Behavioral Maps

Lima dos Santos, Edilton; Fortz, Sophie; Schobbens, Pierre-Yves; Perrouin, Gilles

Published in:
Software Architecture - 15th European Conference, ECSA 2021 Tracks and Workshops, Revised Selected Papers

DOI:
10.1007/978-3-031-15116-3_8

Publication date:
2022

Document Version
Peer reviewed version

Behavioral Maps: Identifying Architectural Smells in Self-Adaptive Systems at Runtime

Edilton Lima dos Santos[1 [0000−0003−2231−3852]], Sophie Fortz[1 [0000−0001−9687−8587]], Pierre-Yves Schobbens[1 [0000−0001−8677−4485]], and Gilles Perrouin[1 [0000−0002−8431−0377]]

PReCISE, NaDI,
Faculty of Computer Science, University of Namur, Namur, Belgium
{edilton.limados,sophie.fortz,pierre-yves.schobbens,gilles.perrouin}@unamur.be

Abstract. Self-adaptive systems (SAS) change their behavior and structure at runtime, depending on environmental changes and reconfiguration plans and goals. Such systems combine architectural fragments or solutions in their (re)configuration process. However, this process may negatively impact the system's architectural qualities, exhibiting architectural bad smells (ABS). Also, some smells may appear in only particular runtime conditions. This issue is challenging to detect due to the combinatorial explosion of interactions amongst features. We initially proposed the notion of *Behavioral Map* to explore architectural issues at runtime. This extended study applies the *Behavioral Map* to analyze the ABS in self-adaptive systems at runtime. In particular, we look for Cyclic Dependency, Extraneous Connector, Hub-Like Dependency, and Oppressed Monitor ABS in various runtime adaptations in the Smart Home Environment (SHE) framework, Adasim, and mRUBiS systems developed in Java. The results indicate that runtime ABS identification is required to fully capture SAS architectural qualities because the ABS are feature-dependent, and their number is highly variable for each adaptation. We have observed that some ABS appears in all runtime adaptations, some in only a few. However, some ABS only appear in the publish-subscribe architecture, such as Extraneous Connector and Oppressed Monitor smell. We discuss the reasons behind these architectural smells for each system and motivate the need for targeted ABS analyses in SAS.

Keywords: Architectural Smells · Dynamic Software Product Lines · Runtime Validation · Self-adaptive Systems · Behavioral Maps.

1 Introduction

Self-adaptive systems (SAS) must adjust their structure or behavior, depending on environmental changes and (re)configuration plans to work in such environments. Moreover, (re)configurations may also negatively affect architectural qualities at runtime. It happens because the (re)configuration process combines different architectural fragments or solutions via feature binding/unbinding at

runtime. Thus, Architectural Bad Smells (ABS) may emerge, implying reduced system maintainability [12, 1]. ABS result from a set of architectural design decisions that negatively impact system lifecycle properties, such as understandability, testability, maintainability, extensibility, and reusability [1, 6, 9]. Consequently, ABS indicate possible design and implementation issues and fixing them can improve the system's quality. ABS are well-studied for single systems [12, 1, 6, 9, 8, 13]. Yet, fewer works exist for SAS [15, 21, 19, 16, 20]. Additionally, these studies do not analyze the impact of runtime variability on smell detection and evolution as the SAS adapts.

In this paper, we extend our previous work [19] to analyze the Architectural Bad Smell in self-adaptive systems at runtime. In particular, we described the feature identification process used to instrument the source code of the self-adaptive systems to detect architectural bad smells at runtime. We look for Cyclic Dependency (CD), Extraneous Connector (EC), Hub-Like Dependency (HL), and Oppressed Monitor (OM) architectural bad smells in various runtime adaptations in the Smart Home Environment (SHE) framework [17], Adasim [24], and mRUBiS [23] systems developed in Java.

Our results suggest that runtime ABS assessment is required to fully capture SAS architectural qualities because the ABS occurrences vary along each self-adaptation. In summary, this paper provides the following contributions:

1. A first study to identify architectural bad smells for SAS at runtime;
2. An analysis based on two runtime adaptations of SHE, 40 runtime adaptations of Adasim, and 16 runtime adaptations of mRUBiS, demonstrate that runtime variability affect the type and occurrence of smells found;
3. A replication package containing the results and scripts to process behavioral maps is also available:
`https://github.com/edilton-santos/BehavioralMapExtendedStudy`.

The remainder of the paper is as follows. Section 2 formally defines the Behavioral Map (BM) and presents the framework. Section 3 discusses the studied systems, and the architectural bad smells identified through the **BM** and illustrated on the SHE framework [17]. We describe our results in Section 4. Section 5 addresses the threats to validity. Section 6 presents the related work. Finally, Section 7 wraps up the paper.

2 Behavioral Map

Inspired by Dynamic Software Product Lines (DSPLs) [5, 4, 3, 17], we consider SAS adaptations as *configurations of interacting features*. In a (D)SPL, one describes features and their dependencies in Feature Model (FM) [11] and trace their realization in the code via *e.g.,*, annotations. Not all SASs are DSPLs, and FM as well as traceability of features throughout the implementation may be absent. Our BM process copes with this issue (see Section 2.2). Then, the role of a **Behavioral Map** is to capture interactions between features of a specific (re)configuration to be analyzed before it gets deployed [18]. Such configurations

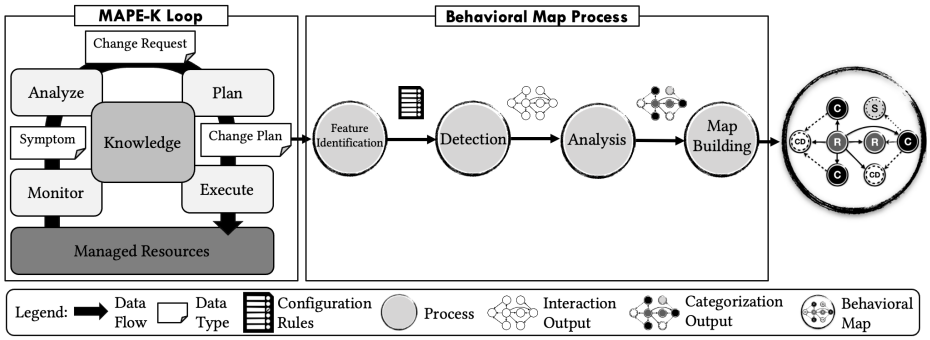

Fig. 1: Behavioral Map (BM) process overview.

