DOCTOR OF SCIENCES

Behavioral Maps

A Framework to Assess and Validate Self-Adaptive Architectures at Runtime

Lima dos Santos, Edilton

Award date:
2023

Awarding institution:
University of Namur

Behavioral Maps: A Framework to Assess and Validate Self-Adaptive Architectures at Runtime

Edilton Lima dos Santos

Jury

Prof. Marie-Ange Remiche
Université de Namur, Belgium

Prof. Claudia Raibulet
Vrije Universiteit Amsterdam, Netherlands

Prof. Kim Mens
Université Catholique de Louvain, Belgium

Prof. Vincent Englebert
Université de Namur, Belgium

Dr. Gilles Perrouin
Université de Namur, Belgium

Prof. Pierre-Yves Schobbens
Université de Namur, Belgium

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Supervised by Dr. Gilles Perrouin and Prof. Pierre-Yves Schobbens

University of Namur

PReCISE Research Center

Cover design: © Presses universitaires de Namur

Rue Grandgagnage 19
B – 5000 Namur (Belgium)
pun@unamur.be - www.pun.be

Registration of copyright: D/2023/1881/14
ISBN: 978-2-39029-177-0
Printed in Belgium.

“If you can dream it, you can do it.”

— ***Walt Disney***

ABSTRACT

A Self-adaptive System (SAS) is a specialized system designed to handle changes that may occur in the operating environment. This system accomplishes this by triggering necessary adaptations at runtime. These adaptations can change the system's structure, behavior, or even its adaptation mechanism. However, it is essential to note that these changes can introduce defects and architectural issues (*e.g.,* architectural bad smells) into the system, which can cause it to fail during runtime. As such, it is crucial to carefully monitor and manage these adaptations to maintain the system's reliability. In order to ensure system performance and integrity, it is essential to conduct thorough testing and architectural analysis while the system is running. Although previous studies available in the literature have focused mainly on analyzing architectural issues and testing during the design phase, evaluating the system at runtime is equally essential. Thus, this thesis proposed the Behavioral Map framework.

During system-under-test (SUT) execution, our framework can recognize feature interactions and architectural bad smells (ABS), such as Cyclic Dependency, Extraneous Connector, Hub-Like Dependency, and Oppressed Monitors. Also, the Behavioral Map generates a graphical map that describes the configuration analyzed at runtime. This map enables us to determine the testing boundaries and to dynamically generate test cases based on the selected scope, which could be determined through either Feature Relationship Analysis or ABSs Analysis. Also, the test case generation processes can generate tests using the following strategies: i) adaptive-random test generation, ii) evolutionary algorithms to generate tests, and iii) combinatorial test design to generate test cases.

The Behavioral Map has been implemented in two versions. The first, the Behavioral Map White Box, is implemented as reusable building blocks that allow their incorporation into the system under analysis. The last version, the Behavioral Map Black Box, automatically executes the SUT methods from the host Java Virtual Machine. This allows the Behavioral Map Black Box to identify features loaded at runtime based on parameters defined by software developers without requiring code instrumentation and source code recompilation of SUT, unlike the Behavioral Map White Box. Through our research, we have conducted various studies to evaluate our approach. In the first study, we analyzed the process of identifying feature interaction and ABS detection at runtime in three SASs. Our findings indicated that some ABS only appear in specific system configurations or architectures. The second study compared ABS detected at runtime to those detected at design time, revealing

differences between the two. Lastly, we focused on assessing the feasibility of our testing approach, and our results show that it is feasible to select the test scope at runtime to SASs.

Keywords: Self-adapting system, Software architecture, Architectural Smells, MAPE-K loop, Software testing, Runtime Validation, Behavioral Maps.

RÉSUMÉ

Un système auto-adaptatif (SAA) est un système spécialisé conçu pour traiter les changements qui peuvent survenir dans l'environnement opérationnel. Ce système y parvient en déclenchant les adaptations nécessaires au moment de l'exécution. Ces adaptations peuvent modifier la structure du système, son comportement, ou même son mécanisme d'adaptation. Cependant, il est essentiel de souligner que ces changements peuvent introduire des anomalies et des problèmes architecturaux (par ex. les mauvaises pratiques ("smells") architecturales) dans le système, ce qui peut entraîner des défaillances en cours d'exécution. Il est primordial de surveiller et de gérer attentivement ces adaptations afin de maintenir la fiabilité du système. Pour assurer la performance et l'intégrité du système, il est essentiel de mener des tests approfondis et une analyse architecturale lorsque le système est en cours d'exécution. Bien que les études antérieures dans la littérature se sont principalement concentrées sur l'analyse architecturales de manière statique, l'évaluation du système en cours d'exécution est également essentielle. Cette thèse a ainsi proposé le Framework "Behavioral Map".

Pendant l'exécution du système à tester (SAT), notre Framework peut identifier l'interaction des fonctionnalités ou les smells architecturaux, telles que la dépendance cyclique, le connecteur externe, la dépendance de type "hub" et les capteurs désynchronisés. En effet, la Behavioral Map génère une représentation graphique qui décrit la configuration analysée au cours de l'exécution. Cette cartographie permet de déterminer les frontières des tests et de produire dynamiquement des cas de test pour un sous ensemble déterminé du système, qui peut être établi par l'analyse des relations entre les caractéristiques ou par l'analyse des smells architecturaux. De plus, les processus de génération de cas de test peuvent générer des tests en utilisant les stratégies suivantes : i) la génération de tests aléatoires adaptatifs, ii) les algorithmes évolutionnaires pour produire des tests, et iii) la conception combinatoire de scénarios de test.

La Behavioral Map a été mise en œuvre en deux versions. La première, la Behavioral Map boite blanche, est mise en œuvre sous forme de composants réutilisables qui autorisent leur incorporation au système analysé. La deuxième version, la Behavioral Map boite noire, exécute automatiquement les méthodes du SAT à partir de la machine virtuelle Java de l'hébergeur. Cela permet à la Behavioral Map boite noire d'identifier les fonctionnalités chargées au moment de l'exécution en fonction des paramètres définis par les développeurs de logiciels sans nécessiter d'instrumentation du code et une recompilation du code source du SAT, au contraire

de la Behavioral Map boite blanche. Dans le cadre de nos recherches, nous avons mené plusieurs études pour évaluer notre approche. Dans la première étude, nous avons analysé le processus d'identification de l'interaction des caractéristiques et la détection de smells architecturaux en cours d'exécution dans trois SAAs. Nos résultats ont indiqué que certains smells ne sont présents que dans des configurations spécifiques de certains SAAs. La deuxième étude a comparé les smells détectés à l'exécution à ceux détectés à la conception, en révélant des différences entre les deux. Enfin, nous nous sommes concentrés sur l'évaluation de la faisabilité de notre approche de test, et nos résultats ont montré qu'il est effectivement possible de sélectionner la portée du test à l'exécution pour les SAAs.

Mots clés : Système auto-adaptatif, Architecture logicielle, Architectural Smells, MAPE-K, Test Logiciel, Behavioral Map

ACKNOWLEDGEMENTS

I want to start by expressing my gratitude to God for the many blessings in my life. I am also thankful for my parents, Everaldo Lima dos Santos and Maria Lúcia Lima dos Santos, who have always encouraged and supported me. Special thanks go to my wife, Andréa, and daughter, Ingrid, who have been a constant source of inspiration and love.

Also, I would like to thank my brothers, Elton and Elder, as well as all the members of the Lima family who have directly or indirectly influenced and motivated me.

My Ph.D. studies began in September 2019 and have presented numerous challenges, including adjusting to a new country, a new job, and being away from family and friends. However, I have also had many rewarding experiences, including the freedom to conduct research without industry limitations and the guidance of exceptional advisors like Gilles Perrouin and Pierre-Yves Schobbens.

I want to express my deep appreciation to Prof. Gilles Perrouin for his invaluable guidance and insightful discussions. Additionally, I am grateful to my friends Dhennya, Davilene, Carmen, Thaís, Larissa, Nádia, Cristiano, Cíntia, Joel, Cássia, Dona Nildes, Maria Helena, Aleci, Leonardo (Léo), Miguel (Miguelito), Maria Eduarda (Duda), Lorena, Aishwarya, Rabeb, Ahmed, Maouaheb, Tárcius, Taís, Danillo, and Sueny for their unwavering support and friendship. I extend my gratitude to my research team, especially to James.

I would like to express my gratitude to the jury members for dedicating their time to reading my manuscript and providing feedback. My sincere thanks go to Prof. Marie-Ange Remiche, Prof. Claudia Raibulet, Prof. Kim Mens, and Prof. Vincent Englebert.

I want to thank Prof. Ivan Machado and Prof. Eduardo Almeida from the Federal University of Bahia in Brazil for believing in me. Without their support, this thesis would not have been possible.

Finally, I am grateful to the University of Namur for their support through the CERUNA grant.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Context and problem statement

Self-Adaptive Systems (SAS) change their behavior depending on environmental changes and (re)configuration plans and goals [25, 47, 71, 109]. Dynamic Software Product Line (DSPL) engineering is a way of implementing SASs, where features are enabled or disabled at runtime according to a feature model [13]. However, validating the DSPLs can be complex due to the exponential growth in the number of configurations possible with the number of features [5, 28, 119]. Furthermore, unexpected and undesired interactions between features can occur, especially if the system can update itself (for example, by downloading new features to interface with a sensor newly plugged into the system) [21]. While the feature interaction problem is well-researched for systems where features are bound at the specification or design time [3, 5, 6, 26, 28, 58, 85], it is less explored for runtime interactions [21, 102].

Beyond that, 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 [51, 52, 77]. In these circumstances, Architectural Bad Smells (ABS) may appear, implying reductions in system maintainability [29, 79]. ABSs result from a set of architectural design decisions that negatively impact the system's properties (understandability, testability, maintainability, extensibility, and reusability) [29, 42, 52]. However, an ad-hoc literature review identified only two studies exploring ABS in SAS at design time [106, 114]. In addition, there is a gap in evaluating the impact or identification of ABS in SAS at runtime [90].

In this context, SASs testing is a complex undertaking that poses a significant challenge. Testing every operational scenario that a SAS may encounter at runtime is impractical [48]. Furthermore, the number of potential configurations that may arise at runtime grows exponentially with the number of features provided by the system. As a result, SAS demands testing strategies that can address unknown variability space at runtime or untested configurations before software deployment. Runtime tests are essential to ensure that the SAS functions as expected in unforeseen situations. Testing SAS to verify that the new configurations satisfy the requirements at runtime is imperative. Therefore, test generation during runtime is necessary to guarantee the relevance of testing, where relevance refers to the applicability of a test case to its environment in constant change [47]. Thus, we must face two

main questions: i) How to deal with Runtime variability's impact on SAS Tests? This umbrella question is discussed in [30, 47, 117, 118]. Also, this thesis addresses some open questions about the runtime variability's impact on SAS Tests, such as How to deal with the exponential growth of SAS configurations that should be tested? and ii) How can the number of test cases be reduced while maintaining the fault detection capability? In this direction, we must generate test cases for each configuration detected at runtime [104, 126, 139].

Contributions

This thesis introduces the Behavioral Map (**BM**) formalism, which gathers information from various sources, such as feature models and source codes, to identify feature interactions and architectural issues (*e.g.*, ABS). The **BM** is a directed graph that captures interactions defined in the feature model while taking control and data flow interactions inferred from the candidate reconfiguration implementation. Typically, Dynamic Software Product Line (DSPL) engineering involves representing the features of a system family along with their commonalities, variabilities, and relationships. This model is highly abstract and serves as a foundation for feature selection and product derivation during design time or runtime. However, it fails to capture control and data flow interactions inferred from the SAS, which are crucial in identifying unpredictable behavior or relationships among features at runtime.

Thus, the **BM** supports the feature interaction issues identification, ABS identification, and testing selection 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 deployment. Consequently, the system will not execute the faulty configuration and will keep the last valid configuration until a new one gets computed. The main 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.

Behavioral Map Framework: We created two versions of the Behavioral Map in Java 8. The first version, the Behavioral Map (**BM**) White Box, is a set of reusable building blocks that can be integrated into the analyzed system. The BM White Box uses the Neo4J[1] graph database (including Cypher[2] queries) to create a map visualization and the WALA API [66] to extract data. This API provides Call Graph [55, 56] and the Control-Flow Analysis (CFA) [55, 87, 88] algorithms for extracting data. The API uses static analysis to identify dependencies within the class hierarchy and performs interprocedural dataflow analysis to identify relationship types. Additionally,

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

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

we can retrieve information about each feature's installation at runtime using the manifest file, which describes the feature and its dependencies. The **BM** White Box also can identify Architectural Bad Smells (ABS) at runtime.

The second, Behavioral Map Black Box, is an extension of Java Pathfinder (JPF)[3] that can automatically execute System Under Test (SUT) methods without requiring code instrumentation or recompilation of SUT. This feature helps software developers to identify features loaded at runtime based on parameters. The **BM** Black Box also supports the detection of ABS at runtime based on the map created using the Neo4J graph database. Also, the tool is equipped with JUnit[4], allowing for seamless execution of test units if the SUT implements them. Additionally, it can generate unit-level test cases using TackleTest[5] [124] based on ABS or feature interaction detected at runtime.

Structure of the thesis

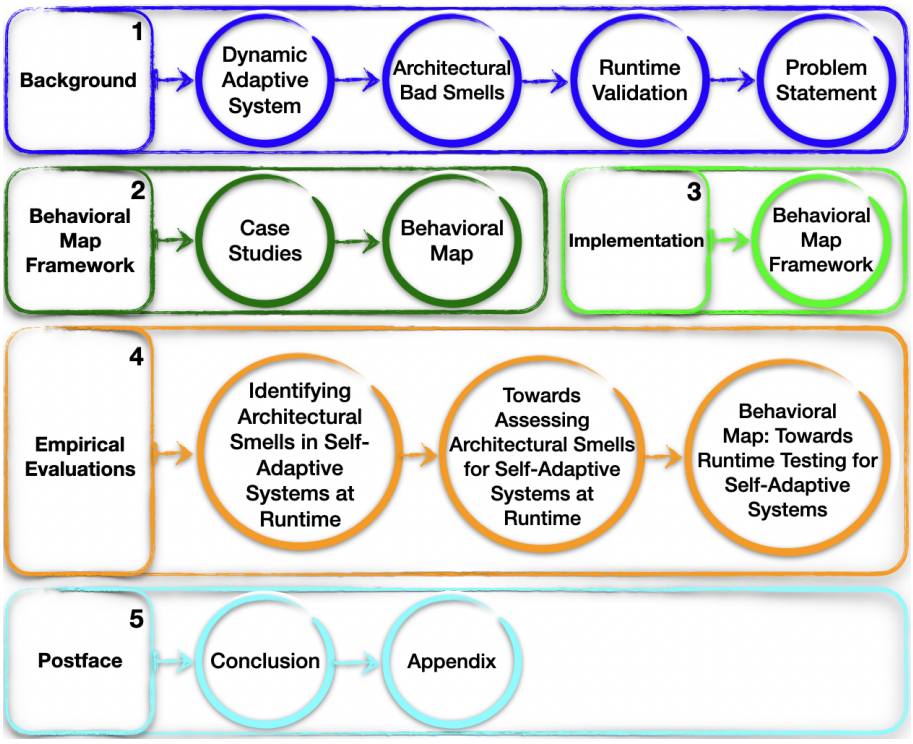

Figure 1: Thesis Structure Overview.

[3]Java Pathfinder is an extensible software model-checking framework for Java bytecode programs - Additional information is available at https://github.com/javapathfinder/jpf-core

[4]JUnit - Additional information is available at https://junit.org/junit5/docs/5.0.0-M5/user-guide/

[5]TackleTest - https://github.com/konveyor/tackle-test-generator-cli

This thesis is divided into five parts, with a diagram of the structure provided in Figure 1. The introductory section is followed by Part I, which covers the background information needed for the rest of the thesis. Chapter 1 introduces Dynamic Adaptive Systems, focusing on DSPL and SAS. Chapter 2 describes ABS, including their characteristics, quality impact, and trade-offs. Chapter 3 provides the necessary background on runtime validation. In Chapter 4, we explore the difficulties that arise with detecting Architectural Bad Smells and conducting runtime testing in the face of runtime variability. Part II contains the case studies discussed in Chapter 5 and main contributions of the thesis, with the Behavioral Map framework presented in Chapter 6. Implementation details are described in Part III on Chapter 7, including the Behavioral Map White Box and Behavioral Map Black Box. The Empirical Evaluations are presented in Part IV, with Chapter 8 focusing on assessing ABS for SASs using the Behavioral Map. Chapter 9 compares design time and runtime ABS detection for SASs. In contrast, Chapter 10 addresses the applicability of Behavioral Map to select scope tests, generate test cases, and execute tests at runtime in SASs. Finally, Part V presents Chapter 11, conclude the thesis and offers research perspectives. Also, the thesis appendixes include some configurations files used to run the Behavioral Map Black Box during the experimentation.

Publications

The content of this thesis is based upon, reuses, and extends the following peer-reviewed publications of the author:

Book Chapter

[77] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Behavioral maps: Identifying architectural smells in self-adaptive systems at runtime. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, **Software Architecture**, pages 159–180, Cham, 2022. Springer International Publishing.

Conferences

[78] Edilton Lima dos Santos, Pierre-Yves Schobbens, Ivan Machado, and Gilles Perrouin. Architectural bad smells for self-adaptive systems: Go runtime! In **Proceedings of the 17th International Working Conference on Variability Modelling of Software-Intensive Systems**, VaMoS '23, page 85–87, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Edilton Lima dos Santos, Pierre-Yves Schobbens, and Gilles Perrouin. Featured scents: Towards assessing architectural smells for self-adaptive systems at runtime. In **19th International Conference on Software Architecture**, pages 71–74. IEEE, 2022.

[35] Edilton Lima dos Santos, Sophie Fortz, Gilles Perrouin, and Pierre-Yves Schobbens. A vision to identify architectural smells in self-adaptive systems using

behavioral maps. In **15th European Conference on Software Architecture (ECSA 2021)**, page 1. CEUR Workshop Proceedings, 2021.

[34] Edilton Lima dos Santos. Stars: Software technology for adaptable and reusable systems. In **Proceedings of the 25th International Systems and Software Product Line Conference (SPLC)**, pages 13–17. ACM, 2021.

[112] Edilton Lima dos Santos, Gilles Perrouin, and Pierre-Yves Schobbens. Stars: software technology for adaptable and reusable systems phd research project. In **Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems**, pages 1–2, 2020.

Part I

Background

CHAPTER 1

DYNAMICALLY ADAPTIVE SYSTEMS

Nowadays, the software industry has been developing software systems capable of anticipating how and when they will need to self-adapt [17, 83, 109, 122]. The self-adaptation has been widely accepted as an effective methodology for handling modern software systems' increasing complexity and dynamicity [133]. Such behavior is implemented in adaptation mechanisms to manage the adaptation process and assurance software integrity during and after the adaptation. These software systems are named Dynamic Adaptive Systems (DAS). The DASs monitor themselves and their execution environment to answer the environmental changes, user requests, and reconfiguration plans and goals at runtime [35, 78]. For this purpose, the DASs combine architectural fragments or solutions in their adaptation process [77]. Also, DASs can be divided into two parts: i) **Adaptation policies or adaptation strategies, specifying how the system must react according to the environmental changes or user's requirements at runtime**, and ii) **the set of features (configuration options) used to (re)configure the system** [91, 132]. Consequently, these systems can be customized or reconfigured to specific needs via the bind and unbind of different features at runtime, a phenomenon known as runtime variability.

The DASs can be conceptualized as a dynamic software product line (DSPL) [89] or self-adaptive system [13] in which variabilities are bound at runtime [1, 13, 57]. However, SASs differ from DSPLs as the former do not provide models of the system's variability [105]. This means that a SAS can be a DSPL only if implemented following

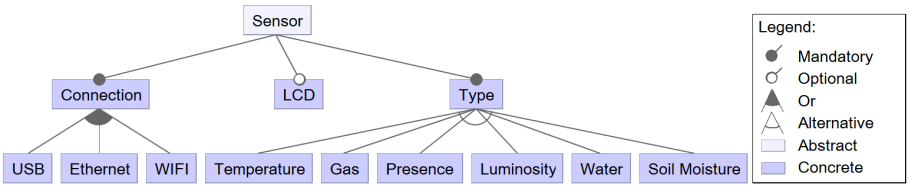

Figure 1.1: Feature model of the Sensor software product line.

the DSPL engineering principles [13, 37]. In the following sections, we define both systems. Thus, this chapter presents the main ideas and concepts behind DAS, focusing in DSPL engineering and SAS used in this thesis. Section 1.1 presents DSPL concepts. Section 1.2 discusses the SAS concepts and their characteristics.

1.1 Dynamic Software Product Line

Dynamic software product lines (**DSPL**) **engineering exploits the knowledge acquired in Software Product Line** (SPL) [27] **engineering to develop systems capable of self-adapt at runtime** (*e.g.,* highly-configurable, context-aware, and self-adaptive) [10]. Thus, DSPL incorporates the property of similarities and variability in the family of software like SPL. Moreover, DSPL provides a mechanism to automatically derive product instances that dynamically adapt at runtime to accommodate particular user needs or likely changes in environmental conditions and resource constraints [111]. Consequently, DSPL engineering helps us design more dynamic software architectures and build more adaptable software to handle autonomous decision-making depending on environmental changes and (re)configuration plans to work in such environments [20, 77]. Moreover, it emphasizes variability analysis and design at development time and variability binding and reconfiguration at runtime [115]. Such reconfiguration at runtime requires variability mechanisms or adaptive mechanisms to trigger the reconfigurations or adaptations at runtime.

DSPL dynamically binds or unbinds features, via an adaptive mechanism, at runtime according to a Feature Model (FM) [9]. A **feature model represents commonalities and variabilities in a family of systems as well as relationships amongst features** [35, 69]. Thus FM represents the structural relationship consisting of a logical grouping of features (*e.g.,* **the features are grouped into Mandatory, Alternative, or Optional**). It thus describes which valid (re)configurations can be performed [35]. Also, the model has a high abstraction level and is used as a starting point for the feature selection to new products instantiation in design time or runtime. In this context, a **feature is a characteristic of a software system that satisfies a requirement, represents a distinctive user-visible aspect, and provides a potential configuration option** [2].

Figure 1.1 presents a concrete example of a Feature Model that represents a set of permitted selection features that can be used to instantiate a new type of Sensor in a software product line. This model is hierarchically arranged in a set of features, which with the main feature, *Sensor*, conceptually represents the product line domain. It

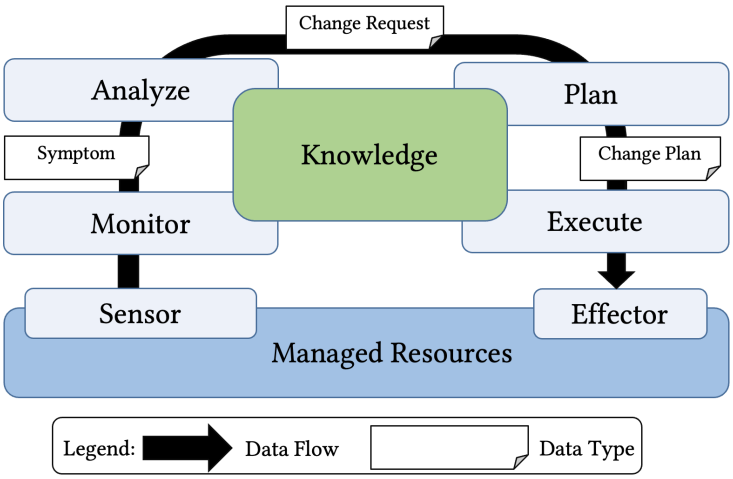

Figure 1.2: The MAPE-K loop.

has two mandatory compound features, *Connection* and *Type*. The mandatory feature *Type* has six child features, which are alternative: *Temperature*, *Gas*, *Presence*, *Luminosity*, *Water* and *Soil Moisture*, respectively. However, the mandatory feature *Connection* has three child features within an OR-feature relationship: *USB*, *Ethernet*, and *WIFI*. Besides, the model includes an optional feature: *LCD*.

1.2 Self-adaptive System

A self-adaptive system (**SAS**) **comprises a closed-loop system capable of evaluating and changing its behavior at runtime due to its requirements (*e.g.*, environment constraints and adaptation policies) and users' needs** [83,92]. Thus, SAS is a closed-loop system with feedback from the *self* (software requirements) and the *context* (operating environment) [109]. Consequently, the self-adaptations are typically implemented **using a closed-loop through key activities** responsible for the adaptation process: **Monitor, Analyze, Plan, and Execute over a shared Knowledge base, together forming a MAPE-K loop** [65, 134].

MAPE-K: The MAPE-K loop was proposed by IBM [65] and an example of this approach is shown in Figure 1.2. The Monitor gathers, aggregate, filters, and report details (such as metrics, configuration parameters, or other information that can trigger the adaptations) collected from a *managed resource* (hardware or software). In this context, a *managed resource* can be any resource type (hardware or software) that exists in the runtime environment of a system, and that can be managed (*e.g.*, sensor/probe and effector/actuator). In this context, a sensor serves as an interface that provides access to information regarding a managed resource's current state as well as any changes made to that state. On the other hand, an effector is an interface that allows for modifications to be made to the state of managed resources.

