STARS

Lima dos Santos, Edilton

Published in:
SPLC '21

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Link to publication

Citation for pulished version (HARVARD):
Lima dos Santos, E 2021, STARS: software technology for adaptable and reusable systems. in M Mousavi & P-Y Schobbens (eds), *SPLC '21: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B.* vol. B, ACM Press, Leicester, pp. 13-17.

STARS: Software Technology for Adaptable and Reusable Systems

Edilton Lima dos Santos
edilton.limados@unamur.be
PReCISE, NaDI,
Faculty of Computer Science, University of Namur
Namur, Belgium

ABSTRACT

Dynamic Software Product Lines (DSPLs) engineering implements self-adaptive systems by dynamically binding or unbinding features at runtime according to a feature model. However, these features may interact in unexpected and undesired ways leading to critical consequences for the DSPL. Moreover, (re)configurations may negatively affect the runtime system's architectural qualities, manifesting architectural bad smells. These issues are challenging to detect due to the combinatorial explosion of the number of interactions amongst features. As some of them may appear at runtime, we need a runtime approach to their analysis and mitigation. This thesis introduces the Behavioral Map (BM) formalism that captures information from different sources (feature model, code) to automatically detect these issues. We provide behavioral map inference algorithms. Using the Smart Home Environment (SHE) as a case study, we describe how a BM is helpful to identify critical feature interactions and architectural smells. Our preliminary results already show promising progress for both feature interactions and architectural bad smells identification at runtime.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; • **Computer systems organization → Self-organizing autonomic computing**.

KEYWORDS

Software Product Line Engineering, Dynamic Software Product Lines Engineering, Self-adapting system, Software architecture, MAPE-K loop, Software testing

ACM Reference Format:
Edilton Lima dos Santos. 2021. STARS: Software Technology for Adaptable and Reusable Systems. In *25th ACM International Systems and Software Product Line Conference - Volume B (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3461002.3473068

1 INTRODUCTION AND MOTIVATION

Self-adaptive systems change their behavior depending on environmental changes and reconfiguration plans and goals. Dynamic Software Product Line (DSPL) engineering implements self-adaptive systems (SAS) by dynamically enabling or disabling features at runtime as prescribed by a feature model [5]. Consequently, the DSPLs validation process is complex because the number of possible configurations grows exponentially with the number of features, and features may interact in both unexpected and undesired ways [2, 8, 28]. Such problems are further amplified if the system can update itself (for example, by downloading new features to interface with a sensor newly plugged into the system) [6]. The feature interaction problem is well-studied for systems where features are bound at specification or design time [1–3, 7, 8, 15, 19], but runtime interactions are less explored [6, 24].

Adaptations at runtime may affect architectural qualities and properties. For instance, the (re)configuration process may add a new architectural solution in an inappropriate context, combine architectural fragments with undesirable behaviors, or apply architectural abstractions at the wrong granularity level via new features loaded at runtime. In these circumstances, Architectural Bad Smells (ABS) may appear, implying reductions in system maintainability [9, 18]. ABS are a set of architectural design decisions that negatively impact the system's properties (understandability, testability, maintainability, extensibility, and reusability) [9, 11, 14]. However, an ad-hoc literature review identified only two studies exploring ABS in SAS at design time [25, 27]. In addition, there is a gap in evaluating the impact or identification of ABS in SAS at runtime [20].

This thesis advocates a model-based approach to the aforementioned issues. We tackle the feature interaction and architectural issues (*e.g.,* ABS) by introducing the Behavioral Map (**BM**) formalism, a directed graph capturing interactions defined in the feature model but also capturing control and data flow interactions inferred from the candidate reconfiguration implementation. Besides, DSPL engineering generally represents the features of a system family (their commonalities and variabilities) and their relationships. Such a model has a high abstraction level and is used as a starting point for the feature selection and product derivation in design time or runtime. However, such a model does not capture control and data flow interactions inferred from the SAS. This information is essential to identify unpredictable behavior or unpredictable relationships among features at runtime.