are produced within an adaptation loop. We rely on the well-known MAPE-K loop (*Monitor*, *Analyze*, *Plan*, and *Execute* over a shared *Knowledge* base) proposed by IBM in [10]. We depicted it left side of Figure 1, though any type of control loop may interact with a **BM**. Thus, the **BM** needs to interact with the component responsible for defining the *Change Plan* used in the adaptation process at runtime and retrieving the configuration rules. We used the *Change Plan* of the self-adaptive system selected to create the map based on its configuration rules. This strategy was adopted because we assume that the system implements a MAPE-K loop [10] to manage the adaptation process at runtime. We thus avoid building a **Behavioral Map** for an invalid configuration. Furthermore, the Behavioral Map can look for architectural bad smells in a self-adaptive system independently of the adaptation mechanism employed in the reconfiguration process at runtime. However, to facilitate the presentation of the Behavioral Map process, we decided to use MAPE-K loop because it is more intuitive and the most used adaptation mechanism for developing SASs.

To build a **BM**, we follow the process described in Figure 1. The MAPE-K loop *monitors* continuously a set of managed resources and gathers the results in *symptoms*. Then the loop *analyses* symptoms and determines if an adaptation is necessary based on *Knowledge* (which in our case includes the DSPL feature model). If such an adaptation is necessary, it will issue a *change request* for the plan phase that will determine the appropriate configuration (a set of enabled and disabled features) to *execute* as prescribed by its *Change Plan*. The **BM** building process (right side of Figure 1) interacts with this *Change Plan* containing, besides the candidate configuration, a set of *configuration rules* noted $\mathbb{CR}$. These rules contain information on the features and their dependencies (versions, imported and exported packages) obtained via extraction (see Section 2.3). The map building process comprises the following steps: *Feature Identification*, *Detection*, *Analysis* and *Map Building*. In the following, we define the **BM** formalism and explain the **BM** building process.

2.1 Behavioral Map Definition

A **BM** is a hybrid structure, mixing structure, data, and control information about one configuration of the DSPL. Formally, a **BM** is a tuple:

$BM = (C, V, VTypes, vtype, E, ETypes, A, vattributes)$, where:

- C is a configuration, i.e. a selection of interacting features in a given planned SAS adaptation,
- $V \subseteq C$ is a set of vertices,
- $VTypes = \{\text{Core, Controller, Sensor, Actuator, Presenter}\}$,
- $vtype : V \times \mathcal{P}(VTypes) \setminus \emptyset$ is a function giving the types of a vertice. We suppose that a vertice/feature can have multiple types. For example, a feature can be core (*i.e.,* present in all configurations) and also serves as a controller,
- E is a set of edges such as $\forall e \in E,\ e = (v, v', r)$ where $v, v' \in V$ and $r \in ETypes = \{\text{Controls, Reads, Suppresses, Requires}\}$,
- A is the set of all attributes,
- $vattributes : V \times P\{A\}$ is a function giving the value of all the attributes for a given vertice.

2.2 Behavioral Map Building Process

In the remaining, we describe the **BM** process shown in the right side of the figure 1.

Feature Identification. We describe the manual process used to identify features in source code based on information available in the system's repository. The feature identification process uses the *Feature Trace* provide by Data Extractor (see Section 2.3 for details) to track features at runtime. This process is necessary because the self-adaptive systems available in Self-adaptive System Community[1] do not use a feature model to define their features. The feature identification process consists of four steps.

Step 1 - Identifying the features: We first identify the features available in the selected systems by examining articles (published in the literature), software requirements documents, architecture descriptions, and other information provided by developers in the software repository (*e.g.,* GitHub) used to describe the software requirements and implementation. These documents describe the systems, including adaptive mechanisms, applicability, test scenarios, and source code.

Step 2 - Identifying the core features in the source code: We used the feature name (or description) identified in step 1 and adaptive mechanisms (see the Table 1) implemented in the system to guide the identification of the core features in the source code. The Core features are executed in every (re)configuration of the system. In addition, we selected only the main concrete class responsible for implementing the feature behavior because the class is the

[1] `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/`

main point for the feature implementation. Consequently, we use this class to identify the hierarchy of dependencies at runtime via Data Extractor.

Step 3 - Identifying the optional features in the source code: We used the feature name (or description) and the scenario where each feature is activated to define the optional features in the source code. Also, we analyzed the source code comments used to describe the class or method implementation to support feature identification in the source code. Thus, we associated the information collected in step 1 with the source code information to find each feature. We selected the main concrete class responsible for implementing the optional feature.

Step 4 - Behavioral Map Feature Trace: The features (class) identified in steps 2 and 3 are included in the *Feature Trace* provided by the Data Extractor.

Detection. *Detection* determines interacting features using pairwise analysis [22] and their directed relationships based on the configuration rules $\mathbb{CR}$. Moreover, we assume that in the $\mathbb{CR}$, there are all features and their configuration policy (including feature dependencies) required to address a specific context at runtime. For example, the feature installation process used the constraints available in the manifest file to identify the feature and its dependencies. Besides, this process can use complementary information defined in the *Change Plan* to guide the installation, configuration, and adaptation processes at runtime.

In this context, we will use the $\mathbb{CR}$ defined in the *Change Plan* to identify the features and directions of each relationship. Thus, the Detection process selects a feature in the $\mathbb{CR}$ and identifies its dependencies based on the configuration information of the feature. Let us consider a *Feature A*, which requires loading a *Feature B* at runtime. This dependency is defined in the $\mathbb{CR}$ file and used by the Detection process to create an arrow from feature A to feature B, indicating the direction of the relationship between the features. The process repeats for each feature until all interactions are detected and created on the map.

