Online Testing of Dynamic Reconfigurations w.r.t. Adaptation Policies

Frédéric Dadeau, Jean-Philippe Gros, and Olga Kouchnarenko

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

Abstract. Self-adaptation of complex systems is a very active domain of research with numerous application domains. Component systems are designed as sets of components that may reconfigure themselves according to adaptation policies, which describe needs for reconfiguration. In this context, an adaptation policy is designed as a set of rules that indicate, for a given set of configurations, which reconfiguration operations can be triggered, with fuzzy values representing their utility. The adaptation policy has to be faithfully implemented by the system, especially w.r.t. the utility occurring in the rules, which are generally specified for optimizing some extra-functional properties (e.g. minimizing resource consumption).
In order to validate adaptive systems' behaviour, this paper presents a model-based testing approach, which aims to generate large test suites in order to measure the occurrences of reconfigurations and compare them to their utility values specified in the adaptation rules. This process is based on a usage model of the system used to stimulate the system and provoke reconfigurations. As the system may reconfigure dynamically, this online test generator observes the system responses and evolution in order to decide the next appropriate test step to perform. As a result, the relative frequencies of the reconfigurations can be measured in order to determine whether the adaptation policy is faithfully implemented. To illustrate the approach the paper reports on experiments on the case study of platoons of autonomous vehicles.
Keywords: Component system, Adaptation policy, Online testing, Usage model

1 Introduction

Context and motivations Adaptive systems can be driven by adaptation policies which describe when reconfiguration operations can be triggered. In this paper, a policy is formally defined by a set of rules that indicate, for a given set of states of the system, also called configurations, which reconfiguration operations can be triggered. Thus, adaptation rules are used to specify the correct behavior of the system, where for a reconfiguration each rule specifies a scope, a guard, and a utility. The guard and the scope may integrate temporal patterns [9], and the scope limits the reconfiguration application (e.g., to a set of configurations, or

depending on events' occurrence). To indicate the utility of the reconfiguration, i.e., the interest for the system to activate a reconfiguration operation, a fuzzy value (e.g., high, medium, low) is used, as put up in fuzzy logic. For example, it is possible to indicate that after entering a tunnel, if the available power of an autonomous vehicle is limited, its GPS component should be removed.

As a consequence, different implementations of a given adaptation policy are possible, provided that the resulting executions of the system do not violate any temporal properties. Verification and validation (V&V) of such systems is a difficult task, due to these multiple, but correct, implementations of a system under adaptation policies with temporal properties. These additional artifacts– temporal properties and adaptation policies–provide means to ensure that the executions of the system are correct. For example, supervising temporal properties at runtime allows ensuring that no violation of the properties is detected [19]. However, the question of establishing whether an adaptation policy is faithfully implemented (its validation), has been very little targeted. This is mainly due to the non-prescriptive nature of adaptation policies.

In general model-based approaches to validate systems' implementations rely on their behavioral models, used as a reference for testing activities. For a system under not mandatory adaptation policies, looking for a model would imply to make implementation choices w.r.t. the adaptation policies, and would require the system to make the same choices to respect the policies, which is too restrictive. This paper describes an approach to validate implementations of the adaptation policies using only themselves as a reference. In addition, as the adaptation policies describe the reconfiguration needs depending on the environment in which the system is executed, we suggest to use a model of the environment instead of a system's behavior model. It allows taking into account occurrences of the events together with the subsequent reconfigurations. In a previous work [6], we have addressed the issue of validating the system implementation w.r.t. the adaptation policy, by checking that the reconfigurations that are triggered during the execution correspond to those authorized in the adaptation policy.

Contributions In this work, we propose to go further by addressing the issue of validating that the system execution respects the utilities of the reconfiguration rules of the adaptation policy. To achieve this, we propose a statistical analysis of the frequencies of the reconfigurations. In order to have a significant number of executions that perform reconfigurations, we perform an automatic generation of large test suites. They are obtained by randomly exploring a model of the system environment, describing how the adaptive system can be stimulated, namely by external events. As reconfigurations may occur without control from the tester, the test generation process is performed online: the test generator observes the reconfigurations in response to events in the system environment, in order to compute the next test step to be executed.

In model-based testing, an approach using coverage criteria ensures that the test cases provide a good representation of the possibilities of the system by covering them. Following this approach, we rely on the previous work [6] which aimed to define test coverage criteria focusing on two artifacts, namely (a) the

adaptation policy, and (*b*) a set of temporal properties that the system has to fulfill. They can be used to illustrate various execution scenarii. These artifacts are used to ensure the variety of the execution traces that will be produced, thus providing a confidence in the results obtained by the frequency analysis of reconfigurations.

This paper makes the following contributions: (*i*) an *online* test generation process that involves a usage model of the system's environment, and aims to produce large sets of test cases that fulfill the coverage criteria defined in [6]; (*ii*) a measure of the reconfiguration operations occurrences based on the frequency analysis, aiming to validate adaptation policies; (*iii*) the experimental validation of this process on a case study of vehicles and their platoons on the road.

Outline The paper is organized as follows. Section 2 presents the necessary background on adaptive component systems and the adaptation policies for them. Then, Section 3 introduces the test generation process, which relies on a usage model of the system environment. The evaluation of the utility fuzzy values based on a frequency analysis is summarized in Sect. 4. Section 5 reports on the experimentation with platoons of vehicles, performed to evaluate this paper proposals. Finally, Section 6 describes related works and concludes.

2 Example and Prerequisites

2.1 Adaptive Systems on the VANet Component-based Example

Example 1. We start by describing an example of a component-based system for the Vehicle Ad-hoc Network (VANet) case study, which is displayed in Fig. 1. The VANet system is composed of vehicles which are either in solo mode, or organized in some platoons. Each platoon is led by a leader vehicle. Any vehicle in solo mode can ask to join a platoon or decide to create a new platoon with another vehicle in solo mode. Each vehicle in a platoon can ask to quit it either because of the destination reached, or to refill its battery. The platoon may change its leader because of another vehicle having more autonomy or a further destination. Some external events happen in the system environment. For example, a new vehicle can arrive on the road, or a driver may decide to quit the platoon on his way to a new destination.

Let us now introduce the notion of component-based systems in relation with this example. Components are entities that can be assembled to design complex systems. The component-based systems under consideration are hierarchical, with two types of components. Primitive, or basic, components provide data or services, while composite components contain other components. Required and provided interfaces are interaction points between components. A component realizes a service and provides it via a provided interface, whereas a required interface is needed for the component to run. Composite components may delegate their interfaces to inner components. We refer to [9,18] for the definition of components, interfaces, variables, bindings, etc. for their consistent

assembly. A state of a component system, also called 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. A reconfiguration can be considered as a transition from one configuration to another one. Components are independent and can be implemented independently as such. A component can be instantiated several times, e.g. Platoon and Vehicle components.

Example 2. Figure 1 displays a VANet instance corresponding to the situation on the right-hand side picture. The compound Road component encapsulates Vehicles, which can be platooned (1.X, 2.X) or be solo (3, 4, 5). Vehicles are connected together via interfaces allowing them to share information.