Thus, we envision that **BM** will support the feature interaction issues identification, ABS identification, and testing prioritization based on the analysis of a runtime configuration. Furthermore, we can include the **BM** in the system adaptation process to verify the selected configuration before its deployment. Consequently, the system will not execute the faulty configuration and will keep the last valid configuration until a new one gets computed. STARS contributions are: i) usage of the **BM** to derive an ABS catalog dedicated to SAS; ii) the exploitation of identified feature interactions to derive test generation and selection algorithms for the configuration under study, notably when new features emerge via hot-plugging mechanisms; and finally, iii) evaluation of map inference mechanisms on several case studies. This evaluation will allow the performance assessment of our inference and prioritization algorithms.

The rest of the paper is organized as follows: Section 2 states the research questions that we address in this thesis. Section 3 shows the methodology, our approach, and threats to validity. In Section 4, we present our results on map inference and architectural bad smell identification. Finally, in Section 5, we recap our progress and provide a monthly work plan.

2 RESEARCH QUESTIONS

In this thesis, we aim to answer the following research questions (RQ):

RQ1 How to model DSPL architectures at runtime? We seek to understand how to model DSPL architectures based on their configuration at runtime.

RQ1.1 What are the necessary concepts needed to identify architecture issues? We aim at discovering the concepts required for architecture issues analyses.

RQ1.2 How to infer such a model? This question covers the techniques able to learn our model automatically from running artifacts describing the DSPL configuration.

RQ2. What are the validation means the Behavioral map can support? We want to evaluate empirically the benefits of our model for feature interactions and ABS identification as well as test prioritization.

3 RESEARCH METHODOLOGY AND APPROACH

3.1 Research Methods

We have defined a research methodology divided into four steps, as described below.

Step 1: We conducted a literature review that aims to identify what strategies are used to test dynamically adaptive systems and raise evidence on techniques and tools that achieve high defect detection even in unpredictable contexts. As a result, we found no readily applicable technique able to perform defect detection in unpredictable runtime contexts.

Step 2: We conducted an *ad-hoc* literature review to identify which types of ABS can occur in SAS and how to identify each bad smell. There are ABS catalogs in the literature [4, 13], but their role in self-adaptive architectures is less known, and we identified only two works in this case [25, 27].

Step 3: We defined a new formalism and a framework implementation that allows the inference of behavioral map models at runtime. The framework uses static analysis implemented via Call Graph and the Context-Flow Analysis (CFA) algorithms to support the data extraction process. Also, we selected the Neo4J platform

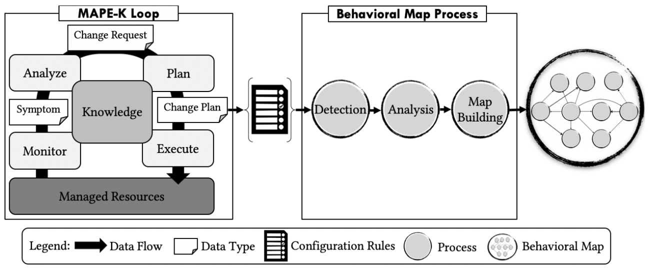

Figure 1: Behavioral Map (BM) process overview.

[21] and the Neo4j APOC Library [22] to implement the graph analyses, map visualization, and storage of behavioral maps. Neo4J is a graph database management system that supports analyses via the Cypher query language [23]. Cypher allows us to extract information about the feature interactions and ABS using pre-defined queries.

Step 4: We will conduct empirical experiments to assess that the Behavioral Map supports bad smell identification and feature interaction analyses. In addition, we will evaluate the prototype developed on a small scenario in the Smart Home domain based on the SHE system [10, 26]. Also, we plan to conduct other advanced evaluations using different SAS types available in Software Engineering for Self-Adaptive Systems website [1] to check the **BM** feasibility.

3.2 Proposed Approach

This thesis intends to answer the research questions by offering a Behavioral Map definition and architecture specification to identify feature interaction problem and ABS at runtime. A **BM** maps the interactions and influences that a feature has on other features in a specific configuration for a given runtime context, *i.e.,* a *context configuration*. Consequently, 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* selected by the SAS to create the map based on its configuration rules. Such a strategy was adopted because we assume that the system implements a MAPE-K loop [16] to monitor, analyze, plan and execute the adaptation process at runtime according to the application feature model. We thus avoid building a **BM** for an invalid configuration.