The **Monitor** also **correlates all information collected into *Symptoms* that can be further analyzed in Analyze**. The Analyze correlates and models complex situations (for example, time-series forecasting and queuing models) to understand the current system state, learn about the environment and help predict future situations. Thus, the **Analyze identifies whether the current system needs an adaptation at runtime and performs the *Change Request***.

The **Plan constructs the *Change Plan* based on the *Change Request* to achieve the goals and objectives for which the system was implemented, using the adaptations policy to guide its work**. The policy is a set of considerations designed to guide the decisions affecting software (*managed resource*) behavior.

The **Execute changes the behavior of the software (*managed resource*) using effectors based on the *Change Plan* built by the Plan**.

The **Knowledge is a set of data** (*e.g.*, symptoms and adaptation policies) **shared among MAPE activities and used in the closed loop to decide when the system needs to trigger an adaptation at runtime**. The data shared includes topology information, historical logs, metrics, symptoms, and adaptation policies, etc.

Self-* Properties: Self-adaptation is a process of (re)configuring or adapting the software architecture of a system as a reaction to changes in the managed resources or environment (context) of the system [14, 49, 74]. Thus, the systems implement the self-* properties to perform self-adaptation. One of the well-known sets of **self-* properties are self-configuring, self-healing, self-optimizing, self-protecting, self-awareness, and context-awareness** [75, 109, 127].

In the following, we further elaborate on each property's details based on [75, 109, 127].

- Self-Configuring is the ability to reconfigure automatically and dynamically in reaction to changes by installing, updating, and composing/decomposing software components at runtime.
- Self-Healing is the ability to discover, diagnose, and react to failure. This property can help to anticipate possible problems and takes proper actions to prevent failure.
- Self-Optimizing is managing performance and resource allocation to satisfy different users' requirements. For instance, this property can manage the response time, throughput, and workload.
- Self-Protecting is the ability to detect security breaches and recover from their effects. It has two aspects: i) defending the system against malicious attacks and ii) predicting problems and taking actions to avoid or mitigate their effects.
- Self-Awareness means that the system is aware of its self-states and behaviors. This property is based on self-monitoring, which reflects what is monitored.
- Context-Awareness means that the system knows its context (operational environment). Context is any information that can be used to characterize the situation of an entity (person, place, or object) that is considered relevant to the interaction between a user and an application [32].

1.3 Wrap up

This chapter presented the standard Dynamic Adaptive Systems (DAS) and focused on the Dynamic Software Product Line (DSPL) and Self-Adaptive System (SAS). Also, we discuss the concepts of runtime variability, feature model, feature, MAPE-K loop, and context awareness used in the approaches proposed in this thesis to identify the architectural issues and select tests at runtime.

CHAPTER 2

ARCHITECTURAL BAD SMELLS

In the software engineering literature, smells are categorized as code smells [41], design smells [121], and Architectural Smells (AS) or Architectural Bad Smells (ABS) [52]. The categorization depends on factors such as its scope and the impact on the rest of the system [110]. For instance, code smells have a limited impact at the class level. On the other hand, **ABS span multiple components impacting them at the system level**. Moreover, ABSs are analogous to code smells because they represent standard solutions that are not necessarily faulty or errant but still negatively impact software quality at different levels [51, 110].

Several authors [29, 42, 52] define an ABS as a set of **architectural design decisions that negatively impact system lifecycle properties**, such as understandability, testability, maintainability, extensibility, and reusability. ABS may arise by applying a design solution in an inappropriate context [52], combining design fragments with undesirable behaviors, or applying design abstractions at the wrong level of granularity [51, 77]. Consequently, ABS indicate possible design and implementation issues and fixing them can improve the system's quality [77].

The following section presents software architecture concepts (Section 2.1) and describes one of the literature's well-known sets of Architectural Bad Smells. Section 2.2.1 describes Connector Envy (CE). We define the Scattered Functionality (SF) smell in section 2.2.2. Ambiguous Interfaces (AI) smell is presented in section 2.2.3.

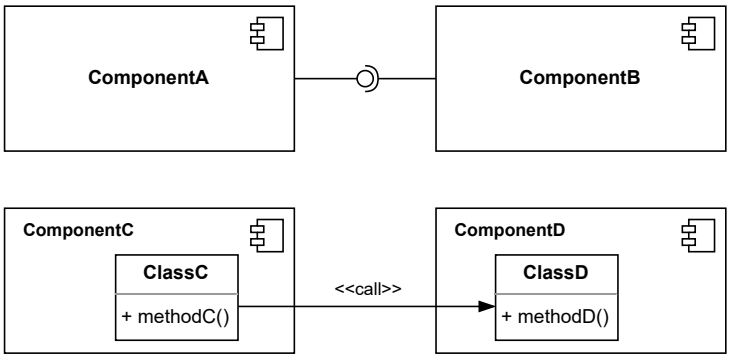

Figure 2.1: Example component and connectors implemented using UML diagrams.

The Extraneous Connector (EC) smell characteristics are introduced in section 2.2.4. Section 2.2.5 defines Cyclic Dependency (CD) smell. Hub-Like Dependency (HL) smell is explained in 2.2.6. Section 2.2.7 presents Oppressed Monitors (OM) smell. Finally, Section 2.3 discusses the related work.

2.1 Software Architecture

Bass *et al.* [11] define software architecture as a structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them [11]. This structure contains various models that depict different perspectives on the system's structure. To formalize software architecture and ensure some of its desirable properties, *architectural styles* can be used to describe component and connector types. In this context, architectural styles are a set of canonical architectural solutions to problems [11]. Thus, components provide specific functions for the system, while connectors enable communication between components. Figure 2.1 shows a component diagram illustrated in UML[1] notation. This diagram provides a visual representation of how different components interact with each other by employing diverse connectors. For instance, `ComponentA` and `ComponentB` communicate with others through an interface (connector type). Also, an object of type `ClassC` in `ComponentC` communicates with `ClassD` in `ComponentD` via a synchronous method call. An *architectural configuration* consists of a single component or multiple components communicating via connectors [97].

2.2 Architectural Bad Smells Description

This section describes seven different ABSs proposed in [50–52] and [114]. We aim to identify and fully grasp these smells by providing precise and concise definitions

[1] UML stands for Unified Modeling Language. For more information, please visit: https://www.omg.org/spec/UML

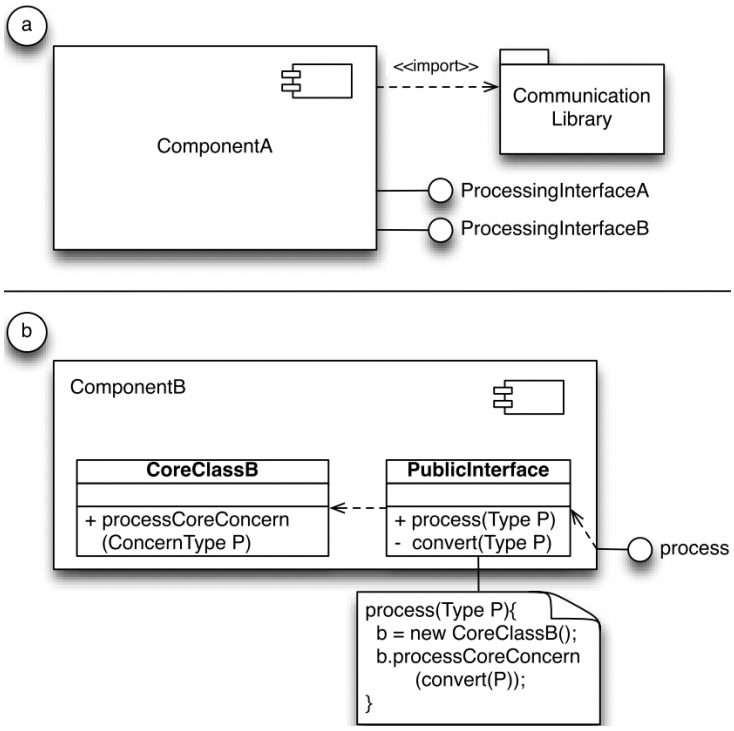

Figure 2.2: The first diagram (a) depicts Connector Envy smell involving communication and facilitation services. The second diagram (b) shows Connector Envy involving a conversion service [51].

based on standard architectural building blocks like components, connectors, interfaces, and configurations [51, 52]. We have also included one or more UML diagrams to represent each smell visually.

2.2.1 Connector Envy (CE)

Description: The components affected by **Connector Envy cover too much functionality that should be delegated to a connector.** Thus, this smell impacts the components encompassing extensively one or more of the following interaction services:

- **Communication** concerns the data transfer (*e.g.*, messages, computational results) between architectural elements.
- **Coordination** concerns the transfer of control between architectural elements.
- **Conversion** translates different interaction services between architectural elements (*e.g.*, conversion of data formats and types).
- **Facilitation** concerns the mediation, optimization, and streamlining of interaction (*e.g.*, load balancing or fault tolerance).

Figure 2.2 (a) illustrates an occurrence of a CE smell, where `ComponentA` imple-

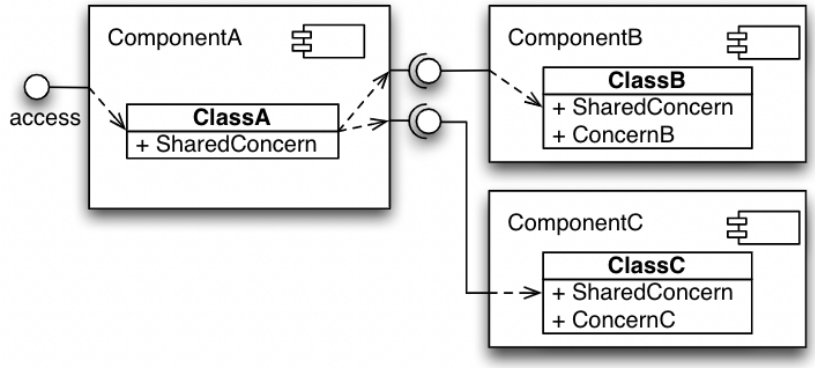

Figure 2.3: Scattered Functionality occurring across three components [51].

ments communication and facilitation services. `ComponentA` imports a communication library because it provides low-level networking facilities implementing remote communication. Consequently, naming, delivery, and routing services handled by remote communication are a type of facilitation service. Figure 2.2 (b) depicts another Connector Envy smell in which `ComponentB` performs a conversion during processing. `ComponentB` implements the `PublicInterface` class. `PublicInterface` implements the process method by calling a conversion method that transforms the `Type` parameter into a `ConcernType`.

Quality Impact and Trade-offs: The quality attributes affected by the occurrence of this smell are: i) **Reusability** is reduced because the dependencies are created between interaction services and application-specific services are hard to reuse either without including the other; ii) **Understandability** because disparate concerns are mixed. Thus, the component has functionality and connection responsibilities; iii) **Testability** because application and interaction functionality cannot be tested separately. For instance, a test failure could result from a fault in the interaction functionality or the application logic. Thus, developers must investigate two potential sources for the error instead of just one.

2.2.2 Scattered Functionality (SF)

Description: Garcia *et al.* [51] argue that SF smell describes **a system with multiple components responsible for realizing the same high-level concern**. Also, some of those components are responsible for orthogonal concerns. Consequently, they violate the principle of separation of concerns in two manners: i) this smell scatters a single concern across multiple components; and ii) at least one component addresses multiple orthogonal concerns. The components realizing scattered concerns depend on each other, and as a result, their reusability and modularity are reduced.

Figure 2.3 depicts three components responsible for the same high-level concern, `SharedConcern`, while `ComponentB` and `ComponentC` are responsible for orthogo-

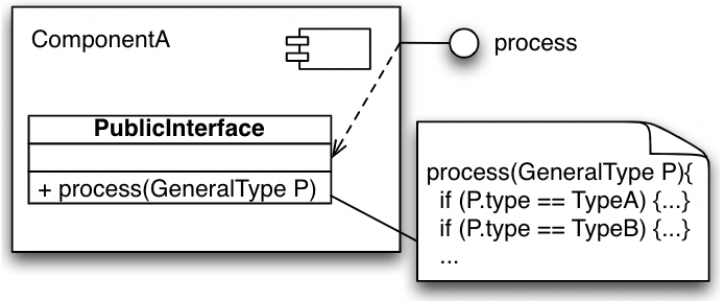

Figure 2.4: An Ambiguous Interface is implemented using a single public method with a generic type as a parameter [51].

nal concerns. Thus, `ComponentB` and `ComponentC` violate the principle of separation of concerns because they are responsible for multiple orthogonal concerns. Also, they cannot be combined into one component with `ComponentA` without creating a component that deals with more than one concern.

Quality Impact and Trade-offs: Scattered Functionality smell impacts modifiability, understandability, testability, and reusability in the components affected. Consequently, when the concern `SharedConcern` (depicted in Figure 2.3) needs to be modified, there are three possible components where `SharedConcern` can be updated and tested. Also, we cannot reuse `ComponentA` without `ComponentB` and `ComponentC`, thus affecting the reusability of each component. The smell reduces understandability because two components (`ComponentB` and `ComponentC`) are responsible for implementing orthogonal concern `SharedConcern`. Garcia *et al.* [51] argue that the Scattered Functionality smell may be acceptable when the shared concern must be provided by multiple off-the-shelf (OTS) components whose internals are unavailable for modification.

2.2.3 Ambiguous Interfaces (AI)

Description: **This smell arises when an interface offers only a single and general entry point into a component.** For instance, this smell **appears in *event-based publish-subscribe* systems** because the interactions are not explicitly modeled, and multiple components exchange event messages via a shared event bus (*e.g., communication broker*). Figure 2.4 depicts an occurrence of Ambiguous Interface smell, where two aspects are relevant: i) the interface offers only one public service or method (*e.g.,* `process(GeneralType P)`), even though its component offers and processes multiple services (*e.g.,* `TypeA` and `TypeB`); and ii) since the interface only offers one entry-point, the accepted type (*e.g.,* `GeneralType`) is consequently overly general.

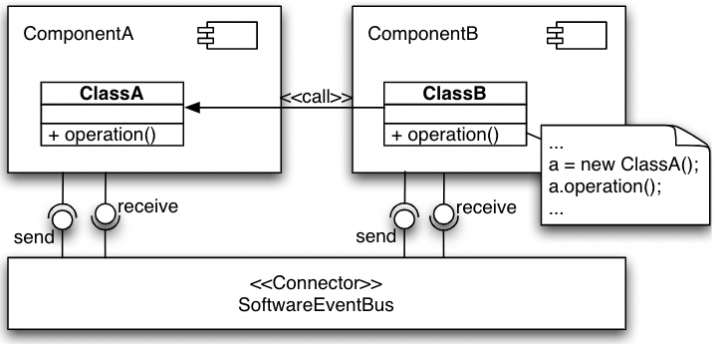

Figure 2.5: An Extraneous Connector occurrence example with an event- and a procedure-call-connector between two components [51].

Quality Impact and Trade-offs: The quality attributes affected by the occurrence of this smell are: i) **Analyzability** because the smell does not reveal which services a component is offering. Thus, the related component's implementation must be inspected before the services can be used. Garcia *et al.* [51, 52] argue that Ambiguous Interfaces also **reduce static analyzability at the architectural level** and can occur independently of the implementation-level constructs that realize them; ii) **Understandability** because the component does not specify what type of message is being passed through the interface.

2.2.4 Extraneous Connector (EC)

Description: **This issue occurs when two different types of connectors are used to connect two components.** While this problem can apply to any connector, **we will only focus on procedure call and event connectors.** Figure 2.5 displays an example of this issue, where `ComponentA` and `ComponentB` are connected through a combination of procedure call and event connectors. In this scenario, both components send events to the `SoftwareEventBus`, which then forwards them to the appropriate recipient. Additionally, an object of type `ClassB` in `ComponentB` communicates with `ComponentA` through a synchronous method call to transfer data and control via a service interface.

Quality Impact and Trade-offs: The quality attributes affected by this smell are: i) **Adaptability** because the senders and receivers of events are unaware of each other. However, the components with procedure calls may be difficult to adapt, as shown in Figure 2.5, because we should adapt both components; ii) **Reusability** is affected because two different links are used to connect the components. Consequently, they have a strong dependency on each other, and this aspect makes it difficult to reuse in different contexts; iii) **Understandability** is affected because it is unclear under what circumstances the additional communication occurs between the two components.

Garcia *et al.* [51, 52] argue that EC smell may be acceptable when a standalone desktop application uses both connector types to handle user input via a graphical user interface (GUI). In this context, event connectors are not used for adaptability advantages but to allow the asynchronous user handling of GUI events.

2.2.5 Cyclic Dependency (CD)

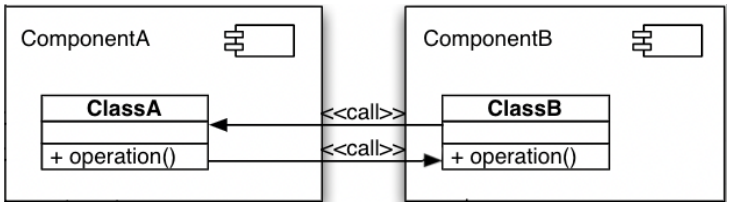

Figure 2.6: An Cyclic Dependency occurrence between two components.

Description: Ganesh *et al.* [50] claim that **this smell arises when two or more class-level abstractions depend on each other directly or indirectly** to function correctly. The components involved in a cycle dependency can hardly be maintained, tested, or reused in isolation. Figure 2.6 depicts two components (`ComponentA` and `ComponentB`) involved in cyclic dependency via `ClassA` and `ClassB` that has direct dependency between both classes.

Quality Impact and Trade-offs: The quality attributes affected by the occurrence of this smell are: i) **Reusability** is reduced because the dependencies between components are hard to reuse without including the other. For instance, to reuse `ComponentA` (in Figure 2.6), the developers should include `ComponentB`; ii) **Testability** because tracking and properly testing all components involved in CD may be difficult depending on the system's complexity. CD forces to execute unrelated system parts, increasing testing complexity; iii) **Maintainability**: when a component related to a concern needs to be modified, it may trigger negative impacts on components involved in CD.

2.2.6 Hub-Like Dependency (HL)

Description: Díaz-Pace *et al.* [33] claim that **this smell arises when an abstraction (*e.g.*, classes or packages) has *outgoing* and *ingoing* dependencies with many other abstractions**. Figure 2.7 depicts seven components, where `ComponentA` provides a message service to send and receive data. In this context, `ClassA` uses `ClassF` and `ClassG` to implement a message service provided to `ComponentB`, `ComponentC`, `ComponentD`, and `ComponentE`. Consequently, `ClassA` is a highly used system class because all message service is concentrated on it. This structure is undesirable, as it increases the potential effort necessary to change all abstractions involved.

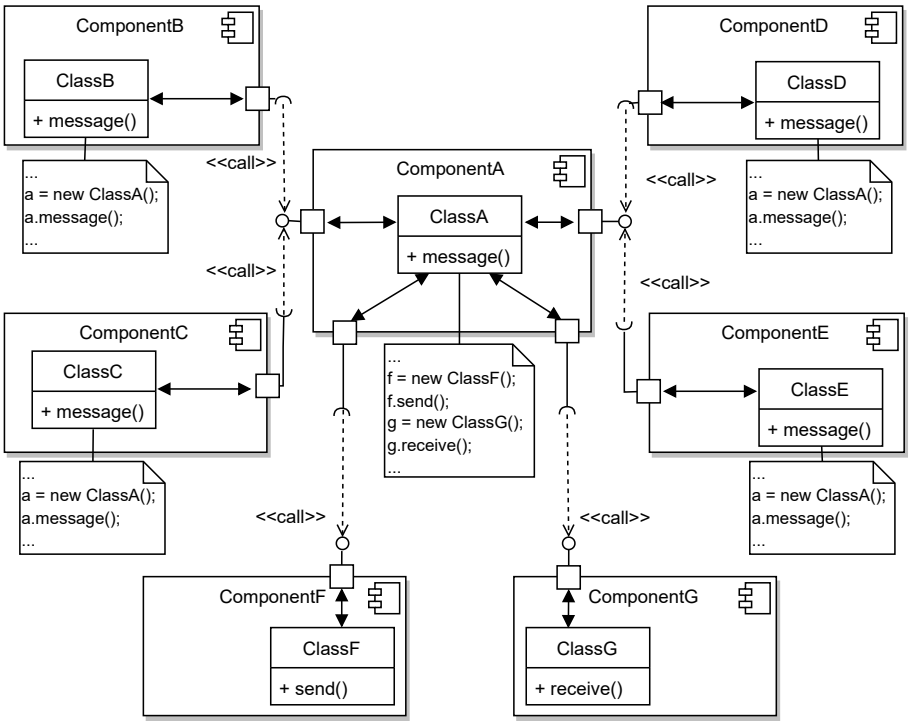

Figure 2.7: Hub-Like Dependency example.

Quality Impact and Trade-offs: The quality attributes affected by the occurrence of this smell are: i) **Maintainability** because even a slight modification in a class affected by HL smell will generate a ripple effect in all classes or packages directly depending upon them. For instance, a modification in `ClassA` (Figure 2.7) will require adapting at least all the dependent classes; ii) **Testability** because the class affected by HL smell cannot be tested separately due to ripple effects triggered by its modification. Therefore, all the dependent classes should be tested with the central component.

2.2.7 Oppressed Monitors (OM)

Description: Serikawa *et al.* [114] claim that this **smell is characterized by a set of monitors** that shows the following characteristics:

- **They are independent of each other concerning the data manipulated.**
- **They have the same polling rates.**
- **The execution order of the monitors is predetermined in compilation time and unmodifiable at runtime.** Such a situation occurs due to bad implementation decisions leading all monitors to be included in a unique loop that takes from their autonomy.

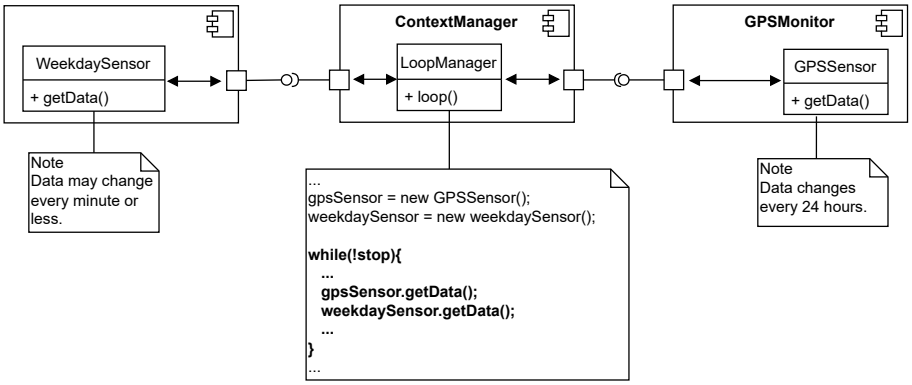

Figure 2.8: Oppressed Monitors smell example.

Consequently, OM indicate that the monitors are forced to have the same polling rate and an immutable execution order at runtime. Figure 2.8 illustrates OM smell occurrence in a simplified scenario composed of three components (`ContextManager`, `WeekdayMonitor`, and `GPSMonitor`). The `ContextManager` component manages the loop to collect the data provided by `WeekdayMonitor` and `GPSMonitor`. In the `WeekdayMonitor`, the data changes every 24 hours. Moreover, the `GPSMonitor` data may change every minute or less, depending on the context detected. However, the loop structure implemented in `LoopManager` class needs to comply with the GPS's high polling rate of change leading to a useless weekday monitoring process implemented in `WeekdaySensor` class. This smell leads to unnecessary waste of resources because the `WeekdayMonitor` is operating at a polling rate faster than necessary.

Quality Impact and Trade-offs: This smell is acceptable only when there are simple monitors with similar polling rates because creating separated monitors as parallel processes may lead to an overhead problem. Also, the quality attributes affected by the occurrence of this smell are: i) **Maintainability** of the system is impacted due to the lack of modularization. Therefore, if a monitor needs to be adjusted, it may directly affect the behavior of other monitors once they are highly coupled; ii) **Testability** because the monitors affected by OM smell cannot be tested in isolation due to having the same polling rate and unchangeable execution order. Moreover, this leads to unnecessary data capturing, performance issues, or data loss because the polling rate was unrespected. Consequently, test results may not represent the actual behavior of each monitor under the test.

2.3 Related Work

In this thesis, we are interested in studying the detection of ABS at runtime in self-adaptive systems. So, we searched the literature and found two works dedicated to identifying ABS in self-adaptive systems. The first study [106] relies on the Arcan [44]

tool to identify ABS in 11 self-adaptive systems. Arcan [43] 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. In this thesis, we want to detect ABSs in self-adaptive systems at runtime. Thus, the approach proposed in Chapter 6 of this thesis also uses a graph for ABS detection, but there are two differences:

(i) **We create a map for each SAS configuration identified at runtime**;

(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.

Also, our work presented in Chapter 9 involves the comparison of Arcan and the BM for runtime smell detection [36].

The second study [114] presents two new **ABSs specific to self-adaptive systems: Obscure Monitor and Oppressed Monitors**. Also, it defines the algorithms to identify each ABS at design time. To validate the proposed smells, the authors identified the proposed smells in 8 SASs and discussed how to refactor the system affected by those smells. We try to detect the Oppressed Monitors ABS at runtime using the approach proposed in this thesis in Chapter 6.