Analysis. During the analysis stage, we further refine the interactions identified during detection in categories. We identify several relationship types ($ETypes$) as relevant to highlight runtime interaction problems. The currently supported types are: **i) Controls:** a relationship where a feature has control over another feature, but does not suppress its behavior; **ii) Suppresses:** a relationship where a feature suppresses the behavior of another one. Also, we consider as suppressed the relationship between features where one controlled feature needs to be uninstalled or unbound by its controlling feature; **iii) Requires:** a relationship in which a feature is part of another feature's implementation. In this relationship, there is no suppression or control over the feature's behavior that is part of the main feature; **iv) Reads:** This type of relationship occurs when one feature reads data produced by another feature, but there is no control or suppression of the feature's behavior.

Map Building. Based on interaction detection and analysis, we can build the Behavioral Map for a configuration of the SAS. We represent this map as a directed graph where features form the vertices and relationships form the edges.

```
1  table ← loadConfigurationRulesFile(ℂℝfile);
2  verticesOnMap ← createVerticesOnMap(table);
3  foreach vertex in verticesOnMap do
4      foreach row in table do
5          if row.name.equals(vertex.name) then
6              foreach relation in row.getAllRelationships() do
7                  if relation.relationship is not null then
8                      createEdge(vertex, relation.relationship_type, relation.featureName);
9                  end
10             end
11         end
12     end
13 end
```

Algorithm 1: Behavioral Map algorithm.

Algorithm 1 captures the whole BM building process. The algorithm begins by loading the ℂℝ file as a `table` (line 1 at algorithm 1) and instantiates the vertices (features) on the map (`createVerticesOnMap`, line 2). The next step is to look for each created vertex (feature) and identify its relationships in the *Configuration Rules* (`table`). Consequently, we create three loops, as shown lines 3, 4, and 6. The first loop selects a vertex on the map and then looks for its information in the `table` using the second loop. Line 5 checks whether each `row` of the table contains the selected vertex. Line 6 retrieves all relationships (`row.getAllRelationships()`) related to the selected vertex on the map. For each relationship, `createEdge` creates an edge in the map based on the following arguments: **i)** the vertex from which the edge starts; **ii)** the relationship type represented by the edge; **iii)** the destination vertex (`relation.featureName` in line 8). The loop on line 6 will repeat until all edges are created.

2.3 Framework Implementation

We conceived a framework to infer Behavioral Maps whose architecture is shown in Figure 2. The framework uses the Neo4J[2] platform and its Cypher[3] query language. The top-most layers, **Map Builder**, **Analyzer**, and **Interaction Detector** perform the processes defined in Section 2. In the following, we focus on the remaining elements of the framework.

The **Integration Layer (IL)** serves as an interface between the DSPL and the map building components, receiving the data used to build the map. Also, this layer defines the ℂℝ *file* data type used to build the map as follows: **i) name** is the feature name in the system; **ii) friendly_name** is friendly name of feature

[2] Neo4j - `https://neo4j.com/product/`
[3] Cypher - `https://neo4j.com/docs/cypher-manual/current/introduction/`

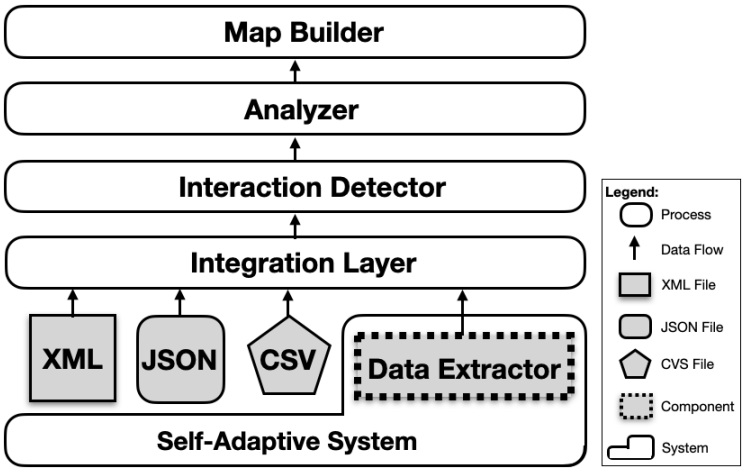

Fig. 2: Behavioral Map Architecture overview.

shown to the user; **iii) exported_packages** lists the exported packages or services offered via features; **iv) imported_packages** lists the packages used by features to compose their functionality; **v) version** represents the feature version; **vi) status** defines if the feature is active or inactive; **vii) type** defines the feature type; **viii) relationships** is a collection composed of relationship types and associated features as describe as follows: **a) relationship_type** represents the relationship type, as defined in *ETypes*; **b) feature_name** is the *feature name* associated with the *relationship_type* field. The **IL** reads data via *Data Extractor* or $\mathbb{CR}$ *file* in formats *XML*, *JSON*, or *CSV*.

The **Data Extractor (DE)** realizes the runtime integration between the *Integration Layer* and the Self-Adaptive system. The **DE** runs over the *Plan* function (see Figure 1), reading the *Change Plan* information at runtime and relating the features and $\mathbb{CR}$ after the system triggers the adaptation process. Hence, the **DE** identifies all features used and their relationships regarding the *Change Plan* configuration to be deployed. Thereafter, the **DE** builds a $\mathbb{CR}$ *file* including all involved features and sends it to the *Integration Layer*. Listing 1.1 shows a small part of the $\mathbb{CR}$ (in JSON format), created by **DE** with one feature (Presence), some properties (*e.g.,* name, status, and type), and relationships at runtime (*e.g.,* line 9).

```
1 {
2  "name":"Presence",
3  "friendly_name":"presence"
4  "exported_packages":["com.she.core.presence"],
5  "imported_packages":["com.she.core.listener"],
6  "version":"1.0.0",
7  "status":"Active",
8  "type":"Sensor",
```

```
 9   "relationships":[{"relationship_type":"Requires","
        feature_name":"Listener"}]
10  }
11  ...
```

Listing 1.1: Presence feature configuration rules.

The **DE** component performs static analysis using the WALA API[4]. Static analysis allows to identify the dependency relationships among the class hierarchy used by selected features or perform interprocedural dataflow analysis and identify relationships' types. Also, manifest files, used to install each feature of the candidate configuration before its deployment, are exploitable. The **DE** component can be implemented for all types of adaptation processes because this component receives as a parameter the features and their $VTypes$, the features implementation path in the packages, and Jar files. Also, we used these parameters to maps the relation between features and components that implements each feature. Besides, the **DE** provides a *Feature Trace* used to identify the features executed at runtime based on the features identified in the source code by the developers or researchers following the process defined in section 2.2. The *Feature Trace* gets all the information used to build the $\mathbb{CR}$ file at runtime and sends all collected information to **DE** for each monitored adaptation.

