Experimenting with a Multi-Approach Testing Strategy for Adaptive Systems

Bento R. Siqueira
DC/UFSCar - Brazil
bento.siqueira@ufscar.br

Misael Costa Júnior
SSC/ICMC/USP - Brazil
misaeljr@usp.br

Fabiano C. Ferrari
DC/UFSCar - Brazil
fcferrari@ufscar.br

Daniel S. M. Santibáñez
DC/UFSCar - Brazil
daniel.santibanez@ufscar.br

Ricardo Menotti
DC/UFSCar - Brazil
menotti@ufscar.br

Valter V. Camargo
DC/UFSCar - Brazil
valtervcamargo@ufscar.br

ABSTRACT

Context: Testing adaptive systems (ASs) is particularly challenging due to certain characteristics such as the high number of possible configurations, runtime adaptations and the interactions between the system and its surrounding environment. Therefore, the combination of different testing approaches in order to compose a strategy is expected to improve the quality of the designed test suites. *Objective:* To devise and experiment with a testing strategy for ASs that relies on particular characteristics of these systems. *Method:* We ranked testing approaches for ASs and devised a strategy that is composed of the three top-ranked ones. The rankings address the challenges that can be mitigated by the approaches, activities from a typical testing process, and characteristics observed in some AS implementations. The strategy was applied to two adaptive systems for mobile devices. *Results:* The approach was applied to both systems. We observed partial gains in terms of fault detection and structural coverage when results are analysed separately for each system, even though no improvements were obtained with the application of the third approach. *Conclusion:* The strategy, despite being incipient, is promising and motivates a deeper analysis of results and new experiment rounds. Furthermore, it can evolve as long as the rankings are updated with new approaches.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Adaptive systems; testing strategy; testing challenges

ACM Reference Format:
Bento R. Siqueira, Misael Costa Júnior, Fabiano C. Ferrari, Daniel S. M. Santibáñez, Ricardo Menotti, and Valter V. Camargo. 2018. Experimenting with a Multi-Approach Testing Strategy for Adaptive Systems. In *17th Brazilian Symposium on Software Quality (SBQS), October 17–19, 2018, Curitiba, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3275245.3275257

SBQS, October 17–19, 2018, Curitiba, Brazil

ACM ISBN 978-1-4503-6565-9/18/10…$15.00
https://doi.org/10.1145/3275245.3275257

1 INTRODUCTION

Adaptive systems (ASs), also referred to as self-adaptive systems [20], play an important role in supporting peoples' social and professional lives. Such systems can adapt themselves to operate according to changes in their surrounding environment [47].

It is well-known that ASs exhibit certain complex characteristics that distinguish them from conventional systems [5, 31]. Some few examples are: the need of monitoring the environment; the high number of adaptation policies to be managed; and the exponential growth of the number of possible configurations. Moreover, it is also well-known that the architecture of ASs can be conceptually divided into a managed and a managing subsystems [11, 37, 45]. The managed one is responsible for fulfilling the functional requirements, whereas the managing one is responsible for the adaptation policies. Therefore, the managing subsystem monitors the environment (or even the managed subsystem), collects relevant data, and, if necessary, elaborates plans and applies the chosen adaptations.

Another important specific property of ASs is the presence of control loops in the managing subsystem [19, 40, 45]. A control loop involves four basic components that are responsible for encapsulating the adaptations: monitors, analysers, planners and executors.

Due to the aforementioned characteristics of ASs, planning, designing and maintaining these systems using traditional approaches are error-prone and time-consuming [20]. Ensuring the reliability of ASs is a fundamental effort, because if failures happen, many users will be affected [43]. To increase reliability, software testing can be applied to reveal faults [28]. In general, testing consists in running the software in any executable abstraction model, with the use of specific input data. It then checks whether the observed behaviour is in agreement with the expected behaviour [28].

Even though testing can reveal faults, it cannot prove their absence without exhaustive - hence impractical – execution [28]. Nevertheless, systematic testing may contribute for increasing the confidence in the correctness of the software functions [14]. In spite of that, in the context of ASs, traditional testing approaches are ineffective due to the inherent characteristics of these systems [10, 15, 43]. Exercising implementations efficiently and detecting faults in ASs are not trivial tasks [17]. According to our previous research [8, 39], in the context of ASs, the two main issues that make AS testing a daunting task for software developers are: (i) the combinatorial explosion of variant composition alternatives, many of them probably unforeseen at design time, and (ii) the nature of this type of system, which includes runtime adaptations.

In this context, devising and applying a customised testing strategy is expected to be an effective way to improve the quality of ASs. Despite our prior work regarding the characterisation of challenges for AS testing [8, 39], to the best of our knowledge, there is not any initiative to create a mapping between challenges and existing testing approaches, in order to compose a testing strategy that is likely to cover a range of harmful situations. Establishing such a strategy is the main goal of our research, as reported in this paper. More specifically when compared to the state-of-the-art, the contributions of this paper are:

- An analysis and establishment of relationships between testing challenges and approaches for ASs testing, and between ASs testing approaches and specific testing activities;
- The definition of a testing strategy for ASs that takes into account existing testing approaches, characterised challenges, and activities in an usual testing process;
- The customisation of three AS testing approaches that compose the strategy, to be applied on behavioural models; and
- An exploratory study for the initial assessment of the strategy.

We describe the steps of our research and the achieved results along this paper as follows: Section 2 presents an overview of the main concepts about software testing, adaptive systems,, and some of main testing challenges for this type of system. Section 3 provides an overview of how we devised the multi-approach testing strategy described in this work. Section 4 describes the exploratory study we performed to evaluate the proposed strategy. Section 5 presents general results and their analysis with respect to data obtained with the application of the strategy. Section 6 discusses limitations of our research. Related research is summarised in Section 7. Finally, Section 8 concludes the paper and indicates our next steps.

2 BACKGROUND

Software testing is a process of running a program with the aim of revealing faults [28], thus helping engineers to increase the system reliability [3]. It consists of a set of dynamic activities that aim to execute a program with specific inputs in order to verify its behaviour [3]. Even though the testing activity may reveal faults, it is impossible to prove their absence, which would require exhaustive execution, with all possible input data [28]. However, when performed in a systematic and rigorous way, the test may contribute to improving user confidence in the software [14].

Testing techniques and criteria provide the developer with a systematically and theoretically grounded approach for defining effective test cases by constructing a mechanism to assess the quality and adequacy of the tests. The techniques can be classified in functional testing, structural testing, and fault-based testing. The main difference between the techniques consists in the source of information for deriving the test requirements (and, hence, the test cases). While functional testing relies on the requirements specifications, structural testing focuses on structural representation (*e.g.* design structure or source code structure) and, finally, fault-based testing relies on common faults that occur in the software.

An **adaptive system – (AS)** is able to evaluate its own behaviour and to change it (*i.e.* to *adapt* it) when the evaluation indicates that the system is not accomplishing what the software is intended to do, or when better functionality or performance is possible [33]. It is composed by two interrelated subsystems; the managing subsystem and the managed subsystem. To perform an adaptation, the former monitors, analyses, plans and executes changes on the managed subsystem. The latter implements the application domain.