2.4 Wrap up

This Chapter presented the essential background on Architectural Bad Smells, discussing the most known ABS available in the literature. We described the **ABS characteristics, quality impact, and trade-offs** for each ABS. Also, we included the two ABSs detected only in self-adaptive systems because we are interested in studying ABS detected in such systems. However, all ABSs presented are **detected at design time** and do not consider the self-adaptability employed by Self-adaptive systems at runtime. Such an aspect may hide or increase the number of ABSs detected in SASs [36, 77]. We define in **Chapter 6 our approach to detecting Architectural Bad Smells in Self-adaptive Systems based on the bind and unbind of features at runtime**. This thesis will focus only on CD, EC, HL, and OM smells because they have been studied previously in SASs [106, 114].

CHAPTER 3

RUNTIME VALIDATION

Self-Adaptive Systems (SAS) are equipped with feedback loops (adaptation mechanisms) to self-adapt to user requirements or environment changes. Consequently, identifying and testing every operational context that a SAS may encounter at runtime is impossible [48]. Also, the number of possible (re)configurations at runtime varies exponentially with the number of features provided by the system. Thus, SAS needs test strategies to deal with an unknown variability space at runtime or untested configurations before the software deployment. In this context, runtime tests may ensure that the SAS functions appropriately, even in unpredictable scenarios. The test of SAS should provide that the new configurations satisfy requirements at runtime and consequently may require test generation at runtime to guarantee testing relevance, where relevance indicates the applicability of a test case to its environment [47].

This Chapter discusses the software testing background and test approaches used to generate test cases to support runtime testing. Section 3.1 presents the software testing concept. Section 3.2 discusses the test approaches focusing on Random Testing, Search-based Software Testing, Combinatorial Test Design, and MAPE-T.

3.1 Software Testing

Within software development, software testing is an essential component of the overall development process. Multiple interpretations exist of what constitutes software testing [16, 82, 103, 117], but regardless of the specific definition, it remains a critical step in ensuring the quality and functionality of the final product. In this direction, the Software Engineering Body of Knowledge (SWEBOK) [16] defines *software testing* as: "*Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.*".

In the realm of software testing, when we use the term *Dynamic*, we are referring to the practice of executing a program with specific inputs to assess its functionality. However, it is necessary to remark that this approach only tests a *finite* subset of all potential scenarios, which have been selected based on their level of risk and priority. The *selection* of tests can significantly vary depending on the technique employed. Thus, it is crucial for software engineers to understand that different selection criteria can have a major impact on the effectiveness of testing. Lastly, the term *Expected* pertains to the ability to evaluate whether the observed outcomes of testing meet the predetermined acceptance criteria. Without this ability, testing would serve no purpose. Thus, testing refers to the process of dynamically analyzing software by executing the system and comparing the results to the expected outputs [82].

The testing process is of utmost importance in identifying defects within software systems. It is widely recognized as the fundamental method for evaluating software quality in real-world scenarios. However, the testing accomplished at each level of software development is different and has distinct objectives [82], as described in the following.

- **Unit Testing** involves **testing the smallest testable piece of software**, also known as the "*unit*", "*module*", or "*component*". This is made at the most basic level to ensure that the software functions properly.
- **Integration Testing** **combines two or more tested units into a more significant structure**. This type of testing is carried out on the interfaces between the components and the larger structure being built, especially if the quality of the structure cannot be evaluated based on its components alone.
- **System Testing** is a **way to ensure the overall quality of an entire system** from beginning to end. It relies on the functional and requirement specifications of the system but also checks for non-functional quality attributes like reliability, security, and maintainability.
- **Acceptance Testing** is **intended for the delivery of software to customers or users**. This testing aims to ensure that the system is operating as intended rather than to discover errors.

In the following Sections, we present some test approaches available in the literature.

3.2 Test Approaches

The following sections present some test approaches used to generate and execute tests at runtime.

3.2.1 Random Testing

Random testing is a technique that proves to be highly beneficial in specific scenarios where there is no need for an in-depth understanding of the internal structure of the software [59, 70]. This method is particularly effective when the output of each test can be automatically verified [70]. To generate random values, one can either opt for a manual generation or employ a pseudo-random number generator.

Kaur and Singh [70] have pointed out that relying solely on pure randomness for generating test cases is a seldom-used approach. Instead, modern testing tools such as Randoop[1] [98] employs a more sophisticated technique known as Feedback-directed Random Test Generation [99]. This approach involves using feedback obtained from executing test inputs to steer the search toward sequences that produce new and legal object states. Inputs that lead to redundant or illegal object states are disregarded. By applying this technique, the search space is significantly reduced, leading to more efficient and reliable test case generation.

3.2.2 Search-based Software Testing

Search-Based Software Testing involves using meta-heuristic optimizing search techniques, such as Genetic Algorithms, to automate or partially automate testing tasks [84, 116]. This technique involves creating a group of potential test cases called candidate solutions. The algorithm then utilizes a fitness function to determine how close these candidate solutions are to meeting a coverage goal. The search is guided by this fitness function, which improves with each generation until a solution is found. This method is effective in ensuring comprehensive coverage of the testing process [84].

Modern tools like EvoSuite[2] [45] use a search-based approach integrating cutting-edge techniques like hybrid search [63], dynamic symbolic execution [53] and testability transformation [62] to automatically generates test cases with assertions for Java code classes. The tool supports coverage criteria such as line, branch, method, exception, output, and weak mutation testing as test objectives. Also, Evosuite minimizes the tests, meaning only the ones contributing to achieving coverage are retained [46].

[1] Randoop is a unit test generator for Java. More information is available at: https://randoop.github.io/randoop/

[2] EvoSuite is an automated tool for generating test cases for Java classes. For more information, please visit: https://www.evosuite.org/

3.2.3 Combinatorial Test Design

Combinatorial Test Design (CTD), also referred to as Combinatorial Testing (CT) [76, 94] or Combinatorial Interaction Testing (CIT) [28, 94], is a widely recognized approach used to develop efficient and highly effective tests [124]. The methodology involves the identification of combinations of input values, allowing for testing multiple scenarios in a single test case. This approach reduces the number of tests required while ensuring that all possible combinations are thoroughly examined, resulting in improved test coverage and a more comprehensive testing process.

The use of CTD offers two advantages [94,124]. Firstly, it allows for coverage goals to be calculated based on an adjustable level of interaction, which helps manage testing costs. Secondly, CTD generates an optimized set of coverage goals at a specific interaction level, resulting in a reduced number of tests needed. These aspects are essential to test self-adaptive systems (SAS) at runtime because they allow dealing with the exponential growth of SAS configurations that should be tested. They provide a practical way to detect failures caused by parameter interactions with a good trade-off between cost and efficiency. Thus, CTD can help to detect failures triggered by the interactions among parameters in the SUT [94].

The CTD is supported by the TackleTest[3] [124] tool for automatically generating of unit-level test cases for Java applications. It also combines different code coverage criteria, such as statement, branch coverage, and type-based combinatorial coverage [73].

3.2.4 MAPE-T

The MAPE-T is a feedback loop used to enhance testing strategies with runtime capabilities [48]. It is based on the MAPE-K [65] architecture for adaptive systems, which involves monitoring, analysis, planning, and execution. Consequently, the MAPE-T comprises four parts described as follows.

- **Monitor:** Monitor and identify changes within the system and its environmental context.
- **Analyze:** Identify individual test cases to adapt and select specific tests to execute at runtime.
- **Plan:** Adapts test inputs and outputs as needed and schedules runtime testing.
- **Execute:** Perform the planned test plan and analyze results, adjust expected outcomes, and trigger adaptations within the SAS, its requirements, or the testing framework.

Such a testing approach manages the adaptation and execution of runtime testing for a SAS as it self-adapts [47]. Consequently, the process of monitoring MAPE-T involves gathering data about the system as well as its surrounding environment [48]. This data is then utilized by the analysis and planning processes to determine which components of the SAS require testing and how to adjust the inputs and outputs of test cases. The execution process runs and evaluates the results of tests to decide if adaptation is required.

[3] TackleTest - https://github.com/konveyor/tackle-test-generator-cli

3.3 Wrap up

This Chapter presented the necessary background on software testing and test approaches focusing on Random Testing, Search-based Software Testing, Combinatorial Test Design, and MAPE-T. In this thesis, we employ the test strategy Random Testing, Search-based Software Testing, and Combinatorial Test Design to generate test cases at runtime. Such strategies were adopted to face the runtime test challenge described in Section 4.2 of Chapter 4.

CHAPTER 4

PROBLEM STATEMENT

In this chapter, we discuss the challenges runtime variability raises on Architectural Bad Smells (ABS) detection and argue that we should analyze Self-Adaptive Systems (SAS) architectures at runtime, as addressed in our paper Architectural Bad Smells for Self-Adaptive Systems: Go Run-time! [78][1]. Additionally, we highlight the runtime testing challenge for SAS, focusing on the impact of runtime variability on tests and how to generate test cases automatically.

4.1 Architectural Bad Smells Challenge

The literature on Self-Adaptive Systems (SAS) encompasses approaches to support **Architectural Bad Smells (ABS) identification at design time through static analysis** [106]. Such approaches enable the program source code analysis statically without executing it. However, **it does not consider the system's (re)configuration process at runtime** [106] **and the variability space**. In particular, we argue that we cannot infer the whole variability space for two reasons. First, since most SAS do not document configuration options, it is difficult to analyze them automatically. Second, SAS are realizing open variability [125] at runtime thanks to variability mechanisms such as polymorphism and via the possibility to download new features on the

[1] Paper presented at 17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2023) - https://vamos2023.sdu.dk/

fly (*e.g.,* the code for a *plug-and-play* sensor [111]). A particular characteristic of a SAS is to reconfigure dynamically at runtime. A SAS might change its behavior due to unexpected environmental changes, reconfiguration plans, and goals [34]. The adaptations at runtime may affect architectural qualities and properties, given that the (re)configuration process may combine architectural fragments or apply architectural abstractions at the wrong granularity level through the newly loaded features [77].

Based on such observations and our experience [77], we have devised the following seemingly controversial idea: ***achieving an effective ABS identification in SAS will only be possible at runtime***, once variability is bound. Accordingly, we ***strongly encourage carrying out dynamic analysis in addition to/rather than solely relying on static analysis***. It contradicts the common practice of identifying architectural smells only at design time. The following section motivates why this current practice is doomed to fail.

4.1.1 Runtime variability's impact on SAS architectures

Even if not implemented as such (see below), one can see SAS (re)configuration as activating and deactivating features at runtime. Not taking this aspect into account leads to inaccurate reports on the existence and importance of ABS runtime. For instance, in a recent study, we compared ABS detected at design time and runtime [36]. We observed significant differences between smells' occurrences at such different binding times for the Adasim project [137]. In addition, some smells appearing at runtime could not be found at design time for the mRUBiS project [128].

4.1.2 Lack of variability documentation in SAS

Variability management is crucial for SAS [36], and this lead the research community on variability management to coin the concept of *dynamic software product lines* [13] (DSPLs). DSPLs realize SAS by carefully modeling SAS adaptations using variability models and tracing variability down to implementation artifacts, *e.g.,* SHE [111] is a SAS implemented using the DSPL engineering. This would allow the variability-aware analysis of SAS and possible extension of the variability-aware code smells [40, 120] to ABS. However, most SAS are not implemented as DSPLs. For example, none of the Java-based exemplars provided by the SEAMS community[2] had any variability documentation (feature model, feature annotations). Dos Santos *et al.* introduced a manual process to identify source code features based on information available in the system's repository [77], more details available in Section 6.3.1. However, adding a mechanism in the systems' source code for ABS identification requires expertise and time because the mandatory and variable features are not documented.

```
1: procedure ADAPTATIONMECHANISM
       while !isFinished() do
       dataLoad();
       dataAnalysis();
       runAdaptationStep();
       end
2: end procedure
3: procedure RUNADAPTATIONSTEP
4:     featureIdentification();
5:     bindingFeatures();
6: end procedure
```

Algorithm 1: Interception loop design.

4.1.3 Capturing adaptations

For identifying ABS at runtime, it is necessary to run the system and identify the exact moment each adaptation starts and ends [107]. It is also necessary to capture all features and dependencies loaded in the adaptation loop at runtime. This task is challenging because it is necessary to identify the method responsible for executing the adaptation loop and the invoked methods inside it. Algorithm 1 illustrates such a scenario using a simplified MAPE-K loop [65] implementation. The `adaptationMechanism()` method is responsible for executing the system's adaptation mechanism. It uses a loop to execute the adaptation process encompassing `dataLoad()`, `dataAnalysis()`, and `runAdaptationStep()` methods. The first method reads the data from the environment, *e.g.*, sensor data, and sends them to the `dataAnalysis()` method. The `dataAnalysis()` defines the features we should activate to support the adaptation required at runtime. Then, the `runAdaptationStep()` method performs the adaptation. We adopt a runtime monitoring approach to this challenge by observing the evolution of methods and objects, progressively identifying the code responsible for the adaptation, and tracing methods entries and exits [77].

4.1.4 Handling polymorphism at runtime

Some SAS architectures are implemented based on polymorphism through abstract classes or interfaces. Polymorphism is a strategy to support variability at runtime [107]. Such a strategy could hide the absolute number of features involved in Cyclic Dependency (CD) and Hub-Like Dependency (HL), particularly when the analysis (of ABS) only considers the design time [36]. This is due to the analysis taking only concrete classes into account. Figure 4.1 shows a simplified architecture model of a Traffic Routing system. The model shows that the `Vehicle` class uses the Core and `Vehicle Routing` interface to bind a specific routing (*e.g.*, `QRouting`, `LearningRouting`, and `LinearRouting`) mechanism at runtime for each `Vehicle`

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

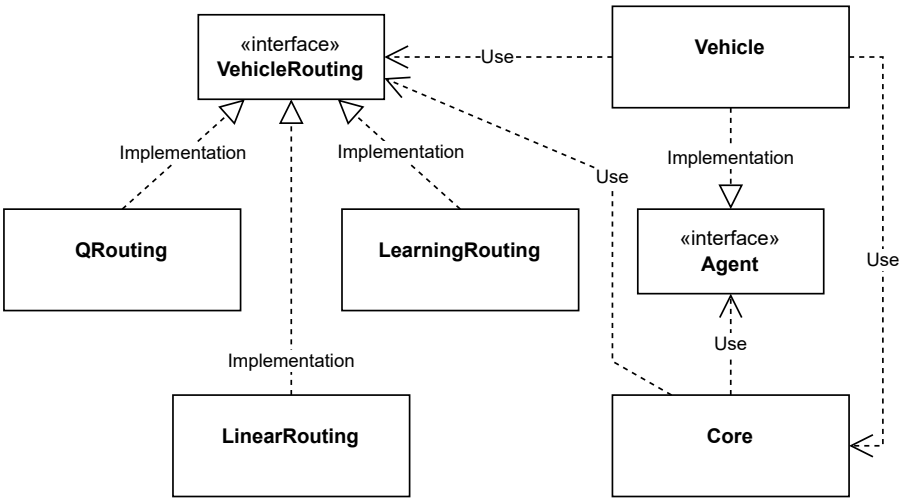

Figure 4.1: Traffic Routing system simplified architecture (simplified).

instantiated. Also, the `Vehicle` class implements the `Agent` interface used to connect the system core, and each agent type is instantiated at runtime. The system core can use `VehicleRouting` (*e.g.,* `LinearRouting`) to manage vehicles with a specific routing type at runtime. In this scenario, the cyclic dependency between `Vehicle` and `Core` will happen only at runtime. Thus, the static analysis does not identify that type of ABS at design time because there is no direct relationship among all classes involved in CD. Also, the same situation may happen with classes involved in HL.

4.2 Runtime Test Challenge

Self-Adaptive Systems (SAS) target environments with hard-to-predict, highly dynamic, and comparatively resource-constrained conditions [31]. Consequently, SAS characteristics imply **the most significant challenge in testing such systems because tests at design time cannot foresee or anticipate all (re)configurations supported by the system at runtime**. SAS testing should ensure that the new configurations satisfy requirements at design time and runtime [47]. Thus, due to the runtime aspects of SAS, it is necessary to perform the Validation and Verification (V & V) tasks incrementally and combined, both at design and runtime [30]. In this context, runtime testing arises as a complementary technique to support the V & V of SAS at runtime. The following sections discuss the challenge of performing runtime validation in SASs.

4.2.1 Runtime variability's impact on SAS Tests

The unpredictability of configurations of the SAS is the most significant challenge to testing because the number of possible configurations goes exponentially with the number of features provided by the system. Thus, the test team needs to deal with the following challenges, as identified in [117, 118].

(i) **How to deal with the exponential growth of SAS configurations that should be tested** - there is considerable difficulty in defining the **test suite scope** to execute on a SAS due to its number of possibilities for (re)configuration, as discussed in [22, 48, 86, 91, 123, 130, 136].

(ii) **How to guarantee the correctness of SAS configurations that have never been tested in advance** - the challenge is to define **test cases** to cover unforeseen configurations and the **testing oracle**. Many works [19, 95, 96, 130] mentioned the difficulty of testing a system that does not have a clear boundary due to (re)configurations at runtime. Also, other work focuses on the difficulty of defining a testing oracle for testing the SAS [80, 86, 100].

(iii) **How to detect and avoid during the testing activity incorrect system configurations defined at runtime** - In this context, it is not easy to dynamically define test cases to avoid incorrect settings so that they guarantee continuous system operation. Consequently is not easy to detect and avoid incorrect system configurations at runtime [86, 95, 96].

(iv) **How to anticipate all the relevant context changes and when they could impact the behavior of SASs** - the SAS context changes may affect their behavior at any time during the execution. In this scenario, the challenge is identifying reliably significant adaptive changes within the system and its execution environment. For instance, a reconfiguration may trigger a **feature interaction problem** at runtime due to an unknown context at design time and consequently crash the SASs. Such an issue is **unpredictable at design time** due to the number of possible configurations at runtime. Thus, building a test set that properly encompasses all relevant context variables with representative values is a significant issue [24, 118] because design-time approaches for testing SASs focus on a specific subset of the known context.

(v) **How to deal with context-dependent control and data flow in SASs** - Applying data flow testing criteria in SAS is arduous due to environmental interference and the context-aware nature of control flow and data flow-related faults [80, 91]. Therefore, the challenge is designing test models that include control flow graphs (with associated data flow information) and defining test cases to cover the properties of those models [118].

(vi) **How to simulate a realistic SAS execution environment and workload** - This issue permeates to building SAS testing to simulate execution environments and workloads realistically [71, 81, 86, 108]. However, unpredictability and unclear system boundaries directly impact SAS realistic simulations and workloads.

(vii) **How to define formal models for testing of changing behavior** - The focus of this challenge is to combine model-checking and testing techniques.

Weyns [131] discussed the challenge of defining formal models to validate by considering the systems' adaptive properties. Another issue is the need to specify and formalize SAS context-aware behavior to enable verification tasks [86, 138].

(viii) **How to define generic testing approaches for any adaptation process** - Defining generic validation approaches for any system adaptation process can be complicated because each adaptive system uses different adaptive mechanisms to manage its adaption process [117]. For example, Adasim [137] uses a parameter-based routing algorithm, while mRUBiS [128] can use various adaptation mechanisms (*e.g.,* MAPE-K, Event-Condition-Action) depending on the selected configuration. The challenge is to conceive a testing approach that can be applied to both systems during runtime. Some papers [39, 64] have discussed this challenge and suggested defining validation approaches that should be generic for any system adaptation process or, at least, to a specific subset of applications.

4.2.2 Test cases generation

One challenge to testing SASs at runtime is to generate test cases for the changing environment [117, 118]. Consequently, we must generate test cases for each configuration detected at runtime [104, 126, 139]. It may increase the system overload and consequently decrease SAS performance at runtime due to the time employed in test case generation, the number of test cases generated, and the time used to execute the tests. Thus, the challenge is reducing the number of automatically generated tests. In this context, how can the number of test cases be reduced while maintaining the fault detection capability? Vassev *et al.* [126] addressed this issue using two techniques for test case generation: one using random selection and another using a change-impact analysis approach. The change-impact analysis determines the test attributes concentrating on the effect of a change in particular events or actions employed by an execution path. Thus, the generated test suites are composed of fluent execution paths and test attributes [126].

4.3 Wrap up

This chapter presented the challenges to detecting Architectural Bad Smells (ABS) at runtime. Also, we made the case to switch from the classic design time and static detection of architectural bad smells to a more dynamic, runtime perspective when considering intrinsically variability-aware self-adaptive systems. Additionally, we have been involved in developing this perspective, providing methods (see Chapter 6) and tools (see Chapter 7) to identify smells at runtime and overcoming the previous challenges [77].

Furthermore, we have thoroughly discussed the intricacies of conducting runtime testing and creating automated test cases for SAS. Our solution involves leveraging the power of the Behavioral Map Black Box (refer to Section 7.3) to seamlessly

and automatically select, generate, and execute testing based on the interaction of features or ABS detected at runtime for every adaptation performed.

Part II

Behavioral Map Framework

CHAPTER 5

CASE STUDIES

In this Chapter, we present a variety of case studies used throughout the thesis. These case studies include the Smart Home Environment (SHE) framework [111], Adasim [137], and mRUBiS [128] systems. They were featured in our published book chapter, **Behavioral Maps: Identifying Architectural Smells in Self-Adaptive Systems at Runtime** [77]. Additionally, we have included the DeltaIoT system [67], which was used in Chapter 10.

We employed the Behavioral Map (BM) framework to assess the software quality of four distinct systems: SHE, Adasim, mRUBiS, and DeltaIoT. The commonality among these systems is that they were all developed using Java 8 programming language. These specific systems were selected because they exhibit a range of adaptive mechanisms, and their descriptions and implementations were readily accessible. Also, our analysis focused solely on self-adaptive systems with complete implementation. It did not consider the frameworks (such as JDEECo[1] [72]) utilized during

[1] More information available at: https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/v2v-deeco/

Table 5.1: Systems used in this thesis.

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
DeltaIoT	Architecture-based adaptation	MAPE-K	Internet of Things

development or the tools (such as Lotus@Runtime[2] [8]) used for SAS verification at runtime.

The Adasim, mRUBiS, and DeltaIoT systems were obtained from the SAS community repository[3], which provides a variety of SAS reference systems. These systems have been thoroughly examined and analyzed during the “Software Engineering for Adaptive and Self-Managing Systems" symposium and community[4]. Additionally, they were chosen for a study on ABS for SAS in the design time published in [106] and runtime published in [35, 77].

Table 5.1 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** - 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 and the runtime variability's impact on self-adaptive systems tests at runtime. In the following sections, we present each selected system and its configurations under evaluation.

5.1 Smart Home Environment (SHE)

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*,

[2]Complementary information available at: https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/lotusruntime/

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

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

Figure 5.1: Behavioral Map (BM) for one SHE configuration.

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. 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. This configuration of SHE is depicted Figure 5.1. 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.

5.2 Adasim

Adasim is a simulator for the Automated Traffic Routing Problem (ATRP)[5], implemented as an agent-based system [106, 137]. 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 [137]. The system employs adaptive mechanisms to deal with the scalability problems and the unpredictable changes in the environment, for instance, an accident.

5.3 mRUBiS

mRUBiS[6] is a marketplace based on RUBiS [101], comprising 18 components and can arbitrarily host many shops. These shops manage items, users, auctions/purchases, inventory, and authenticate users. Also, mRUBiS is a model-based architectural self-healing and self-optimization exemplar that can be expanded upon [101]. Consequently, the system supports different adaptive mechanisms [128], as shown in table 5.1.

5.4 DeltaIoT

DeltaIoT [67] is a multi-hop communication Internet-of-Things (IoT) system. It means that each IoT device (mote) must be able to communicate with other devices in order to reach the gateway. The system is designed to adjust its settings in response to different uncertainties. DeltaIoT[7] also includes a simulator for offline testing and a physical setup of 25 motes[8] accessible for remote field experimentation. The system comprises features: i) Probe used to collect data; ii) Effector used to execute an action; iii) Mote used to represent the devices; iv) Link; v) LinkSettings contains the source and destination node of the link, the transmission power and spreading factor to be used to communicate via the link and the distribution factor for the link.

5.5 Threats to validity

The case studies introduced in this Chapter are used to evaluate architectural bad smells detection and test case selection at runtime. It is important to note that the case studies we have provided may not be representative of all dynamic adaptive systems currently in existence. To mitigate the validity threats, we choose systems from various application domains with different adaptive mechanisms, as presented in Table 5.1.

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

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

[7] Find more information at: https://people.cs.kuleuven.be/~danny.weyns/software/DeltaIoT/

[8] The motes use LoRa radio technology supporting long-range communication, more information available at: https://www.lora-alliance.org/What-Is-LoRa/Technology

5.6 Wrap up

This Chapter considers different case studies that represent different kinds of systems and come from different sources.