Adaptation policies for these systems are seen as artifacts that describe needs for system's adaptation, which are, however, not mandatory. For example, a platoon may change its leader by a relay between the leader and another vehicle of the platoon either when the leader has not enough autonomy to stay leader, or when another vehicle has a farther destination and has more autonomy than the leader.

2.2 Models

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.

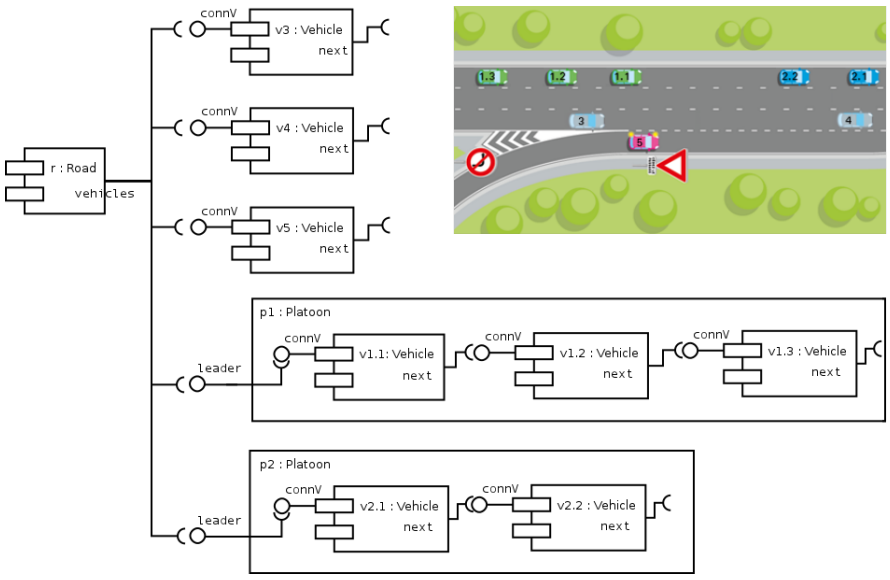

Fig. 1. Component architecture (a) vs. the considered VANet system state (b)

In particular, configuration propositions are used to define consistent configurations. For example, they specify global invariance properties (on components, interfaces, bindings, etc.), that component-based architectures must satisfy. An *interpretation* function $l : \mathcal{C} \rightarrow CP$ gives the largest conjunction of $cp \in CP$ evaluated to true on $c \in \mathcal{C}$, that characterizes the current configuration in the most precise way.

Let us consider a set $\mathcal{R}$ of reconfiguration operations, which 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, etc. Let $\mathcal{R}_{run} = \mathcal{R} \cup \Theta \cup \{run\}$ be a set of actions, where $\mathcal{R}$ is a finite set of reconfiguration operations, Θ is the set of operations triggered by external events ($\mathcal{R} \cap \Theta = \emptyset$), and run is the name of a generic action used to represent all the running operations of the component-based system, different from reconfigurations. We assume, as in [17], that external events are captured by the system' sensors and processed immediately by triggering internal methods from Θ.

Definition 1 (Labelled transition system). *A labelled transition system is the tuple $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ where $\mathcal{C}$ is a set of configurations, $\mathcal{C}^0 \subseteq \mathcal{C}$ is a set of initial configurations, $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ is the reconfiguration relation, and $l : \mathcal{C} \rightarrow CP$ is a total labelling function.*

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

Definition 2 (Path). *Given S, a (reconfiguration) path σ of S is a sequence of steps $c_0 \stackrel{ope_0}{\rightarrow} c_1, c_1 \stackrel{ope_1}{\rightarrow} c_2, \ldots$ such that $\forall i \geq 0.\ (c_i, ope_i, c_{i+1}) \in \rightarrow$. Given σ, its trace is a word $tr(\sigma) = ope_0.ope_1 \ldots ope_i \ldots$ of operation names occurred in σ.*

We write c_i or $\sigma(i)$ to denote the starting i-th configuration of σ's i-th step. The notation σ_i denotes the suffix path starting at $\sigma(i)$, and σ_i^j the path segment in between $\sigma(i)$ and $\sigma(j)$. Let Σ_S denote the set of paths of S, and Σ^f ($\subseteq \Sigma$) the set of finite paths. A configuration c' is reachable from c when there is a path $\sigma = c_0 \stackrel{ope_0}{\rightarrow} c_1, \ldots c_{n-1} \stackrel{ope_{n-1}}{\rightarrow} c_n$ in Σ^f s.t. $c = c_0$ and $c' = c_n$. An execution is a path σ in Σ s.t. $\sigma(0) \in \mathcal{C}^0$.

2.3 Adaptation Policies

Adaptation policies are composed of rules that indicate, for a given set of configurations, which reconfiguration operations can be triggered, with a utility level associated. Reconfiguration operations in adaptation rules are guarded by sequences of events that may either exploit a configuration proposition, or involve temporal logic properties. This section recalls the FTPL[1] temporal patterns [9], before integrating them into adaptation policy rules.

[1] FTPL comes from the fusion between TPL (Temporal Pattern Language) and 'F' for First order logic consistency constraints over components.

FTPL is a pattern language used to specify properties over execution traces, on which the reconfiguration operations are expected. Let $Prop_{FTPL}$ denote the set of the FTPL formulae obeying the FTPL grammar in Fig. 2. In addition to configuration propositions (cp) in CP from Sect. 2, FTPL contains temporal properties ($temp$) together with events (external events as well as events of reconfiguration operations). Following Dwyer's seminal work on patterns, trace properties ($trace$) are embedded into temporal properties.

The FTPL semantics described in [18] is basic for configuration propositions (e.g., like in PLTL[2]). For other temporal patterns, it makes use of scopes (before, after, until) while considering trace properties and occurrence of events. Like in [17], let suppose that the external events occur instantaneously and can be seen as invocations of methods from Θ performed by system' 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 1. a guard cp_{ext}, which is a first-order logic formula over the parameters specified in the invocation of the method corresponding to ext, and 2. an assertion $happens_\sigma$, valued in $\{\top, \bot\}$.

<*FTPL*>	::=	<*temp*> | <*events*> | *cp*
<*temp*>	::=	**after** <*events*> <*temp*>
	|	**before** <*events*> <*trace*>
	|	<*trace*> **until** <*events*> | <*trace*>
<*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

Following the semantics of the Event Calculus [20,23], $happens_\sigma(cp_{ext}, i, j) = \top$ if there is at least one occurrence of ext between i-th and j-th states of σ, s.t. $cp_{ext} = \top$; otherwise, $happens_\sigma(cp_{ext}, i) = \bot$.