Monitors probe the managed subsystem by means of sensors to get the current state of context variables. Analysers correlate the context values received from monitors with reference values to decide about the need for adapting the system. Based on business policies, planners define the maintenance activities to be executed to adapt or evolve the system. Executors implement the set of activities defined by planners [25].

This type of system exposes one or more of the following properties: self-configuration, self-optimization, self-protection and self-healing [33]. These properties are also referred to as the self-* adaptation properties and concern the *how* (means for runtime evolution), *what* (artifacts to be evolved), and *why* (reasons for evolution) perspectives of the runtime software evolution dimensions.

With respect to concepts of ASs used in this work, *context* is a collection of data that represents the environment in which an AS is inserted [24]. Each change in an AS environment generates a new context, so that, a *context variable* is an attribute of an AS with respect to the environment [24]. A *context instance* is an instantiated context variable. Given a timeline t, each context is located in a specific time of t. Thus, as long as the environment changes, a new context is generated with an increment of t [43]. A *context flow* is given by a time series of several *context instances*. *Context diversity* measures the number of context changes inherent in a *context flow* [42]. Usually, we can compute the *context diversity* by applying the total of *Hamming distance* [13] in the *context flows*.

In the context of ASs, due to the likelihood of a combinatorial explosion of configurations, many of those probably unforeseen at specification and design time, testing is a very difficult task for software developers. Thus, traditional testing approaches are inefficient in the context of ASs [43]. There are major challenges of AS testing [39], which are briefly introduced in the next section.

In prior work, we characterised **challenges for AS testing** [39]. We presented a list of 34 specific challenges (some of them are listed in Table 1). We defined a *specific challenge* (SC) as a challenge that has been described as specific for AS concepts, technologies or development approaches. Based on that list, and considering the relationship between the challenges, we defined 12 (so-called) *generic challenges* for AS testing. Among them, "*How to deal with the exponential growth of AS configurations that should be tested.*" and "*How to test AS that run in a distributed and heterogeneous environment.*" are the most recurring challenges discussed by the research community, so that software engineers should take them into account for testing ASs.

3 ESTABLISHING A CANDIDATE STRATEGY FOR TESTING ADAPTIVE SYSTEMS

We define a *testing strategy* as a composition of testing approaches that can be systematically applied to systems under test. A testing approach comprises a set of practices used for testing a specific type of system (*e.g.* adaptive systems), as proposed by authors in the literature. That said, in our work we composed a testing strategy as a selection of testing approaches that encompass and/or address particular sets of characteristics.

Table 1: Specific testing challenges for ASs (adapted from our prior paper [39]).

SC	Description
1	The impossibility to guarantee the correctness of a changing system, whose number of configurations may be unforeseen and growing [26, 27, 41, 44].
2	The issue of limiting (or not) the ability to adapt [10].
3	The issue of when it is possible to stop the testing [46].
4	The dynamicity of ASs, which have no clear boundary, so that the use of the configuration variants are not predictable [4, 29, 30, 44].
5	The difficulty of defining testing oracles [23].
...	...

In our work, the process for composing the strategy takes as input three indicators, as follows: (1) challenges to AS testing; (2) phases of a testing process, focusing on design of test cases and finalisation; and (3) specific AS-related characteristics of the systems under test. The output is the strategy itself (*i.e.* the combination of testing approaches) to be applied to the systems under test. In addition, the strategy can evolve based on other indicators, according to the importance those new indicators might have (*e.g.* a specific testing technique; or a specific type of testing model).

We next present the basis dataset (Section 3.1), the analysis we performed upon the dataset (Section 3.2), and the details of the approaches that compose the strategy (Section 3.3).

3.1 The Basis Dataset

To achieve the goal of establishing a strategy for AS testing, we reused results from our prior work [39]. In short, in our prior paper we analysed and characterised testing-related challenges for ASs which are described in the literature. Other goals of that work were: (i) identifying approaches of software testing that may be customised and applied to ASs; and (ii) characterising the types of faults that may occur in ASs. For this purpose, we performed a Systematic Literature Review (SLR) on AS testing, whose results have been partially reported [39]. Once we are now interested in devising a testing strategy for ASs, we figured out that leveraging our prior analysis would give us an interesting starting point. We selected primary studies that fulfilled at least one of the inclusion criteria: (i) define or apply testing approaches to ASs (34 studies, in total); (ii) characterise challenges for AS testing (25 studies, in total); or (iii) characterise types of faults that are specific to ASs (8 studies, in total). The list of 38 selected studies is available at <http://goo.gl/Cc9N1W>. The original SLR protocol can be found at <http://goo.gl/dnuHP6>.

3.2 The Mappings that Underlie the Strategy

We analysed that set of studies and created mappings to select testing approaches to compose our strategy. The first approach was selected based on the relationship between testing approaches and testing challenges. The second approach was selected based on the relationship between testing approaches and phases of a typical testing process. Finally, the third approach was selected based on the relationship between testing approaches and characteristics of the systems to be tested. Details are next given.

Selection of the first testing approach: given the characterisation of testing challenges [39], we identified the testing approach with the largest number of relationships with challenges for AS testing. In particular, we selected approach S03, by Tse et al. [41], which is associated to 19 out of 34 testing challenges. An excerpt of the matrix created for this analysis is shown in Table 2. A description of Tse et al.'s approach is presented later in this section.

Selection of the second testing approach: at this step, we identified the testing approach that was mostly related with activities throughout the *Test Case Design* and *Finalisation* phases of a testing process. Both phases comprise the creation of test cases (according to some testing technique and criteria) and the evaluation of their execution (*i.e.* measurement of the results). Thus, we selected approach S05, by Lu et al. [24], which was associated 4 out of 5 activities from the considered phases. Specifically, Lu et al.'s approach addresses (1) testing criteria and intended coverage; (2) details about how to apply the approach; (3) information for creating test cases; and (4) which environment requirements should be used.

Selection of the third testing approach: For selecting the third testing approach, we analysed the implementation of two systems, which are both Android applications with adaptive functionalities. The first, called *PhoneAdapter* [34], is driven by rules and can adapt automatically according to the environmental changes, in order to support its users' activities. The second, called *Self-protected* [6], it is a context-aware Android application and its main objective is to provide a feature that uses contextual information (GPS, networks etc.) as an encryption key to encode and decode files. More details about both systems are presented in Section 4.

The analysis of these systems took into account their properties and testing challenges associated with those properties. This let us identify the testing approaches that would mitigate challenges during the testing activity, specifically for the analysed systems. We performed a mapping between system properties, the testing challenges and the testing approaches, defining a tri-dimensional matrix (*i.e.* Properties x Challenges x Approaches). Thus, we analysed the matrix and, as a result, we obtained a ranking that is partially depicted in Table 3. In short, the numbers presented in column "Total" indicate the number of relationships between challenges for AS testing – which were associated to properties of ASs found in *PhoneAdapter* and *Self-protected* – and the analysed testing approaches. Given that approaches S03 and S05 were already selected, and approaches S22, S15, S02, S07 and S09 are too specific, we therefore selected approach S14, by Munoz and Baudry [27].

