. F. Dadeau, K. Cabrera Castillos, and J. Julliand. Coverage criteria for model-based testing using property patterns. In A.K. Petrenko and H. Schlingloff, editors, *MBT 2014, 9th Workshop on Model-Based Testing, join to ETAPS 2014*, volume 141 of *EPTCS*, pages 29–43, Grenoble, France, apr 2014.

9. R. De Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.
10. J. Dormoy and O. Kouchnarenko. Event-based adaptation policies for Fractal components. In *AICCSA 2010, ACS/IEEE Int. Conf. on Computer Systems and Applications*, pages 1–8, Hammamet, Tunisia, may 2010.
11. J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In L. Barbosa and M. Lumpe, editors, *FACS*, volume 6921 of *LNCS*, pages 200–217. Springer Berlin Heidelberg, 2012.
12. B. Eberhardinger, H. Seebach, D. Klumpp, and W. Reif. Test Case Selection Strategy for Self-Organization Mechanisms. In Mario Winter, Andreas Spillner, and Andrej Pietschker, editors, *Test, Analyse und Verifikation von Software – gestern, heute, morgen.* dpunkt Verlag, 2017.
13. B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. Towards testing self-organizing, adaptive systems. In M. G. Merayo and E. M. de Oca, editors, *Testing Software and Systems*, pages 180–185, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
14. A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Asp. Comput.*, 24(2):163–186, 2012.
15. A. Filieri, G. Tamburrelli, and C. Ghezzi. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Software Eng.*, 42(1):75–99, 2016.
16. M. Guériau, B. Dafflon, and F. Gechter. VIPS: A simulator for platoon system evaluation. *Simulation Modelling Practice and Theory*, 77:157–176, 2017.
17. M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, 2008.
18. C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. D. Wagner, C. Le Goues, and P. Koopman. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg*, pages 276–285, 2018.
19. E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont. Environment-model based testing of control systems: Case studies. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 636–650, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
20. D. Jia, K. Lu, J. Wang, X. Zhang, and X. Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE Communications Surveys Tutorials*, 18(1):263–284, 2016.
21. M. Kim, I. Lee, J. Shin, O. Sokolsky, et al. Monitoring, checking, and steering of real-time systems. *ENTCS*, 70(4):95–111, 2002.
22. O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS, 10th Int. Symp. on Formal Aspects of Component Software*, volume 8348 of *LNCS*, pages 234–253. Springer, 2014.
23. O. Kouchnarenko and J.-F. Weber. Decentralised evaluation of temporal patterns over component-based systems at runtime. In I. Lanese and E. Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *LNCS*, pages 108 – 126, Bertinoro, Italy, sep 2015. Springer.
24. O. Kouchnarenko and J.-F. Weber. Practical analysis framework for component systems with dynamic reconfigurations. In M. Butler, S. Conchon, and F. Zaïdi, editors, *ICFEM'15, 17th Int. Conf. on Formal Engineering Methods*, volume 9407, pages 287–303. Springer, 2015.
25. M. Utting. How to design extended finite state machine test models in Java. In *Model-Based Testing for Embedded Systems*, Series on Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 147–170. CRC Press, 2011.
26. M. Mayeda. *Evaluating Software Testing Techniques: ASystematic Mapping Study.* PhD thesis, University of Denver, Colorado, 2019.
27. V. Poladian, D. Garlan, M. Shaw, M. Satyanarayanan, B. R. Schmerl, and J. Pedro Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007*, pages 214–223, 2007.

28. R. Rajkumar, I. Lee, L. Sha, and J. A. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 731–736, 2010.
29. M. Schneider, J. Grossmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker. Behavioral fuzzing operators for UML sequence diagrams. In *Proceedings of the 7th International Conference on System Analysis and Modeling: Theory and Practice*, SAM'12, pages 88–104, Berlin, Heidelberg, 2013. Springer-Verlag.
30. A. Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.
31. T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand. Can robot navigation bugs be found in simulation? an exploratory study. In *2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017*, pages 150–159, 2017.
32. G. Steinbauer and F. Wotawa. Model-based reasoning for self-adaptive systems - theory and practice. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, volume 7740 of *Lecture Notes in Computer Science*, pages 187–213. Springer, 2013.
33. C. Szyperski. Component software: beyond object-oriented programming. 1998. *Harlow, England: Addison-Wesley*, 1995.
34. S. Taha, J. Julliand, F. Dadeau, K. Cabrera Castillos, and B. Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Aspects of Computing*, 27(4):641–664, dec 2015.
35. A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.
36. M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier, 2006. 550 pages, ISBN 0-12-372501-1.
37. N.M. Villegas, H.A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 80–89, 2011.
38. J.-F. Weber. Tool support for fuzz testing of component-based system adaptation policies. In *13th International Conference on Formal Aspects of Component Software*, volume 10231 of *LNCS*, pages 231 – 237, 2016.
39. J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, Oct 1994.
40. F. Wotawa. Testing self-adaptive systems using fault injection and combinatorial testing. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Companion, Vienna, Austria, August 1-3, 2016*, pages 305–310. IEEE, 2016.