The BM framework allows to compute a graph depicting core and variable features as well as the different interactions between them (see the figures 3, 4, 5, 6, 7, and 8). Though these maps may be used for visual inspection, they mainly serve as support for further analyses thanks to the Neo4J graph database[5].

3 BM-Based ABS Detection

This section presents the SAS under study and describes the architectural bad smells that the Behavioral Map can identify. Furthermore, we describe the process for identifying each architectural smell and discuss its impact on the SAS' architecture.

3.1 SAS under study

We applied our **BM** framework on SHE [17], Adasim [24] and mRUBiS [23] systems, all written in Java programming language. The motivation for these choices also relies on the fact that these systems have different adaptive mechanisms, and their description and implementation are available. Furthermore, the last two systems were selected as part of a previous study on ABS for SAS [15]. Table 1 shows the main characteristics of each selected system as follows: i) System - the name of the system; ii) Architectural Model - The type of architectural model used to implement the system under evaluation; iii) Adaptive Mechanisms

[4] WALA - `https://github.com/wala/WALA`
[5] `https://neo4j.com/product/neo4j-graph-database/`

Table 1: Systems used in this study.

System	Architectural Model	Adaptive Mechanisms	Application Domain
SHE	Publish-Subscribe	MAPE-K	Internet Of Things
Adasim	Agent-based	Parameter-based routing algorithm	Automated traffic routing
mRUBiS	Architectural model-based	Architecture-based MAPE-K Event-Condition-Action State based feedback loop	Marketplace

- The mechanisms used to trigger the adaptations at runtime; iv) Application Domain - Information about the application domain of the systems selected in this study. These characteristics are essential to help us understand the impact of each smell in the selected systems. We present each selected system and its configurations under evaluation in the following.

SHE is a smart home system that uses the MAPE-K loop to identify changes (such as a new sensor being plugged in) and make the appropriate changes to the dashboard (*e.g.,* display data coming from that sensor). The SHE core is composed by *Manager*, *Listener*, *Loader*, *Installer*, and *Presentation Layer*. These layers are responsible for controlling the adaptation, communication, and data presentation at runtime. Also, we included four optional features as follows: i) ***Luminosity*:** used to read data from the luminosity sensor; ii) ***Presence*:** used to read data from the presence sensor; iii) ***lampController*:** responsible for controlling Lamp feature's behavior using the information read from *Luminosity* and *Presence* features; iv) ***Lamp*:** an actuator used to switch on and off lights based on the *lampController* feature's data. This configuration of SHE is depicted Figure 3. Also, we analyzed a second version of the SHE that uses the same features described above and includes the *water*, *climateController*, *temperature*, and *airConditioner* features.

Adasim is a simulator for the Automated Traffic Routing Problem (ATRP)[6], implemented as an agent-based system [24, 15]. The system is composed of six abstract components: i) a map; ii) vehicles; iii) agents - make routing decisions; iv) sensors; v) uncertainty filters - utilized to control the noise and other sources of uncertainty in the sensor; and vi) data privacy policies - used by vehicles and streets to restrict part or all information about themselves from sensors [24]. The system employs adaptive mechanisms to deal with the scalability problems and the unpredictable changes in the environment, for instance, an accident.

[6] `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-atrp/`

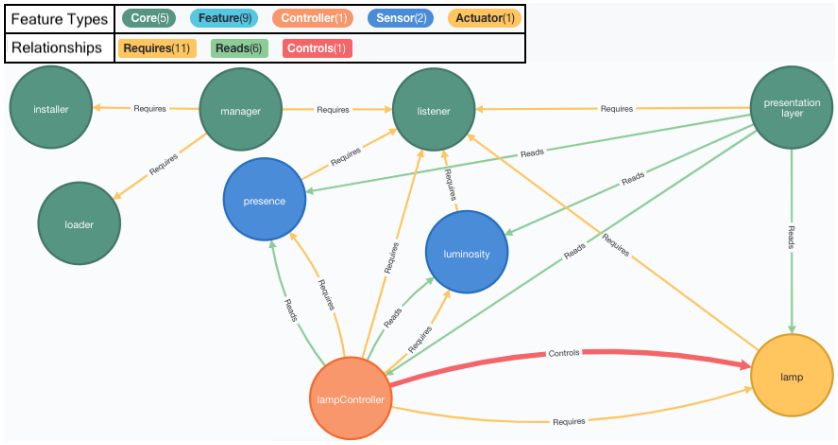

Fig. 3: Behavioral Map (BM) for one SHE configuration.

mRUBiS is a marketplace based on RUBiS [14], comprising 18 components and can arbitrarily host many shops. These shops manage items, users, auctions/purchases, inventory, and authenticate users. mRUBiS[7] allows different adaptive mechanisms [23], as showed in table 1.

3.2 Identifying Architectural Bad Smells

Table 2: Selected Architectural Bad Smells for Self-Adaptive Systems.

Smell Name	Detection
Cyclic Dependency (CD) [2]	Full
Extraneous Connector (EC) [8]	Full
Hub-Like Dependency (HL) [2, 15]	Full
Oppressed Monitors (OM)[21]	Partial

While ABS catalogs exist in the literature [2, 8], their role in self-adaptive architectures is less known [15, 21]. Table 2 presents a list of smells we believe to be relevant for assessing self-adaptive architecture as well as their level of support through the **BM**. For each of them, we briefly describe how they can be identified via the **BM**, and we provide a short discussion on their impact. We also provide a replication package on GitHub[8] with a tutorial to configure the Neo4J platform, CR files, and the scripts used to create the map and analyze ABS.

[7] `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/mrubis/`
[8] `https://github.com/edilton-santos/BehavioralMapExtendedStudy`

Cyclic Dependency [2]: This smell occurs when two or more components depend on each other directly or indirectly [2]. Components involved in a dependency cycle can hardly be released, maintained, or reused in isolation [7].

Identification Guidelines. We determine cycles in the sub-graph of the BM formed by the features and the relationships of type *Requires* using a Depth-First traversal strategy.