To conclude, with the selected approaches (S03, S05 and S14), we aimed to devise a strategy that is able to: deal with a large number of AS testing challenges; cover particular activities of a testing process; and address specific characteristics of the systems under test. Descriptions of these approaches are provided in the sequence.

3.3 The Selected Testing Approaches

Tse et al.'s approach (study S03) is based on the concept of metamorphic relations, which consists in generating test cases that are

Table 2: Relationship between testing approaches and testing challenges for adaptive systems.

Testing challenges	Testing approaches															
	S02	S03	S04	S05	S06	S07	S09	S10	S11	S12	S13	S14	S15	S16	S17	...
SC-1	x	x	x	x	x	x	x	x				x	x			...
SC-2								x				x				...
SC-3													x			...
SC-4	x	x	x	x	x	x	x	x				x	x			...
SC-5		x						x				x	x			...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Total	12	19	3	12	7	3	7	15	7	0	0	14	4	12	5	...

Table 3: Top-ranked approaches with respect to the testing challenges observed in the analysed systems.

ID	Author	Total
S03	Tse et al. [41]	13
S22	Akour et al. [1]	10
S15	Niebuhr et al. [30]	9
S02	Flores et al. [9]	8
S05	Lu et al. [24]	8
S07	King et al. [18]	8
S09	Niebuhr and Rausch [29]	7
S14	Munoz and Baudry [27]	7
S35	Griebe and Gruhn [12]	7

related with each others. Each metamorphic relation is responsible for a test set, thus, if a test case fails, all test case that are related to it must be analysed. The authors define relationships for dealing with asynchronous behaviour and explosion of combinations. Since it is a black box approach, the internal explosion of combinations of context is encapsulated, that is, the tests can be applied even when the boundary of the system is uncertain. In our analysis, this approach is related to 19 out of 34 specific challenges for AS testing (Table 2). In addition, this approach is related to 13 specific challenges that we observed in the ASs tested in our exploratory study (see in Table 3 and details in next sections).

The approach of study S05, by Lu et al. [24], relies on structural-based testing. The authors proposed a graph that represents context-aware information of the system under test (the so-called *Context-aware Flow Graph – CaFG*). As an example, be x and y two data flows that run in parallel processes. If a variable v is used in x and changed in y, this can lead to inconsistent value between the flows. The usage of the CaFG lets one identify relationships between different data flows, thus supporting the generation of test cases that exercise the definition and uses of variables. The approach also uses the concepts of *situation* and *association* that, in short, allow the application of the conventional def-use-based data flow criteria, but taking into account data flows related to context-aware data. Based on these concepts, the authors defined four test selection criteria (not described herein due to space limitation).

With respect to approach S14, by Munoz and Baudry [27], it proposes the use of the so-called *Artificial Shaking Table Testing* – ASTT. When the AS has several changes in a short period for specific context variables, Munoz and Baudry argue that this is an interesting scenario for evaluation, since it is fault prone. According to the authors, there are variations in the system environmental conditions that, when exercised, are very likely to reveal faults. In particular, their approach uses "artificial earthquakes" that simulate violent changes in the context variables. To apply the approach to an AS under test, at first the tester needs to identity context variables, context instances, context flows and context diversities. Thus, by analysing the context flows, the tester can see the changes those context variables can go through along the system execution, and hence model the context diversity that may lead to fault detection. In general, the "earthquakes" involve several changes in context flow in a short space of time.

4 EVALUATION STUDY

To evaluate the proposed testing strategy, we ran an exploratory study with the intent of answering the following research questions:

- RQ1: *How much effort is required for applying the strategy (regarding numbers of test cases)?*
- RQ2: *How does the structural coverage evolve with the application of the strategy (regarding statement coverage)?*

The structural coverage was a means of assessing whether the respective testing approaches covered parts of the software we considered important (*e.g.* adaptive properties). Thus, we compared the level of coverage and faults we found.

We conducted the study using two systems (*PhoneAdapter* and *Self-protected*). The study was conducted on a laptop with a Intel Core i7 2.4 GHz processor and 16GB RAM. Results are presented in Section 5. Details of the targeted systems are presented next.

4.1 The Targeted Systems

System 1 - *PhoneAdapter* [22]: This system allows the user to set up the system rules and system profiles. It has been originally proposed as a project to develop a rule-based context-aware pervasive application for research purposes and has been used by other authors in the literature [32, 34–36].

With respect to adaptation-related features, *PhoneAdapter* comprises two main components: *ContextManager* and *AdaptationManager*. Listing 1 shows part of *PhoneAdapter* code. It contains a snippet from the *ContextManager* component and refers to the behaviour of a monitor that receives context data from sensors. The `MyBroadcastReceiver` class inherits from `BroadcastReceiver` that contains features to receive context data and messages from the Android runtime environment. It shows that data is received from the Bluetooth sensor (this is handled by the `onReceiver` event in line 3), and then sent to a component responsible for analysing the context data, using the `mBtDeviceList` collection (line 10).

Sama et al. [34] devised a customised finite state machine, the so-called *Adaptation Finite-State Machine - A-FSM*, to demonstrate changes of contexts in ASs. They used *PhoneAdapter* as a running example, and identified 28 valid sequences of transitions that involve context changes and require system adaptation. In our work,

we extended Sama et al.'s A-FSM into a Mealy Machine [2] to simulate the execution of test cases (in the remainder of this paper we use the term A-FSM to refer to this extended A-FSM). The machine is depicted in Figure 1.a. It contains the states ($q0 - q8$), predicates ($a - p$) and adaptations (ME, MA, MI, VI and AV).

Listing 1: Receiving sensor data from the bluetooth sensor.

```
1  public class MyBroadcastReceiver extends BroadcastReceiver{
2      @Override
3      public void onReceive(Context c, Intent i) {
4          String action=i.getAction();
5          ...
6          else if(action.equals(BluetoothDevice.ACTION_FOUND)){
7                 BluetoothDevice device = i.getData(BluetoothDevice.EXTRA_DEVICE);
8
9                 if(!listContainsMac(mBtDeviceList, device.getAddress())){
10                      mBtDeviceList.add(device.getAddress());
11                }
12         }
13         ...
14     }
15 }
```

Listing 2: Uncrypt method with adaptation strategy

```
1  file = (SmartFile) this.getIntent().
2  getSerializableExtra(ActivityContent.BUNDLE_FILE);
3
4  public void nextIndex(){
5         index++;
6         if (index >= file.getPolicies().size())
7                 uncrypt();
8         else
9                 senseContext();
10 }
11 ...
12 public void uncrypt(){
13
14     if (file.decodeFile(this)){
15                Intent i = new Intent(this, ActivityContent.class);
16                i.putExtra(ActivityContent.BUNDLE_FILE, file);
17
18                done = true;
19                Toast.makeText(this, uncrypted, Toast.LENGTH_SHORT).show();
20                this.finish();
21                startActivity(i);
22        }
23        else{
24                this.finish();
25        }
26 }
```

System 2 - *Self-protected* [6]: In this system, the user encodes a file in a given environment and can only decode it if the information used in the encryption is the same as the current user information. As a strategy to perform the adaptation, *Self-protected* captures the user's information automatically, updating the decoding method of the encrypted message files. Thus, if the contextual information of the user is the same as the encoding information of a specific message file, the decoding method is enabled.