3.2.1 *Behavioral Map Building Process*

To build a **BM**, we follow the process described in Figure 1. The MAPE-K loop *monitors* continuously a set of managed resources and correlates them into *symptoms*. Then the **Analyze** loop analyzes the *symptoms* to determine whether an adaptation is necessary based on *knowledge* (including the DSPL feature model). If an adaptation is needed, it will create a *change request* for the **Plan** phase that will determine the appropriate configuration (a set of enabled and disabled features) to **execute** according to the *change plan*. The **BM** process (right part of Figure 1) takes as input this *change plan* containing the candidate configuration and a set of *configuration rules* noted $\mathbb{CR}$. The **BM** process comprises the following: **i)** *Detection* determines

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

interacting features using pairwise analysis [28] and their relationships based on the $\mathbb{CR}$; **ii)** *Analysis* further classifies interactions in categories according to the *ETypes* set (see Section 4.1). A feature can thus *control*, *read* some information from, *suppress* the behavior of, or *require* an other feature; **iii)** Finally, the *Map Building* build the map.

3.3 Threats to Validity

There are some general threats to validity that have to be considered, threatening the internal validity of results themselves or their generalization.

Internal. There are specific architectural styles that can impede the precision of our map inference and analysis algorithms. When multiple components exchange event messages via a shared event bus (*e.g., publish-subscribe* architectures) [14], interactions are more challenging to identify [9]. To mitigate such a threat, we analyze the class hierarchy that composes each feature and its configuration instruction. Thus, we identified which features depend on the event bus to establish communication with other features that composes the system, regardless of the communication topics used at run time.

External. It is not easy to find real-world SAS based in the MAPE-K loop with open source code and a distinct feature model. Therefore, our main issue for conducting experiments to evaluate the **BM** framework is finding suitable case studies. We used a smart home system developed for academic study [10, 26] to test our **BM** framework in a small scenario. Thus, evaluation results may not generalize to all real-world SAS. To address this threat, we plan to use the self-adaptive systems exemplars available in Software Engineering for Self-Adaptive Systems community website.

4 PRELIMINARY RESULTS

The results in this section are early results addressing the **BM** approach (definition, algorithm, and framework) answering **RQ1** and the preliminary outcome of the research as Behavioral Map example and Architectural Smell Identification answering **RQ2**.

4.1 Behavioral Map Definition

A **BM** can be seen as 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 valuation of features from the feature model,
- $V \subseteq C$ 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 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.

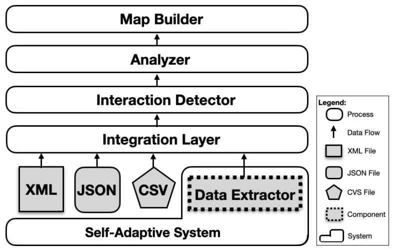

Figure 2: Behavioral Map Architecture overview.