Definition 3 (FTPL semantics). *Let $\sigma \in \Sigma$. The FTPL semantics is defined on $\Sigma \times Prop_{FTPL} \to \mathbb{B}_2$ 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	$i > 0 \wedge \sigma(i-1) \neq \sigma(i) \wedge \sigma(i-1) \stackrel{ope}{\rightarrow} \sigma(i) \in \rightarrow$
$\sigma(i) \models ope$ **exceptional**	if	$i > 0 \wedge \sigma(i-1) = \sigma(i) \wedge \sigma(i-1) \stackrel{ope}{\rightarrow} \sigma(i) \in \rightarrow$
$\sigma(i) \models ope$ **terminates**	if	$\sigma(i) \models ope$ **normal** $\vee\ \sigma(i) \models ope$ **exceptional**
$\sigma(i) \models ext$	if	$eval_\sigma(cp_{ext}, i) = \top$
$\sigma(i) \models event, events$	if	$\sigma(i) \models event \vee \sigma(i) \models events$
For the trace properties:		
$\sigma \models$ **always** cp	if	$\forall i.(i \geqslant 0 \ \Rightarrow \sigma(i) \models cp)$
$\sigma \models$ **eventually** cp	if	$\exists i.(i \geqslant 0 \wedge \sigma(i) \models cp)$
$\sigma \models trace_1 \wedge trace_2$	if	$\sigma \models trace_1 \wedge \sigma \models trace_2$
$\sigma \models trace_1 \vee trace_2$	if	$\sigma \models trace_1 \vee \sigma \models trace_2$
For the temporal properties:		
$\sigma \models$ **after** *event temp*	if	$\forall i.(i \geqslant 0 \wedge \sigma(i) \models event \Rightarrow \sigma_i \models temp)$
$\sigma \models$ **before** *event trace*	if	$\forall i.(i > 0 \wedge \sigma(i) \models event \Rightarrow \sigma_0^{i-1} \models trace)$
$\sigma \models trace$ **until** *event*	if	$\exists i.(i > 0 \wedge \sigma(i) \models event \wedge \sigma_0^{i-1} \models trace)$

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

[2] PLTL, or Propositional Linear Temporal Logic

For the detailed formal semantics, the reader is refered to [18]. The short FTPL presentation above aims to illustrate temporal patterns integrated into adaptation policies, which make them more expressive than policies in [4] using only propositional properties and invariants over configurations.

Example 3. Let us consider an FTPL property for the VANet system:

$\varphi 1$: **after** *Join normal* **always** $Battery \geq 5 \wedge Dist > 2$ **until** *Quit normal*
This property specifies that after a vehicle joins a platoon, its autonomy and remaining distance are greater than respectively 5% and 2km, until it quits the platoon. It applies to each vehicle.

We now define adaptation policies by adapting [4] to integrate temporal patterns, when simplifying the notations from [18].

Definition 4 (Adaptation policy). *Let S be an LTS with its reconfigurations $\mathcal{R} \cup \Theta$, CP be a set of configuration propositions labelling its states, and Ftype a finite set of fuzzy types. An adaptation policy is defined as $A = \langle R_N, R_R \rangle$, where:*

- *$R_N \subseteq \mathcal{R} \cup \Theta$ is a finite (non-empty) set of reconfiguration operations,*
- *R_R is a finite set of adaptation rules $(b, g, ir) \in Prop_{FTPL} \times CP \times I$, where $ir \in I \subseteq R_N \times Ft$ is a pair that associates a utility fuzzy value f from $Ft \in Ftype$ with a reconfiguration operation $ope \in R_N$.*

Let us note that I is a relation, and thus the designer can connect different fuzzy values to one reconfiguration: sometimes, the utility is `high`, sometimes it is `low`. Moreover, some operations (in $\mathcal{R} \cup \Theta \setminus R_N$) may be not concerned by designed policies, and may have no utility value associated.
Each adaptation rule (in R_R) is built using several keywords as follows: **when** b **if** g **then utility of** *ope* **is** f. The b property after the `when` keyword determines the temporal scope of the reconfiguration operation to occur. Within the defined temporal scope, the guard g on system's configurations is then described after the `if` keyword. It defines the system's configurations where a reconfiguration can be triggered. Afterwards, in order to bind the utility to the reconfiguration operation, $ope \in R_N$ is associated with its utility $f \in Ft$ by using the `then utility of` and `is` keywords. The utility is specified by a fuzzy value (e.g., `high, medium, low`).

when after *Join normal* until *Quit normal* **and** *VehicleId.battery* < 33
if *state* = *leader* **then utility of** *PassRelay* **is** *high*

when after *Join normal* until *Quit normal* **and** *VehicleId.battery* $>$ *Leader.battery*
if *state* = *platooned* **then utility of** *GetRelay* **is** *medium*

Example 4. Let us consider 2 adaptation rules involving the *PassRelay* and *GetRelay* reconfigurations. Intuitively, the above rules apply to all vehicles and are used to determine when it is possible to have a relay between the leader and another vehicle of the platoon. In the first case, the *PassRelay* reconfiguration

triggers when the leader has not enough autonomy to stay leader. In the second case, the *GetRelay* reconfiguration triggers when the autonomy of a vehicle is greater than the autonomy of the leader. Notice that the addition/removal of a component is not strictly requested, but rather suggested with a utility value (e.g. with $Ft = \{$ high, medium, low$\}$). Hence, there is no guarantee that the system eventually executes the considered reconfiguration operation.

We now define how the adaptation policies affect the behaviour of S. For an LTS S and a finite set AP of adaptation policies, let $S \triangleleft AP$ denote S under policies in AP.

Definition 5 (LTS under Adaptation Policies). *The restriction of S by adaptation policies in AP is defined as $S \triangleleft AP = \langle \mathcal{C}_{\triangleleft AP}, \mathcal{C}^0_{\triangleleft AP}, \mathcal{R}_{run}, \rightarrow, l \rangle$, where $\mathcal{C}_{\triangleleft AP}$ is the least set s.t. if $c \in \mathcal{C}$ and $A \in AP$ then $c_{\triangleleft A} \in \mathcal{C}_{\triangleleft AP}$, $\mathcal{R}_{run} \cap (\cup_{A \in AP} R_N) \neq \emptyset$, $l : \mathcal{C}_{\triangleleft AP} \rightarrow CP$ is a total labelling function, and for every $ope \in \mathcal{R}_{run}$, the transition relation $\rightarrow \in \mathcal{C}_{\triangleleft AP} \times \mathcal{R}_{run} \times \mathcal{C}_{\triangleleft AP}$ is the least set of triples $(c_{\triangleleft A}, ope, c'_{\triangleleft A})$ satisfying the following rules:*

$$[ACT1] \quad \frac{c \xrightarrow{ope} c'}{c_{\triangleleft A} \xrightarrow{ope} c'_{\triangleleft A}} \quad (ope \in \bigcup_{A \in AP} R_N) \wedge b \wedge g$$

$$[ACT2] \quad \frac{c \xrightarrow{ope} c'}{c_{\triangleleft A} \xrightarrow{ope} c'_{\triangleleft A}} \quad ope \notin \bigcup_{A \in AP} R_N$$

This definition means that the transitions of S under AP result from performing either reconfiguration operations obeying adaptation policies (Rule *[ACT1]*), or reconfigurations which are not involved in adaptation policies (Rule *[ACT2]*)). Regarding evaluation of Rule *[ACT1]* side condition, namely its temporal b part, it is possible to proceed in a decentralised manner, using progressive semantics, like in [2,19].