Self-protected comprises three main components: *ContextManager*, *FileEncode*, and *FileDecode*. *ContextManager* corresponds to a set of classes that aim to obtain contextual information. The information is obtained periodically and the contextual information is added to the encode policies of the message file.

FileEncode encodes a message file and adds the contextual information used in the encoding through a policy manager. Finally, *FileDecode* decides whether a message file will be decoded or not.

Listing 2 presents the method responsible for decoding the message file. The `uncrypt` method is called from the `nexIndex` method (line 7) to decode or verify the possibility of decoding a message file. Initially, the information from a given file is installed (line 1) and, from the `nextIndex` method (line 4), the similarity between the contextual encoding information and the current contextual information of the user is verified. If the contextual information encoding a message file is similar to the current contextual information, the file decoding function is enabled (lines 14–22).

In *Self-protected*, context information management methods are called periodically by the application and the state of the `uncrypt` method is updated as the contextual information of the user is changed. Figure 1.b presents the A-FSM we created for this system, upon which we designed the test cases following our strategy. For this system, we identified 15 valid sequences of transitions that involve context changes and require system adaptation. It is important to emphasise that the derivation of A-FSM for *Self-protected* followed the same methodology used in the A-FSM of the PhoneAdapter (Figure 1.a).

Table 4 shows some basic metrics for these systems. LOC-related metrics only consider executable lines of code. Columns labelled with "# classes adapt." and "LOC adapt.", respectively, refer to the number of classes that fully or partially deals with system adaptations, and their associated LOC.

Table 4: Basic metrics for *PhoneAdapter* and *Self-Protected*.

System	LOC	# classes	# classes adapt.	LOC adapt.
PhoneAdapter	2.583	91	22	720
Self-Protected	980	38	21	820

4.2 Applying the Testing Approaches

Using the A-FSMs as the underlying test models, we customised the application of the selected testing approaches as following explained. We highlight that these customisations are also a contribution of this work, since the original authors [24, 27, 41] did not suggest the application of their approaches to behavioural models.

Tse et al. [41]'s approach: The main characteristic of this approach is that if a given test case t does not pass, all inter-related test cases (*i.e.* test cases in a metamorphic relation (MR) that includes t) are all seen as failing test cases and should be analysed. That said, applying this approach involved in advance the use of domain restrictions, focusing on a specific user profile. Thus, for each MR we defined paths involving states and transitions between states. For these transitions we also defined predicates (which involves variables that store input from sensors) and oracles (derived from outputs of the A-FSM).

To illustrate how we applied the concept of MR to the targeted systems, let us consider the A-FSM designed for *PhoneAdapter* (Figure 1.a). One of the defined MRs includes the following test cases (notice that these tests lead to, or include the *Meeting* – q_7):

(1) $\lambda(q_0, k)$
(2) $\lambda(q_0, k)\lambda(q_6, m)$
(3) $\lambda(q_0, k)\lambda(q_6, m)\lambda(q_7, n)$
(4) $\lambda(q_0, k)\lambda(q_6, m)\lambda(q_7, n)\lambda(q_6, l)$

Test case (1) involves predicate k and exercises the transition $q_0 \rightarrow q_6$. By running (1), the adaptation VI (*Vibrator mode*) takes place and the final state is q_6 (*Office*). In (2), the sequence of transitions $q_0 \rightarrow q_6 \rightarrow q_7$ is covered; in this case, the adaptation AV

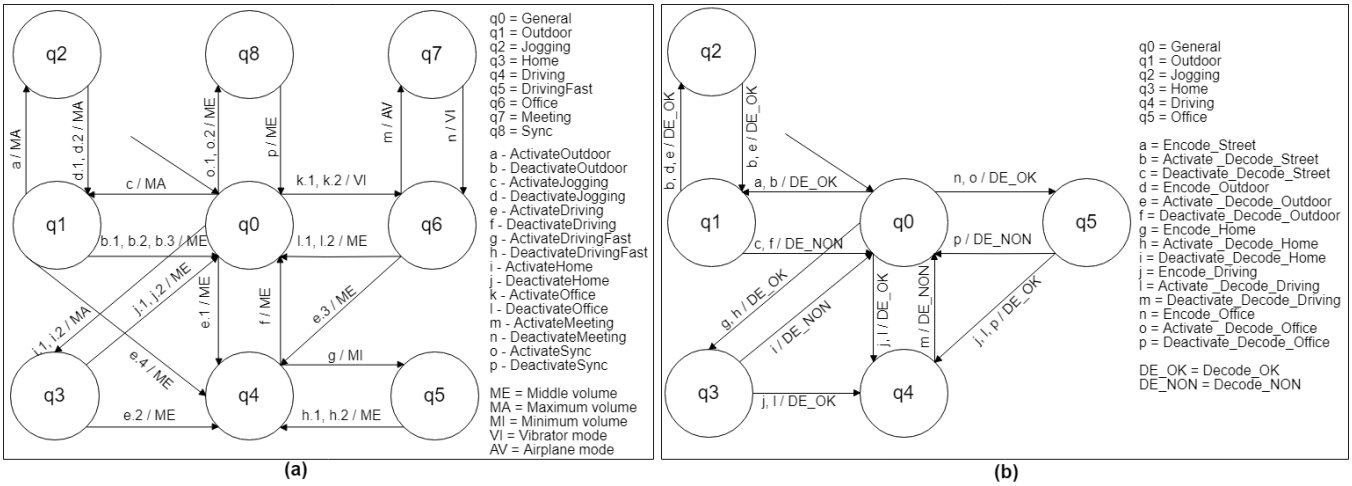

Figure 1: A-FSMs (extended with Mealy Machine notation) used as a testing model for *PhoneAdapter* (a) and *Self-protected (b).*

(*Airplane mode*) takes place and the final state is q_7 (*Meeting*). This sequence of transitions is presented in Listing 3, as an example of the test cases we developed during this work using JUnit. As we can see, for each transition there are an execution (*i.e. lambda*) and a collect (*i.e. output*) of test results.