4.2 Behavioral Map Algorithm

The **BM** building process is summarized by Algorithm 1. It starts from a `table` loaded by the `loadConfigurationRulesFile` procedure (line 1 at listing 1) and creates the vertices (features) on the map (`createVerticesOnMap`, line 2). It then looks for each created vertex (feature) and identify its relationships in the *Configuration Rules* (`table`). 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 for each `row` of the table whether it 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 last loop (line 6) will repeat until all edges are created.

```
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.

4.3 Behavioral Map Framework

Figure 2 shows the implemented framework to infer behavioral maps whose architecture. The framework uses the Neo4J platform [21] and its Cypher query language [23]. We defined the top-most layers (**Map Builder**, **Analyzer**, and **Interaction Detector**) processes in Section 3.2.1. In the following, we focus on the remaining elements of the framework. The **Integration Layer (IL)** provides a interface between DSPL (via *Data Extractor*) and the map building components. Also, the framework supports different $\mathbb{CR}$ file formats: *XML*, *JSON*, or *CSV*, see Figure 2.

The **Data Extractor (DE)** performs the runtime integration between the *Integration Layer* and the SAS. The **DE** relies on the *Plan* function (see Figure 1), reading the *Change Plan* information

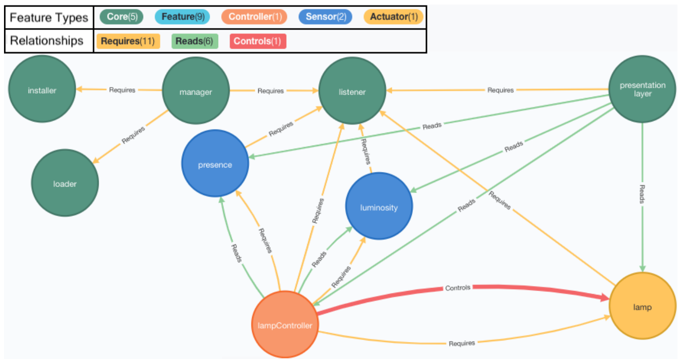

Figure 3: Behavioral Map (BM) for one SHE configuration.

at runtime. The **DE** identifies all features used and their relationships involved in the new configuration defined in the *Change Plan*. Thereafter, the **DE** builds a $\mathbb{CR}$ *file* including all involved features and sends it to the *Integration Layer*. The **DE** performs static analysis using the WALA API [17]. Such analysis allows identifying the dependency relationships among the class hierarchy used by selected features or performing interprocedural dataflow analysis and identifying relationships' types. Also, complementary information that is available in manifest files (used to install each feature of the candidate configuration before its deployment) can be used to identify the relationships.

The **DE** can be implemented for all adaptation process types, it just needs to receive the following parameters: the features and their $VTypes$, set of source code paths in the packages and the related Jar files. Also, we used these parameters to map the relations between features and components implementing them.

4.4 Behavioral Map example

We exemplify the **BM** framework on the SHE [10, 26] system. SHE is a smart home system relying on a MAPE-K loop to adapt to new situations (such as a new sensor being plugged in) and updates a dashboard (*e.g.,* adding a widget for the new sensor). The SHE core features are: *Manager*, *Listener*, *Loader*, *Installer*, and *Presentation Layer*. They control the adaptation, communication, and data presentation. They are optional features: 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 based on information gathered from *Luminosity* and *Presence* features; iv) ***Lamp*****:** an actuator used to switch on and off lights based on the *lampController* feature's data. This example configuration is presented Figure 3. An implementation of this example with a tutorial to perform **BM** construction and ABS identification is available on our companion website[2].

4.5 Architectural Bad Smell Identification

We selected the architectural smells shown in Table 1 because they were proposed for self-adaptive systems [25, 27]. In the SHE configuration analyzed, we identified the Hub-Like Dependency (HL) and Extraneous Connector (EC) smells. The former appears when a component has (incoming or outgoing) dependencies with a large number of other abstractions (*e.g.,* components) or concrete classes [4, 25]. Since the **BM** is a graph, computing the in/out-degree for each vertex (feature) is easy, features having high in/out-degrees suffer from the HL smell. In Figure 3, the *Listener* feature is subjected to the HL smell as it is involved in most of the *Requires* of the **BM**. The *publish-subscribe* architecture adopted by the SHE framework is the cause of this smell. Indeed, the *Listener* centralizes all the communication processes in this software architecture and works as a communication broker. While acceptable in this case [4, 12], hubs may greatly impact the systems if they fail.

Table 1: Selected Architectural Bad Smells for Self-Adaptive Systems.

Smell Name	Detection
Cyclic Dependency (CD) [4]	Full
Extraneous Connector (EC) [13]	Full
Hub-Like Dependency (HL) [4, 25]	Full
Oppressed Monitors (OM)[27]	Partial

The latter smell arises when two connectors of different types are used to link the same pair of components [13]. It is easy to identify this smell as edges and vertices have types, colors providing visual cues. As depicted Figure 3, the *lampController* uses two types of connectors to connect with *Presence*, *Luminosity*, and *Lamp* features. 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*. Computation of paths between vertices may support the automated identification of this smell.

5 WORK PLAN