Discussion. Based on relationship categories, other forms of cyclic dependencies may be uncovered, such as control ones which may cause concurrent accesses to resources and/or deadlocks.

Extraneous Connector (EC) [8]: This smell happens when two connectors of different types are used to link a pair of components [8]. This paper focuses on only the impact of combining procedure call and event connectors (*e.g.,* communication via publish-subscribe).

Identification Guidelines. The automatic identification of extraneous connectors proceeds by analyzing paths between pairs of vertices in the **BM**. In a complementary way, a designer can visually identify EC smells on the **BM**. The *lampController* (Figure 3) uses two types of connectors to connect with the features *Presence*, *Luminosity*, and *Lamp*. The *lampController* uses the *Listener* (*Publish-Subscribe* client to implement the Reads edge) and procedure call communication (represented by the Requires edge) with *Presence*, *Luminosity*, and *Lamp*.

Discussion. This smell increases the coupling between features of the DSPL, negatively impacting its variability, and thus its adaptability [9]. However, a direct connection may be justified for concurrent operation [8] and may increase the system's resiliency in case of failure of the publish-subscribe architecture.

Hub-Like Dependency (HL): This smell appears when a component has (incoming or outgoing) dependencies with a large number of other abstractions (*e.g.,* other components) or concrete classes [2, 15].

Identification Guidelines. Thanks to its graph structure, the **BM** allows to automatically compute the in/out-degree (number of incoming or outgoing edges) for each vertex (feature). Features having high in/out-degrees are subjected to the HL smell. In Figure 3, we see that the *Listener* feature is subjected to the HL smell since it is involved in most of the *Requires* relationships of the **BM**. Besides, if a feature has only many outgoing Requires edges, it is a Hub type called *Overreliant Class* [2].

Discussion. The presence of the HL smell in the Listener feature is motivated by the publish-subscribe architecture adopted by the SHE framework. The *Listener* centralizes all the communication processes in this software architecture and works as a communication broker. It is therefore acceptable in this case [2, 7]. However, hubs form points of attention in case of failure.

Oppressed Monitors [21] (OM): According to [21], this smell is characterized by a set of monitors (retrieving information from sensors) independent from each

other that are managed with the same data polling rate and predefined execution order, yielding sub-optimal data acquisition and failure of subsequent monitors if one monitor in the sequence fails.

Identification Guidelines. Fully identifying this smell involves delving into the source code and getting information about polling rate since sequencing of sensor calls is not present on the map. Yet, if several sensors are controlled by the same controller, the map can help locating the features to look for this smell.

Discussion. In some cases, this smell is acceptable, especially when there are simple monitors with similar polling rates [21]. However, this smell limits the adaptability and resiliency of the system, which are important criteria for self-adaptive systems.

These examples illustrate the two complementary usages of the **BM**. First, the BM is a formal model amenable to automated detection of smells using graph algorithms. Second, visual representations help designers and engineers to visualize runtime configurations.

Identification Process: The **BM** framework thus comes with dedicated algorithms to identify ABS [19], as described in section 3.2. These algorithms are implemented via the Cypher[9] language, allowing to query the graph. We used provided queries to identify CD, EC, HL, and OM on the map created for the SASs under study. For example, listing 1.2 shows how to compute cyclic dependencies on the map. All queries used in this study are available on GitHub[10].

```
1 MATCH (f:Feature)-[:Requires]->(f2:Feature)-[:Requires]->(f)
2 OPTIONAL MATCH (f2)-[:Requires]->(f3:Feature)-[:Requires]->(
      f)
3 RETURN f, f2, f3
```

Listing 1.2: Cypher query used to look for CD in the BM.

4 Results

The following sections describe the results and discuss the reasons behind each architectural smell identified in the self-adaptive systems under study.

4.1 SHE Framework Results

The **SHE Framework** performed two self-adaptations and activated 22 features at runtime, nine in the first adaptation and 13 in the last adaptation. Table 3 presents in detail the features involved in ABS during the SHE Framework adaptations. The *listener* is involved in HL smell in both adaptations, but the number of outgoing increases in the second adaptation. This situation occurred

[9] Cypher - `https://neo4j.com/docs/cypher-manual/current/introduction/`
[10] `https://github.com/edilton-santos/BehavioralMapExtendedStudy`

Table 3: ABSs identified adaptation 1 and 2 of the SHE.

		Adaptation 1			Adaptation 2		
Feature Name	**Feature Type**	**EC**	**HL**	**OM**	**EC**	**HL**	**OM**
listener	Core	Yes (6)			Yes (10)		
lampController	Optional	Yes (3)		Yes	Yes (3)		Yes
climateController	Optional				Yes (2)		

because the *water*, *climateController*, *temperature*, and *airConditioner* features were activated at runtime, increasing the number of the *Requires* relationships on the *listener* feature, as shown in Figure 4.

Fig. 4: Behavioral Map for SHE in adaptation 2.

Also, the BM identified *lampController* as involved in EC and OM smells in both adaptations. The EC smell occurred because the *lampController* uses the *listener* (the communication broker) and procedure call to exchange messages with *presence*, *luminosity*, and *lamp*. The procedure call is represented as the relationship *Requires* or *Controls* on the BM, as illustrated in Figure 4. The *Requires* relationships among *lampController* and *presence*, *luminosity*, *lamp* represent an architectural bad smell.

The BM identified the *lampController* and *presentation layer* as a possible OM smell. However, after analyzing the source code together with the SHE Framework developers, we identified that only the *lampController* uses the same data polling rate and predefined execution order to retrieve data from the sensors. Thus, only *lampController* feature was classified as OM smell. Finally, the *climateController* feature activated in adaptation two was classified as EC smell. While the BM supports the identification of potential OM smells, manual source code analysis is necessary to eliminate false positives.

4.2 Adasim Results

The Adasim system was executed using two different parameter files because we identified two adaptation modes: *QLearningRoutingAlgorithm* and *AdaptiveRoutingAlgorithm*.

Table 4: ABSs identified in adaptation 1 and 2 of the Adasim - QLearningRoutingAlgorithm.