Adaptation policies have to be faithfully implemented by the system, especially wrt. the utility values occurring in the rules, which are generally specified for optimizing some extra-functional properties (e.g. minimizing resource consumption). To this end, various relations can be used, e.g., a refinement relation [21], simulation relations [24], a satisfaction relation [15], etc. However, deciding those relations to compare implementations wrt. specifications under adaptation policies may be a hard problem, which becomes trickier when considering events from the system environment. This problem is in general undecidable for infinite-state systems. Instead, in the next section, we propose a usage-model-based methodology to validate implementations of adaptive systems with adaptation policies.

3 Online Testing with a Usage Model

In this section we first motivate the use of an online testing approach before describing the usage models and the test generation algorithm needed to compute the test cases.

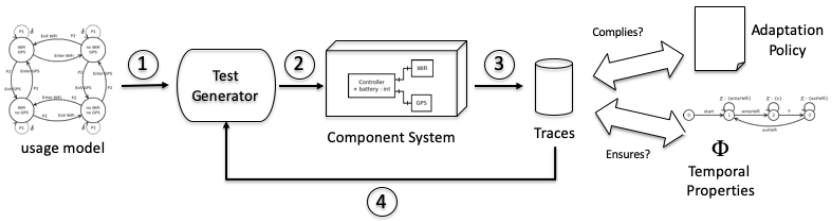

Fig. 3. Online test generation process

3.1 Online Test Generation Process

Adaptive systems react to external events, according to adaptation policies that provide guidelines to detect when to execute reconfigurations on the system. There are numerious different but correct implementations for a system under given adaptation policies. Let us call a frequency of a reconfiguration the ratio of the number of times a reconfiguration is applied to the number of times that it could have been. When considering traces of executions and relative frequencies of reconfiguration operations, if a reconfiguration operation with a high utility has a lower relative frequency than an operation with a low utility, it means that either the system implementation incorrectly takes the utility value into account, or the utility value specified in the adaptation policy needs to be modified. As different implementations of a system under given adaptation policies are possible, describing a unique reference model of these implementations could be complicated: it would require to make many design choices on the behavior of the system, and then would force the implementations to comply to these choices.

In order to avoid the description of LTSs of implementations and their validation wrt. the specification–an LTS under adaptation policies, whose application is not mandatory–this paper suggests to consider a usage model of the system under test [30]. Thus, in our case, it is not a model of the adaptive system under policies itself, but a model of its environment [29] focused on the events that occur, and to which the system may react according to adaptation policies. Such models are usually smaller than models describing the whole system, and simpler to design manually by a validation engineer.

As described before, the system may evolve in response to the external events from Θ (also seen as controllable events, e.g., sent by the tester for validation purpose), its state changes as well. It may trigger internal (uncontrollable) events that are logged to be observed from the outside [3]. This influences the generation of the next possible external events that will be sent to the system, as a reconfiguration may prevent some external events from being effective on the system. Thus, it is important for the test generation process to avoid generating tests that perform irrelevant actions. It should also take into account the internal events, which will only be observable, and adjust the next test generation

[3] These two sets are disjoint, as set in Sect. 2 for reconfiguration operations and reflected in the FTPL grammar.

step accordingly. As a consequence, the usage model also considers the internal events that correspond to reconfigurations that may occur in the system.

To be able to deal with such situations, we propose an online testing approach, in which: (*a*) each step of the test is directly executed on the system under test (SUT for short), (*b*) the evolution of the SUT is detected by the test generation algorithm, and taken into account for generating the next test step. The proposed process is depicted in Fig. 3. The usage models are explored by a dedicated algorithm, which aims to compute the next test steps. They are given as an input to the test generator (1), which is connected to the SUT.

In an online manner, the algorithm randomly chooses a component and computes, by browsing its usage model, a test step (2), namely an event that will be sent to the SUT. The SUT may possibly react to the event, and perform a reconfiguration, which will be logged onto the execution trace (3). To compute the next step, the test generator also considers the updated execution trace, which is used to update the current state of the generator (4). In parallel, the execution traces are used to detect the possible violation of temporal properties integrated into the policies (as described in [19]), or the triggering of unauthorized reconfiguration operations, as described in [6]. These aspects, studied before, are not in the focus of this paper.

3.2 Test Cases and Reconfiguration Paths

We define test cases as sequences of controllable events that are sent to the system under test, at a given rate. In the case of adaptive systems that are hybrid systems, in which discrete and continuous time are mixed, a discretized approach can be adopted, like in [10] for handling events. In this case, the events used to stimulate the system are sent at a given rate, symbolized by clock ticks whose duration is parameterized. In addition, we rely on the notion of delay, denoted by δ, which consists in performing no action on the SUT.

Example 5 (Test case for the VANet example). The following test case is produced for exercizing the VANet system:

$$\delta; \delta; V1.join; \delta; ...; \delta; V2.join; \delta; ...V2.forceQuit; ...$$

It displays frequent occurrences of the delay δ representing a period of time during which the system state evolves without any request from the environment; concretely the vehicles' battery resource decreases. In this sequence, *join* is a request from a vehicle (external to a platoon) to joint it, and $forceQuit$ is a request from a driver to quit a platoon.

When a test case is executed on the SUT, it produces a reconfiguration trace. It can then be analyzed to decide if the system complies with the various specifications, namely, the adaptation policies and the expected temporal properties.

Example 6 (A reconfiguration path wrt. external events). Let us consider two sequences illustrated in Fig. 4: the sequence of external events (namely, the test

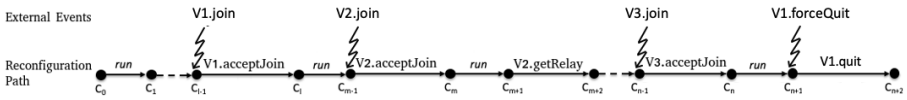

Fig. 4. A reconfiguration path according to external events

case) made of *join* and *quit* requests at the top-level line, and the reconfiguration path (namely the execution trace) of the system generated in response to the external events, at the bottom-level line. Sequences of external events are sampled with a clock tick on the same frequency, while execution paths report on the reception of events and triggering of reconfigurations in response to these events.

In the VANet system, several external events can occur. The *join* event happens when vehicles are close enough to merge, the system can either accept to merge and trigger the *acceptJoin* reconfiguration, or decide to refuse the request and trigger the *refuseJoin* reconfiguration. When a driver decides to quit the platoon (an external event to the autonomous vehicle), the *forceQuit* event is emitted and the system reacts to it with a (*quit*) reconfiguration. The system can also react to internal events. For example, the leader may change with the (*getRelay*) reconfiguration.

3.3 Usage Models for Online Testing

This section describes the artifact that will be exploited to generate such test cases: usage models of the components.