Listing 3: Example of an implemented test case.

```
...
   @Test
   public void testCase11(){
       TestCaseOutput output1 = lambda_q0_k1();
       boolean result1 = ActivateOffice(output1);
       TestCaseOutput output2 = lambda_q6_m();
       boolean result2 = ActivateMetting(output2);
       TestCaseOutput output3 = lambda_q7_n();
       boolean result3 = DeactivateMetting(output3);
       TestCaseOutput output4 = lambda_q6_l();
       boolean result4 = DeactivateOffice(output4);

       assertTrue(result1 && result2 && result3 && result4);
   }
```

Lu et al. [24]'s approach: We applied the four testing criteria proposed by Lu et al. [24], using CaFGs and A-FSMs we designed for each system. The main characteristics of using these criteria is about defining graphs according to situations and associations between context-aware data flows for generating test cases.

We highlight the fact that the graphs do not intend to represent the full control flow of the program. Instead, they only represent data flows that involve context variables. Figure 2 brings some examples of CaFGs. It shows three graphs (each surrounded by the large rectangles, namely, *s1*, *sgps* and *cm*). Nodes are denoted by n_i, where i is an ID number, and each graph has an entry node and an exit node. Double rectangled nodes (*e.g.* node $n1$ of *cm* graph (Figure 2.b) represent context variables that are defined (def) in a graph and used (use) in another. Continuous arrows indicate sequences of nodes, and dashed arrows indicate associations between nodes of two distinct data flows.

Initially, we identified the context variables, which, in this case, store values that come from sensors. For *PhoneAdapter*, we defined two graphs to deal with these variables. The first graph, named *sgps*

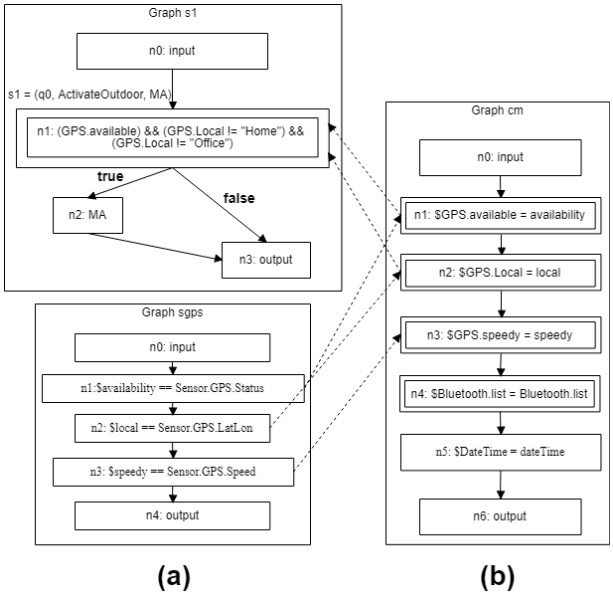

Figure 2: Examples of CaFGs designed for *PhoneAdapter*. The dashed arrows indicate inter-graph definition-use of context-related variables.

in Figure 2.a, has definitions for the following context variables: *$availability*, responsible for storing whether the sensor is active or not; *$local*, responsible for storing geographic coordinates; and *$speedy*, responsible for storing the maximum speed of movement based on the GPS sensor. The *cm* graph (Figure 2.b) has the definition of *$Bluetooth.list*, which is responsible for storing the Bluetooth devices to be used by the system adaptations. Finally, each situation s_i is represented as a graph that contains a predicate that defines the adaptations. An example is shown in Figure 2.a (graph *s1*).

Based on the A-FSM and on the CaFGs created for each system, we identified the relationships between situations (from CaFGs) and

transitions (from A-FSMs). Taking situation *s1* as an example (Figure 2.a, graph *s1*), from state q_0 the adaptation *MA* can be triggered if the predicate *ActivateOutdoor* is true. Based on such kind of relationships, we defined the test cases to cover the requirements of the four criteria proposed for Lu et al. [24].

Munoz and Baudry [27]'s approach: As required by Munoz and Baudry [27]'s approach, we needed to identify all changes in context variables that may happen during the system execution. At first, based on the A-FSMs, we applied the pairwise technique for generating the possible pairs of states. For this, we analysed all states and calculated the context diversity between two different states. As an example, taken from the *Self-protected* system, we observed the possible combinations of pairs between the six states of the A-FSM (Figure 1.b). Starting from each state q_i, we observed its paths to all other reachable states q_j. We then identified the shortest path between q_i and q_j (*i.e.* each path was considered a context flow). Again based on the example of Figure 1.b, the shortest path between q_0 and q_2 is $q_0 \rightarrow q_1 \rightarrow q_2$. Finally, we calculated the "artificial earthquakes" (*i.e.* large variations of context diversity in context flows) taking into account the start and final states of those shortest paths, and test cases were designed to simulate them.

5 RESULTS AND ANALYSIS

This section presents and analyses the results of our study with respect to the research questions. Further discussions address fault detection effectiveness, as well as some examples of faults revealed in the *PhoneAdapter* system with the application of the strategy.

5.1 Results

Table 5 summarises the results of the experimental studies we conducted over *PhoneAdapter* and *Self-protected*. The table includes:

(i) The *study configuration* used to obtain one specific testing result (prefixes "1" and "2" refer to *PhoneAdapter* and *Self-protected*, respectively).
(ii) The *testing approach* applied to the targeted systems.
(iii) The *SUT*s (*i.e.* the targeted systems).
(iv) The *number of test cases* designed to fulfil a given testing approach (all tests were implemented as JUnit tests).
(v) The *number of failures* observed during the set set executions.
(vi) The *number of faults* revealed during the analysis of failures.
(vii) The *percentage of covered adaptive-related code*, which is the percentage of covered code, with focus on classes that fully or partially deals with system adaptations.

The Eclemma tool was used to more thoroughly evaluate the coverage of the test cases in our study. Detailed information produced by Eclemma is shown in Table 6 includes: number of instructions, number of missed instructions, number of classes, number of missed classes, number of methods, and number of missed methods. Note that one LOC in Table 4 is considered executed when at least one instruction that is assigned to this LOC has been executed. Moreover, note that a single LOC can contain several instructions.

5.2 Analysis and Discussion

The proposed strategy comprises three testing approaches. By applying them in sequence, we were able to analyse the results from different perspectives, according our research questions.

Initially, it is important to note that Table 5 shows only relative application costs for each testing approach. By *relative* we mean that test cases from previous approaches were reused, thus only complementary test cases were added to cover all test requirements. In other words, to apply Lu et al. [24]'s approach we reused test cases created to cover test requirements of Tse et al. [41]'s approach. Similarly, to apply Munoz and Baudry [27]'s approach we reused test cases created to fulfil test requirements derived from both Tse et al.' and Lu et al.'s approaches.

RQ1: *How much effort is required for applying the strategy (regarding numbers of test cases)?*

The first applied approach [41] is driven by the metamorphic relations defined for subsets of test cases. Its application in *PhoneAdapter* required the creation of 49 test cases that are associated with 9 metamorphic relations. For *Self-protected*, 80 test cases were created. They are associated to 5 metamorphic relations defined for that system. Lu et al. [24]'s approach was applied next. It considers the point of view of analysing associations between different program data flows, according to characteristics of parallel processes of this type of system. Its application in *PhoneAdapter* required the addition of 40 test cases (increase of 81%). For *Self-protected*, 5 test cases were added to the current test set (increase of 6%).