		Adaptation 1		**Adaptation 2**	
Feature Name	**Feature Type**	**CD**	**HL**	**CD**	**HL**
TrafficSimulator	Core	Yes		Yes	
RoadSegment	Core	Yes	Yes (13)	Yes	Yes (12)
Vehicle	Core	Yes	Yes (14)	Yes	Yes (13)
VehicleManager	Core	Yes		Yes	
RoadVehicleQueue	Core	Yes		Yes	
AdasimMap	Core	Yes		Yes	
QLearningRoutingAlgorithm	Optional	Yes			
SimulationXMLBuilder	Core		Yes (9)		Yes (9)

Adasim QLearningRoutingAlgorithm. Adasim performed 13 self-adaptations and activated 18 features at runtime. However, we identified that the variability of the features at runtime only triggered different numbers of ABS detected between adaptations one and two. Such behavior was observed because Adasim did not enable/disable other features (after adaptation two), which may add new ABS at runtime. It means that the system continued executing the adaptations process using the features and data produced by each loop until it completed its adaptation cycles.

Table 4 presents in detail the features involved in ABS during the two first adaptations. The *QLearningRoutingAlgorithm* is an optional feature involved in CD only in adaptation one with the feature *RoadSegment*, and *Vehicle*, as shown in Figure 5. Such figures show all features involved in CD, the features in green

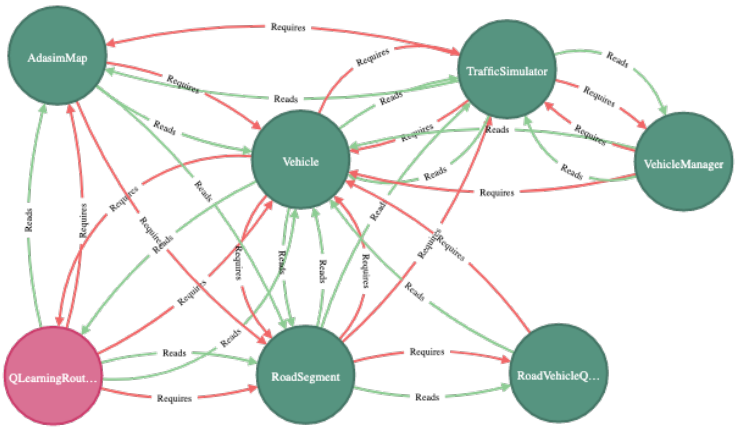

Fig. 5: CD identified in Adasim QLearningRoutingAlgorithm in adaptation 1.

are core features and the optional feature in pink. The relationship defined as Requires amongst features in CD indicates that all features involved can hardly be released, maintained, or reused in isolation. Thus, if the developers decide to reuse the feature *Vehicle*, they should reuse all features presented in Figure 5.

Nevertheless, the absence of the *QLearningRoutingAlgorithm* (in adaptation two) reduces the numbers of dependency in the features *RoadsSegment* and *Vehicle* involved in HL, see Table 4. This situation occurred because *RoadSegment* and *Vehicle* are not sharing *QLearningRoutingAlgorithm* in adaptation two. Also, performing evolutionary or corrective maintenance on the *RoadSegment* and *Vehicle* features is an arduous task, as poorly planned maintenance can trigger unexpected behavior in the system, like bugs. Moreover, a hub (as *RoadSegment* and *Vehicle*) with a mixture of ingoing/outgoing dependencies could be a problem because of its lack of architectural logic [7]. These aspects negatively impact system maintenance and reusability. In addition, the *SimulationXMLBuilder* feature has been identified as HL. Thus, we have identified three features involved in HL, as shown in Figure 6.

Adasim AdaptiveRoutingAlgorithm. The Adasim executed 27 self-adaptations and activated 20 features at runtime. We observed that the variability of the features at runtime impacted the numbers of ABS detected between adaptation 1 and 2, as identified in the Adasim QLearningRoutingAlgorithm. Table 5 presents the ABS identified during adaptations 1 and 2. Additionally, it is possible to observe that the number of CD identified increase or decrease depending on the number of optional features required in each adaptation process. This situation also impacts the number of HL identified in each adaptation, mainly because the features identified as CD and HL concentrated on the core features. Also, there is a strong relation of dependency among them at runtime. Thus, we detected that the *Vehicle* feature identified as HL in Adaptation 1 was not identified in Adaptation 2. Such a situation occurred because the optional fea-

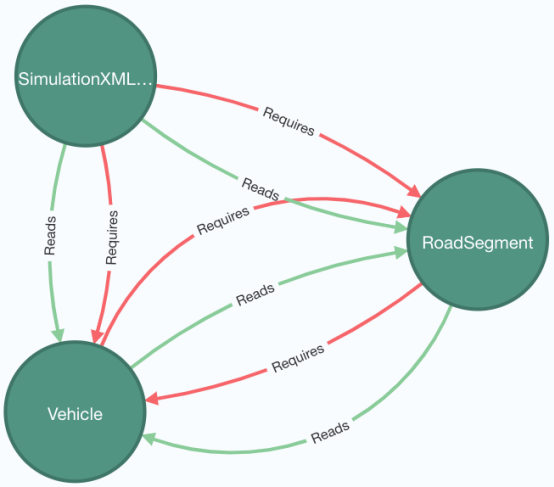

Fig. 6: Features involved in HL identified in Adasim.

tures *AdaptiveRoutingAlgorithm*, *QLearningRoutingAlgorithm*, and *Lookahead-ShortestPathRoutingAlgorithm* are not used in adaptation 2. Consequently, the BM identified in adaptation 2 the *RoadSegment* feature as a new HL.

However, we did not identify the Extraneous Connector and Oppressed Monitors smells in Adasim because the system does not use publish-subscribe architecture or loops to collect data in the sensors.

4.3 mRUBiS Results

The mRUBiS system is divided into self-healing and self-optimization versions. However, during the feature identification process, we identified four versions of mRUBiS: i) self-healing with adaptation mechanism Event-Condition-Action (ECA) feedback loop is composed of 22 features; ii) self-healing with adaptation mechanism State-Based Feedback Loop (SBFL) is composed by 18 features; iii) self-healing with adaptation mechanism MAPE-K is composed of 22 features, and iv) self-optimization with adaptation mechanism MAPE-K is composed by 27 features.

mRUBiS self-optimization: Figure 7 depicts the first configuration of mRUBiS self-optimization with one optional feature (in pink). We started looking for ABS in the system based on this configuration. The BM identified the *SelfOptimizationConfig*, *MRubisModelQuery*, and *EventBasedMapeFeedbackLoop* as HL in four adaptation loops. Thus, these features are core used in all configurations of mRUBiS self-optimization. We observed in the *SelfOptimizationConfig* a decrease in the numbers of dependencies used in the second adaptation. This situation occurred because the feature is responsible for adding the validators and other parameters for self-optimization to the simulator. However, the number of validators used at runtime decreases, impacting the dependencies identified.

