A Vision to identify Architectural Smells in Self-Adaptive Systems using Behavioral Maps

Edilton Lima dos Santos[1], Sophie Fortz[1], Gilles Perrouin[1] and Pierre-Yves Schobbens[1]

[1]Faculty of Computer Science, University of Namur, Namur, Belgium

Abstract

Self-adaptive systems can be implemented as Dynamic Software Product Lines (DSPLs) via dynamically enabling or disabling features at runtime based on a feature model. However, the runtime (re)configuration may negatively impact the system's architectural qualities, exhibiting architectural bad smells. Such smells may appear in only very specific runtime conditions, and the combinatorial explosion of the number of configurations induced by features makes exhaustive analysis intractable. We are therefore targeting smell detection at runtime for one specific configuration determined through a MAPE-K loop. To support smell detection, we propose the Behavioral Map (BM) formalism to derive automatically key architectural characteristics from different sources (feature model, source code, and other deployment artifacts) and represent them in a graph. We provide identification guidelines based on the BM for four architectural smells and illustrate the approach on Smart Home Environment (SHE) DSPL.

Keywords

Architectural Smells, Dynamic Software Product Lines, Runtime Validation, Self-adaptive Systems

1. Introduction

Self-adaptive systems (SAS) operate in unpredictable, heterogeneous environments, are prone to crashes and various other types of involuntary or required changes at runtime. Thus, such systems must adjust their structure or behavior, depending on environmental changes and (re)configuration plans to work in such environments. Dynamic Software Product Line (DSPL) engineering implements SAS by dynamically binding or unbinding features at runtime as prescribed by a feature model [1]. A feature model represents commonalities and variabilities in a family of systems as well as relationships amongst features [2]. It thus describes which valid (re)configurations can be performed. DSPLs are challenging to validate because the number of possible configurations grows exponentially with the number of features, and this problem is worse if the DSPL can self-update (*e.g.,* by downloading new features to interface with a sensor newly plugged into the system) [3].

Moreover, (re)configurations may also negatively affect architectural qualities. Some specific feature interactions may not appear in the feature model (because it does not capture data and control flows, only accessible via source code analysis) or the resulting configuration's architecture may be prone to other issues. 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 reductions in system maintainability [4, 5]. Several authors [5, 6, 7] define an ABS as a set of architectural design decisions that negatively impact system lifecycle properties, such as understandability, testability, maintainability, extensibility, and reusability. Also, it indicates possible design and implementation issues and helps improving the quality of the system.

While ABS are well-studied for other type of systems [4, 5, 6, 7, 8, 9], ABS in SAS are less explored [10, 11]. To the best of our knowledge, the only studies targeting SAS search for ABS at design time: this may lead to smell detection for invalid configurations (false positives) [10]. Additionally, some ABS may appear more frequently in a specific configuration. Thus, we are interested in identifying ABS in SAS that may arise after the (re)configuration process determined which configuration to deploy, allowing to have access to all the necessary artifacts for a more accurate smell identification.

In this paper, we introduce the Behavioral Map (**BM**) formalism: a directed graph capturing interactions defined in the feature model but also capturing control and data flows interactions inferred from the candidate reconfiguration implementation. We describe the **BM** framework implementation and how to automatically build such a map from reconfiguration plans and actual code using static analysis techniques. We provide guidelines to identify four ABSs based on the inferred map. We exemplify our approach on SHE [12], a smart home DPSL. Our BM example and smell detection

CASA'21: Context-aware, Autonomous and Smart Architecture Workshop, September 13–17, 2021, Växjö, Sweden

edilton.limados@unamur.be (E. L. d. Santos); sophie.fortz@unamur.be (S. Fortz); gilles.perrouin@unamur.be (G. Perrouin); pierre-yves.schobbens@unamur.be (P. Schobbens)

0000-0003-2231-3852 (E. L. d. Santos); 0000-0001-9687-8587 (S. Fortz); 0000-0002-8431-0377 (G. Perrouin); 0000-0001-8677-4485 (P. Schobbens)

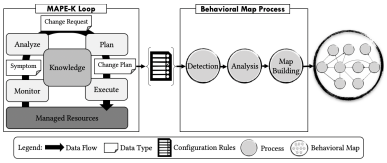

Figure 1: Behavioral Map (BM) process overview.

scripts are available on the companion website: https://github.com/edilton-santos/BehavioralMapExample.