CHAPTER 6

BEHAVIORAL MAP

This chapter gives the Behavioral Map (BM) concept proposed in this thesis. The **BM** supports **feature interaction issues** detection, **Architectural Bad Smell** (ABS) identification, and **testing selection** based on **the analysis of each runtime configuration** [34, 77]. We tackle the feature interaction, testing selection, and architectural issues (*e.g.,* ABS) by introducing the **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. Also, Sections 6.1, 6.2, 6.3, and 6.4 were published in the book Software Architecture (Lecture Notes in Computer Science, vol 13365) as part of the chapter **Behavioral Maps: Identifying Architectural Smells in Self-Adaptive Systems at Runtime** [77].

6.1 Overview

Inspired by Dynamic Software Product Lines (DSPLs) [9, 13, 20, 111], we consider SAS adaptations as *interacting feature configurations*. In a (D)SPL, one describes features and their dependencies in Feature Model (FM) [69] and traces their realization in

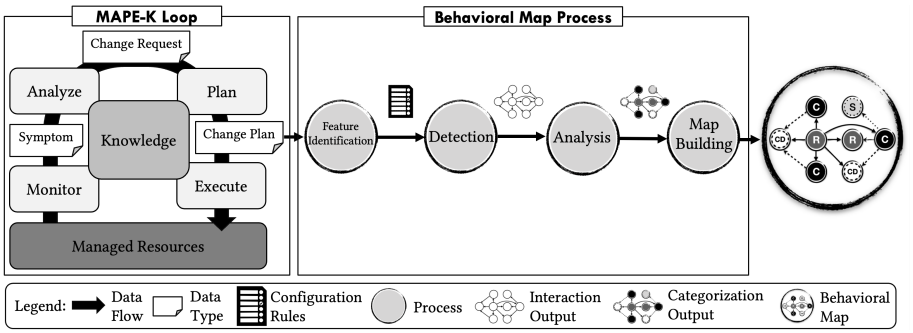

Figure 6.1: Behavioral Map (BM) process overview.

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 6.3). 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 [34]. 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 [65]. We depicted it on the left side of Figure 6.1, 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. The *Change Plan* is a plan or series of actions that outlines the necessary modifications to be performed at runtime to execute a particular adaptation [65]. For instance, the plan includes the list of features that should be binding and unbinding or the configuration policy used to install each feature.

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 [65] 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 [4, 17, 25, 61, 77].

To build a **BM**, we follow the process described in Figure 6.1. 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 6.1) 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 7.1). 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.

6.2 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 in a configuration,
- $VTypes$ = {Core, Optional, Controller, Sensor, Actuator, Presenter} where:
  - **Core** - represents the features that are included in all product configurations [12, 69];
  - **Optional** features refer to features that can be included in one configuration of the software but omitted in others [12, 69];
  - **Controller** is a feature that adjusts the target system by performing actions through an actuator based on sensor information [18]. Also, a feature controller can be control the behavior from other feature at runtime [111];
  - **Sensor** is a feature used to obtain data from the managed resources [65];
  - **Actuator** is a feature used to perform operations on the managed resource [65];
  - **Presenter** is a feature responsible for displaying data from the features that communicate with the system's features [111],
- $vtype : V \rightarrow \mathscr{P}(VTypes) \setminus \emptyset$ 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 $r \in ETypes$ = {Controls, Reads, Suppresses, Requires} where:
  - **Controls:** In the context of self-adaptive systems, "controls" refers to a relationship where one feature can influence another feature's behavior without completely suppressing it [65]. An example of this would be a Controller managing the behavior of an Actuator [17];
  - **Reads:** This kind of relationship happens when one feature reads data collected or produced by another feature, such as a sensor [65], without controlling or suppressing the feature's behavior;
  - **Suppresses:** This type of relationship happens when a feature suppresses the behavior of another one. Also, we consider as suppressed the relationship between features where a controlled feature (like Actuator or another Optional feature) needs to be uninstalled or unbound by its controlling feature (like Controller);

  - **Requires:** a relationship in which a feature is part of another feature's implementation [12]. In this relationship, there is no suppression or control over the feature's behavior that is part of the main feature,
- 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.

6.3 Behavioral Map Building Process

In the remaining, we describe the **BM** process shown on the right side of Figure 6.1.

6.3.1 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* provided by Data Extractor (see Section 7.1.2 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 use the feature name (or description) identified in Step 1 and adaptive mechanisms (see Table 5.1) 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 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 use the feature name or description and the scenario in which each feature is activated to identify optional features in our source code. We also analyze the comments used to describe class or method implementation to support our feature identification process. By associating the information collected in Step 1 with our source code information, we can locate each optional feature and select the main concrete class responsible for implementing it.

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 (more detail in Section 7.1.2).

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

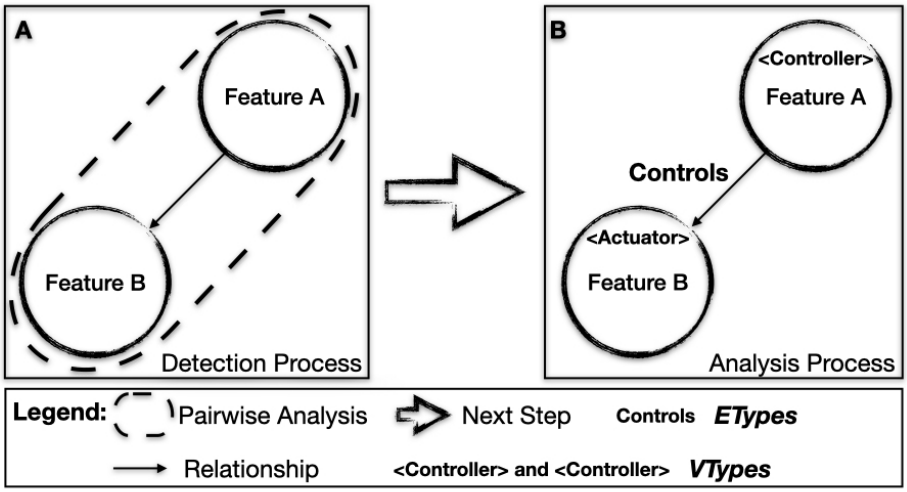

Figure 6.2: An overview of Detection (A) and Analysis (B) processes are used to identify and categorize the features on Behavioral Map.

6.3.2 Detection

Detection determines interacting features using pairwise analysis [119] 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, as depicted in Figure 6.2 - A. The process repeats for each feature until all interactions are detected and created on the map.

6.3.3 Analysis

During the analysis stage, we further refine the interactions identified during detection (in Figure 6.2 - A) in categories. For this, we analyzed the selected interaction (Feature A and Feature B depicted in Figure 6.2 - A) to identify the type of feature type ($VTypes$) implemented, according to the information available in the $\mathbb{CR}$ file. Thus,

let us consider Feature A as a Controller ($VTypes$) and Feature B as an Actuator ($VTypes$). Consequently, the relationship between Feature A and B, as shown in Figure 6.2 - B, is categorized as Controls ($ETypes$) because a Controller feature controls the behavior of features categorized as Actuator. For example, a Controller can read data provided by Sensor. The relationship between a Controller and a Sensor is categorized as Reads.

6.3.4 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 2: Behavioral Map algorithm.

Algorithm 2 captures the whole BM building process. The algorithm begins by loading the ℂℝ file as a `table` (line 1 at algorithm 2) 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.

6.4 Identifying Architectural Bad Smells

This Section describes the process performed to identify the architectural bad smells using the Behavioral Map. We recall the definition (on a high level) of ABS detected in the Behavioral Map for this. Additionally, we present the identification guidelines and discussion for each ABS.

Table 6.1: Selected Architectural Bad Smells for Self-Adaptive Systems.

Smell Name	Detection
Cyclic Dependency (CD) [7]	Full
Extraneous Connector (EC) [51]	Full
Hub-Like Dependency (HL) [7, 106]	Full
Oppressed Monitors (OM) [114]	Partial

While ABS catalogs exist in the literature [7, 51], their role in self-adaptive architectures is less known [106, 114]. Table 6.1 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[2] with a tutorial to configure the Neo4J platform, CR files, and the scripts used to create the map and analyze ABS.

6.4.1 Cyclic Dependency [7]

This smell occurs when two or more components depend on each other directly or indirectly [7]. Components involved in a dependency cycle can hardly be released, maintained, or reused in isolation [44].

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.

6.4.2 Extraneous Connector (EC) [51]:

This smell happens when two connectors of different types are used to link a pair of components [51]. This thesis 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 5.1) 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 [52]. However, a direct

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

connection may be justified for concurrent operation [51] and may increase the system's resiliency in case of failure of the publish-subscribe architecture.

6.4.3 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 [7, 106].

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 5.1, 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* [7].

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 [7, 44]. However, hubs form points of attention in case of failure.

6.4.4 Oppressed Monitors [114] (OM):

According to [114], 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 [114]. 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 the automated detection of smells using graph algorithms. Second, visual representations help designers and engineers to visualize runtime configurations.

6.4.5 Identification Process

The **BM** framework thus comes with dedicated algorithms to identify ABS [35], as described in Section 6.4. These algorithms are implemented via the Cypher[3] language, allowing to query the graph. We used provided queries to identify Cyclic

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

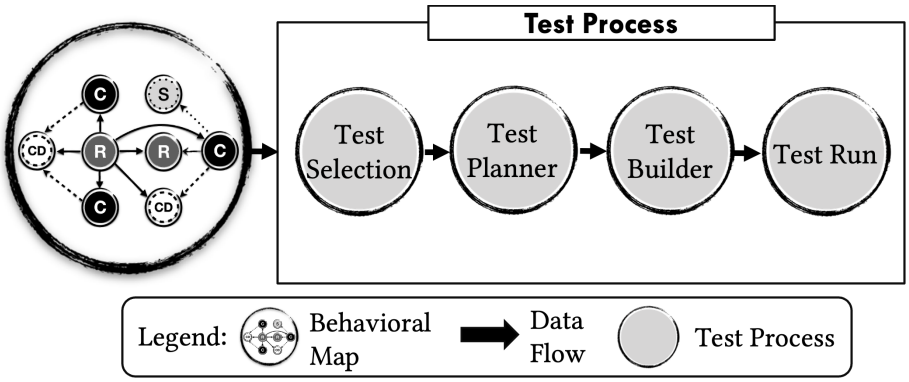

Figure 6.3: Behavioral Map (BM) test process overview.

Dependency (CD), Extraneous Connector (EC), Hub-Like Dependency (HL), and Oppressed Monitors (OM) on the map created for the SASs under study. For example, listing 6.1 shows how to compute cyclic dependencies on the map. The first line checks if feature F needs feature F2 to be bound during runtime and if F2 requires F at the same time. The second line checks if F2 needs F3 and F3 requires F. Finally, the last line returns features involved in CD. All queries used in this study are available on GitHub[4].

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

Listing 6.1: Cypher query used to look for Cyclic Dependency on the Behavioral Map.

6.5 Test Process

This section will explain how the Behavioral Map (BM) supports **the generate automated test cases and performing runtime testing for Self-Adaptive Systems (SAS)**. To illustrate the Test Process, we will adhere to the process outlined in Figure 6.3. The Test Process comes into play once the behavioral map processing has been completed and comprises four steps (Test Selection, Test Planner, Test Builder, and Test Run), which we will discuss in the subsequent sections.

6.5.1 Test Selection

The Test Selection process selects the **test suite scope** based on the behavioral map built at runtime for the System Under Test (SUT). Thus, the BM supports test selection at runtime based on the **Feature Relationship Analysis** or **Architectural Bad Smells Analysis**.

[4]https://github.com/edilton-santos/BehavioralMapExtendedStudy

Feature Relationship Analysis

The Behavioral Map (BM) will select the features based on the relationship types ($ETypes$) relevant to highlight runtime interaction problems, as defined in Section 6.3.3.

- **Feature interactions with shared resources:** The **BM** selects the first features that are controlled by two or more features that are classed as Controls simultaneously. Moreover, the features that are classed as Controls and are part of the shared resources identified will be included in the test selection. Consequently, the controlled features and the feature controls are included in the test selection. And then, the Behavioral Map executes the same process to identify the required features involved in shared resources relationships. Finally, select all features suppressed by two or more features at runtime. This strategy allows selecting all features involved in shared resources relationships detected at runtime. We adopted such a strategy because the features involved in shared resources relationships are more likely to trigger unexpected behavior at runtime, widely known as feature interaction problems.
- **Feature interactions without shared resources:** When working with feature interactions, it is crucial to consider their relationships. In this case, we identify the features that fall under the Controls relationship, followed by those involved in the Requires relationship. Next, we need to select the features that are part of the Suppresses relationship. Finally, we select all features that are classified under the Reads relationship.

Architectural Bad Smells Analysis

The Behavioral Map (BM) selects only the features involved in architectural bad smells identified on the map. For this, we use the processes outlined in Section 6.4.5.

6.5.2 Test Planner

The Test Planner identifies all classes and their dependent classes used to implement the selected features. For this purpose, the Test Planner obtains the name of the class responsible for implementing the selected feature and dependent features in the configuration rules ($\mathbb{CR}$) file. Also, the Test Planner retrieves complementary information based on each class load at runtime that has a relationship to the selected feature via Data Extractor (more detail in Section 7.1).

6.5.3 Test Builder

Test cases are expertly crafted by the Test Builder, utilizing the list of classes employed to implement the selected feature, as the Test Planner process identified. The Test Builder can generate tests using the following strategies:

- Adaptive-random test generation;
- Evolutionary algorithms to generate tests;
- Combinatorial Test Design (CTD) to generate test cases.

6.5.4 Test Run

The Test Run performs the tests based on the test suite scope built by Test Builder at runtime and generates the test reports.

6.6 Uncovered Aspects

This thesis aims to provide a framework to support the feature interaction detection, architectural bad smells identification and test suite scope selection at runtime for self-adaptive systems. This intricate process involves multiple aspects, and it is essential to note that certain areas will not be covered in this work.

6.6.1 Feature model definition

It is assumed that the developers have already defined and validated the feature model beforehand, as the Behavioral Map (BM) cannot support this task. However, the **BM** can assist in identifying the core and optional features in the source code (as explained in Section 7.2.1) or using the parametrization process to identify feature types (core and optional) loading at runtime, as described in Section 7.3.1.

6.6.2 Product derivation

The Behavioral Map does not support product derivation and only runs (supported only by Behavioral Map Black Box[5]) and monitors the system's adaptations under test based on the parameters defined previously by the users for the system execution.

6.6.3 Predict (Re)Configuration

The Behavioral Map cannot infer the SAS (re)configuration before the system defines the adaptation plan based on its context. Thus, the map shows precisely the configuration loaded at runtime.

6.7 Wrap up

This chapter introduced the Behavioral Map approach and explained its features. Also, we discussed the limitations of our approach in Section 6.6. We will provide a detailed explanation of the various components of the Behavioral Map framework implementation in Chapter 7.

[5] More information available at Section 7.3

Part III

Implementation

CHAPTER 7

BEHAVIORAL MAP FRAMEWORK

Behavioral Map (BM) is the framework we developed to support the feature interaction detection, architectural bad smells identification, and testing selection scope described in this thesis. This framework was published as part of a book chapter entitled **Behavioral Maps: Identifying Architectural Smells in Self-Adaptive Systems at Runtime** [77] in the book Software Architecture (Lecture Notes in Computer Science, vol 13365), specifically Sections 7.1, 7.1.1, and 7.1.2.

7.1 Framework Implementation

We conceived a framework to infer Behavioral Maps whose architecture is shown in Figure 7.1. The framework uses the Neo4J[1] platform and its Cypher[2] query language, as well as the Neo4j APOC (Awesome Procedures on Cypher) Library [93]. The topmost layers, **Map Builder**, **Analyzer**, and **Interaction Detector** perform the processes defined in Chapter 6. In the following, we focus on the remaining elements of the framework.

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

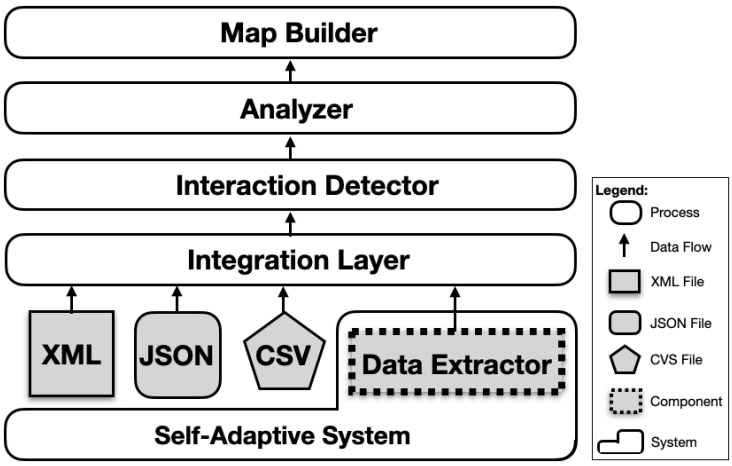

Figure 7.1: Behavioral Map Architecture overview.

7.1.1 Integration Layer

The **Integration Layer (IL)** serves as an interface between the Self-adaptive System 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:

- **name** is the feature name in the system;
- **friendly_name** is the friendly name of the feature shown to the user;
- **exported_packages** lists the exported packages or services offered via features;
- **imported_packages** lists the packages used by features to compose their functionality;
- **version** represents the feature version;
- **status** defines if the feature is active or inactive;
- **type** defines the feature type;
- **relationships** is a collection composed of relationship types and associated features described as follows:
  - **relationship_type** represents the relationship type, as defined in $ETypes$;
  - **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*.

7.1.2 Data Extractor

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 6.1), 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 7.1 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 7.1: Presence feature configuration rules.

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 map 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 6.3.1. 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 computing a graph depicting core and variable features as well as the different interactions between them (see the figures 5.1, 8.1, 8.2, 8.3, 8.4, and 8.5). Though these maps may be used for visual inspection, they mainly serve as support for further analyses thanks to the Neo4J graph database[3].

7.2 Behavioral Map White Box

The **Behavioral Map (BM) White Box** is implemented as reusable building blocks that allow their incorporation into the system under analysis. The **BM** White Box uses the Neo4J graph database (including Cypher queries) to build map visualization and WALA API [66] to perform the data extraction. Such API provides Call Graph [55, 56] and the Control-Flow Analysis (CFA) [55, 87, 88] algorithms used in the Data Extraction component implementation. The static analysis allows us to identify the dependency relationship among the class hierarchy used by selected features or perform interprocedural dataflow analysis and identify relationships' types. Also, in a complementary way, we can retrieve information using the manifest file used to install each feature at runtime. Such a file describes the feature and its feature dependency. Furthermore, the **BM** White Box offers **support to identify Architectural Bad Smells (ABS) at runtime**. The following sections will delve into the framework's various resources.

[3]https://neo4j.com/product/neo4j-graph-database/

7.2.1 Feature Trace

```
1  public class ECAFeedbackLoop {
2   ...
3   public static void main(String[] args) { // Main to run the simulator and feedback loop.
4    // Behavioral Map Data Extractor component.
5    FeatureTrace.activeOnlyOneJarFile("./mrubis-selfhealing.jar", "de/mdelab/simulator/mrubis/
         ↪ ");
6    FeatureTrace.setCoreFeatureTrace(ECAFeedbackLoop.class);
7    // load the model.
8    Architecture architecture = SelfHealingConfig.loadModel("./model/mRUBiS.comparch");
9    FeatureTrace.addRequiresFeatureInsideMethods(ECAFeedbackLoop.class, SelfHealingConfig.
         ↪ class);
10          ...
11   UtilityFunction utilityFunction= new MRubisSelfHealingUtilityFunction(exceptionThreshold);
12   FeatureTrace.addRequiresFeatureInsideMethods(ECAFeedbackLoop.class,
         ↪ MRubisSelfHealingUtilityFunction.class);
13   Scenario scenario = new SelfHealingScenario(architecture, exceptionThreshold);
14   FeatureTrace.addRequiresFeatureInsideMethods(ECAFeedbackLoop.class, SelfHealingScenario.
         ↪ class);
15      ...
16   ECAFeedbackLoop loop = new ECAFeedbackLoop(exceptionThreshold);
17   // run the simulation
18   do {
19    simulator.injectIssues();
20    loop.run(architecture, simulator.getChangeEventsQueues());
21    simulator.validateModel();
22         ...
23     // Behavioral Map.
24     InteractionDetector interactionDetector = new InteractionDetector(new Integration(new
           ↪ DataExtractor()));
25     interactionDetector.doAnalysis();
26     // Save the Configuration Rules in JSON format.
27     interactionDetector.saveResult("./BMResult/mRubis-BM/selfhealing/ECAFeedbackLoop");
28     // Remove the optional feature available in the feature trace collection.
29     FeatureTrace.removeOptionalFeatures();
30       ...
31   } while (!simulator.isSimulationCompleted());// repeat until the simulation is completed
32   simulator.showResults();
33  }
34
35    // Runs the feedback loop.
36    public void run(Architecture architecture, ChangeEventQueues queues) {
37   Queue<ComponentStateChangeEvent> componentStateChangeEvents = queues.
         ↪ getComponentStateChangeEvents();
38   if (!componentStateChangeEvents.isEmpty()) {
39    new ComponentStateChangeEventProcessor().process(componentStateChangeEvents);
40    // Behavioral Map Data Extractor component.
41    FeatureTrace.addRequiresFeatureInsideMethods(ECAFeedbackLoop.class,
          ↪ ComponentStateChangeEventProcessor.class);
42   }
43   ...
44  }
45 }
```

Listing 7.2: *Feature Trace code instrumentation implemented in mRUBiS [128].*

The Feature Trace is a Java static class that offers a set of methods used to feature identification according to the process described in Section 6.3.1. In the code snippet provided in Listing 7.2, we instrumented mRUBiS [128] (version Event-Condition-Action (ECA) feedback loop) with Feature Trace to detect loaded features based on the *Change Plan*. Specifically, we focused on class `ECAFeedbackLoop`, which runs the simulator and feedback loop. Line 5 adds the Jar file and main package to be

```
1  public class LinearTrafficDelayFunction implements TrafficDelayFunction {
2   private static final Logger logger = Logger.getLogger(LinearTrafficDelayFunction.class);
3
4   public int getDelay(int weight, int capacity, int number) {
5    logger.info("Delay computed.");
6
7    // Behavioral Map Data Extractor component.
8    FeatureTrace.addOptionalFeature(LinearTrafficDelayFunction.class, true);
9    return Math.max(weight, number - capacity + weight);
10  }
11 }
```

Listing 7.3: *Feature Trace code instrumentation implemented in Adasim [137].*

analyzed when mRUBiS is executed. In contrast, line 6 identifies `ECAFeedbackLoop` as a core feature because it implements the feedback loop for self-healing mRUBiS. This adaptation mechanism is implemented between lines 36 and 44. To identify the relationship between `ECAFeedbackLoop` and other loaded features, we analyze the objects loaded inside the methods provided by classes. For this, the Feature Trace provides the method `FeatureTrace.addRequiresFeatureInsideMethods` used to make the association between the class (*e.g.,* `ECAFeedbackLoop.class`) and dependency (*e.g.,* `SelfHealingConfig.class`) loaded inside its methods, the lines 9, 12, 14, and 41 show an example of this instrumentation. However, this information will only be included in the Feature Trace if the method was executed at runtime.

The method `addRequiresFeatureInsideMethods` provided by Feature Trace allows include features available inside methods in the analysis because the WALA API cannot identify objects inside executed methods at runtime. The other relationships are detected based on class properties retrieved by WALA API through the class hierarchy built. Additionally, Feature Trace offers a valuable method (*e.g.,* `FeatureTrace.removeOptionalFeatures()`) for erasing any optional features present in the trace collection prior to the commencement of a fresh adaptation cycle, as illustrated in line 29. Consequently, each map will include only the optional features loaded in a specific adaptation loop.

In addition to this, Listing 7.3 also showcases the implementation of the optional feature, the Linear Traffic Delay Function available in Adasim [137]. This feature is responsible for setting the delay at a node to be equal to the number of cars at the node minus a given maximum capacity. The Feature Trace includes this optional feature through line 8, utilizing the `addOptionalFeature` method. This method allows us to include the class responsible for implementing the optional feature and define whether the class is a controller, as defined in Section 6.3.3. In this case, it means that the Linear Traffic Delay Function controls the behavior of other features, so the second parameter of the method is `true`. With the help of the Feature Trace, we were able to gain a thorough understanding of the features present in the systems and their relationship with each other.