Introduced in [29] usage models mainly aim to specify the various events that may occur in system's environment. These controllable events can be sent to the system under test to which it reacts. In usage models, it is also possible to use delay representing an absence of external events for a given time period.

The most common way to design usage models is to rely on probabilistic automata. In this paper it is suggested to associate them with components. Intuitively, to each state of such an automaton corresponds a set of events that can be triggered on it, i.e., which the component is able to respond to. They will be used by the test generator to produce a test case.

As indicated in Sect. 3.1, reconfiguration operations, that are not controllable, can be observed on the execution traces. They are related to the observable events which also appear in the usage model. They will be used to detect and capture an evolution of the component current state, which is necessary to send relevant events to the system.

Definition 6 (Usage model probabilistic automaton). *For each component C, its usage model is defined as a deterministic probabilistic automaton $\mathcal{A}_C = \langle Q, q_0, E_\delta \cup O, F, P \rangle$, where Q is a set of states, $q_0 \in Q$ is the initial state, E_δ is the set of controllable external events*[4] *together with δ for the delay (the absence of external events), O is the set of uncontrollable events, made of*

[4] giving rise to reconfigurations from Θ, Sect. 2.2

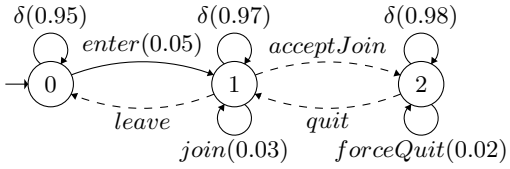

Fig. 5. Usage model of the VANet vehicle

the possible reconfiguration operations observed on the system[5]*, F is a transition relation $F \in Q \times (E_\delta \cup O) \times Q$, and P is the probability*[6] *of a transition $P : Q \times E_\delta \to [0; 1]$ such that $\forall q \in Q \Rightarrow \Sigma_{e \in E_\delta} P(q, e) = 1$.*

Each automaton is associated with a component, which can be either a high-level virtual component (such as the road in our example), or a low-level component (such as a vehicle).

Example 7 (Usage models of the VANet vehicles). In the VANet example, each vehicle responds to three external events: *enter*, denoting that a vehicle enters the road; *join*, which denotes that a vehicle requests to join another vehicle or an existing platoon; and *forceQuit*, which indicates that the vehicle exits the platoon due to its driver's intervention.

In addition, there are three not controllable but observable events relative to vehicles: *leave*, which denotes that the vehicle leaves the road; *acceptJoin*, which represents the reconfiguration of a platoon to accept the vehicle's request; and *quit*, which represents the reconfiguration operation for the vehicle to exit its platoon. The usage model for this example is displayed in Fig. 5. Let us assume that these events only occur with a user-defined probability (a number in parentheses). In this figure, the δ-labelled transitions represent a delay of one time unit, and the other transitions with labels represent either the controllable events *enter*, *join* and *forceQuit*, or the observable events *leave*, *acceptJoin* and *quit*, denoting reconfigurations in the system. The latter are represented with the dashed lines.

Based on usage models of individual components, a composition of usage models of components can be built by using e.g., a component encapsulation and refinement, a composition of (extended) interface automata, or hierarchical input/output automata, etc. To avoid this (memory- and time-consuming) construction, it is possible to proceed in a decentralised manner, like in [2,19] for a run-time property evaluation. In our approach, the component-based model is used while proceeding in a decentralized manner.

3.4 Test Generation with Usage Models

The test generation process relies on the usage models of the components in the SUT, summarized in Fig. 3. As the components may evolve in an uncontrollable

[5] reconfigurations from $\mathcal{R}$, Sect. 2.2

[6] We assume that if no outgoing transition of the current state is labeled by an event, its probability is 0.

Algorithm 1 Online test generation algorithm

```
1: for all A_C do
2:    state(A_C) ← q_0(A_C)
3: end for
4: i ← 0
5: while i < n do
6:    C ← selectComponent()
7:    state(A_C) ← update(A_C, trace)
8:    e ← pick(E_δ, state(A_C))
9:    if e ≠ δ then
10:      C.send(e)
11:      state(A_C) ← update(A_C, [e])
12:      i ← i + 1
13:   end if
14:   await()
15: end while
```

way, the online approach also relies on the trace of the system's execution, which is expected to log onto the occurred reconfigurations.

A test case is defined as a sequence of events obtained by traversing transitions of the probabilistic automata of the components usage models. Let $A_\delta(C)$ be the usage model associated with component C. For a given component system, a test case, based on a set of usage models, is a finite sequence of events $C_0^j.e_0^j; C_1^k.e_1^k; ...C_n^l.e_n^l$ (of length $n+1$) in which, at i-th step ($0 \leq i \leq n$), the triggered event e^j is associated with component C^j and has a probability $P_{C^j}(q^j, e^j) > 0$.

In order to compute a set of test cases, a.k.a. test suite, a Markov random walk [27] is performed on the probabilistic usage models of components. However, the exploration of components' usage models is performed in a random manner. The test generation algorithm, represented in Algorithm 1, repeats the following steps until the maximal length of test cases is reached. First, one of the components is randomly selected (line 6). The current state of the automaton is updated w.r.t. the execution trace of the system (line 7). This step consists in taking into account different reconfiguration operations (i.e., the observable events) that may have occurred since the last selection of the component. A transition is then randomly selected among the outgoing transitions of the current state w.r.t. its associated probability (line 8). If the transition is not a delay (line 9), the event is sent to the system under test via the considered component (line 10-12), and the current state is updated. At the end (line 14), the test generation process awaits (depending on the sampling rate of events) before computing the next test step.

This algorithm can be employed to generate test suites of arbitrary size and test cases of arbitrary length. It provides means to evaluate whether the fuzzy values associated with reconfiguration operations in the adaptation policy are faithfully implemented, or not. To achieve that, the analysis of the relative fre-

quencies of the reconfigurations over traces of the executions is performed, which is now described.

4 Adaptation Policies Compliance

Our goal is to evaluate whether an implementation of an adaptation policy complies with the utility of the reconfigurations specified by using fuzzy values. In a previous work [6], we have presented the use of the adaptation policy rules as a coverage metrics to evaluate the relevance of a test suite. We now present a new and complementary use of such a coverage measure, in order to assess the compliance of the implementation w.r.t. the triggering of reconfiguration operations when applying adaptation policy rules.

4.1 Coverage, Eligibility and Frequency of a Rule

Coverage of a rule We first define a function that counts the number of executions of a rule on a given reconfiguration path.

Definition 7 (Number of executions of a rule). *Let σ be a reconfiguration path, and $actual_{\sigma(i)}$ the actual reconfiguration operation occurring at $\sigma(i)$. 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 $actual_{\sigma(i)} \in dom(I)$[7] *in which I is the relation linking reconfiguration operations with fuzzy values.*

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 (property b or guard g in Def. 4). For such cases, we propose to count the number of times these different parts of the rule are satisfied.