The remainder of the paper is as follows. Section 2 formally defines BM and presents the framework allowing them to build them automatically from a given DSPL. Section 3 discusses the architectural bad smells identified through the **BM** and illustrates them on the Smart Home Environment (SHE) [12] case study. Section 4 presents the related work. Finally, Section 5 wraps up the paper.

2. Behavioral Map

The role of a **Behavioral Map** is to capture interactions between features (described in a feature model [2]) of a specific (re)configuration to be analyzed before it gets deployed [13]. Such configurations 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 [14]. We depicted it left 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* selected by the self-adaptive system 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 [14] to manage the adaptation process at runtime according to the feature model. We thus avoid building a **BM** for an invalid configuration.

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 part 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 of *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 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 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 part of the figure 1.

2.2.1. Detection

Detection determines interacting features using pairwise analysis [15] and their directed relationships based on the $\mathbb{CR}$. Moreover, we assume that in the $\mathbb{CR}$, there are all features and their configuration policy (including feature dependencies) used to answer a specific context at runtime. For example, the feature installation process used the constraints available in the manifest file used to describe the feature and its dependencies. Also, this process can use complementary information defined in the *Change Plan*. Such information is used to guide the installation, configuration, and adaptation process 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 to load a *Feature B* at runtime. This dependency is defined in the $\mathbb{CR}$ file and is 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.

2.2.2. 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.

2.2.3. Map Building

Based on interaction detection and analysis, we can build the **BM** 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(ℂℝ file);
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.

The whole building process is summarized by Algorithm 1. This algorithm begin from a `table` loaded by the `loadConfigurationRulesFile` procedure (line 1 at listing 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 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 loop on line 6 will repeats until and thus edges are created.

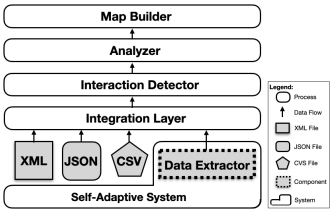

Figure 2: Behavioral Map Architecture overview.

2.3. Framework Implementation

We conceived a framework to infer Behavioral Maps whose architecture is shown in Figure 2. The framework uses the Neo4J[1] platform and its Cypher[2] 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 $\mathbb{CR}$ *file* data type used to build the map as follows: **i) name** is the feature name in the system; **ii) exported_packages** lists the exported packages or services offered via features; **iii) imported_packages** lists the packages used by features to compose their functionality; **iv) version** represents the feature version; **v) status** defines if the feature is active or inactive; **vi) type** defines the feature type; **vii) 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

[1]Neo4j - https://neo4j.com/product/

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

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*. The **DE** component performs static analysis using the WALA API[3]. 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 the data extract needs to receive 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.

3. BM-Based ABS Detection

3.1. Case Study: The SHE Framework

We applied our **BM** framework on SHE [12]: 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.

3.2. Identifying Architectural Bad Smells

While ABS catalogues exist in the literature [16, 8], their role in self-adaptive architectures is less known [10, 11]. Table 1 presents a list of smells we believe to be relevant for assessing self-adaptive architecture as well as their level of support via 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. Also, we provided a package in GitHub[4] with a tutorial to do the configuration of the Neo4J platform, two CR files, and the scripts used to create the map and analyze the ABSs.

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

Smell Name	Detection
Cyclic Dependency (CD) [16]	Full
Extraneous Connector (EC) [8]	Full
Hub-Like Dependency (HL) [16, 10]	Full
Oppressed Monitors (OM)[11]	Partial

3.2.1. Cyclic Dependency [16]:

This smell occurs when two or more components depend on each other directly or indirectly [16]. Components involved in a dependency cycle can hardly be released, maintained, or reused in isolation [17].

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 that may be uncovered, such as control ones which may cause concurrent accesses to resources and/or deadlocks.

3.2.2. Extraneous Connector (EC) [8]:

This smell happens when two connectors of different types are used to link a pair of components [8].

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 [7]. 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-and-subscribe architecture.