The Feature Trace supports detecting the type of object load at runtime because SASs architectures are implemented based on polymorphism through abstract classes or interfaces. Listing 7.4 shows the implementation of the method

```
1  private Vehicle buildVehicle(int i, ConfigurationOptions opts, AdasimMap g) throws
       ↪ ConfigurationException {
2    RoutingAlgorithm cs = randomVehicleStrategy( opts.getStrategies() );
3    cs.setMap(g);
4    List<RoadSegment> nodes = g.getRoadSegments();
5    RoadSegment start = randomNode( nodes );
6    RoadSegment end;
7    do {
8     end = randomNode(nodes);
9    } while ( start.equals(end) );
10
11   // Behavioral Map Data Extractor component.
12   FeatureTrace.addRequiresFeatureInsideMethods(SimulationBuilder.class, cs.getClass());
13   FeatureTrace.addRequiresFeatureInsideMethods(SimulationBuilder.class, Vehicle.class);
14   return new Vehicle( start, end, cs, i);
15 }
```

Listing 7.4: *Code snippet of class SimulationBuilder implemented in Adasim [137].*

`buildVehicle` used to instantiate an object `Vehicle`. Such a method associates the routing algorithm (`RoutingAlgorithm` in line 2) used to move the vehicle on the map (`AdasimMap`) and their position on the road (`RaodSegment` in lines 5 e 6). However, the `RoutingAlgorithm` is an interface used to specify the behavior of a class by providing an abstract type. Thus, line 2 loads at runtime any object that implements the `RoutingAlgorithm` interface. As a result, the architectural bad smell analyses at design time cannot detect the relationship between `SimulationBuilder` and classes that implement the `RoutingAlgorithm` interface because only concrete classes are included in the analyses. To solve this issue, the Feature Trace gets as a parameter the runtime class of the object instantiated, as shown in line 12. There are two possibilities to set the class type: i) All Java classes offer a static method called `class`, which can be used to return the runtime class of an object, as used in line 13, and ii) To discover the object type loaded via an interface or abstract class, we should use the method `getClass()` provided by the object loaded at runtime because this method returns the runtime class of object. For instance, in line 12, we used the `cs.getClass()` to obtain the concrete class loaded at runtime. Thus, we can identify the actual type of object loaded through an interface or abstract class and consequently analyze its dependencies. The strategy employed to set the class type was adopted because the Feature Trace should be generic and work with different types of system implementation.

7.2.2 Interaction Detector

The Interaction Detector (defined in Section 6.3.2) is a component that operates within the adaptation loop, as seen in lines 24, 25, and 27 in Listing 7.2. Its purpose is to inform the Behavioral Map about the end of the adaptation loop at runtime so that it can adequately process the configuration rules ($\mathbb{CR}$) file based on the data collected by Data Extractor (see line 24) from *Change Plan*. Thus, it processes the $\mathbb{CR}$ file before starting a new adaptation loop via the method `interactionDetector.doAnalysis()`, as illustrated in line 25. This method performed static analysis to identify class interaction using the CFA algorithm. The last process of Interaction

Detector is to save the $\mathbb{CR}$ in JSON format, as shown in line 27.

7.2.3 Map Building Implementation

```
1  CALL apoc.load.json("file:///crFileSHEstudy1.json") YIELD value
2  CALL apoc.create.node(['Feature',value.type], {name:value.name, friendly_name:value.
       ↪ friendly_name,
3  exported_packages:value.exported_packages, imported_packages:value.imported_packages,
       ↪ version:value.version,
4  status:value.status, type:value.type}) YIELD node
5  MERGE (f:Feature {name: node.name})
6  WITH value, node, f
7  UNWIND value.relationships AS relation
8  MERGE (fr:Feature {name: relation.feature_name})
9  WITH value, relation, f, fr
10 WHERE relation.relationship_type <> ""
11 CALL apoc.merge.relationship(f,relation.relationship_type,{},{},fr) YIELD rel
12 RETURN value, f, fr, rel
```

Listing 7.5: Cypher query used to import the $\mathbb{CR}$ file and build the Behavioral Map.

The Map Building defined in Section 6.3.4 is implemented using the Noe4j platform. To illustrate the creation of a behavioral map using the $\mathbb{CR}$ file in JSON format, we present Listing 7.5. The first line of the code loads the $\mathbb{CR}$ file into the Neo4J graph database, while the second line generates nodes on the map. Moving forward, line 7 establishes the relationships between the various features as described in Section 6.3.2, and line 11 generates relationships for each feature on the map based on the relations identified in line 7, as defined in Section 6.3.3.

7.2.4 Architectural Bad Smell Identification

Identifying Architectural Bad Smells (ABS) on the map is implemented via a set of Cypher queries executed on a behavioral map. Thus, the Behavioral Map framework implements a Cypher query for each ABS described in Section 6.4. All queries used to identify ABSs supported by the Behavioral Map framework are available on GitHub[4].

7.3 Behavioral Map Black Box

The **Behavioral Map Black Box** is implemented as an extension of Java Pathfinder (JPF) [5] that uses the jpf-nhandler[6] to automatically execute the System Under Test (SUT) methods from the host Java Virtual Machine (JVM). This allows the Behavioral Map Black Box to identify features loaded at runtime based on parameters defined by software developers without requiring code instrumentation and source code

[4] https://github.com/edilton-santos/BehavioralMapExtendedStudy

[5] Java Pathfinder is an extensible software model-checking framework for Java bytecode programs - Additional information is available at https://github.com/javapathfinder/jpf-core

[6] jpf-nhandler is an extension of Java PathFinder (JPF) - see more information at https://github.com/javapathfinder/jpf-nhandler

Listing 7.6: *Behavioral Map Black Box configuration engine.*

```
1  ## We don't check the standard libraries or frameworks used to build the SUT.
2  behavioralmap.excludeLibraries=org.netbeans.*,org.openide.*,com.ibm.*,org.apache.*,org.
       ↪ eclipse.*,org.json.*,org.osgi.*,org.jdom.*
3  ## Define the path used to save the CR file used to build the Behavioral Map.
4  behavioralmap.crFilePath =${jpf-behavioralmap}/configurationRules
5  ## By setting this to true, all CR files created in the configurationRules directory,
6  ## are removed once the search starts.
7  behavioralmap.cleanConfigurationRulesDir =false
8  ## Define the path used to save the result file after the analysis.
9  behavioralmap.resultFilePath =${jpf-behavioralmap}/result
10 ## By setting this to true, all result files created in the result directory, are
11 ## removed once the search starts.
12 behavioralmap.cleanResultDir =false
13 ## Define dependency level used to analyze the architectural smell Hub-Like Dependency.
14 behavioralmap.architecturalDependencyLevel =7
15 ## By setting this to true, save the SUT stack trace.
16 behavioralmap.saveSutStackTrace =true
17 ## By setting this to true define the SUT as a SAS and analyzes each adaptation.
18 behavioralmap.sutIsSelfAdaptiveSystem =true
19 # The root directory for java JDK 8 installation.
20 behavioralmap.javaJDKHome =/usr/lib/jvm/java-8-openjdk-arm64
```

recompilation of SUT, unlike the Behavioral Map White Box. Furthermore, the Behavioral Map Black Box as the Behavioral Map White Box **supports the Architectural Bad Smells (ABS) detection at runtime**.

The BM Black Box tool is also a powerful solution for **facilitating runtime testing**. It comes equipped with JUnit[7], which allows for the seamless execution of test units if the system under test (SUT) implements them. The tool also provides an interface with the TackleTest[8] [124], which automatically generates unit-level test cases. Furthermore, the Neo4J graph database is used to produce map visualizations, as seen in the Behavioral Map White Box.

7.3.1 Behavioral Map Configuration Process

The Behavioral Map Black Box tool is a solution that employs parametrization based on the JPF configuration file to specify the parameters necessary for creating the behavioral map and executing relevant analyses (*e.g.*, Architectural Bad Smells (ABS) detection and Test Selection). The configuration file offers support for a wide range of processes, including but not limited to:

Configuration Engine

The Behavioral Map Black Box configuration engine is responsible for the general setup of the tool. In Listing 7.6, we can see a section of the engine that pertains to the general configuration applied to all systems undergoing analysis. Line 2 shows which standard libraries or frameworks used to build the system under test (SUT) should not be included in the analysis. Lines 4 and 9 define the path to save the configuration rules ($\mathbb{CR}$) files and reports generated after analysis of each SUT

[7]JUnit - Additional information is available at https://junit.org/junit5/docs/5.0.0-M5/user-guide/
[8]TackleTest - https://github.com/konveyor/tackle-test-generator-cli

Listing 7.7: *Feature Trace configuration example used to run the Adasim [137].*

```
1  ## SUT what the Behavioral Map should run. Sets the Main class name and path.
2  target=adasim.TrafficMain
3  ## SUT app name.
4  behavioralmap.sutAppName =Adasim
5  ## The main package of the system under analysis.
6  behavioralmap.sutMainPackageFilter =adasim
7  ## List of the packages used to storage the core features of system under analysis.
8  behavioralmap.sutCoreFeaturePackages=adasim,adasim.agent,adasim.filter,adasim.generator
       ↪ ,adasim.model.*,adasim.util
9  ## List of the packages used to storage the optional features of system under analysis.
10 behavioralmap.sutOptionalFeaturePackages=adasim.algorithm.*
11 ## List of the classes or packages used to control other classes behavior at runtime.
12 behavioralmap.sutControllerFeatures=adasim.algorithm.*,adasim.agent.RoadClosureAgent
13 ## List of classes whose behavior is controlled by classes defined as controllers.
14 behavioralmap.sutControlledFeatures=adasim.model.RoadSegment,adasim.model.Vehicle
15 ## Define the method used in the adaptation loop.
16 behavioralmap.sutAdaptationLoopMethodName =adasim.model.TrafficSimulator.
       ↪ takeSimulationStep()
```

adaptation at runtime. The standard configuration saves the file in the Behavioral Map path (*e.g.,* `${jpf-behavioralmap}`). Lines 7 and 12 (on Listing 7.6) define if the configuration rules directory and the result directory will be cleaned before starting the Behavioral Map execution.

The configuration engine has a feature that allows us to set the dependency level for analyzing the architectural smell (Hub-Like Dependency), see line 14. By adjusting this configuration, we can improve the accuracy of the analysis. Additionally, the tool can save a stack trace report for each SAS adaptation made at runtime (line 16 in Listing 7.6). This feature allows us to track the entire history of adaptations from the known initial state to the last one executed. The history includes all the methods executed and the values of their respective parameters at runtime. This history can help face the issue of tracing the complete history of adaptations and their execution states mentioned in [117].

The Behavioral Map Black Box can analyze architectural bad smells and generate tests in self-adaptive or non-adaptive systems. This parametrization can be set via the parameter `behavioralmap.sutIsSelfAdaptiveSystem` presented in line 18 in Listing 7.6. By setting `behavioralmap.sutIsSelfAdaptiveSystem` to `true`, the user defines the system under test (SUT) as a self-adaptive system, and each adaptation will be analyzed at runtime. Otherwise, running the system analysis as a non-adaptive system and the adaptation loop not be verified. The parameter `behavioralmap.javaJDKHome` defines the root directory for Java Development Kit (JDK) 8 installation because the Java Runtime Environment (JRE) will not suffice to build the tests.

Feature Trace

To utilize the Feature Trace, we must configure the configuration file provided in Listing 7.7. This particular listing displays only a small section of the Behavioral Map configuration file, which is used to operate and evaluate the Adasim [137] system within the Behavioral Map framework. The first step is to define the class of SUT

that the Behavioral Map should run to start the SUT. For this, the Feature Trace provides the parameter `target` used to define the class with the method *main* used to run the SUT, see line 2. The parameter `behavioralmap.sutAppName` (in line 4) is used to define the name of the system under test. Additionally, the parameter `behavioralmap.sutMainPackageFilter` (line 6) defines the main package of the system that should be analyzed at runtime. Consequently, the parameter guides the architectural bad smell analyses and test generation.

Additionally, the Feature Trace allows identifying the core and optional features through the parameters `behavioralmap.sutCoreFeaturePackages` and `behavioralmap.sutOptionalFeaturePackages`, as illustrated in lines 8 and 10 of the Listing 7.7. Such parametrization allows the users to inform the set of packages or classes responsible for implementing the core and optional features. We employed this strategy to address the issue of lack of variability documentation in SAS discussed in Section 4.1.2. Similarly, we can define the controller features (line 12) and controlled features (line 14) using the same method to identify the core and optional features. The parametrization presented in lines 8, 10, 12, and 14 supports the Feature Identification process described in Section 6.3.1.

In the Feature Trace, there is a parameter called `behavioralmap.sutAdaptationLoopMethodName` (refer to line 16) which addresses the issue of capturing adaptations at runtime, as discussed in Section 4.1.3. According to line 16, the Behavioral Map should keep track of the `adasim.model.TrafficSimulator.takeSimulationStep()` method, as a new adaptation begins once this method finishes executing. Thus, The Behavioral Map can progressively identify the code responsible for the adaptation and trace methods entries and exits [77]. Consequently, this parameter supports the identification process of the *Change Plan* at runtime and the process of retrieving the Configuration Rules ($\mathbb{CR}$) used to generate the Behavioral Map. Also, more information about all configurations supported by Feature Trace can be found in Section A.1 of Appendix A.

Test Engine Configuration

The Behavioral Map test engine was conceived to tackle the challenge discussed in Sections 4.2 and 4.2.2. Thus, the test engine configuration provides the parametrization necessary to select the test scope, generate test cases and run testing at runtime based on feature interaction type and Architectural Bad Smells identified. Listing 7.8 present all parameter necessary to select and generate testing at runtime via the Behavioral Map. The parameter `behavioralmap.sutTestPrioritizationBasedOn` (line 2) allows setting the criteria (*e.g.,* Feature Relationship Analysis or Architectural Bad Smells Analysis) to select the classes that should be tested at runtime, as defined in Section 6.5.1. Also, there is a possibility to define if the tool will generate the test cases or use the tests case provided by the self-adaptive system under analysis to run the tests, see line 4.

The Behavioral Map utilizes the Tackle Test tool for generating test cases. To set the Tackle Test installation directory, use the parameter `behavioralmap.tackleTestGeneratorHome` (line 6). Additionally, the `behavioralmap.testBaseTest-`

Listing 7.8: *Test Engine Configuration of Behavioral Map Black Box.*

```
1  ## Test selection based on: interaction or smells
2  behavioralmap.sutTestPrioritizationBasedOn=interaction
3  ## By setting this to true, the Behavioral Map will generate the test classes.
4  behavioralmap.sutGenerateTestClasses=true
5  ## The root directory for tackle-test-generator-cli installation.
6  behavioralmap.tackleTestGeneratorHome =/tackle-test-generator-cli
7  ## The test generator: evosuite, randoop, and combined (evosuite and randoop).
8  behavioralmap.testBaseTestGenerator=combined
9  ## If you select the options evosuite or combined, you shoud select one or more
       ↪ Evosuite generation Criterion.
10 behavioralmap.testEvosuiteGenerationCriterion =LINE,BRANCH,EXCEPTION,WEAKMUTATION,
       ↪ OUTPUT,METHOD,METHODNOEXCEPTION,CBRANCH
11 ## Do not augment CTD-guided tests with coverage-increasing base tests.
12 behavioralmap.testNoAugmentCoverage =false
13 ## Do not generate CTD coverage report.
14 behavioralmap.testNoCtdCoverage =true
15 ## CTD interaction level (strength) for test-plan generation.
16 behavioralmap.testInteractionLevel =2
17 ## Number of executions to perform to determine pass/fail status.
18 behavioralmap.testNumSeqExecutions =2
19 ## Time limit per class (in seconds) for evosuite/randoop test generation.
20 behavioralmap.testTimeLimit =2
21 ## Maximal heap size (in MB) used for obtaining coverage data.
22 behavioralmap.testMaxMemoryForCoverage =4096
23 ## A boolean flag indicating that error-revealing tests should be generated.
24 behavioralmap.testBadPath =false;
25 ## Reuse existing base test cases. Default: false.
26 behavioralmap.testReuseBaseTests =false;
27 ## Generate code coverage report with JaCoCo agent.
28 behavioralmap.testCodeCoverage =true
29 ## When test suites are generated per module, create a combined coverage report.
30 behavioralmap.testCombineModulesCoverageReports =false
```

`Generator` parameter helps create JUnit test cases using different strategies such as `evosuite`[9], `randoop`[10], and `combined` (EvoSuite and Randoop). Randoop uses adaptive-random test generation, while EvoSuite applies evolutionary algorithms to create tests that meet code-coverage targets (such as method, statement, branch, and exception coverage). Thus, the code-coverage targets are defined using the parameter `behavioralmap.testEvosuiteGenerationCriterion` in line 10. Within line 8 of the code, an option called `combined` utilizes a technique known as Combinatorial Test Design (CTD) to generate test cases that examine various combinations of the declared method parameter types. The CTD modeling approach enhances the types of combinations examined and provides the ability to select different interaction levels as coverage goals [124]. Thus, this ultimately results in a more comprehensive testing process. Furthermore, the test generation can be configured on whether error-revealing tests are generated. For this, the parameter `behavioralmap.testBadPath` should be `true`, indicating that error-revealing tests should be generated.

Also, the tool has other parameters used to enhance the performance and code coverage. For instance, by setting the parameter `behavioralmap.testNoAugmentCoverage` (line 12) to `false`, the tool will augment CTD-guided tests with coverage-

[9] EvoSuite - Additional information is available at https://www.evosuite.org/
[10] Randoop - More information is available at https://randoop.github.io/randoop/

Feature Name	Package	Feature Type	Status
SimulationXMLReader	adasim.model.internal	Core	Activated
RoadVehicleQueue	adasim.model.internal	Core	Activated
Filters	adasim.model.internal	Core	Activated
SimulationXMLBuilder	adasim.model.internal	Core	Activated
SimpleErrorHandler	adasim.model.internal	Core	Activated
Version	adasim	Core	Activated
VehicleManager	adasim.model.internal	Core	Activated
TrafficMain	adasim	Core	Activated
TrafficSimulator	adasim.model	Core	Activated
IdentityFilter	adasim.filter	Core	Activated
ConfigurationOptions	adasim	Core	Activated
FilterMap	adasim.model.internal	Core	Activated
Vehicle	adasim.model	Core	Activated
PrivacyFilterMap	adasim.agent	Core	Activated
AdasimMap	adasim.model	Core	Activated
RoadSegment	adasim.model	Core	Activated
ReflectionUtils	adasim.util	Core	Activated
Total of Features Activated:	17		

Figure 7.2: Report of Features Activated at runtime.

increasing base tests. In a complementary way, we can reuse the existing base test cases. For this, the field `behavioralmap.testReuseBaseTest` (line 26) should be set as `true`. While the parameter `behavioralmap.testInteractionLevel` specifies the CTD interaction level for a test-plan generation. For example, by setting the interaction level for 2 results in pair-wise testing. Consequently, the test plan will include all possible combinations of subtypes for each pair of method parameters. The parameter `behavioralmap.testNumSeqExecutions` (line 18) defines the number of executions to determine the pass/fail status of generated sequences.

The tool also allows setting the time limit per class (in seconds) for test generation, see line 20. Thus, reducing the time used to generate the test cases to the selected testing scope. The last parameter used to enhance the performance is `behavioralmap.testMaxMemoryForCoverage` (line 22). This parameter sets the maximal heap size (in MB) used for obtaining coverage data. The Behavioral Map generates test reports as CTD coverage (line 14), code coverage (28), and combine modules coverage (line 30).

7.3.2 Behavioral Map Reports

The Behavioral Map provides a comprehensive collection of reports for every adaptation analyzed at runtime. These reports provide valuable insights into the system's quality detected at runtime. The reports are automatically saved in the "result" directory inside the Behavioral Map installation or any other directory specified in the configuration file, as shown in Listing 7.6.

Report of Features Activated

The report displayed in Figure 7.2 provides a list of the features that were loaded at runtime based on the *Change Plan* executed by the self-adaptive system. Thus, this report is created using the map developed during each adaptation's analysis. It presents a **comprehensive overview of the activated features**, describing the feature name, the package in which it is implemented, the type of feature, its activation status, and the total number of features activated throughout the adaptation process.

Report of Features Involved in Architectural Bad Smells

The Behavioral Map **generates a report for each type of architectural bad smell (ABS) detected at runtime**. These reports help to identify problematic areas that may require refactoring to improve the overall architectural quality of the system. An example report, illustrated in Figure 7.3, offers a comprehensive overview of the features involved in Cyclic-Dependency ABS during Adasim's [137] adaptation loop. Such a report comprises a list of classes (Features) and packages in Java format name, the feature type, and the total of features involved in ABS. These informations are crucial for deciding which classes should be refactored to improve the overall system's architectural quality. Also, the Behavioral Map allows us to use this information to select test scope and generate test cases at runtime, as discussed in Section 7.3.1.

Feature Name	Package	Feature Type
adasim.model.internal.VehicleManager	[adasim.model.internal]	Core
adasim.model.internal.RoadVehicleQueue	[adasim.model.internal]	Core
adasim.model.AdasimMap	[adasim.model]	Core
adasim.model.RoadSegment	[adasim.model]	Core
adasim.model.TrafficSimulator	[adasim.model]	Core
adasim.model.Vehicle	[adasim.model]	Core
adasim.algorithm.routing.QLearningRoutingAlgorithm	[adasim.algorithm.routing]	Controller
Total of Features involved in Architectural Smell(s):	7	

Figure 7.3: Report of Features Involved in Cyclic-Dependency architectural bad smell.

Invoked Methods Report

The Invoked Methods Report illustrated in Figure 7.4 **shows the methods most invoked at runtime**. This report is built based on the methods executed during the feature interactions at runtime. Thus, the report lists the most frequently executed methods, ordered from most to least. This information is helpful for developers because it highlights which methods require extra attention during system maintenance. If these methods have issues, the system may crash. The report can also assist with the decision of which unit tests should be implemented because developers can focus on the most executed methods at runtime.

Invocations	Method Name
6	adasim.model.RoadSegment#equals(Object)
6	adasim.model.Vehicle#isFinished()
1	adasim.model.internal.VehicleManager#isFinished()
1	adasim.model.TrafficSimulator#isFinished()
Total of invoked method(s):	4

Figure 7.4: Invoked Methods Report shows the last methods executed in the last adaptation performed by Adasim [137].

Stack Trace Report

```
0:     adasim.model.TrafficSimulator.isFinished()
0:      adasim.model.internal.VehicleManager.isFinished()
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@91b3)
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@e4e0)
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@10626)
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@b2f9)
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@e4e0)
0:      adasim.model.Vehicle.isFinished()
0:       adasim.model.RoadSegment.equals(adasim.model.RoadSegment@91b3)
```

Figure 7.5: Stack Trace Report shows the last method executed in the last adaptation performed by Adasim [137].

The Behavioral Map generates Stack Trace Report is used to **trace the complete history of adaptations starting at a known initial state**, including the parameter values used to execute each method at runtime. Thus, the report is built based on the order of the methods executed during the feature interactions detected at runtime by the Behavioral Map. Consequently, the reports can be used to support unit test implementation to deal with context-dependent control and data flow in SASs because it shows each executed method and their respective parameters used to answer a specific adaptation at runtime, as illustrated in Figure 7.5.

Test Code Coverage Report

The Behavioral Map generates the Test Code Coverage Report[11] based on the selected test scope process (Section 6.5.1) and test strategy (Section 6.5.3) for each adaptation that has been tested at runtime. This report indicates the percentage of code statements and branches that were covered during the execution of test cases, providing insight into how much of the code was actually executed during

[11]The Code Coverage Report was generated using JaCoCo. See more at https://www.jacoco.org/jacoco/trunk/index.html

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
simulator		37%		72%	14	40	69	125	11	29	1	4
domain		71%		63%	45	128	69	288	26	91	0	12
deltaiot		87%		43%	10	27	25	149	4	19	1	4
deltaiot.mapek		86%		77%	13	53	10	90	2	19	0	3
deltaiot.client		77%		91%	6	23	8	51	5	17	0	1
deltaiot.services		91%		83%	3	54	3	105	2	51	0	3
deltaiot.main		100%		n/a	0	5	0	16	0	5	0	1
Total	1,118 of 4,391	74%	59 of 198	70%	91	330	184	824	50	231	2	28

Figure 7.6: Test Code Coverage Report.

automated tests, as depicted in Figure 7.6. Also provides other important metrics listed in the following:

- **Branches Coverage** refers to the proportion of executed branches in code that includes if/else and switch statements. This metric considers the total number of branches in a method and determines whether they were executed or not. By examining the branch coverage, developers can gain insights into the effectiveness of their testing procedures and identify areas for improvement.
- **Cyclomatic Complexity** [129] is a metric used to evaluate the complexity of code based on the number of paths necessary to cover all potential paths in the code using a linear combination.
- **Methods** are considered executed once at least one instruction has been executed.
- **Classes** are considered executed as soon as their methods are executed.

Unit Test Report

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
deltaiot	5	0	0	0	100%	0.012
deltaiot.main	1	0	0	0	100%	0.101
deltaiot.mapek	6	0	0	0	100%	0.015
deltaiot.client	3	0	0	0	100%	0.023
deltaiot.services	24	0	0	0	100%	0.015

Figure 7.7: Unit Test Report shows an overview of the tests performed at runtime.

The Behavioral Map framework generates a comprehensive Unit Test Report based on the selected test scope (Section 6.5.1) for each adaptation tested at runtime. This report provides a concise **summary of all the test case results**. A visual depiction of the report can be found in Figure 7.7.