Table 5: ABSs identified in adaptation 1 and 2 of the Adasim AdaptiveRoutingAlgorithm.

		Adaptation 1		**Adaptation 2**	
Feature Name	**Feature Type**	**CD**	**HL**	**CD**	**HL**
TrafficSimulator	Core	Yes		Yes	
RoadSegment	Core	Yes		Yes	Yes (13)
Vehicle	Core	Yes	Yes (17)	Yes	
VehicleManager	Core	Yes		Yes	
RoadVehicleQueue	Core	Yes		Yes	
AdasimMap	Core	Yes		Yes	
AdaptiveRouting Algorithm	Optional	Yes			
QLearningRouting Algorithm	Optional	Yes			
LookaheadShortest PathRoutingAlgorithm	Optional	Yes			
SimulationXMLBuilder	Core		Yes (11)		Yes (11)

The *MRubisModelQuery* and *EventBasedMapeFeedbackLoop* maintain the same numbers of dependencies in all adaptations. Also, the BM framework did not identify other types of ABS during the adaption loop.

mRUBiS self-healing: The BM does not identify ABS in the self-healing version with adaptation mechanism ECA and SBFL after four reconfiguration processes at runtime. The BM identified one instance of HL in the core feature *StateBasedMapeFeedbackLoop* in four adaptations loops to mRUBiS self-healing version with adaptation mechanism MAPE-K. The feature is the main entry point to other features such as *Monitor*, *Action*, *Plan*, *Execute*, *SelfHealingConfig*, *SelfHealingScenario*, and *MRubisSelfHealingUtilityFunction*. Also, the knowledge is captured in the model described in CompArch [23] language, provided by the framework CompArch implemented outside the mRUBiS implementation. This model is utilized as a parameter on the feature *StateBasedMapeFeedbackLoop* to validate the self-healing issues at runtime. Thus, the HL identified is a feature of the architecture instead of an issue. This situation happened because the *StateBasedMapeFeedbackLoop* has been chosen as a controlled entry point to separate the adaptive mechanism (MAPE-K) logically from the self-healing configuration (implemented via *SelfHealingConfig*). We can observe this situation in Figure 8 through the relationship between *StateBasedMapeFeedbackLoop* (highlighted in red) and *SelfHealingConfig* (highlighted in blue). Also, Figure 8 presents all features available in adaptation 1 of the mRUBiS Self-Healing MAPE-K loop. The features *CF1_ Injector* (in pink) and *LightWeightRedeployComponent* (in yellow) are optional features activated at runtime.

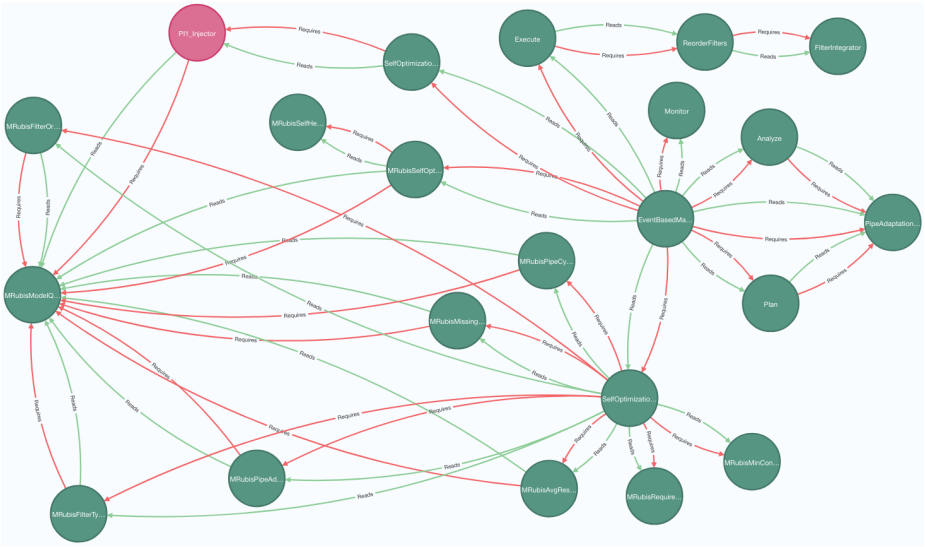

Fig. 7: Behavioral map of the first configuration of mRUBiS Self-Optimization.

5 Threats to Validity

As for any empirical study, we consider threats to the internal validity of results themselves or their generalization.

5.1 Internal Validity

The absence of a feature model and feature annotations in the source code may reduce the precision of the feature identification process in the source code. To mitigate this threat, we used the Eclipse IDE[11] tool to verify the feature implementation and to debug the systems' source code to check the execution of each feature identified using the process defined in Section 2.2. Also, the systems under study provide a log system that we used to check whether the main class used to implement the features identified using our methodology were present in the system log. Thus, we checked whether each core or optional feature was correctly identified in the source code.

5.2 External Validity

Our results may not generalize to all SAS since we selected only three systems in our study. Additionally, it is impossible to run all possible system adaptations or estimate their number. We selected systems with different architectural models, adaptation mechanisms, and application domains, as presented in Table 1. This

[11] Eclipse IDE - `https://www.eclipse.org/downloads/packages/`

Fig. 8: Behavioral map of the first configuration of mRUBiS Self-Healing MAPE-K loop.

diversity contributes to the mitigation of this threat. In this study, our goal was to reveal and explain the existence of the runtime architectural bad smells using the Behavioral Map. We left for future work with a more quantitative assessment.