Eligibility of a rule We now define the number of triggerings of a rule as the number of configurations in which its triggering property became true.

Definition 8 (Number of triggerings of a rule). *Let σ be a reconfiguration path, and let $trig_{\sigma(i)}$ be the set of reconfiguration operations that can be triggered at $\sigma(i)$, i.e., whose b-triggers are true at this state. The number of triggerings of a rule $r \in R_R$ 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*

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

[7] which equals to 1 if the predicate holds, and 0 otherwise

As a reconfiguration may occur at any configuration within the scope of the rule, while such a b property holds, the number of configurations in which the property holds does not provide the information on the real situation with reconfigurations. This is why for a given path σ we define the number of eligibilities of a rule, which is the number of states of σ where the rule became eligible. The rules whose b-triggers and g-guards are true at $\sigma(i)$ are called eligible in this state.

Definition 9 (Number of eligibilities of a rule). *Let σ be a reconfiguration path, and $elig_{\sigma(i)}$ the set of eligible rules that could have been applied at $\sigma(i)$. The number of eligibilities of a rule $r \in R_R$ 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*

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

These measures, reported for each rule, help determine 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, we compute the frequency of the adaptation rules as an additional measure.

Frequency of a rule Given a test suite, the frequency of a rule activation is measured as the number of activations on the number of configurations in which this rule became eligible.

Definition 10 (Frequency of a rule). *For a given test suite TS composed of test cases tc, the frequency of a rule $r \in R_R$ is defined as follows:*

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

Notice that if the rule is not eligible in any states, i.e., if its eligibility number is equal to 0, its frequency is also set to 0.

In order to obtain meaningful frequencies, we rely on the test generation process described in Sect. 3. Thanks to this algorithm it is possible to produce large test suites, while covering the rules of adaptation policies. 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 specified in the rule, in order to detect a potential inconsistency in the adaptation policy implementation. Notably, starting from a particular set of initial configurations and based on an expected system behavior, the frequency analysis allows detecting a rule with a *high*-utility reconfiguration that is less frequently applied than a rule with a *low*-utility reconfiguration.

4.2 Compliance with the Adaptation Policy

Intuitively, the compliance of the implementation w.r.t. the adaptation policy can be explained thanks to utility values used in adaptation rules.

Given an adaptation rule $r = (b, g, ir)$, by Def. 4 the pair $ir = (ope, f)$ associates in r utility f with reconfiguration *ope*. Assuming that the fuzzy values are ordered. Then the validation engineer may expect each rule with utility N to be applied at a greater frequency than every rule with utility $N-1$. Formally, for a given adaptation policy A, and a given test suite TS, we say that the adaptation policy is faithfully implemented if

$$\forall r, r' \in A, f^r > f^{r'} \Rightarrow freq^r(TS) > freq^{r'}(TS)$$

Let $\sqsubseteq$ denote the well-known sub-word embedding relation. Given the set $E_\delta \cup O$ of controllable and uncontrollable events, we use $\sqsubseteq$ modulo δ, written $\sqsubseteq_\delta$, after removing δ from considered words.

Proposition 1. *If the adaptation policy A is faithfully implemented then $\forall tc \in TS$, $\exists \sigma \in \Sigma_{S_\triangleleft A}$ s.t. $tc \sqsubseteq_\delta tr(\sigma)$. Moreover, $\forall i \geq 0$, configuration $\sigma(i)$ is reachable by performing reconfiguration operations given by set $elig_\sigma$ of adaptation rules eligible on σ.*

Proof. By construction. Every test case $tc \in TS$ is generated by applying Algorithm 1 and using Def. 6. By Def. 5 for $S_\triangleleft A$, each state $\sigma(i)$ on path σ corresponding to test case tc is obtained by applying either rule *[ACT1]* in the case of a reconfiguration by one of the adaptation rules, or rule *[ACT2]* in the case of an observable reconfiguration.

Starting from initial configurations, in the case when rule *[ACT1]* is applied at step i, the target state results from the reconfiguration actually performed ($actual_{\sigma(i)}$), which has been choosen accordingly to its utility among the reconfigurations from set $elig_{\sigma(i)}$ of rules, which are eligible in this configuration. This utility-based reconfiguration choice during the system execution increases the reconfiguration frequency, and thus it makes the adaptation policy faithfully implemented. The reachability of configurations on σ is ensured by construction, thanks to Def. 5. Finally, the sub-word $tc \sqsubseteq_\delta tr(\sigma)$ relationship between the test case and the execution trace relies on Defs. 6, 1 and 5.

5 Experimentation

In order to validate the proposed approach, this section describes the experiments, where inconsistencies in the adaptation policies implementation wrt. specified utilities have been detected. We start this section with the research questions (RQ) before describing the experimentation. We then report on the results and discuss the threats to validity.

[RQ1.] To what extent our approach is efficient to generate large amount of pertinent test cases? The goal is to evaluate the capability of the test generator to produce large test suites with a good coverage of the adaptation policy rules. Notably, we want to ensure that the test cases are able to reach system states in which reconfigurations are eligible.

[RQ2.] To what extent the frequency analysis makes it possible to assess the fuzzy values? In order to be able to measure frequencies with accuracy, the goal is to verify that a significant number of reconfigurations are performed.

[RQ3.] To what extent suspicious results can be detected? Our overall objective is to evaluate whether the process is able to detect inconsistencies in the reconfigurations that are performed w.r.t. the fuzzy values described in the adaptation policy.

Experimental procedure To address these research questions, the following experiment has been designed. Several simulators of the VANet have been implemented in Java, which differ in their strategy to perform the reconfigurations. These different implementations are inspired by typical implementation styles that we have found in the literature. For example, in [9] the authors translate a fuzzy value for reconfiguration operation with a ranking, the utility of applying operations is guaranteed by a threshold value. The rules of the adaptation policies implemented in Tangram4Fractal [4] are considered in the declaration order, meaning that the first applicable rule that is declared in the adaptation policy is selected. In [26] the authors classify the reconfigurations according to defined priorities: if two operations with same priority are eligible, the first one is chosen.

The designed experiment for the VANet case study has 9 reconfiguration operations, among them 3 (getRelay, passRelay and quit) are triggered by 8 rules of the adaptation policy, named R1-R8. Informally, R1 rule specifies when the leader vehicle can pass the relay, R6 rule specifies when a vehicle can replace the current leader. Other rules specify the different cases when a vehicle may quit its current platoon, namely by running out of energy, or reaching its destination. Rules R1-R4 have a high utility, rules R5-R6 are with a medium utility, and rules R7-R8 display a low utility.

In the considered setup, it is supposed that vehicles can be dynamically added onto the road, and they are removed once they reach their destination. During the running of the experiments, the number of vehicle has oscillated between 100 and 250. In sum, in addition to vehicles in solo mode, several platoons have been constituted (from 4 to 25), with up to 8 vehicles in each of them. This mechanism has ensured the renewal of vehicles on the road and increased the chances to reach configurations that were likely to trigger possible reconfigurations. Notice that the generation of relevant initial configurations that minimizes the number of components is not in the focus of this study.