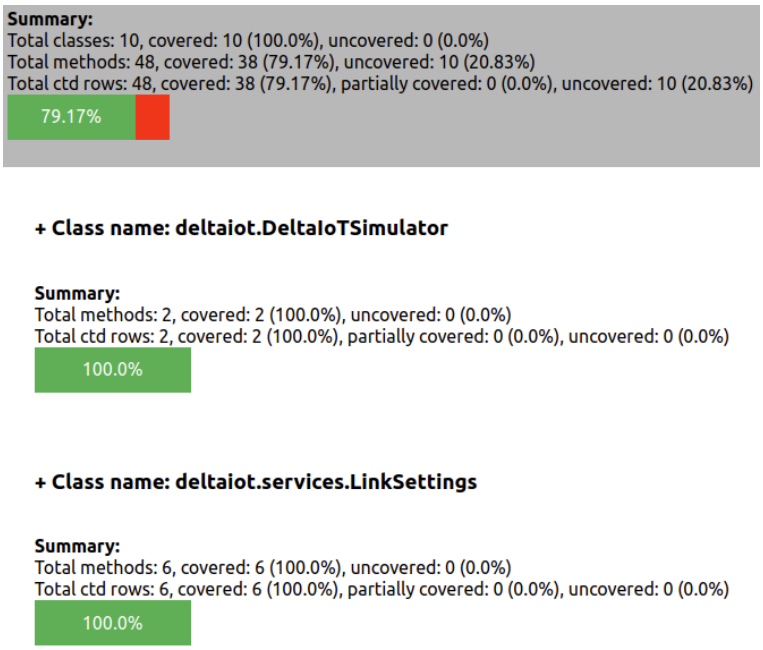

Figure 7.8: Combinatorial Test Design Report example.

Combinatorial Test Design Report

After Combinatorial Test Design (CTD) guided test generation is finished, the Behavioral Map framework provides a coverage report that **summarizes the achieved CTD test plan coverage**. The report indicates the number of rows in the CTD test plan for which the tool could generate unit tests. However, it is essential to note that the percentage of test plan coverage[12] reported here does not reflect the achieved code coverage. The code coverage can be verified in the Test Code Coverage Report 7.3.2. The CTD report is available in both JSON format (for visualization tools) and HTML format, which allows the user to drill down from class to method to CTD test plan row level, as shown in Figure 7.8.

7.4 Wrap up

In this chapter, we discussed the implementation of the Behavioral Map framework, explicitly highlighting the features of the Behavioral Map White Box and Behavioral Map Black Box. Additionally, we explored how the tool supports architectural smells identification and runtime testing to face the issue presented in Chapter 4.

[12] More information available at https://github.com/konveyor/tackle-test-generator-cli/blob/main/doc/unit/user_guide.md#output-artifacts

Part IV

Empirical Evaluations

CHAPTER 8

IDENTIFYING ARCHITECTURAL SMELLS IN SELF-ADAPTIVE SYSTEMS AT RUNTIME

We present in this chapter the case study published in the book Software Architecture (Lecture Notes in Computer Science, vol 13365), titled **Behavioral Maps: Identifying Architectural Smells in Self-Adaptive Systems at Runtime** [77] used to introduce the Behavioral Map (BM) formalism (presented in Section 6) and validate the **Behavioral Map White Box**, described in Section 7.2. Also, this case study is an extended version of the paper **A Vision to identify architectural smells in self-adaptive systems using behavioral maps** [35]. We analyze the Architectural Bad Smell in self-adaptive systems at runtime.

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 chapter provides the following contributions:

(i) A first study to identify **architectural bad smells** for SAS at runtime;
(ii) An analysis based on two runtime adaptations of Smart Home Environment (SHE), 40 runtime adaptations of Adasim, and 16 runtime adaptations of mRUBiS, demonstrate that **runtime variability affect the type and occurrence of smells found**;

Table 8.1: 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)		

(iii) A replication package containing the results and scripts to process behavioral maps is also available: https://github.com/edilton-santos/BehavioralMapExtendedStudy.

8.1 Behavioral Map - Based Architectural Bad Smells Detection

We search for architectural smells in various runtime adaptations within the Smart Home Environment (SHE) framework [111], Adasim [137], and mRUBiS [128] systems developed in Java. These smells include Cyclic Dependency (CD), Extraneous Connector (EC), Hub-Like Dependency (HL), and Oppressed Monitors (OM), defined in Chapter 2. To identify these ABSs, we use the Feature Identification process described in Section 6.3.1 and the process to identify the ABS using the Behavioral Map described in Section 6.4. Additionally, we instrumented the system source code following the process introduced in Section 7.2.1, which is based on the system's characteristics presented in Sections 5.1, 5.2, and 5.3.

8.2 Results

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

8.2.1 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 8.1 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 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 8.1.

Also, the BM identified *lampController* as involved in EC and OM smells in two adaptations. The EC smell occurred because the *lampController* uses the *listener*

Figure 8.1: Behavioral Map for SHE in adaptation 2.

(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 8.1. 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.

Table 8.2: 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)

8.2.2 Adasim Results

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

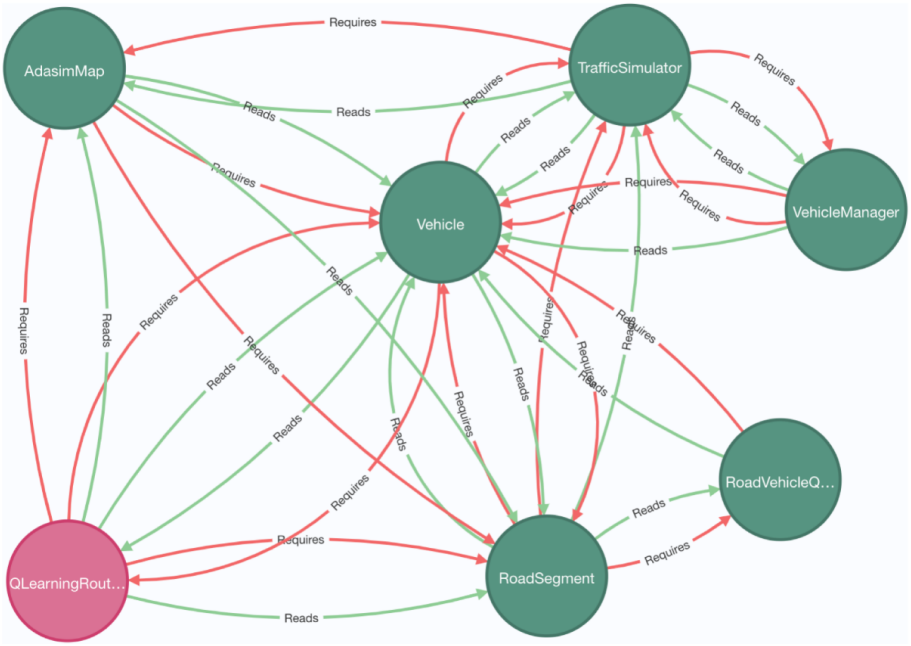

Figure 8.2: CD identified in Adasim QLearningRoutingAlgorithm in adaptation 1.

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 8.2 presents in detail the features involved in ABS during the two first adaptations. The *QLearningRoutingAlgorithm* is an optional feature involved in CD (only in the first adaptation) with the feature *RoadSegment*, and *Vehicle*, as shown in Figure 8.2. Such figures show all features involved in CD, the features in green 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 8.2.

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 8.2. 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 [44]. 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 8.3.

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 adaptations 1 and 2, as identified in the Adasim QLearningRoutingAlgorithm. Table 8.3 presents the ABS identified during adaptations 1 and 2. Additionally, it is possible to observe that the number of CD identified increases or decreases 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 features *AdaptiveRoutingAlgorithm*, *QLearningRoutingAlgorithm*, and *LookaheadShortestPathRoutingAlgorithm* 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.

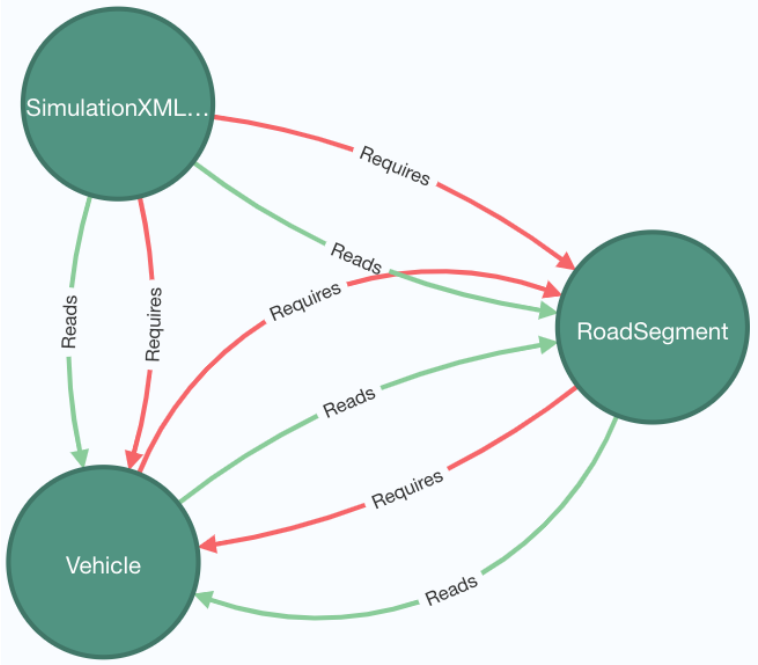

Figure 8.3: Features involved in HL identified in Adasim.

8.2.3 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 8.4 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. 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.

Table 8.3: 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)

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 [128] 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 8.5 through the relationship between *StateBasedMapeFeedbackLoop* (highlighted in red) and *SelfHealingConfig* (highlighted in blue). Also, Figure 8.5 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.

Figure 8.4: Behavioral map of the first configuration of mRUBiS Self-Optimization.

8.3 Threats to Validity

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

8.3.1 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 miti-

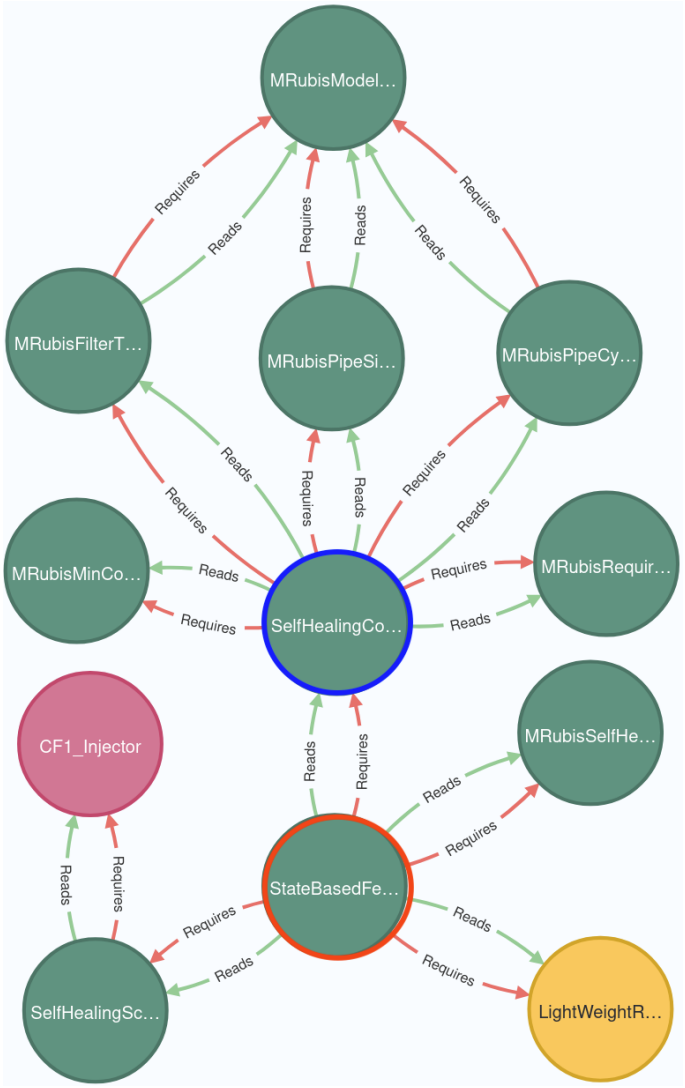

Figure 8.5: Behavioral map of the first configuration of mRUBiS Self-Healing MAPE-K loop.

gate this threat, we used the Eclipse IDE[1] 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 6.3.1. 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

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

code.

8.3.2 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 5.1. This 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.

8.4 Related Work

We found two works dedicated to the identification of ABS in self-adaptive systems. The first study [106] relies on the Arcan [44] 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 [36].

The second study [114] 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.

8.5 Wrap up and perspectives

In this chapter, we argued for the assessment of architectural bad smells (ABS) in self-adaptive systems (SAS) at runtime using the Behavioral Map (BM). We evaluated three SAS (SHE Framework, Adasim, and mRUBiS) and identified ABS during runtime on various system reconfigurations. Our results showed that certain ABS only appear in specific system configurations or architectures, such as the EC and OM smell in the publish-subscribe architecture used in SHE Framework. We also observed that the type and quantity of ABS found in SAS depend on the configuration analyzed at runtime. For example, in Adasim's AdaptiveRoutingAlgorithm, the Behavioral Map detected nine CD and three HL smells in the first adaptation. Also, only six CD smells in the second can be explained by binding and unbinding

certain runtime features. Therefore, the Behavioral Map framework is a valuable tool for evaluating the architectural qualities of a given runtime adaptation. However, identifying ABS during runtime requires expertise and time because the core and variable features are not documented.

We envision three future works:

(i) We intend to conduct an empirical study to compare the differences between smells detected at design time and those occurring during runtime in self-adaptive systems;

(ii) We plan to create a dedicated ABS tool that operates at the bytecode level to reduce the engineering costs associated with analyzing SAS at runtime;

(iii) We aim to expand our findings by evaluating more self-adaptive systems.

CHAPTER 9

TOWARDS ASSESSING ARCHITECTURAL SMELLS FOR SELF-ADAPTIVE SYSTEMS AT RUNTIME

Nowadays, more and more Self-Adaptive Systems (SAS) are required to run without interruptions in heterogeneous environments. SAS vary their behavior through the (de)activation of *features* depending on environmental changes and reconfiguration plans and goals to answer such requirements. Such a reconfiguration combines architectural fragments or different solutions at runtime that may negatively impact their architectural qualities. Thus, Architectural Bad Smells (ABS) may emerge, implying reductions in system maintainability [29, 79]. ABS results from architectural design decisions that negatively impact system lifecycle properties (*e.g.,* testability and maintainability) [52].

Variability management is key for SAS as well as for Software Product Lines (SPL), where one derives a family of variants based on the *core* (present in all variants) and *optional* (present in some variants) features [27]. Many studies target ABS in single systems or [29, 42, 51, 52, 79, 90]. However, there are less studies focusing on SAS or Dynamic Software Product Line (DSPL) [35, 106, 114]. In particular, these **studies do not discuss the impact of runtime variability on smell detection and evolution** as the SAS adapts. To assess the runtime architectural qualities of SAS, we developed a

framework that instruments SAS to monitor runtime adaptations and captures them in *behavioral maps* (**BM**) [35].

This chapter presents a preliminary comparison between design time and runtime smell detection for SAS published and presented at the *International Conference on Software Architecture (ICSA)*[1] [36]. To perform this comparison, we selected two ABS detection tools, Arcan [43] and our **BM** framework [35]. We motivate the choice of the former tool because it was applied on SAS at design time [106]. **The Behavioral Map framework is the only approach to focus on runtime ABS**. Both tools focus on SAS written in Java. We selected Adasim [137] and mRUBiS [128] for their public availability and the diversity of adaptation mechanisms these SAS use. The characteristics of those systems were introduced in Sections 5.2 and 5.3. As for smells, we considered Cyclic Dependency (CD) and Hub-Like Dependency (HL) [7] as they are supported by both Arcan and **BM** framework tools. Furthermore, additional information about the ABSs selected can be found in Sections 2.2.5 and 2.2.6.

Our results show important differences between smells occurrences at design time and runtime for Adasim, and smells appearing at runtime not found at design time for mRUBiS. ABS occurrences also vary along SAS reconfigurations. Our results suggest that runtime ABS assessment is required to fully grasp SAS architectural qualities. In summary, this chapter provides the following contributions:

(i) **A first empirical comparison of architectural bad smells for SAS detected at design time and at runtime**;

(ii) Our analysis based on 40 runtime adaptations of Adasim and 16 runtime adaptations of mRUBiS, demonstrates that runtime variability affects the type and occurrence of smells found. The results and scripts to process behavioral maps are also available here: https://doi.org/10.5281/zenodo.5814028.

9.1 Study Design

This empirical study aims at investigating differences between smells one detects at design time and smells occurring at runtime. To understand these differences, we formulate the following research questions.

9.1.1 Research Questions

RQ1. Are smells found at design time also found at runtime?

This involves: *i)* running different configurations of self-adaptive systems, *ii)* measure the type and number of occurrence of each smell, and *iii)* compare these results with smells detected at design time.

[1] ICSA 2022 - https://icsa-conferences.org/2022/

RQ2. How does the number of architectural bad smells evolves during the reconfiguration process at runtime?

We aim at qualifying the variations of smell occurrences at runtime, and this is related to the SAS variability.

9.1.2 Architectural Bad Smells Analysis Tools

We selected Arcan [43] and the Behavioral Map (BM) [35, 77] framework to perform ABS detection in self-adaptive systems. Both tools were employed in the past for ABS detection [35, 106]. The former analyses the source code while the latter computes a *behavioral map, i.e.,* a graph representing different relationships (requires, reads, controls) between features at runtime described in Chapter 6. Both tools target software written in Java.

9.1.3 Experimental Setup

We ran Arcan on the Jar files available on GitHub for Adasim[2] and mRUBiS[3]. We instantiated the Behavioral Map (BM) framework on both systems, a result of a two weeks work from the author. Below, we will provide instructions on how to use each of the selected tools.

The ABS Identification in Arcan

We used the Arcan with the Neo4J[4] command and specified the database destination folder. We present below the commands used to identify CD and HL in the system under study. Thus, we executed the tool command as described in the following.

Command to look for CD:
```
java -jar Arcan-1.2.1-SNAPSHOT.jar
-p my_projec_location -out my_output-results_location
-d my_database_destination_location -neo4j -cd
```

Command to look for HL:
```
java -jar Arcan-1.2.1-SNAPSHOT.jar
-p my_projec_location -out my_output-results_location
-d my_database_destination_location -neo4j -hl
```

The ABS Identification in Behavioral Map

In order to accurately identify the ABS in Adasim and mRUBiS systems, we conducted a thorough analysis of the available source code and Jars files on GitHub. We downloaded the source code of each system and reconstructed their development and execution environments based on the information provided by the developers. Furthermore, we expertly instrumented the source code to keep a precise trace of

[2] https://github.com/brunyuriy/adasim
[3] https://github.com/thomas-vogel/mRUBiS
[4] Neo4j - https://neo4j.com/product/

features during runtime. The process for identifying these features is outlined in detail below.

Feature Identification Process: To ensure accurate tracking and integration of features into the Feature Trace (as explained in Section 7.2.1), we adhere to the established process outlined in Section 6.3.1. In mRUBiS, the code instrumentation for the Feature Trace is available in Listing 7.2, while Listing 7.3 showcases the code instrumentation employed in Adasim. Through these code instrumentation methods, we can effectively identify the features loaded during runtime and add them to the Feature Trace collection. This collection is then used to generate the configuration rules ($\mathbb{CR}$) file based on the *Change Plan* detected in the adaptation loop, as demonstrated in lines 24 and 25 of Listing 7.2.

Running the Behavioral Map tool: In order to effectively utilize the Behavioral Map tool, a series of meticulous steps were taken for each system. Within the Adasim system, two parameter files, namely `config.xml` and `config-AdaptiveRoutingAlgorithm.xml`, were employed to determine the routing and traffic delay function algorithm utilized for adaptations during runtime. The Behavioral Map framework created 13 maps for the first file and 27 maps for the second, ultimately resulting in 40 carefully analyzed adaptations at runtime. Similarly, within the mRUBiS system, two parameter files, namely `mRUBiS_performance.comparch` and `mRUBiS.comparch`, were utilized to express the adaptable software's architectural runtime model. The `mRUBiS_performance.comparch` file was utilized to run the self-optimization version with the MAPE-K loop adaptation mechanism, while the `mRUBiS.comparch` file was used to run the self-healing version with adaptation mechanisms based on the Event-Condition-Action (ECA) approach, MAPE-K loop, and State-Based Feedback Loop. Through the Behavioral Map tool, a total of four maps were created for each version of mRUBiS executed, ultimately resulting in 16 adaptations that were thoroughly analyzed overall. These mRUBiS variants were identified during the feature identification process. They were included in the Architectural Bad Smells (ABS) analysis process for comprehensive analysis. Additionally, we employed the strategy outlined in Section 6.4.5 to find the ABSs for each system we examined.

9.2 Results

The following sections describe the results and discuss the reasons behind the differences between each architectural bad smell identified at runtime or design time in each selected self-adaptive system.

9.2.1 Adasim Results

Adasim is a parameter-based routing algorithm adaptive mechanism, and we identified two adaptation modes according to the algorithms initiating the reconfiguration process: QLearningRoutingAlgorithm and AdaptiveRoutingAlgorithm.

Table 9.1: ABS identified by Arcan and Behavioral Map.

Feature Name	Arcan		Behavioral Map		
	CD	HL	CD	HL	Feature Type
TrafficSimulator	Yes		Yes		Core
RoadSegment	Yes	Yes	Yes	Yes	Core
Vehicle	Yes		Yes	Yes	Core
VehicleManager	Yes		Yes		Core
RoadVehicleQueue	Yes		Yes		Core
AdasimMap			Yes		Core
QLearningRoutingAlgorithm			Yes		Optional
SimulationXMLBuilder				Yes	Core

Table 9.2: ABS identified by the BM in adaptation 1 and 2 of the Adasim - QLearningRoutingAlgorithm.

Feature Name	Feature Type	Adaptation 1		Adaptation 2	
		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: Table 9.1 presents the ABS identified by Arcan and **BM**. For the last tool, we show only the smells identified in the first adaption loop. Additionally, the table shows the feature type affected for each ABS. The Arcan analysis identified ABS only in the core features. Thus, the TrafficSimulator, RoadSegment, Vehicle, VehicleManager, and RoadVehicleQueue are Cyclic Dependency (CD), and RoadSegment is Hub-Like Dependency (HL).

BM found the same features identified by Arcan involved in ABS and identified three more additional features, as we depicted in Table 9.1. Thus, the core feature AdasimMap and the optional feature QLearningRoutingAlgorithm were identified as CD. Also, the core feature SimulationXMLBuilder was identified as HL. These ABS were identified in the first adaptation loop executed by Adasim.

We analyzed Adasim during 13 self-adaptations and found that the number of detected ABS only changed between the first and second adaptations. Further adaptations did not affect the architecture further. Table 9.2 details the features involved

in ABS during the two first adaptations. The QLearningRoutingAlgorithm is an optional feature involved in CD only in adaptation 1 with the feature RoadSegment and Vehicle. Nevertheless, the absence of the QLearningRoutingAlgorithm (in adaptation 2) reduces the numbers of dependency in the features RoadsSegment and Vehicle involved in HL, see Table 9.2. This situation occurred because RoadSegment and Vehicle are not sharing QLearningRoutingAlgorithm in the second adaptation.

Adasim AdaptiveRoutingAlgorithm. In this mode, we monitored the system during 27 self-adaptations, involving 20 features. Similarly, we observed differences between the two first adaptations. Table 9.3 presents the ABS identified during adaptations one and two. We observe that the number of CDs identified increases or decreases 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 dependency among them at runtime. We discovered that the Vehicle feature, identified as HL in Adaptation 1, was not identified in Adaptation 2 because the optional features AdaptiveRoutingAlgorithm, QLearningRoutingAlgorithm, and LookaheadShortestPathRoutingAlgorithm were not loaded. As a result, the **BM** identified the RoadSegment feature as a new HL in adaptation 2, as Arcan also identified.

Table 9.3: ABS identified by the BM 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)

Table 9.4: Architectural Bad Smells identified by Arcan and Behavioral Map in mRUBiS Self-Optimization.

	Arcan		Behavioral Map		
Feature Name	**CD**	**HL**	**CD**	**HL**	**Feature Type**
SimulatorUtil		Yes			
ModelParameterPage	Yes				Optional
ModelParameterPage$1	Yes				Optional
SelfOptimizationConfig				Yes	Core
MRubisModelQuery				Yes	Core
EventBasedMapeFeedbackLoop				Yes	Core

Our investigation into Adasim has uncovered discrepancies in identifying architectural bad smells (ABS) at runtime versus design time. Upon analyzing the data, we found that runtime detection could identify a higher number of ABSs, including those present in the core features in all adaptations (RQ1 - see Section 9.1.1). An example is the Adasim AdaptiveRoutingAlgorithm, which had 5 Cyclic Dependency (CD) and 1 Hub-Like Dependency (HL) ABS detected by Arcan, and 9 CD and 2 HL ABS detected by the Behavioral Map in the first adaptation. We also discovered that the number of ABS occurrences was affected by the activation or deactivation of optional features (RQ2 - see Section 9.1.1). For instance, in Adasim AdaptiveRoutingAlgorithm, the **BM** detected 9 CD and 3 HL ABS in the first adaptation but only 6 CD ABS in the second. These findings shed light on the importance of considering runtime and design time detection when analyzing ABSs in software architecture.

9.2.2 mRUBiS Results

The mRUBiS system is divided into self-healing and self-optimization versions.

mRUBiS self-optimization: Table 9.4 details the Architectural Bad Smells (ABS) identified by Arcan and the Behavioral Map (BM) framework. Arcan analysis identified the `SimulatorUtil` as Hub-Like Dependency (HL) and the classes `ModelParameterPage` and `ModelParameterPage$1` as Cyclic Dependency (CD) in the mRUBiS self-optimization version. The `SimulatorUtil` is a class part of the framework used to implement the mRUBiS simulator, but the Arcan tool identified the class as a mRUBiS implementation. Also, the classes `ModelParameterPage` and `ModelParameterPage$1` are optional features responsible for implementing the graphical interface.