Being in the middle of this thesis, we established the main concepts of behavioral maps and designed an inference framework. In the next year, we want to refine the mapping between features and their realizations, currently being one-to-one relationships. We want to introduce "modules" to allow a more fine-grained traceability [29]. We also plan to extend the formalism to support family-based **BM**s to analyze architectural issues for the entire (D)SPL in a static way (as opposed to current configuration level analysis), which may be relevant for smells detection [25]. We plan to work on this challenge between September and February 2022.

The second research direction focuses on providing test generation/prioritization algorithms, at runtime and for one given configuration, that rely on edge types between features. One can give a higher priority to features involved in a *control* relationship rather than those involved in a *reads* one. We plan to work in parallel with the first research direction notably in September-December 2021 and again from March 2022.

ACKNOWLEDGMENTS

I want to thank my promoters Gilles Perrouin and Pierre-Yves Schobbens, for supervising this thesis. This work is funded by a CERUNA grant from the University of Namur.

[2]https://github.com/edilton-santos/BehavioralMapExample

REFERENCES

[1] Sven Apel, Alexander Von Rhein, Thomas ThüM, and Christian KäStner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[2] Joanne M Atlee, Uli Fahrenberg, and Axel Legay. 2015. Measuring behaviour interactions between product-line features. In *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering*. IEEE, 20–25.

[3] Thomas H Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 165–178.

[4] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. 2019. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, IEEE, 88–97.

[5] Nelly Bencomo, Peter Sawyer, Gordon S Blair, and Paul Grace. 2008. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems.. In *SPLC (2)*. 23–32.

[6] Nicolás Cardozo and Ivana Dusparic. 2020. Learning run-time compositions of interacting adaptations. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 108–114.

[7] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*. 321–330.

[8] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 129–139.

[9] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. 2014. Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*. 1–6.

[10] Edilton Lima dos Santos and Ivan do Carmo Machado. 2019. An Architecture Model for DSPL Engineering. In *The 18th Belgium-Netherlands Software Evolution Workshop: BENEVOL*. BENEVOL, CEUR-WS.

[11] Francesca Arcelli Fontana, Paris Avgeriou, Ilaria Pigazzini, and Riccardo Roveda. 2019. A Study on Architectural Smells Prediction. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, IEEE, 333–337.

[12] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. 2016. Automatic detection of instability architectural smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE, 433–437.

[13] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, IEEE, 255–258.

[14] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a catalogue of architectural bad smells. In *International conference on the quality of software architectures*. Springer, Springer, 146–162.

[15] Robert J Hall. 2000. Feature combination and interaction detection via foreground/background models. *Computer Networks* 32, 4 (2000), 449–469.

[16] IBM. 2006. An architectural blueprint for autonomic computing. *IBM White Paper* 31 (2006), 1–6.

[17] IBM. 2020. *The T. J. Watson Libraries for Analysis (WALA)*. IBM. https://github.com/wala/WALA

[18] Martin Lippert and Stephen Roock. 2006. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons.

[19] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 483–494.

[20] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2020. A systematic mapping study on architectural smells detection. *Journal of Systems and Software* (2020), 110885.

[21] Neo4j. 2020. *The Internet-Scale Graph Platform*. Neo4j. https://neo4j.com/product/

[22] Neo4j. 2020. *Neo4j APOC Library*. Neo4j. https://neo4j.com/developer/neo4j-apoc/

[23] Neo4j. 2020. *The Neo4j Cypher Manual v4.1*. Neo4j. https://neo4j.com/docs/cypher-manual/current/introduction/

[24] Gilles Perrouin, Mathieu Acher, Jean-Marc Davril, Axel Legay, and Patrick Heymans. 2016. *A Complexity Tale: Web Configurators*. ACM Press. https://doi.org/10.1145/2897045.2897051

[25] Claudia Raibulet, Francesca Arcelli Fontana, and Simone Carettoni. 2020. A preliminary analysis of self-adaptive systems according to different issues. *Software Quality Journal* (2020), 1–31.

[26] Edilton Santos and Ivan Machado. 2018. Towards an Architecture Model for Dynamic Software Product Lines Engineering. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, IEEE, 31–38.

[27] Marcel A Serikawa, André de S Landi, Bento R Siqueira, Renato S Costa, Fabiano C Ferrari, Ricardo Menotti, and Valter V De Camargo. 2016. Towards the characterization of monitor smells in adaptive systems. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, IEEE, 51–60.

[28] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. 59–66.

[29] Tassio Vale, Eduardo Santana de Almeida, Vander Alves, Uirá Kulesza, Nan Niu, and Ricardo de Lima. 2017. Software product lines traceability: A systematic mapping study. *Information and Software Technology* 84 (2017), 1–18.