Our test generation process described in Algorithm 1 has been used to produce and to execute test cases on these implementations. Based on the execution traces, the rules' frequencies have been computed, according to the formulas provided in Sect. 4. Then they have been compared to the fuzzy values specified in the adaptation policy rules. In our experimentation, the analysis has been performed on test cases composed of 100.000 steps. To ensure that the test suite activates each reconfiguration rule at least once, we rely on the coverage criteria defined in [6].

Measuring the frequencies at each step of the execution allows us to draw graphs which show the evolution of the frequencies over the execution steps. For each reconfiguration these graphs can be used to compare its frequency w.r.t. the utility described in the adaptation rule. At the end of the execution/experiment, it is possible to compare the frequencies altogether so as to check their actual

triggering. As shown in Fig. 6, the frequencies stabilize when the number of steps increases, showing that the number of test cases that we generate makes it possible to perform a reliable frequency comparison (RQ1).

Using plots allows a visual analysis, which makes it possible to detect suspicious behaviors, for example a potential instability in one of the frequencies (RQ3).

In order to evaluate the capability of the approach to detect potential issues in the implementation of an adaptation policy, several implementations (E1-E6) have been developed, which differ in the way a reconfiguration is selected. The first implementation E1 selects reconfigurations according to a priority level. In addition, if an eligible reconfiguration has not been executed, its priority is increased for the next step. The results of executions of E1 can be found on the left-hand side of Fig. 6 and in Table 1. When examining the graph, one can notice that the plot of reconfiguration R5 is under the plots of reconfigurations R7 and R8. These plots show an inconsistency as R5 is of medium priority and should have a higher frequency than low priority reconfigurations R7 and R8. We then modified E1 to create implementation E2 by increasing the priority level of reconfiguration R5 whose results can be found on the right-hand side of Fig. 6 and in Table 1. The plot for R5 is now above those of R7 and R8, complying with specified priorities. Thus, this example shows that the proposed approach helps in detecting an inconsistent implementation choice and then in validating the adjustment made (RQ2).

To go further in the experiments, we have simulated other implementation choices with the priority level adjusted as in E2. Their results are shown in Table 1. In implementation E3, we make use of a fairness assumption: among two *eligible* reconfigurations with the same priority, the one that is not selected will be chosen the next time this scenario happens. In simulation with E4, the reconfigurations are chosen on the base of their priority level. Implementation E5 selects a low-utility rule in 20% of the cases. Finally, implementation E6 selects the first reconfiguration of the adaptation policy that can be triggered.

The results of simulations with E2 and E3 show that the implementation choice made matches the specified fuzzy utility values. Implementation E4 shows acceptable results but reconfiguration R4 occurs too often compared to other reconfigurations with the same utility. Implementation E5 is not consistent, as the plot of reconfiguration R5 of medium priority is below the plots of recon-

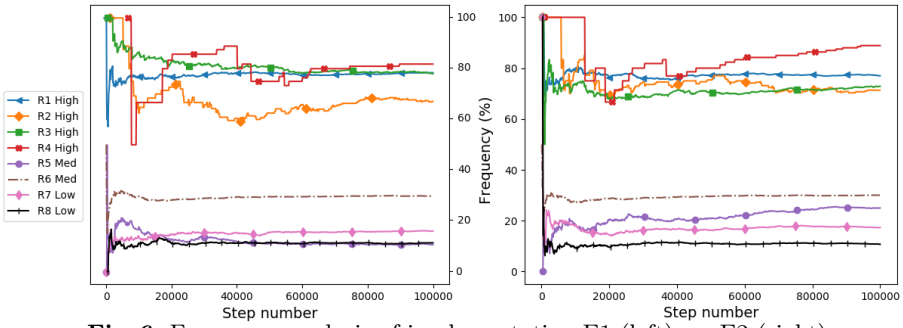

Fig. 6. Frequency analysis of implementation E1 (left) vs. E2 (right)

	R1(%)	R2(%)	R3(%)	R4(%)	R5(%)	R6(%)	R7(%)	R8(%)
	high	high	high	high	med.	med.	low	low
E1	78.4	67.2	78	82.8	10.6	29.7	16	11.5
E2	77	71.3	72.9	87.9	25.5	30	17.3	10.9
E3	68.7	64.7	70.8	76	23.1	31.4	14.9	9.4
E4	79.2	72.3	74.1	92.9	25	30.2	17.2	11.4
E5	64.4	66	56.6	68.2	17	29.7	19.1	24.4
E6	22.9	14.9	0.7	0	4.2	26.2	10.4	23.3

Table 1. Average results for 100.000-step executions

figurations R7 and R8 of low priority. Finally, the simulation with E6 shows several inconsistencies in the frequencies. These experiments show the interest of the proposed approach allowing the user to validate implementations under adaptation policies as well as to identify suspicious behaviors. In this case, a subsequent analysis permits to design a system implementation that complies with the adaptation policy (RQ2).

Threats to validity A first threat to validity relates to the frequency analysis, which may rely on too few occurrences of rules triggering to be able to perform an accurate ordering of the rules actual utilities. In relation with this, notice that during the experiments each reconfiguration has been triggered from several dozen of times to thousands of times on 100.000 step test cases. Thus, it has provided a significant number of eligibility for each rule, which gives relevance to the frequencies that are computed. A second threat to validity is that the experimentation has been performed on only one complex system example, though with several implementations. Extending the experimentation to other case studies is part of the future works. Finally, one more threat to validity is the fact that we have developed all the implementations which may introduce a bias. However, as mentioned before, the implementation choices that we made are inspired from real-world strategies already described in the literature.

6 Related Work and Conclusion

Related work The present paper continues the study of complex systems self-adaptation.

The special issues [8,7] indicate that adaptive and self-adaptive systems are challenging in terms of insurance, at several levels, of such adaptation, automation, and dependencies between rules. They propose a feedback control that measures the behavior of the system and changes it if necessary. We also proceed by system instrumenting. However, instead of direct change of the behavior of the system we chose to inform the validation engineer or the user of a potential error because: 1) there is no direct control of the system when performing black-box testing; and 2) we consider that the error may come from the development phase rather than from the environment itself. In [13] the online testing approach has been discussed; the authors advocate using online detection and diagnosis to recover to a correct system state after a failure. In [5] the authors summarize the

use of models to address the assurance of self-adaptive systems. They identify techniques that rely on evolutionary algorithms to automatically generate test cases. Other approaches [11] make use of a MAPE-T loop to monitor the applicability and utility of test cases during the execution. Our approach is also based on monitoring principles, however we do not analyse the tests themselves. Our overall goal is the validation of implementations of the systems under adaptation policies, so we produce them directly, in an online manner, based on a model of the environment and the systems execution.

Among interaction models for distributed systems, the interaction language in [22] is equipped with operational semantics based on step-by-step execution. Unlike [22] permitting trace validation by reading events one by one in the offline mode, our testing approach with components usage models allows processing events in the online mode. The framework in [1] is used offline as well, to manage component-based system' states. For this the authors use policies with a subset of temporal constraints over states. Differently from our approach, the policies are applied, allowing to build the state space and thus to plan system' reconfigurations aiming to optimize some of its extra-functional requirements.