The last applied approach [27] explores a way to identify context flows which are likely to be error prone due to substantial changes in the context variables (such changes as called "artificial earthquackes"). For *PhoneAdapter*, its application required the creation of additional 13 test cases (increase of 15%). For *Self-protected*, 18 test cases were added to the current test set (increase of 21%).

Overall, the application of the strategy required the creation of 102 test cases for *PhoneAdapter* and 103 test cases for *Self-protected*. Despite the oscillation in terms of required effort noted during the application of the second testing approach, the numbers seemed to be more "stable" when the third approach was applied in both systems. Moreover, if we consider the classes related to adaptations and their associated LOC (see columns 4 and 5 of Table 4), we obtain similar averages of test cases per class and per LOC: 4.64 and 0.14 for *PhoneAdapter*, and 4.90 and 0.13 *Self-protected*, respectively.

In order to draw a conclusion regarding RQ1, let us consider some examples extracted from the literature on software testing. In a mutation testing-related study, Just et al. [16] used several releases of industrial, medium-sized systems obtained from source code repositories. Those authors characterise the original (developer) test suites as "*comprehensive*". In short, the statement coverage of the test suites ranged from 55% to 90%, with average test cases per LOC ranging from 0.02 to 0.15. As another study, which addressed structural testing, Lemos and Masiero [21] estimated an average of 0.22 test cases per LOC in order to obtain adequate test suites for medium-sized systems developed in the academic context.

Based on our results and taking these two pieces of work as examples – from industrial [16] and academic perspectives [21], respectively – we can draw the following conclusion for RQ1: *in terms of the number of required test cases, the application of the strategy required low to moderate effort to be applied in the targeted systems.* Besides that, as next discussed (RQ2), even though Munoz and Baudry [27]'s approach was not able to increase the code coverage, its relative application cost (which can be considered an overhead)

Table 5: Summary of the execution of the testing approaches.

(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)
Study Configuration	testing approach	SUT	# test cases	# failures	# faults	% covered adapt. instructions
1 - S03	Tse et al. [41]	PhoneAdapter	49	4	1	57%
2 - S03	Tse et al. [41]	Self-protected	80	0	0	73%
1 - S05	Lu et al. [24]	PhoneAdapter	49 + 40	4 + 10	1 + 5	58%
2 - S05	Lu et al. [24]	Self-protected	80 + 5	0 + 0	0 + 0	74%
1 - S14	Munoz and Baudry [27]	PhoneAdapter	49 + 40 + 13	4 + 10 + 2	1 + 5 + 0	58%
2 - S14	Munoz and Baudry [27]	Self-protected	80 + 5 + 18	0 + 0 + 0	0 + 0 + 0	74%

Table 6: Summary of the test case coverage report.

(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)
Testing approach	SUT	# instruc.	# missed instruc.	# classes	# missed classes	# methods	# missed methods
Tse et al. [41]	PhoneAdapter	3,739	1,623	22	9	63	32
Tse et al. [41]	Self-Protected	3,495	896	21	3	147	29
Lu et al. [24]	PhoneAdapter	3,739	1591	22	8	63	30
Lu et al. [24]	Self-Protected	3,495	855	21	3	147	29
Munoz and Baudry [27]	PhoneAdapter	3,739	1591	22	8	147	30
Munoz and Baudry [27]	Self-Protected	3,495	855	21	3	147	29

was low: it required the addition of 13 out of 102 (15%) test cases for *PhoneAdapter* and 18 out of 103 (21%) for *Self-protected*, respectively.

RQ2: *How does the structural coverage evolve with the application of the strategy (regarding statement coverage)?*

Regarding the code coverage obtained with application of the strategy, test cases derived from the three approaches covered 58% of *PhoneAdapter* code and 74% of *Self-protected* code. Once again, note that we considered only code related to system adaptation mechanisms while computing coverage with the Eclemma tool.

As shown in Table 5, the incremental application of the strategy slightly incremented the coverage. In particular, this increase could only be observed when we applied the second approach. In that case, the coverage of *PhoneAdapter* increased 1% (from 57% to 58%), likewise the coverage of *Self-protected* (from 73% to 74%). No gains were observed with the application of Munoz and Baudry [27]'s approach in both systems.

For RQ2 we can conclude that *the sequential application of varied testing approaches did not yield substantial gains in terms of code coverage*. Despite that, the small obtained increase led us to reveal some faults in the *PhoneAdapter* system, as discussed in Section 5.3.

5.3 Further Discussion

It is worth to mention one interesting result from this exploratory study: six faults were detected in the *PhoneAdapter* system. This was an interesting outcome, since we tested each system *as is*, that is, we did not apply any fault injection approach (*e.g.* mutation testing). Thus, originally, we did not draw any expectation regarding fault revealing capabilities of the strategy as a whole, likewise for the testing approaches themselves.

Regarding to the relationship between testing effort, code coverage and detected faults, the application of the first approach [41] in *PhoneAdapter* required, 49 test cases, yielded 73% of code coverage and revealed one fault. The second approach [24] required additional 40 test cases, but increased coverage only in 1%. Despite this, and surprisingly, this small coverage increase allowed us to detect five other faults in this system. Even more surprisingly is the fact that *PhoneAdapter* seemed to be a more "mature" system due to its usage in other research studies [32, 34–36]. Our future work will deal with fault injection (*e.g.* mutations) to evaluate the trade-off between effort and number of revealed faults.

Listing 4: Examples of *Nondeterministic Activation faults.*

```
1  ...
2  // getting data from context
3  boolean gpsAvailable =
4            i.getBooleanExtra(ContextName.GPS_AVAILABLE, false);
5  String gpsLocation =
6            i.getStringExtra(ContextName.GPS_LOCATION);
7  double gpsSpeed =
8            i.getDoubleExtra(ContextName.GPS_SPEED, 0.0);
9  String[] deviceMacList =
10           i.getStringArrayExtra(ContextName.BT_DEVICE_LIST);
11 int count = deviceMacList.length;
12 String time = i.getStringExtra(ContextName.TIME);
13 String weekday = i.getStringExtra(ContextName.WEEKDAY);
```

Listing 4 includes 5 faults found in *PhoneAdapter*. This code belongs to a component that executes in parallel processes and aims to obtain values from context variables. Such values will be later used for finding the adaptation predicate and for triggering the desired adaptation. In line 5, if the GPS location is unknown (*i.e.* the device is in a location different from home or office), by default the `gpsLocation` variable is assigned the value of 0 (`false`), and hence the Outdoor state (q_1 in Figure 1.a) will be activated. However, failures were obtained with the execution of test cases that verified sequences of transitions that are affected by this default value. According to Sama et al. [34], such an imprecise (in fact, inconsistent) value assignment represents an instance of *Nondeterministic Activation* fault. Note that the value assignments in lines 3, 7, 9 and 12 also suffer from the same inconsistency issue. Therefore, they also represent faults in *PhoneAdapter*.