However, 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 feature a decrease in the number of dependencies used in the second adaptation. This situation occurred because the feature is responsible for adding the validators and other parameters related to self-optimization. However,

Table 9.5: Architectural Bad Smells identified by Arcan and Behavioral Map in mRUBiS Self-Healing MAPE-K loop.

	Arcan		Behavioral Map		
Feature Name	**CD**	**HL**	**CD**	**HL**	**Feature Type**
SimulatorUtil		Yes			
ModelParameterPage	Yes				Optional
ModelParameterPage$1	Yes				Optional
StateBasedMapeFeedbackLoop				Yes	Core

the number of validators used at runtime decreases, impacting the dependencies identified. 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 adaptation loop.

mRUBiS self-healing: The Arcan identified the same ABS identified in the mRUBiS self-optimization version because the classes the `SimulatorUtil`, `ModelParameterPage`, and `ModelParameterPage$1` are used in the mRUBiS build independent of the version. However, the **BM** does not identify ABS in the self-healing version with the adaptation mechanism Event-Condition-Action (ECA) and State-Based Feedback Loop (SBFL) after four reconfiguration processes at runtime. The difference between the results presented by Arcan and **BM** is triggered by the way how ABSs are identified. Table 9.5 presents in detail the ABS identified by Arcan and **BM** for the self-healing version with adaptation mechanism MAPE-K. The **BM** identified one instance of HL in the core feature StateBasedMapeFeedbackLoop in four adaptation loops. The feature is the main entry point to other features such as Monitor, Action, Plan, Execute, SelfHealingConfig, SelfHealingScenario, and MRubisSelfHealingUtilityFunction.

After conducting a thorough analysis of four different system versions, it has been determined that mRUBiS at runtime has a superior architectural quality. This is due to the fact that we were only able to identify the Architectural Bad Smells (ABS) in the self-healing version that utilized the MAPE-K adaptation mechanism. Specifically, we discovered one Hub-Like Dependency (HL) in the StateBasedMapeFeedbackLoop feature that remained consistently present throughout all four performed self-adaptations (RQ2 - see Section 9.1.1). Furthermore, ABSs identified at design time are not always present at runtime.

9.2.3 Synthesis

Answering RQ1. We observed significant differences between design time and runtime smell detection for both systems. Some smells are only present at runtime, and conversely, some smells only appear at design time. By answering no to RQ1, we motivate the need for further assessment of runtime smells for SAS.

Answering RQ2. We also observed a variation in the occurrences of smells found between the adaptations. For instance, in Adasim AdaptiveRoutingAlgorithm, the **BM** found 9 CD and 3 HL in the first adaptation. However, in the second, the **BM** found 6 CD smells. We could explain this variation by activating and deactivating certain runtime features.

9.2.4 Lessons Learned

The feature identification process was quite complicated because we did not find a feature model or feature annotation in the source code. Consequently, we ran each system using the Eclipse IDE in debug mode to analyze its execution flow to understand how the self-adaptation process works. In addition, we analyze how the features previously identified in the documentation were implemented. After this step, we include the Behavioral Map Feature Trace in the code of all the features identified in the previous phase. However, during this process, we encountered non-explicit features in the documentation. An example of this situation is Adasim's SimulationXMLBuilder and RoadVehicleQueue features. Thus, we worked for two weeks on the process of identifying the features of each system.

A self-adaptive system changes its behavior depending on environmental changes and reconfiguration plans. Thus, these systems have architectures based on interface and abstract classes. This type of architecture impacts the ABS identification because the system's core features use polymorphism (via interfaces or abstract class) to perform the (re)configurations. Such an aspect can interfere with the identification of ABS, such as CD and HL, which consider a concrete class in their identification process. The **BM** framework identifies the actual feature activated at runtime based on the interface or abstract class used to execute the feature. It then establishes the relationship between the features identified at runtime and the core feature. We used the process described to identify (at runtime) the relationships of Adasim's QLearningRoutingAlgorithm features (see Figure 8.2). The core features AdasimMap, Vehicle, and RoadSegment receive an interface called RoutingAlgorithm as a parameter. Thus, the **BM** framework identified the real interactions between each feature at runtime, identifying the real dependencies between features at runtime.

9.3 Threats to Validity

In this empirical study, some general threats to validity have to be considered, threatening the results' internal validity or generalization.

9.3.1 Internal Validity

The lack of a feature model and feature annotations in the source code may hamper the variability identification process. To mitigate this threat, we used the Eclipse IDE[5] tool to verify the feature implementation and to debug the systems' source code to check the execution of each feature identified using the process described

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

in Section 6.3.1. We also analyzed execution logs to ensure that our identification of features was correct. Thus, we verified whether each core or optional feature was identified precisely in the source code. Finding tools to perform Architectural Bad Smells (ABS) detection in self-adaptive systems at design time and runtime is difficult. Therefore, our main issue for conducting experiments is to evaluate if the ABS found at design time is also found at runtime. Thus, we used the Arcan [43] and the Behavioral Map (BM) framework [35, 77] to address this threat because both tools were employed in the past for ABS.

9.3.2 External Validity

Our findings may not be applicable to all SAS, as we only analyzed two specific systems in our research. Also, testing all possible system adaptations and estimating their number is impossible. However, the systems we chose have diverse architectural models, adaptation mechanisms, and application domains, which helps to reduce this limitation. Our aim was not to gather statistical evidence on the differences between runtime and design time architectural bad smells but rather to uncover and explain their existence. We will leave a more quantitative evaluation for future research.

9.4 Related Work

We have come across three research studies that focus on identifying Architectural Bad Smells (ABS) in self-adaptive systems (SAS). The first study [114] presents two new ABS specific to SAS. It also outlines the guidelines for identifying each smell during the design phase. The second study [106] identified ABS in 11 self-adaptive systems at design time. However, these studies do not address how the ABS identification process can be affected by the runtime (re)configuration. Since the system will adapt itself to respond to changes in its environment during runtime, this adaptation can cause the ABS identified during design time to not be found during runtime, or vice versa.

The last study [35] relies on the Behavioral Map (BM) to automatically infer key architectural characteristics from different sources (*e.g.,* feature model, source code, adaptation rules) and represent them in a graph. The graph allows the execution of algorithms to detect the ABS based on the architecture identified at runtime. Also, the author illustrates their applicability to Smart Home Environment (SHE) [111] example. Our study aims to understand how ABS occurs at runtime for different feature combinations and if the ABS identified at design time occurs in all SAS configurations at runtime.

9.5 Wrap up and perspectives

In this chapter, we made a case for assessing architectural bad smells (ABS) for self-adaptive systems (SAS) at runtime, hypothesizing differences between design time and runtime analyses. We selected two SAS (Adasim and mRUBiS) and two ABS

detection tools (Arcan and the Behavioral Map framework) to reveal and explain these differences. We performed design time and runtime smell detection on several systems reconfigurations. Our results showed that there are indeed differences between design time and runtime detection. While we could have assumed that all smells found at runtime would be a subset found at design time, we also found occurrences of smells only found at runtime. We identified the root causes for this seemingly surprising finding, including polymorphism affecting the design time analysis's precision. These differences interest SAS architects in order to more precisely put their maintenance efforts and assess the architectural qualities of a given runtime adaptation. However, instrumenting such SAS for runtime ABS identification requires expertise and time, especially since core and variable features are not documented. This is the main lesson learned from our study besides our results.

There is room for future work. First, we would like to reduce the cost of engineering involved in analyzing SAS at runtime, which is a current impediment to large-scale analyses. In particular, we will design a dedicated ABS tool operating at the bytecode level, easing runtime analyses. Second, we will generalize our findings by assessing more SAS.

CHAPTER 10

BEHAVIORAL MAP: TOWARDS RUNTIME TESTING FOR SELF-ADAPTIVE SYSTEMS

This Chapter introduces a preliminary study that utilizes the Behavioral Map Black Box to select the test scope at runtime based on feature relationship analysis and architectural bad smells analysis approach proposed in this thesis in Section 6.5.1. We show how to generate and execute tests at runtime in two self-adaptive systems.

10.1 Study Design

In this empirical study, we employ the Behavioral Map Black Box to deal with the runtime variability's impact on self-adaptive systems tests at runtime (discussed in Section 4.2.1), and we generate test cases for the changing environment [117, 118]. To face those problems, we state the following research questions in Section 10.1.1.

10.1.1 Research Questions

RQ1. How to define the test suite scope that should be tested at runtime for SASs? One of the main challenges is determining **the test suite scope to test** on a Self-Adaptive Systems (SAS) due to a large number of possible (re)configurations.

RQ2. Which approach to test scope selection at runtime achieves the best code coverage? This question aims to find the most effective approach to choose the test scope that covers the most code while requiring the least time to complete at runtime. **We evaluated Feature Relationship Analysis and Architectural Smells Analysis as test selection strategies** to answer this question via our Behavioral Map Black Box.

10.1.2 Self-adaptive systems under study

We chose Adasim [137] and DeltaIoT [67] as they have Java 8 components implementation and are compatible with Linux operating systems for ARM[1] 64-bit (ARM64) processors. We indeed developed and experimented the Behavioral Map framework on such a platform. As for mRuBiS, we did not select them due to their Java 8 components being incompatible with Linux ARM64.

10.2 Experimental Setup

This Section explains the procedures we used to generate and run tests at runtime using the Behavioral Map Black Box to test the Adasim [137] and DeltaIoT [67].

10.2.1 Feature identification process in the Behavioral Map Black Box

We follow the process outlined in Section 6.3.1 to identify the features in Adasim and DeltaIoT. Thus, we parameterized the Feature Trace engine using the parameters described in the Listing 10.1. Based on this listing, we will describe the feature identification process in the Behavioral Map Black Box.

Step 1 - Identify the features: To begin, we examine papers and information available on the Software Engineering for Self-Adaptive Systems portal to identify the features present in Adasim[2] and DeltaIoT[3]. These papers provide details on the systems, including their adaptive mechanisms, applicability, test scenarios, source code, and system version.

Step 2 - Identify the core features in the source code: To identify the core features in the source code, we used the feature name or description determined in Step 1, along with the adaptive mechanisms presented in Table 5.1. These core

[1] ARM stands for Advanced RISC Machine. For more information, please visit: https://www.redhat.com/en/topics/linux/what-is-arm-processor#overview

[2] Adasim artifacts available at https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-atrp/

[3] DeltaIoT artifacts available at https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/deltaiot/

Listing 10.1: *Feature identification in Behavioral Map Black Box.*

```
1  ## SUT app name.
2  behavioralmap.sutAppName =adasim
3  ## The main package of the system under analysis.
4  behavioralmap.sutMainPackageFilter=adasim
5  ## List of the classes or packages used to storage the core features of system under
       ↪ analysis.
6  behavioralmap.sutCoreFeaturePackages=adasim,adasim.agent,adasim.filter,adasim.generator
       ↪ ,adasim.model.*,adasim.util
7  ## List of the classes or packages used to storage the optional features of system
       ↪ under analysis.
8  behavioralmap.sutOptionalFeaturePackages=adasim.algorithm.*
9  ## List of the classes or packages used to control other classes behavior at runtime.
10 ## The Behavioral Map will use it to verify if the behavior defined at design time
       ↪ happen at runtime.
11 behavioralmap.sutControllerFeatures=adasim.algorithm.*,adasim.agent.RoadClosureAgent
12 ## List of classes whose behavior is controlled by classes defined as controllers.
13 behavioralmap.sutControlledFeatures=adasim.model.RoadSegment,adasim.model.Vehicle
14 ## SUT version.
15 behavioralmap.sutVersion =1.0.0
16 ## Define the method used in the adaptation loop.
17 behavioralmap.sutAdaptationLoopMethodName =adasim.model.TrafficSimulator.
       ↪ takeSimulationStep()
```

features are executed every time the system is reconfigured. We chose the main class responsible for implementing the feature behavior. This class is crucial for feature implementation and helps us identify the hierarchy of dependencies at runtime. Thus, we can set in the Behavioral Map Black Box parameterization engine the core features in the level of class or package (*e.g.,* packages part of the core of the system used in the feature implementation). For instance, Listing 10.1 shows in line 6 the core features identified based on the Adasim source code. In this case, we define the core features-based implementation packages because the system uses the features packages to separate concerns.

Step 3 - Identify the optional features in the source code: We used the feature name or description along with the scenario where each feature is activated to identify optional features in the source code. Additionally, we analyzed the source code comments used to describe the class or method implementation to support feature identification. This information was then associated with the source code information collected in Step 1 to locate each feature. Such as the core features, the optional features can be set in the class or package level, as illustrated in line 8 in the Listing 10.1.

Step 4 - Behavioral Map Feature Trace: The features (class or packages) identified in steps 2 and 3 are included in the Feature Trace engine provided by the Behavioral Map Black Box, as depicted in Listing 10.1 in lines 6 and 8. Also, we define the system's name in line 2 and identify the system's main package (line 4) filter used to guide the analysis at runtime. Consequently, this parameter guarantee that only packages of the system under test will be analyzed at runtime. We define the controller and controlled features in lines 11 and 13. Line 15 sets the version of the system. Finally, line 17 shows which method should be intercepted to capture the adaptations loop at runtime and obtain the *Change Plan* that initiated the adaptation. In this case, the Behavioral Map should intercept the method `adasim.`-

Listing 10.2: *Test Selection Engine Configuration.*

```
1 ## The Behavioral Map can be set to run test selection based on feature interaction
2 ## type and Architectural Bad Smells identified at runtime. Test selection based on:
3 ## - Feature Relationship Analysis = interaction
4 ## - Architectural Bad Smells = smells
5 behavioralmap.sutTestPrioritizationBasedOn=interaction
```

`model.TrafficSimulator.takeSimulationStep()` because this method starts and ends the adaptation loop. As a result, the tool can **capture the environmental updates to generate test cases** to facilitate the test of context-aware programs such as SAS that reflect the system's environmental surroundings and the selected configuration. Sections 7.3.1 and A.5 of Appendix A provide all the configuration files used in Adasim and DeltaIoT analysis. Also, we included a short description for each parameter used to explain their parameter goal.

10.2.2 Test Selection Configuration

We selected the **test suite scope** to test based on the feature interaction for Adasim and DeltaIoT. Thus, the Behavioral Map will determine the test suite scope using the **Feature Relationship Analysis** process outlined in Section 6.5.1. Also, we selected the test scope based on the **Architectural Bad Smells** detected at runtime, as discussed in Section 6.5.1. However, this option was applied only in the Adasim case because we did not find smells in DeltaIoT.

Listing 10.2 shows the parameters used to define how the test suite scope should be selected. Thus, in this first experimentation phase, we set the test selection engine to select the scope based on interaction, as shown in line 5. In the last phase, we set the parameter `behavioralmap.sutTestPrioritizationBasedOn` to *smells*.

10.2.3 Test Generation Configuration

A challenge in testing SASs during runtime is creating test cases for the constantly changing environment. Therefore, we must create test cases for every configuration detected at runtime. To face this challenge, the **Behavioral Map Black Box supports the test case generation for each adaptation detected at runtime**. For this, we configure the Behavioral Map Black Box to generate testing at runtime, as depicted in Listing 10.3. Therefore, we set the option `behavioralmap.sutGenerateTestClasses` (line 4) to `true`. This option activates the test generation process at runtime. At line 6, we define the directory for TackleTest installation. Also, as we presented in Section 6.5.3, the Behavioral Map Black Box can generate test cases at runtime using the following strategies: i) adaptive-random test generation; ii) evolutionary algorithms to generate tests; and iii) Combinatorial Test Design (CTD) to generate test cases.

In this study, we apply the Behavioral Map to generate testing using the CTD combined with an evolutionary algorithm (supported by EvoSuite) and adaptive-random test (via Randoop) to guide the test generation at runtime. Thus, we set the

Listing 10.3: *Test Generation Engine.*

```
1  ########## AUTOMATED TEST-GENERATION CONFIGURATION ##########
2  ## By setting this to true, the Behavioral Map will generate the test classes based on
3  ## the test selection define by the Behavioral Map algorithm.
4  behavioralmap.sutGenerateTestClasses=true
5  ## The root directory for tackle-test-generator-cli installation.
6  behavioralmap.tackleTestGeneratorHome =/edilton/Downloads/tackle-test-generator-cli
7  ## The Behavioral Map supports three strategies for test generation: CTD-guided test
8  ## generation, test generation using EvoSuite standalone, and test generation using
9  ## Randoop standalone.
10 ## The test generator strategy used to generate the building-block test sequences:
11 ## - Test generation using EvoSuite only = evosuite
12 ## - Test generation using Randoop only = randoop
13 ## - Test generation using Evosuite and Randoop = combined
14 behavioralmap.testBaseTestGenerator=combined
15 ## If you select the options Evosuite or combined, you shoud select one or more
       ↪ Evosuite generation criteria.
16 behavioralmap.testEvosuiteGenerationCriterion =LINE,BRANCH,EXCEPTION,WEAKMUTATION,
       ↪ OUTPUT,METHOD,METHODNOEXCEPTION,CBRANCH
17 ## Do not augment CTD-guided tests with coverage-increasing base tests.
18 behavioralmap.testNoAugmentCoverage =true
19 ## Do not generate a CTD coverage report. Default value: false
20 behavioralmap.testNoCtdCoverage =false
21 ## CTD interaction level (strength) for a test plan generation. Default: 2. Its means
       ↪ pair-wise testing, in which all combinations of
22 ## subtypes for each pair of method parameters are included in the test plan.
23 behavioralmap.testInteractionLevel =2
24 ## Number of executions to perform to determine pass/fail status of generated sequences
       ↪ . Default value: 2
25 behavioralmap.testNumSeqExecutions =2
26 ## Time limit per class (in seconds) for evosuite/randoop test generation. Default: 2.
27 behavioralmap.testTimeLimit =2
28 ## Maximal heap size (in MB) used for obtaining coverage data.
29 behavioralmap.testMaxMemoryForCoverage =10240
```

parameter `behavioralmap.testBaseTestGenerator` (line 14) to `combined`. The tool uses the CTD-guided test generation process to create a CTD model for each public method in the specified application classes. This model is then utilized to generate a test plan. Each row of the test plan specifies a set of types for the method parameters, which becomes a coverage goal for test generation. By using a method-level approach, a set of test plans is created to guide the test generation process. The CTD model for a method includes a set of types for each formal parameter, identified statically through type inference and subtypes of the declared parameter type [124]. Also, we selected the test criterion `LINE`, `BRANCH`, `EXCEPTION`, `WEAKMUTATION`, `OUTPUT`, `METHOD`, `METHODNOEXCEPTION`, and `CBRANCH` to generate test cases, as shown in line 16.

Additionally, we do not augment CTD-guided tests with coverage-increasing base tests, as shown in line 18. It increases the time expended in the test generation. We adopted testing pair-wise based, as a consequence, the parameter `behavioralmap.testInteractionLevel` (line 23) was set to 2. Thus, all combinations of subtypes for each pair of method parameters are included in the test plan. Then, we define the number of executions to perform to determine the pass/fail status of generated sequences to 2, see line 25. The time limit per class (in seconds) for test generation was defined in line as 2 seconds. To increase the tool performance, the maximal heap size (in MB) used for obtaining coverage data was set to `10240` MB, as

Table 10.1: Processing time to generate testing at runtime for each SAS under analyses.

		Feature Relationship Analysis	**Architectural Smells**
System	**Adaptation Loops**	**CTD**	**CTD**
Adasim	15	01:18:02	01:08:04
DeltaIoT	96	05:59:00	-

Table 10.2: Adasim test strategy coverage.

Adasim		**Adaptation Loop**														
Scope Selection	**Test Strategy**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
F. Relationship	CTD	50%	50%	50%	50%	51%	51%	51%	51%	51%	51%	51%	51%	51%	51%	51%
Architectural Smells	CTD	52%	53%	53%	53%	53%	53%	53%	53%	53%	53%	53%	53%	53%	53%	53%
		Test Strategy Coverage														

shown in line 29.

10.2.4 Architectural Bad Smells

Our research delved into two types of Architectural Bad Smells (ABS): Cyclic Dependency (CD) and Hub-Like Dependency (HL). We considered examining these ABS because they are highly interconnected and require more maintenance, making them more critical. Additionally, these ABS pose significant challenges related to dependency issues. The selected ABS was introduced in Sections 2.2.5 and 2.2.6.

10.3 Experimental Results

This Section discusses the results obtained by applying the Behavioral Map Black Box to generate and run tests for Self-Adaptive Systems (SAS) at runtime.

10.3.1 Running the Behavioral Map

We used a Linux (Ubuntu 20.04.6 LTS) ARM 64-bit in a Virtual Machine with 32GB RAM, a CPU with 2 gigahertz and four processing cores, and a 256GB hard disk to run the Behavioral Map. Table 10.1 presents the time it took to generate the tests at runtime using the Combinatorial Test Design (CTD) strategy, with the test scope selected based on Feature Relationship Analysis and Architectural Smells. Adasim executed 15 adaptation loops for each scope selection strategy at runtime, resulting in 30 adaptations analyzed at runtime. In comparison, DeltaIoT had no Architectural Bad Smells (ABS), so we could not generate testing based on Architectural Smells. Thus, 96 adaptation loops were analyzed at runtime using Feature Relationship Analysis.

10.3.2 Adasim

The tool generated tests for Adasim at runtime using the testing scope based on Feature Relationship and Architectural Smells as presented in Table 10.1. The Behavioral Map strategy took one hour, eighteen minutes, and two seconds to generate and run tests based on CTD with scope selected via Feature Relationship Analysis. On the other hand, in the test generation guided by Architectural Smells, the tool consumed one hour, eighteen minutes, and 4 seconds to generate and run tests. Consequently, the tool used less time to generate the tests and test the system at runtime.

Table 10.2 summarizes the code coverage of tests generated at runtime using Feature Relationship Analysis and Architectural Smells Analysis guided scoping. In the following sections, we discuss these results.

Feature Relationship Analysis

In Adasim, the number of features activated during runtime varied between 17 and 20. The first adaptation activated 20 features, while the last only activated 17. Meanwhile, the generated test coverage covered 50 percent of the system code between adaptations 1 and 4 for the testing scope selected via Feature Relationship analysis, as shown in Table 10.2. Starting from adaptation five, the code coverage index remained at 51%. We observed that the number of features loaded in each adaptation loop did not affect the accuracy of the CTD because the test generation operates in two steps. Firstly, it uses EvoSuite and/or Randoop to create a set of initial test cases, which are then used to gather "building-block" test sequences and add them to a sequence pool [73]. In the next step, the engine goes through the CTD test plans and attempts to produce a complete test sequence for each row by creating objects/values of the types specified and reusing sequences from the pool. Also, we used pair-wise interaction as the coverage objective to generate the test plans. Thus, many different test plans can achieve the same t-way coverage [124].

Architectural Smells Analysis

The Adasim configuration used in the Feature Relationship Analysis was also used for this strategy. Thus, the number of activated features ranged from 17 to 20. The initial adaptation resulted in 20 activated features, while the final adaptation only resulted in 17. However, the test code coverage results are slightly different, as shown in Table 10.2. The generated test covered 52% of the system code between adaptation 1 and for the other adaptation 53%. This situation happened because we selected the test scope based on architectural bad smells. Thus, using pairwise, the Behavioral Map Black Box will generate tests only for classes involved in ABS and its dependency. Pairwise testing involves testing all combinations of subtypes for each pair of method parameters in the test plan [73]. Table 10.3 shows the ABSs detected at runtime and which adaptation is affected by them. As the table outlines, eight core features are involved in various ABSs in different adaptation loops. It means the system's features are numerous dependencies with other abstractions, such as other

Table 10.3: ABSs identified by the BM in Adasim adaptation loops.

		Architectural Bad Smells			
Feature Name	**Feature Type**	**CD**	**HL**	**CD**	**HL**
TrafficSimulator	Core	Yes		Yes	
RoadSegment	Core	Yes	Yes	Yes	
Vehicle	Core	Yes	Yes	Yes	Yes
VehicleManager	Core	Yes		Yes	
RoadVehicleQueue	Core	Yes		Yes	
AdasimMap	Core	Yes		Yes	
QLearningRoutingAlgorithm	Optional	Yes			
SimulationXMLBuilder	Core		Yes		
SimulationXMLReader	Core		Yes		Yes
Adaptation Loop		1-5, 7-10, 12, 14	1	6, 11, 13, 15	2-15

Table 10.4: DeltaIoT test strategy coverage.

DeltaIoT		**Adaptation Loop**			
Scope Selection	**Test Strategy**	**1-2**	**3-4**	**5-22**	**23-96**
F. Relationship	CTD	72%	73%	74%	75%
		Test Strategy Coverage			

components, and more components rely on each other directly or indirectly [77]. Consequently, the generated test will cover much of the system code mainly because the architectural smells focused on the system's core under analysis, as shown in Table 10.3.