6 Related Work

We found two works dedicated to the identification of ABS in self-adaptive systems. The first study [15] relies on the Arcan [7] tool to identify ABS in 11 self-adaptive systems. Arcan creates a graph database with the structure of classes, packages, and dependencies of the analyzed project, allowing the execution of algorithms on the graph to detect the ABS at design time. Our approach also uses a graph for ABS detection, but there are two differences: i) we create a map for each SAS configuration identified at runtime; and ii) we identify the ABS at the level of features defined in the system's feature model. Thus, to analyze the architecture, we associate the features defined in the model with the structure of classes, packages, and dependencies implemented in the source code. This process allows us to relate a feature to its implementation. Our work in progress involves the comparison of Acran and the BM for runtime smell detection [20].

The second study [21] presents two new ABSs specific to self-adaptive systems: the Obscure Monitor and the Oppressed Monitors. Also, it defines the algorithms to identify each smell at design time. To validate the proposed smells, the authors identified the proposed smells in 8 SASs in the manual and discussed how to refactor the system affected for those smells. We believe that our work on smells identification at runtime may uncover new ABS specific to SAS.

7 Concluding Remarks

In this paper, we made a case for assessing architectural bad smells (ABS) for self-adaptive systems (SAS) at runtime using the Behavioral Map (BM). We selected three SAS (SHE Framework, Adasim, and mRUBiS) and performed runtime smell detection on several systems reconfigurations. Our results showed that some ABS appear only in a specific system configuration or architecture. For instance, the EC and OM smell appear in publish-subscribe architecture, as used in SHE Framework. Also, we observed that the type and amount of ABS found in the SAS depend on the configuration analyzed at runtime. For instance, in Adasim AdaptiveRoutingAlgorithm, the Behavioral Map found nine CD and three HL smells in the first adaptation, but the BM found six CD smells in the second. We could explain this variation by binding and unbinding certain runtime features. Thus, the Behavioral Map framework offers interesting support for assessing the architectural qualities of a given runtime adaptation. However, instrumenting the systems for runtime ABS identification requires expertise and time because the core and variable features are not documented.

We envision three future works: i) we would like to conduct an empirical study to investigate differences between smells one detects at design time and smells occurring at runtime in self-adaptive systems; ii) we would like to reduce the cost of engineering involved in analyzing SAS at runtime. In particular, we will design a dedicated ABS tool operating at the bytecode level, easing runtime analyses; iii) we will generalize our findings by assessing more self-adaptive systems.

Acknowledgements Edilton Lima dos Santos is funded by a CERUNA grant from the University of Namur. Sophie Fortz is supported by the FNRS via a FRIA grant. Gilles Perrouin is an FNRS Research Associate.

References

1. de Andrade, H.S., Almeida, E., Crnkovic, I.: Architectural bad smells in software product lines: An exploratory study. In: Proceedings of the WICSA 2014 Companion Volume. pp. 1–6 (2014)
2. Azadi, U., Fontana, F.A., Taibi, D.: Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 88–97. IEEE (2019)
3. Baresi, L., Quinton, C.: Dynamically evolving the structural variability of dynamic software product lines. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 57–63. IEEE Press (2015)
4. Bencomo, N., Sawyer, P., Blair, G.S., Grace, P.: Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: SPLC (2). pp. 23–32 (2008)
5. Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., Hinchey, M.: An overview of dynamic software product line architectures and techniques: Observations from research and industry. Journal of Systems and Software **91**, 3–23 (2014)

6. Fontana, F.A., Avgeriou, P., Pigazzini, I., Roveda, R.: A study on architectural smells prediction. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 333–337. IEEE (2019)
7. Fontana, F.A., Pigazzini, I., Roveda, R., Zanoni, M.: Automatic detection of instability architectural smells. In: IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 433–437. IEEE (2016)
8. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: 13th European Conference on Software Maintenance and Reengineering. pp. 255–258. IEEE (2009)
9. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In: International conference on the quality of software architectures. pp. 146–162. Springer (2009)
10. IBM: An architectural blueprint for autonomic computing. IBM White Paper **31**, 1–6 (2006)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., CMU-SEI (1990)
12. Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)
13. Mumtaz, H., Singh, P., Blincoe, K.: A systematic mapping study on architectural smells detection. Journal of Systems and Software (2020)
14. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 33–42. IEEE (2012)
15. Raibulet, C., Fontana, F.A., Carettoni, S.: A preliminary analysis of self-adaptive systems according to different issues. Software Quality Journal pp. 1–31 (2020)
16. Raibulet, C., Fontana, F.A., Carettoni, S.: Sas vs. nsas: Analysis and comparison of self-adaptive systems and non-self-adaptive systems based on smells and patterns. In: ENASE. pp. 490–497 (2020)
17. Santos, E., Machado, I.: Towards an architecture model for dynamic software product lines engineering. In: IEEE International Conference on Information Reuse and Integration (IRI). pp. 31–38. IEEE (2018)
18. dos Santos, E.L.: Stars: Software technology for adaptable and reusable systems. In: Proceedings of the 25th International Systems and Software Product Line Conference (SPLC). ACM (2021)
19. dos Santos, E.L., Fortz, S., Perrouin, G., Schobbens, P.Y.: A vision to identify architectural smells in self-adaptive systems using behavioral maps. In: 15th European Conference on Software Architecture (ECSA 2021). p. 1. CEUR Workshop Proceedings (2021)
20. dos Santos, E.L., Schobbens, P.Y., Perrouin, G.: Featured scents: Towards assessing architectural smells for self-adaptive systems at runtime. In: 19th International Conference on Software Architecture. pp. 71–74. IEEE (2022)
21. Serikawa, M.A., Landi, A.d.S., Siqueira, B.R., Costa, R.S., Ferrari, F.C., Menotti, R., De Camargo, V.V.: Towards the characterization of monitor smells in adaptive systems. In: X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). pp. 51–60. IEEE (2016)
22. Soares, L.R., Meinicke, J., Nadi, S., Kästner, C., de Almeida, E.S.: Varxplorer: Lightweight process for dynamic analysis of feature interactions. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems. pp. 59–66 (2018)

23. Vogel, T.: mrubis: An exemplar for model-based architectural self-healing and self-optimization. In: Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems. pp. 101–107 (2018)
24. Wuttke, J., Brun, Y., Gorla, A., Ramaswamy, J.: Traffic routing for evaluating self-adaptation. In: 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 27–32. IEEE (2012)