[3]WALA - https://github.com/wala/WALA

[4]Behavioral Map example - https://github.com/edilton-santos/BehavioralMapExample

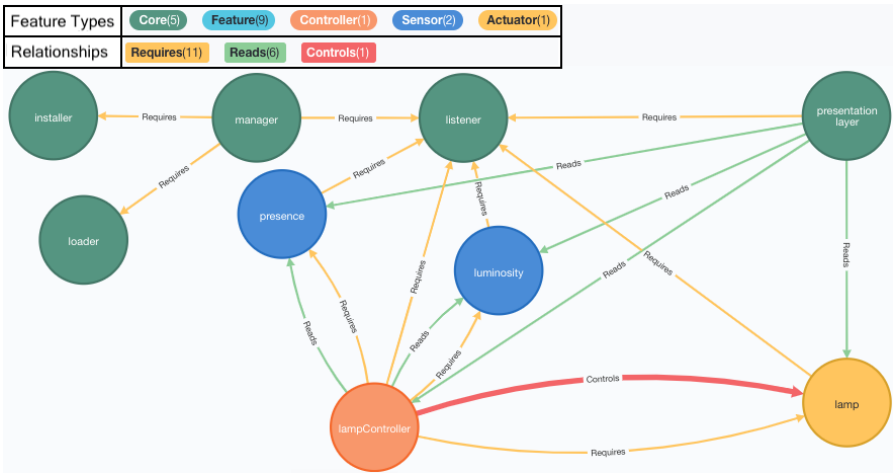

Figure 3: Behavioral Map (BM) for one SHE configuration.

3.2.3. 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 [16, 10].

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 Hub type called *Overreliant Class* [16].

Discussion. The presence of the HL smell in the Listener feature is motivated by the publish-and-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 [16, 17]. However, hubs form points of attention in case of failure.

3.2.4. Oppressed Monitors [11] (OM):

According to [11], 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 [11]. 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.

4. Related Work

We found two works dedicated to the identification of ABS in self-adaptive systems. The first study [10] relies on the Arcan [17] 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.

The second study [11] 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.

5. Concluding Remarks

We defined behavioral maps (**BMs**) to reason on architectural issues in self-adaptive systems. A **BM** is informed by a feature model and by the considered configuration before its runtime deployment. We implemented a flexible framework inferring **BMs** from heterogeneous information sources, relying on static analysis to characterize interactions more finely than with a feature model. We illustrated its application on a smart home dynamic software product line. As a work in progress paper, there is room for future work. First, we want to use the **BM** to provide new smells specific to SAS architectures and perform self-adaptive architecture assessments beyond general ones [10]. Then, we envision other **BM** usages such as test prioritization strategies, notably when new features appear via hot-plugging mechanisms.

Acknowledgments

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

References

[1] N. Bencomo, P. Sawyer, G. S. Blair, P. Grace, Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems., in: SPLC (2), 2008, pp. 23–32.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report, CMU-SEI, 1990.

[3] N. Cardozo, I. Dusparic, 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, 2020, pp. 108–114.

[4] M. Lippert, S. Roock, Refactoring in large software projects: performing complex restructurings successfully, John Wiley & Sons, 2006.

[5] H. S. de Andrade, E. Almeida, I. Crnkovic, Architectural bad smells in software product lines: An exploratory study, in: Proceedings of the WICSA 2014 Companion Volume, 2014, pp. 1–6.

[6] F. A. Fontana, P. Avgeriou, I. Pigazzini, R. Roveda, A study on architectural smells prediction, in: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2019, pp. 333–337.

[7] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Toward a catalogue of architectural bad smells, in: International conference on the quality of software architectures, Springer, 2009, pp. 146–162.

[8] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 255–258.

[9] H. Mumtaz, P. Singh, K. Blincoe, A systematic mapping study on architectural smells detection, Journal of Systems and Software (2020).

[10] C. Raibulet, F. A. Fontana, S. Carettoni, A preliminary analysis of self-adaptive systems according to different issues, Software Quality Journal (2020) 1–31.

[11] M. A. Serikawa, A. d. S. Landi, B. R. Siqueira, R. S. Costa, F. C. Ferrari, R. Menotti, V. V. De Camargo, Towards the characterization of monitor smells in adaptive systems, in: X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), IEEE, 2016, pp. 51–60.

[12] E. Santos, I. Machado, Towards an architecture model for dynamic software product lines engineering, in: IEEE International Conference on Information Reuse and Integration (IRI), IEEE, 2018, pp. 31–38.

[13] E. L. dos Santos, Stars: Software technology for adaptable and reusable systems, in: Proceedings of the 25th International Systems and Software Product Line Conference (SPLC), ACM, 2021.

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

[15] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, E. S. de Almeida, Varxplorer: Lightweight process for dynamic analysis of feature interactions, in: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, 2018, pp. 59–66.

[16] U. Azadi, F. A. Fontana, D. Taibi, Architectural smells detected by tools: a catalogue proposal, in: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE, 2019, pp. 88–97.

[17] F. A. Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 433–437.