Another point to be discussed regards the variations in statement coverage and numbers of test cases for *PhoneAdapter* and *Self-proteced*. In Section 4.1, we described that *PhoneAdapter* comprises two main components: *ContextManager* and *AdaptationManager*.

The *ContextManager* is responsible for monitoring data from the environment to generate contexts. With respect to this, we identified a difficulty in testing the monitoring behaviour due to the combinatorial explosion of possible configurations that is a specific challenge for ASs testing. Therefore, during the experimentation, instead of allowing the generation of context data by the *ContextManager* component, we decided to "fix" the data to narrow the number of possibilities. This means the source code of *ContextManager* was not touched by tests, thus leading to a lower coverage (58%) when compared to the results for *Self-protected*.

Regarding the numbers of tests designed for each system, the application of Lu et al. [24]'s approach resulted in expressive difference: *PhoneAdapter* required the addition of 40 test cases (increase of 81%), whereas *Self-protected* required only 5 new test cases (increase of 6%). The analysis of both systems revealed that in *PhoneAdapter* the context variables are defined and used by different parallel processes. However, it does not happen in *Self-protected*, in which both variables and adaptations are performed in the same process. Consequently, the analysis of *Self-protected* yielded less inter-related graphs to represent data flows.

In regard to runtime testing, which is another issue for ASs testing broadly discussed in the literature [1, 10, 30, 38], it is defined by Silva and De Lemos [38] as testing activities that "*need to be performed during run-time*", including definition of the testing plan. That said, we highlight that our strategy was applied at design-time and involved AS features like run-time and monitoring. In other words, the applied approaches deal with context changes during the AS execution. We foresee the strategy can be applied at run-time, following the process proposed in Silva and De Lemos [38]'s work, with adaptations of models, techniques, criteria and testing level. For instance, in Silva and De Lemos's work, the process generation (comprised by the strategy, tactical and operational phases) involves *existing techniques*. These *existing techniques* may be the approaches applied in our work, which can guide the definition of test cases for each component of the AS. In addition, we are aware of ASs that are (dynamically) self-reconfigurable, so that the run-time testing, specifically at integration level, is essential. However, this characteristic was not present in the ASs we tested, thus we did not explicitly split our test cases based on testing levels. Indeed, the *PhoneAdapter* system could present the challenge of a large number of configurations only if its context variations were not defined by the user, which is not the case.

6 LIMITATIONS

We next discuss some limitations regarding (i) the definition of the strategy and (ii) the procedures and results of the exploratory study. With respect to (i), we defined rankings and characterisations by using a theoretical basis, after the analysis of reference literature and implemented ASs. However, different understanding and procedures (*e.g.* definition of the test model, order of application of multi-approach strategies, or test automation steps) might have led to different results. Nevertheless, we focused on developing our analysis in a discerning and systematic manner, and hence on attempting to mitigate this threat.

The basis database itself, obtained from our prior work [39], may be incomplete, so there may exist other AS testing approaches that may substitute the top-ranked ones to compose the strategy. This threat can be mitigated with regular updates of the SLR.

With respect to the exploratory study, we applied the testing approaches in sequence (S03 [41] → S05 [24] →S14 [27]). Different orders of application might have led to different results, particularly with respect to required effort when test case reuse is considered. This will be explored in future work. Besides that, the application of the three approaches based on a behavioural model (A-FSM) is also a threat, given that the targeted systems may have been designed for specific purposes, so that there might exist other testing approaches that are more well-aligned to be applied to them.

Regarding the results, the total number of test cases designed for *PhoneAdapter* and *Self-protected* are similar (102 and 103, respectively). Despite this, there is not a clear pattern when we consider the increments on the test set along the application of the approaches. This is likely to have occurred due to particularities of both systems such as their size, architecture and context modelling.

7 RELATED WORK

Apart from the studies of Tse et al. [41], Lu et al. [24] and Munoz and Baudry [27], this section summarises some related work that explores hybrid approaches for ASs, as well as research that addressed the *PhoneAdapter* system for testing purposes.

Eze et al. [7] evaluated the complementary properties of AS testing approaches from the following viewpoints: generic approaches that can be adapted to different processes; testing approaches related with the design and execution software lifecycle phases; and integrated testing approaches that are not separated from the architecture of the system under testing. The authors pointed out that complementarity cannot be applied in experimental studies so it is not possible measure their effectiveness. In both research initiatives, authors neither present experimental studies, nor define indicators to define testing strategies. The main difference between this approach and our proposal is that, even though they analysed the complementarity of approaches, they did not apply it as we did.

Sama et al. investigated the problem of exposing faults of ASs that cannot be exposed with traditional testing techniques (*i.e.* testing techniques that do not take into account characteristics of adaptation during the testing) [34–36]. Thus, they defined and applied the A-FSM model as representation to support the detection of faults of incorrect adaptation logic, and asynchronous update of context information. *PhoneAdapter* was used in their approach description and evaluation studies. In our work, we extended the A-FSM model in order to facilitate the generation of test cases for the three testing approaches that compose the strategy.

Qin et al. [32] developed an approach named SIT (Sample-based Interactive Testing) for AS testing. They suggest that an input space of an AS can be systematically split, adaptively explored, and mapped to the testing of the AS's different behaviour. This is achieved by two components, an interactive model and a test generation technique. The former captures characteristics of interactions between an AS and its environment, and the latter uses adaptive sampling to explore an AS input space and test its behaviour. The effectiveness and efficiency of SIT were evaluated by the authors in a case study that targeted the *PhoneAdapter* system.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented the definition of a testing strategy for ASs based on the analysis of testing challenges and testing approaches for ASs, between testing approaches and specific testing activities, and between testing approaches and challenges observed in existing AS implementations.

With respect to the characterisation of challenges, which supported us in our mappings to define the testing strategy, this work used results from our prior research [39]. Thus, the mappings that we performed allowed us to devise a testing strategy comprised by three approaches, which were sequentially applied on behavioural models in our exploratory studies involving two systems. An exploratory study enabled us to observe partial gains in terms of fault detection and structural coverage when results are analysed separately for each system, even though no improvements were obtained with the application of the third approach.

Regarding the effort the apply the strategy (in terms of the number of required test cases), the application of the strategy required low to moderate effort to be applied in the targeted systems, compared to the literature. In addition, the sequential application of varied testing approaches did not yield substantial gains in terms of code coverage, even though a little increase during the second testing [24] approach allowed us identifying some faults in one of the targeted systems (*PhoneAdapter*).

In spite of being incipient, the achieved results are promising and motivates a deeper analyses and new experiment rounds. The new rounds will include variations in the application order of the approaches, as well as the inclusion of other AS implementations to be tested. Furthermore, the strategy can evolve as long as the rankings are updated with new approaches. We also aim to investigate the theoretical and practical costs for the strategy (also considering its possible evolution).

ACKNOWLEDGMENTS

We thank the financial support received from FAPESP, SP-Brazil (process 2016/03104-0), CNPq (process 306310/2016-3), Becas Chile Scholarship (CONICYT), and PPGCC/UFSCar (PROAP).