In the component-based systems area, online testing is used in [12] to detect violation of proprieties in order to rollback the reconfiguration to a normal state. Several articles relate to built-in tests for component systems, with a focus on the architecture of component systems. In [3], an online model-based testing approach is used, where the system under test (SUT) is stimulated by a Markov Decision Process (MDP). The states of the model and behavior of the system are linked in order to generate actions with the usage model that will lead to a targeted behavior. Unlike the previously cited works, our contributions are based on retrieving the information from system's logs to exploit usage models to generate relevant tests for the SUT.

Adaptation rules and adaptation policies are useful to reduce or even resolve non-determinism when multiple actions can be performed by the adaptive system. In the field of dynamic delta modeling for adaptive component systems, in [14] sets of rules with priorities are used as product lines. In [26,28] adaptation policies contain rules with priority values, whereas in [16] utility functions are used to define objectives of the system. On the one hand, our utility notion is close to them as it permits a priority management, in order to select the rule with best utility wrt. aimed objectives. On the other hand, those works do not validate that the sets of rules are faithfully implemented. In [25], the authors improve the reactivity of self-adaptive systems with predictive algorithms. Our approach is more general as it validates utility of reconfiguration rules for large diversity of systems with adaption policies.

Conclusion In this paper, we have presented an approach and a method to automatically generate test cases for validating adaptation policies of component-based systems. This approach aims to produce large test suites, from probabilistic models of the components' environment, so as to be able to evaluate metrics during the system's execution and then to compare occurrence of reconfiguration operations w.r.t. their formal specification in adaptation policies. The developed

experiments have shown that the approach makes it possible to detect implementations that would not properly respect the utility of the reconfigurations. Notice that this approach, applied to component systems in this paper, could be easily adapted to other formal frameworks using rules and policies.

Based on the same approach, one of the future work directions consists in providing the user with the means to validate that an adaptation policy, that is correctly implemented, fulfills extra-functional properties, such as optimized resource-consumption, etc. In this work, we focused on generating test cases as event sequences. One improvement would be the automatic generation of large sets of meaningful initial configurations, which can be decisive in the testing process for reaching specific configurations during the execution of the adaptive system.

References

1. F. Alvares, E. Rutten, and L. Seinturier. Behavioural model-based control for autonomic software components. In *IEEE Int. Conf. on Autonomic Computing, ICAC'15*, pages 187–196. IEEE Computer Society, 2015.
2. A. Bauer and Y. Falcone. Decentralised LTL monitoring. In *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.
3. M. Camilli, C. Bellettini, A. Gargantini, and P. Scandurra. Online model-based testing under uncertainty. In Sudipto Ghosh, Roberto Natella, Bojan Cukic, Robin S. Poston, and Nuno Laranjeiro, editors, *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018*, pages 36–46. IEEE Computer Society, 2018.
4. F. Chauvel, O. Barais, I. Borne, and J.-M. Jézéquel. Composition of qualitative adaptation policies. In *23rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2008)*, pages 455–458. IEEE Computer Society, 2008.
5. B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas. *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*, pages 101–136. Springer Int. Publishing, Cham, 2014.
6. F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Testing adaptation policies for software components. *Software Quality Journal*, 2020. https://doi.org/10.1007/s11219-019-09487-w.
7. R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, and N. Bencomo. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 3–30. Springer, 2017.
8. 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.
9. 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.
10. G. Dupont, Y. Aït Ameur, M. Pantel, and N. Singh. Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B. In M. Butler, A. Raschke,

and K. Reichl, editors, *Int. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2018)*, volume 10817 of *LNCS*, pages 155–170. Springer-Verlag, 2018.
11. E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng. Towards run-time testing of dynamic adaptive systems. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 169–174, May 2013.
12. M. Greiler, H.-G. Gross, and A. van Deursen. Evaluation of online testing for services: a case study. In G. A. Lewis, A. Metzger, M. Pistore, D. B. Smith, and A. Zisman, editors, *Proceedings of the 2nd Int. Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2010*, pages 36–42. ACM, 2010.
13. S. Gupta, A. Ansari, S. Feng, and S. A. Mahlke. Adaptive online testing for efficient hard fault detection. In *27th Int. Conf. on Computer Design*, pages 343–349. IEEE Computer Society, 2009.
14. M. Helvensteijn. Dynamic delta modeling. In E. Santana de Almeida, Ch. Schwanninger, and D. Benavides, editors, *16th International Software Product Line Conference, SPLC'12*, pages 127–134. ACM, 2012.
15. B. Jonsson and K. Larsen. Specification and refinement of probabilistic processes. In *Proc. LICS'91*, pages 266–277. IEEE Computer Society, 1991.
16. J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *5th IEEE Int. Wshp on Policies for Distributed Systems and Networks (POLICY 2004), 7-9 June 2004, Yorktown Heights, NY, USA*, pages 3–12. IEEE Computer Society, 2004.
17. M. Kim, I. Lee, J. Shin, O. Sokolsky, et al. Monitoring, checking, and steering of real-time systems. *ENTCS*, 70(4):95–111, 2002.
18. 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.
19. 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.
20. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Gener. Comput.*, 4(1):67–95, 1986.
21. K. Larsen and B. Thomsen. A modal process logic. In *LICS'88, 1988*, pages 203–210. IEEE Computer Society, 1988.
22. E. Mahe, C.Gaston, and P. Le Gall. Revisiting semantics of interactions for trace validity analysis. In *FASE 2020, Proceedings*, volume 12076 of *LNCS*, pages 482–501. Springer, 2020.
23. R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations. *Electron. Trans. Artif. Intell.*, 3(A):77–105, 1999.
24. R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
25. V. Poladian, D. Garlan, M. Shaw, M. Satyanarayanan, B. R. Schmerl, and J. Pedro Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Proc. of the First Int. Conf. on Self-Adaptive and Self-Organizing Systems, SASO 2007*, pages 214–223. IEEE Computer Society, 2007.
26. D. Romero, C. Quinton, L. Duchien, L. Seinturier, and C. Valdez. Smartyco: Managing cyber-physical systems for smart environments. In D. Weyns, R. Mirandola, and I. Crnkovic, editors, *Software Architecture - 9th European Conference, ECSA*

2015, volume 9278 of *Lecture Notes in Computer Science*, pages 294–302. Springer, 2015.
27. A. Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.
28. F. Trollman, J. Fähndrich, and S. Albayrak. Hybrid adaptation policies: towards a framework for classification and modelling of different combinations of adaptation policies. In Jesper Andersson and Danny Weyns, editors, *Proc. Int. Conf. SEAMS@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 76–86. ACM, 2018.
29. G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Software: Practice and Experience*, 25(1):97–108, 1995.
30. J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Trans. on Software Engineering*, 20(10):812–824, 1994.