REFERENCES

[1] M. Akour, A. Jaidev, and T. M. King. 2011. Towards Change Propagating Test Models in Autonomic and Adaptive Systems. In *ECBS'11*.
[2] S. Bensalem, K. Moez, and T. Stavros. 2008. State identification problems for input/output transition systems. In *WODES'08*.
[3] A. Bertolino. 2003. Software testing research and practice. In *ASM'03*.
[4] J. Cámara, R. Lemos, N. Laranjeiro, R. Ventura, and M. Vieira. 2015. Robustness-Driven Resilience Evaluation of Self-Adaptive Software Systems. *IEEE Transactions on Dependable and Secure Computing* 6, 1 (2015).
[5] B. H. C. Cheng et al. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap.* Chapter Software Engineering for Self-Adaptive Systems.
[6] S. Constantino. 2014. SelfProtectedProject. Online. (2014). http://github.com/SamMcSam/SelfProtectedProject - last access on 25/04/2018.
[7] T. Eze, R. J. Anthony, C. Walshaw, and A. Soper. 2011. The challenge of validation for autonomic and self-managing systems. In *ICAS'11*.
[8] F. C. Ferrari, B. B. P. Cafeo, J. Noppen, R. Chitchyan, and A. Rashid. 2011. Investigating Testing Approaches for Dynamically Adaptive Systems. In *VariComp'11*.
[9] A. Flores, J. C. Augusto, M. Polo, and M. Varea. 2004. Towards Context-Aware Testing for Semantic Interoperability on PvC Environments. In *SMC*.
[10] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng. 2013. Towards run-time testing of dynamic adaptive systems. In *SEAMS'13*.
[11] D. Garlan et al. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (2004).
[12] T. Griebe and V. Gruhn. 2014. A model-based approach to test automation for context-aware mobile applications. In *SAC'14*.
[13] R. W. Hamming. 1950. Error Detecting and Error Correcting Codes. *Bell System Technical Journal* 29, 2 (1950).
[14] M. J. Harrold. 2000. Testing: A Roadmap. In *ICSE'00*.
[15] L. C. Jaw, D. Homan, V. Crum, W. Chou, K. Keller, K. Swearingen, and T. Smith. 2008. Model-based Approach to Validation and Verification of Flight Critical Software. In *AeroConf'08*.
[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *FSE'14*.
[17] T. M. King, A. A. Allen, R. Cruz, and P. J. Clarke. 2011. Safe Runtime Validation of Behavioral Adaptations in Autonomic Software. In *ATC'11*.
[18] T. M. King, A. Ramirez, R. Cruz, and P. Clarke. 2007. An Integrated Self-Testing Framework for Autonomic Computing Systems. *Journal of Computers* (2007).
[19] M. M. Kokar, K. Baclawski, and Y. A. Eracar. 1999. Control theory-based foundations of self-controlling software. *IEEE Intel. Syst. and their Applications* (1999).
[20] P. Lalanda, J. A. McCann, and A. Diaconescu. 2013. *Autonomic Computing*.
[21] O. A. L. Lemos and P. C. Masiero. 2011. A Pointcut-Based Coverage Analysis Approach for Aspect-Oriented Programs. *Information Sciences* (2011).
[22] Y. Liu. 2013. PhoneAdapter. Online. (2013). http://sccpu2.cse.ust.hk/afchecker/phoneadapter.html - último acesso em 27/04/2018.
[23] H. Lu. 2007. A context-oriented framework for software testing in pervasive environment. In *ICSE'07*.
[24] H. Lu, W. K. Chan, and T. H. Tse. 2006. Testing Context-aware Middleware-centric Programs: A Data Flow Approach and an RFID-based Experimentation. In *SIGSOFT'06*.
[25] T. Mens, A. Serebrenik, and A. Cleve. 2014. *Evolving Software Systems*.
[26] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik. 2012. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *KES-AMSTA'12*.
[27] F. Munoz and B. Baudry. 2009. *Artificial table testing dynamically adaptive systems.* Technical Report. INRIA.
[28] G. J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing*.
[29] D. Niebuhr and A. Rausch. 2007. A Concept for Dynamic Wiring of Components: Correctness in Dynamic Adaptive Systems. In *SAVCBS'07*.
[30] D. Niebuhr, A. Rausch, C. Klein, J. Reichmann, and R. Schmid. 2009. Achieving Dependable Component Bindings in Dynamic Adaptive Systems - A Runtime Testing Approach. In *SASO'09*.
[31] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* (1999).
[32] Y. Qin, C. Xu, P. Yu, and J. Lu. 2016. SIT: Sampling-based interactive testing for self-adaptive apps. *Journal of Systems and Software* (2016).
[33] M. Salehie and L. Tahvildari. 2009. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.* (2009).
[34] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang. 2010. Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification. *IEEE Transactions on Software Engineering* (2010).
[35] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. 2008. Model-based Fault Detection in Context-Aware Adaptive Applications. In *FSE'08*.
[36] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. 2010. Multi-Layer Faults in the Architectures of Mobile, Context-Aware Adaptive Applications. *Journal of Systems and Software* (2010).
[37] M. Shaw. 1995. Beyond Objects: A Software Design Paradigm Based on Process Control. *SIGSOFT Software Engineering Notes* (1995).
[38] C. E. Silva and R. De Lemos. 2011. Dynamic Plans for Integration Testing of Self-adaptive Software Systems. In *ICSE'11*.
[39] B. R. Siqueira, F. C. Ferrari, M. A. Serikawa, R. Menotti, and V. V. Camargo. 2016. Characterisation of Challenges for Testing of Adaptive Systems. In *SAST*.
[40] J. G. Truxal. 1961. Computers in Automatic Control Systems. *IRE'61* (1961).
[41] T. H. Tse, W. K. Chan, S. S. Yau, H. Lu, and T. Y. Chen. 2004. Testing Context-Sensitive Middleware-Based Software Applications. In *COMPSAC'04*.
[42] H. Wang and W. K. Chan. 2009. Weaving context sensitivity into test suite construction. In *ASE'09*.
[43] H. Wang, W. K. Chan, and T. H. Tse. 2014. Improving the Effectiveness of Testing Pervasive Software via Context Diversity. *ACM Transactions on Autonomous and Adaptive Systems* (2014).
[44] K. Welsh and P. Sawyer. 2010. Managing testing complexity in dynamically adaptive systems: A model-driven approach. In *ICST'10*.
[45] D. Weyns, M. Iftikhar, and J. Soderlund. 2013. Do External Feedback Loops Improve the Design of Self-adaptive Systems? A Controlled Experiment. In *SEAMS*.
[46] F. Wotawa. 2012. Adaptive autonomous systems - From the system's architecture to testing. *Communications in Computer and Information Science* (2012).
[47] L. Yu and J. Gao. 2014. Generating Test Cases for Context-aware Applications Using Bigraphs. In *SERE'14*.