10.3.3 DeltaIoT

DeltaIoT performed 96 self-adaptations during runtime and loaded eight features for each adaptation loop from the initial adaptations until the second-to-last one. However, the final adaptation loop only included five core features, which means that three optional features were not loaded. To ensure comprehensive testing for DeltaIoT, we employ Feature Relationship Analysis to determine the appropriate testing scope during runtime. Table 10.4 shows the test strategy code coverage for all self-adaptation analyzed at runtime. However, the test code coverage results are slightly different, as shown in Table 10.4. For instance, in adaptation loops 1 and 2, the test cases generated at runtime covered 72% of the system source code. In DeltaIoT, self-adaptation 3 and 4 covered increased by one point, which means 73% covered. However, between the adaptations, loops 5 to 22 covered increased to 74%, and finally, during adaptations 23 to 96, the test code covered stayed at 75%.

Unfortunately, the Behavioral Map Black Box could not find ABS on DeltaIoT be-

cause of how the simulator was designed. DeltaIoT is made up of five sub-packages: `deltaoit.client`, `deltaoit.main`, `deltaoit.mapek`, `deltaoit.scenario_data`, and `deltaoit.services`. Our analysis focused solely on these packages, encompassing DeltaIoT's implementation. We did not examine any third-party framework packages, such as the `domain` and `simulator` used for network infrastructure simulation, as our goal was to analyze the architectural quality of the DeltaIoT system.

10.4 Discussion

This section answers the research questions presented in section 10.1.1.

10.4.1 Research Questions

Answering RQ1. The findings demonstrated the approaches' viability in selecting the test suite scope at runtime based on the adaptation loop under tests. Also, we used the Combinatorial Test Design (CTD) strategy to generate the tests at runtime based on Feature Relationship Analysis and Architectural Bad Smells Analysis. As a result, the time used to generate tests at runtime varied according to the test scope selection approach employed and the quantity of self-adaptation performed by the system under tests.

Answering RQ2. Based on test results performed on the Adasim system presented in Section 10.3.2, we may suggest that the test scope selection based on Architectural Bad Smells Analysis is more effective rather than Feature Relationship Analysis. This is because the number of selected classes through Architectural Bad Smells Analysis is significantly less than the number of selected classes through Feature Relationship Analysis. During the first adaptation loop, Adasim loaded 20 features at runtime. However, only 9 features were selected for testing through Architectural Bad Smells Analysis, while 19 features were selected through Feature Relationship Analysis. Despite the smaller scope size, the coverage of the test generated through CTD is considerable, which is evident in Table 10.2. We cannot make the same assessment with DeltaIoT because we cannot find architectural smells in the system. However, given the DeltaIoT results (6 hours to generate and run the tests for all the adaptations), testing large SAS implementations based on Feature Relationship Analysis is time-consuming.

10.5 Threats to Validity

When conducting an empirical study, it is essential to consider potential threats to validity, which could impact the accuracy of the results or their general applicability. Thus, in the following section, we present the threats to validity faced in this study.

10.5.1 Internal Validity

Identifying variability in source code can be challenging in the absence of a feature model or annotations. To overcome this, we employed the process described in Section 10.2.1. Moreover, we meticulously analyzed execution logs to ensure precise feature recognition. Our ultimate objective is to verify the correct identification of all core and optional features in the source code. In addition, some self-adaptive systems selected have incorporated test classes that may produce discrepancies in the automatic test coverage calculations. As a result, we have decided to delete all implemented test classes from the source code and related dependencies.

10.5.2 External Validity

It is important to note that our research only pertains to two specific SAS systems and may not be applicable to all systems. However, we did evaluate these two systems based on their unique adaptation mechanisms and application domains, which helps to address this limitation. Our aim was not to obtain statistical evidence on the differences in Feature Relationship Analyses and Architectural Bad Smells but rather to assess the viability of the proposed approach for selecting the test scope. Therefore, a more quantitative assessment will be conducted in future work.

10.6 Related Work

This section will discuss the related work that is most relevant to our research. We have chosen papers that focus on creating and implementing tests for SAS.

RETAkE [37] is a method of conducting runtime testing based on the variability of the system's context and the modeling of its features. It verifies the system's adaptation rules using its variability model while also verifying its behavioral properties. The authors evaluated RETAkE using the mutation testing technique [68, 135] with two SAS and measured the overhead introduced when integrating it into the SAS. Our approach allows defining the test scope based on feature interaction or architectural bad smells detected in the selected configuration at runtime through the Behavioral Map. The Behavioral Map then generates test cases using strategies such as Combinatorial Test Design, adaptive-random test generation, and evolutionary algorithms to ensure adequate code coverage (including method, statement, branch, and exception coverage). Once the tests are generated, the tool executes them at runtime.

Eberhardinger *et al.* [38] utilized runtime testing principles to confirm that the system could respond to inputs and reorganize accordingly. They achieved this by simulating the environment and introducing controlled faults into the system under test. On the other hand, the Behavioral Map does not require a simulated SASs environment for testing. Instead, the tool executes SASs in its Java Virtual Machine (JVM) and monitors the feature interactions and workflows. It identifies any architectural issues during runtime, selects the appropriate test scope, generates test cases, and runs the tests.

Proteus [47] is a framework that allows for online adaptive testing to ensure that tests remain relevant to changing operating conditions. It adjusts test suites and test cases at runtime and creates an adaptive test plan for each operating context. This plan includes all possible test suites related to a specific system configuration and environmental parameters. Proteus implements the MAPE-T [48] runtime testing feedback loop to improve testing assurance. As a Proteus, the Behavioral Map runs testing at runtime but does not adjust test suites and test cases. Instead, the tool creates test cases for each operating context during the system self-adaptation based on the selection of the scope (*e.g.,* feature interaction or architectural bad smells).

Hansel *et al.* [60] has introduced a systematic SAS testing scheme that enables engineers to test feedback loops at an early stage of development. They achieve this by utilizing architectural runtime models [15] that are typically available early in the development process. Feedback loop activities commonly use these models during runtime, providing a high-level abstraction that describes test inputs and expected results. Thus, to test for failures, a simulator is used. This simulator injects failures into the runtime model, which emulates the behavior of the adaptable software and environment. It also includes a monitor step that reflects any failures in the model. In contrast, our approach runs the system under test and generates the test case at runtime for each adaptation based on test criteria selected by developers.

10.7 Wrap up and perspectives

In this chapter, we presented a preliminary study to assess the feasibility of selecting the test scope at runtime using a feature relationship analysis and an architectural bad smells analysis approach. The Behavioral Map Black Box, introduced in this thesis, is used to select the test scope, generate the tests at runtime using the Combinatorial Test Design strategy, and run the tests. The findings indicate that using this approach to select the test scope at runtime for self-adaptive systems is feasible. The main limitation of this study is that it is only limited to coverage, we would like to explore other aspects (such as the bug-finding ability, and test suite size) in the future.

Part V

Postface

CHAPTER 11

CONCLUSION

A Self-adaptive System (SAS) is a sophisticated system designed to handle any changes that may arise in its operating environment. This system can trigger necessary adaptations at runtime, changing its structure, behavior, or even its adaptation mechanism. However, it is crucial to state that these changes can introduce architectural issues (*e.g.,* architectural bad smells) and defects into the system, which can cause it to fail during runtime. It is of utmost importance to closely monitor and manage these modifications to uphold the system's dependability.

In this thesis, we introduced the Behavioral Map framework to identify feature interaction and architectural bad smells at runtime. With this framework, we can efficiently determine the testing scope and produce test cases dynamically at runtime based on the scope selected through Feature Relationship Analysis or Architectural Bad Smells analysis.

11.1 Summary of contributions

Performing runtime analysis to verify the behavior (*e.g.,* test at runtime) and architectural issues for self-adaptive systems at runtime is challenging. For this, we proposed the Behavioral Map framework. The framework supports **feature interaction detection and architectural bad smells detection at runtime**. Thus, based on the feature interactions detected at runtime, a **graphical map** is created to show how they interact among them. Additionally, this map is used to identify architectural bad smells that may arise in each reconfiguration at runtime.

Also, the **Behavioral Map framework** supports the **test suite scope selection at runtime** based on feature relationship analysis and architectural bad smells detection. Consequently, our framework **generates test cases and runs unit tests at runtime**. The Behavioral Map framework offers three strategies to generate test

cases: i) **adaptive-random test generation**, ii) **evolutionary algorithms** to generate tests, and iii) **Combinatorial Test Design** to generate test cases.

The Behavioral Map framework is conceived in two versions: Behavioral Map White Box and Behavioral Map Black Box, both implemented in Java 8. The Behavioral Map White Box was implemented as reusable building blocks that allow their incorporation into the system under analysis. On the other hand, the Behavioral Map Black Box automatically executes the system under test (SUT) methods from the host Java Virtual Machine (JVM). This characteristic makes it a convenient option for those who want to avoid the hassle of modifying the SUT's code.

Our empirical assessments are performed on several case studies, using self-adaptive systems of various domains provided by the Software Engineering for Self-Adaptive Systems community[1] with diverse adaptive mechanisms and architectural implementations.

11.2 Perspectives and future work

This section presents our perspectives and future potential research directions to improve the architectural bad smells detection, test case selection, test generation strategy, and Assessment of Test Generation.

11.2.1 Architectural Bad Smells Detection

Architectural Bad Smells detection may be improved in two ways:

- The Oppressed Monitors (OM) [114] architectural bad smell is partially detected because fully identifying this smell involves delving into the source code and getting information about the polling rate since the sequencing of sensor calls is not present on the map. Thus, we need to include a strategy to automatically identify the polling rate for each sensor detected at runtime based on the source code. However, depending on the SAS implementation, this information is unavailable in the source code or system configuration files. As a result, we also need to define a parametrization in the framework that allows us to define the polling rate for each sensor.
- Our goal is to identify new architectural smells using the Behavioral Map framework. We aim to detect Connector Envy, Scattered Functionality, and Ambiguous Interfaces architectural smells as they can adversely affect the quality of the SAS, including its reusability, maintainability, extensibility, understandability, and testability [51].

11.2.2 Test Case Selection

The last version of the Behavioral Map can select testing scope based on Feature Interaction Analysis and Architectural Bad Smells detected at runtime. However, there is a possibility to select the testing scope based on the most executed methods

[1] Software Engineering for Self-Adaptive Systems exemplars - https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

at runtime in a given adaptation. Our framework offers a report (presented in Section 7.3.2) that shows all the most executed methods at runtime. Thus, based on this report, we can select the testing scope and generate the test cases only for the methods most executed and their dependency loaded at runtime. We envision that such a strategy could reduce the time consumed in generating and running the test cases.

11.2.3 Test Generation Strategy

Testing self-adaptive systems at runtime poses a significant challenge as the system may support (re)configurations not anticipated during the design phase. This situation makes ensuring the accuracy of SAS configurations that were not tested beforehand challenging. To address this issue, two questions must be answered: (1) how to create test cases that cover unforeseen configurations, and (2) how to define a testing oracle that covers unforeseen configurations [80, 86, 100]. To face the former question, we proposed in this thesis to utilize the Behavioral Map framework to select the test scope and generate test cases based on the configurations detected during runtime. Additionally, we plan to apply metamorphic testing [23] at runtime to generate the test cases [54, 113] to address the second question.

11.2.4 Assessment of Test Generation

We will evaluate our test methodology to identify the best combination of test scope selection and generation strategies to increase test coverage and reduce the testing time. Thus, we will be able to understand which combination (test scope selection and generation strategies) can present the best cost-benefit ratio. Beyond these criteria, we would like to explore the bug-finding ability of these strategies and conduct user studies.

11.3 Final remarks

The purpose of this thesis is to conduct architectural analysis (*e.g.*, Architectural Bad Smells) and testing of Self-adaptive Systems at runtime. Although the Behavioral Map framework is designed for Self-adaptive Systems or other types of Dynamic Adaptive Systems, it can also be applied to other types of systems. By disabling the monitoring of the adaptation loop performed by Behavioral Map Black Box, the tool will only execute the architectural analysis and run tests when the system under test completes its execution. Thus, our contributions may be extended to other types of systems.

Also, the Behavioral Map Black Box provides two reports (Invoked Methods Report and Stack Trace Report) that can be used to enhance the effectiveness of the unit test developed. These reports combined to provide a comprehensive list of the commonly used methods and their respective parameter values at runtime. By leveraging these reports, developers can prioritize the unit test development strategy effectively and ensure their system's optimal test code coverage.

BEHAVIORAL MAP BLACK BOX

A.1 Feature Trace Configuration Options

Listing A.1 and A.2 present all configurations supported by the Feature Trace engine. Also, in each configuration, include a short explanation about their goal. For instance, the parameter `behavioralmap.sutSuppressorFeatures` defines a list of the classes or packages used to suppress other classes' behavior at runtime. These classes or packages may trigger a negative impact on the behavior of the system at runtime. Thus, we use this configuration to identify the suppressor on the behavioral map. Correspondingly, we can define the list of classes whose behavior is suppressed by classes defined as suppressors. For this, we can use the parameter `behavioralmap.sutSuppressedFeatures`.

The Feature Trace engine supports the definition of the list of the classes or packages used as a presenter. For instance, the class used to implement the dashboard can be defined via the parameter `behavioralmap.sutPresenterFeatures`. Too, it is possible to define the list of the classes or packages used as a sensor. For example, a class that implements read data via a physical sensor can be set using the parameter `behavioralmap.sutSensorFeatures`. The parameter `behavioralmap.sutActuatorFeatures` defines a list of the classes or packages used as an actuator. All parameters are available in the Feature Trace engine are used in Behavioral Map Black Box to validate and create a behavioral map for each configuration detected at runtime.

A.2 Chapter 10 - Case Study Configuration Files

This section presents all configuration files used in the Behavioral Map Black Box to run the case study proposed in Chapter 10. Section A.2.1 presents the configuration

Listing A.1: *Feature Trace configuration options to run the SUT part 1.*

```
1  ## SUT what the Behavioral Map should run. Sets the Main class name and path.
2  behavioralmap.sutAppName =
3  ## The main package of the system under analysis. Example: myApp or com.myApp
4  behavioralmap.sutMainPackageFilter =
5  ## List of the packages used to storage the core features of system under analysis.
6  behavioralmap.sutCoreFeaturePackages =
7  ## List of the packages used to storage the optional features of system under analysis.
8  behavioralmap.sutOptionalFeaturePackages =
9  ## List of the classes or packages used to control other classes behavior at runtime.
10 behavioralmap.sutControllerFeatures =
11 ## List of classes whose behavior is controlled by classes defined as controllers.
12 behavioralmap.sutControlledFeatures =
13 ## List of the classes or packages used as a suppressor of other classes' behavior
14 ## at runtime.
15 behavioralmap.sutSuppressorFeatures =
16 ## List of classes whose behavior is suppressed by classes defined as suppressor.
17 behavioralmap.sutSuppressedFeatures =
18 ## List of the classes or packages used as a presenter, for instance class used to
       ↪ implement the dashboard.
19 behavioralmap.sutPresenterFeatures =
20 ## List of the classes or packages used as a sensor, for instance class used to
       ↪ implement read data via physical sensor.
21 behavioralmap.sutSensorFeatures =
22 ## List of the classes or packages used as an actuator.
23 behavioralmap.sutActuatorFeatures =
24 ## Define the classpath the place where we can find the compiled test class without
       ↪ dependency.
25 behavioralmap.sutTestClassCompiledClasspath=
26 ## SUT version.
27 behavioralmap.sutVersion =
```

file used to run the Adasim and Section A.2.2 DeltaIoT configuration file.

A.2.1 Adasim Configuration Files

Listings A.3 and A.4 show the configuration necessary to run the Adasim system on the Behavioral Map Black Box and intercept the adaptation loop at runtime.

Listing A.3, line 2 loads the Behavioral Map Black Box, and line 4 targets the main class used to run the Adasim. Line 6 defines the methods that should be executed directly on Java Virtual Machine (JVM). Thus, the parameter nhandler.spec.delegate should delegate a JVM to all methods implemented in third-party frameworks. Lines 17 until 56 are used to define the classpaths of the system. Thus, line 17 defines where we can find the .class file or other systems dependency used to run the system. Line 28 defines the localization of the source code, and line 36 defines the classpath to compile the test class without dependency. Line 43 the classpath for compiled test class without dependency. The last parameter set path to the Adasim path to JavaPathFinder.

Listing A.4 shows the Feature Trace configuration. Thus, we define the system under the test name (line 2) and the system's main package in line 4. Lines 6 until 12 define packages or classes to implement each feature type. Line 16 sets the adaptation loop method that should be monitored at runtime. Finally, line 18 defines the type of architectural smells that should be verified at runtime.

Listing A.2: *Feature Trace configuration options to run the SUT part 2.*

```
1  ## Define the method used in the adaptation loop. Note: If the SUT there is more than
       ↪ one method inside of the
2  ## adaptation loop, inform on the parameter sutAdaptationLoopMethodName only the last
       ↪ method executed or invoked inside of the adaptation loop.
3  ## Example of method name:
4  ## - Method less parameter: com.myAppPackage.ClassName.methodName() - > real name: com.
       ↪ myApp.MainClass.adaptationLoop()
5  ## - Method with primitive type parameter, you need to include the parameter type as
       ↪ follow:
6  ## ---> com.myApp.MainClass.adaptationLoop(int) or com.myApp.MainClass.adaptationLoop(
       ↪ boolean) or
7  ## com.myApp.MainClass.adaptationLoop(double) or om.myApp.MainClass.adaptationLoop(int,
       ↪  double) or
8  ## com.myApp.MainClass.adaptationLoop(long) or om.myApp.MainClass.adaptationLoop(long,
       ↪ double)
9  ## - Method with object type parameter, you need to include the parameter type as
       ↪ follow:
10 ## ---> com.myAppPackage.ClassName.methodName(ObjectTypeName) or com.myAppPackage.
       ↪ ClassName.methodName(ObjectTypeName1, ObjectTypeName1)
11 ## ---> com.myApp.MainClass.adaptationLoop(Object) or com.myApp.MainClass.
       ↪ adaptationLoop(String) or
12 ## ---> com.myApp.MainClass.adaptationLoop(Object, Object) or com.myApp.MainClass.
       ↪ adaptationLoop(String, String)
13 ## - Method with array or collection (as Set, HashSet, etc.) type parameter, you need
       ↪ to include the parameter type as follow:
14 ## ---> com.myAppPackage.ClassName.methodName(Object[]) or com.myAppPackage.ClassName.
       ↪ methodName(Set) or
15 ## ---> com.myAppPackage.ClassName.methodName(String[]) or com.myAppPackage.ClassName.
       ↪ methodName(HashSet)
16 behavioralmap.sutAdaptationLoopMethodName =
17 ## By setting this to true, the Behavioral Map will look for the Extraneous Connector (
       ↪ EC),
18 ## Oppressed Monitors (OM), Cyclic dependency (CD), and Hub-Like Dependency (HL).
       ↪ Otherwise,
19 ## the Behavioral Map will look for only CD and HL. Because architectural smell happens
       ↪  only
20 ## in publish-subscribe systems. Also, you should inform the communication broker Class
       ↪  in
21 ## the parameter behavioralmap.sutCommunicationBrokerClass.
22 ## Note: If you activated for TRUE but the system doesn't was implemented as a publish-
       ↪ subscribe
23 ## system, the analysis results for the EC and OM architectural smells will exhibit the
       ↪  wrong results.
24 behavioralmap.sutIsPublishSubscribeSystem =false
25 ## Define the Class used in the SUT to implement the Communication Broker Class. This
       ↪ class on used to
26 ## exchange messages among the system features. Example: com.myApp.
       ↪ CommunicationBrokerClass
27 behavioralmap.sutCommunicationBrokerClass =
```

A.2.2 DeltaIoT Configuration Files

The configuration presented in the Listing A.5 shows the configuration necessary to run the DeltaIoT system on the Behavioral Map Black Box and intercept the adaptation loop at runtime. For instance, line 2 loads the Behavioral Map Black Box, and line 4 targets the main class used to run the DeltaIoT. Line 6 defines the system's name under tests, and line 8 defines the methods that should be executed directly on Java Virtual Machine (JVM). Thus, the parameter nhandler.spec.delegate should be used to delegate a JVM to all methods implemented in third-party frameworks. On the other hand, the parameter nhandler.resetVMState is used to reset the virtual

machine state. Lines 15 until 51 are used to define the classpaths of the system. For instance, line 15 defines where we can find the .class file or other systems dependency used to run the system. Line 27 defines the localization of the source code, and line 39 defines the classpath to compile the test class without dependency.

Also, the Listing A.5 shows how the Features were identified and how to define the method responsible for perform the adaptations at runtime, as illustrated in line 65.

Listing A.3: *Adasim Configuration File (part 1) used to run the system under Behavioral Map.*

```
## Loads the Behavioral Map
@using=jpf-behavioralmap
## Main class name and path.
target=adasim.TrafficMain
## Delegate methods to JVM host.
nhandler.spec.delegate=org.jdom.input.SAXBuilder.*,
org.jdom.Element.*,
org.apache.xerces.parsers.AbstractSAXParser.*,
org.apache.xerces.util.SAXMessageFormatter.*,
java.util.ResourceBundle$Control.newBundle,
java.util.ResourceBundle.loadBundle,
java.util.ResourceBundle.getBundleImpl,
java.util.ResourceBundle.findBundle,
java.util.ResourceBundle.getBundle,
java.util.ResourceBundle.getClassContext
## The place where we can find the .class file or other systems dependency.
classpath=${sut-workspace}/adasim/bin,
${sut-workspace}/adasim/visual,
${sut-workspace}/adasim/lib/jdom-1.1.2.jar,
${sut-workspace}/adasim/lib/log4j-1.2.16.jar,
${sut-workspace}/adasim/lib/jopt-simple-3.2.jar,
${sut-workspace}/adasim/lib/junit-4.8.2.jar,
${sut-workspace}/adasim/lib/pjunit-0.2.jar,
${sut-workspace}/adasim/resources/test,
${sut-workspace}/adasim/resources,
${sut-workspace}/adasim/resources/xml
## System source code.
sourcepath=${sut-workspace}/adasim/src,
${sut-workspace}/adasim/test,
${sut-workspace}/adasim/resources/test,
${sut-workspace}/adasim/resources,
${sut-workspace}/adasim/resources/xml,
${sut-workspace}/adasim/releases,
${sut-workspace}/adasim/visual
## System test classpath (compiled test class with dependency).
test_classpath=${sut-workspace}/adasim/bin,
${sut-workspace}/adasim/test,
${sut-workspace}/adasim/resources/test,
${sut-workspace}/adasim/resources/xml,
${sut-workspace}/adasim/resources,
${sut-workspace}/adasim/visual
## Define the classpath to compiled test class without dependency.
behavioralmap.sutTestClassCompiledClasspath=${sut-workspace}/adasim/bin
## Used by the nhandler
native_classpath=${sut-workspace}/adasim/bin,
${sut-workspace}/adasim/visual,
${sut-workspace}/adasim/lib/jdom-1.1.2.jar,
${sut-workspace}/adasim/lib/log4j-1.2.16.jar,
${sut-workspace}/adasim/lib/jopt-simple-3.2.jar,
${sut-workspace}/adasim/lib/junit-4.8.2.jar,
${sut-workspace}/adasim/lib/pjunit-0.2.jar,
${sut-workspace}/adasim/resources/test,
${sut-workspace}/adasim/resources,
${sut-workspace}/adasim/resources/xml,
${sut-workspace}/adasim/releases,
${sut-workspace}/adasim/visual
```

Listing A.4: *Adasim Configuration File (part 2) used to run the system under Behavioral Map.*

```
## SUT app name.
behavioralmap.sutAppName =adasim
## The main package of the system under analysis.
behavioralmap.sutMainPackageFilter=adasim
## List of the packages used to storage the core features of system under analysis.
behavioralmap.sutCoreFeaturePackages=adasim,adasim.agent,adasim.filter,adasim.generator
    ↪ ,adasim.model.*,adasim.util
## List of the packages used to storage the optional features of system under analysis.
behavioralmap.sutOptionalFeaturePackages=adasim.algorithm.*
## List of the classes or packages used to control other classes behavior at runtime.
behavioralmap.sutControllerFeatures=adasim.algorithm.*,adasim.agent.RoadClosureAgent
## List of classes whose behavior is controlled by classes defined as controllers.
behavioralmap.sutControlledFeatures=adasim.model.RoadSegment,adasim.model.Vehicle
## SUT version.
behavioralmap.sutVersion =1.0.0
## Define the method used in the adaptation loop.
behavioralmap.sutAdaptationLoopMethodName =adasim.model.TrafficSimulator.
    ↪ takeSimulationStep()
## The Behavioral Map will look for only CD and HL.
behavioralmap.sutIsPublishSubscribeSystem =false
```

Listing A.5: *DeltaIoT Configuration File used to run the system under Behavioral Map.*

```
## Loads the Behavioral Map
@using=jpf-behavioralmap
## Main class name and path.
target=deltaiot.main.SimpleAdaptation
## System under test name.
behavioralmap.sutAppName =deltaiot
## Delegate methods to JVM host.
nhandler.spec.delegate=com.google.gson.annotations.Expose.*,
com.google.gson.Gson.*,
com.google.gson.GsonBuilder.*,
java.lang.Thread.currentThread().getContextClassLoader().getResourceAsStream
## Used to reset virtual machine state.
nhandler.resetVMState =true
## The place where we can find the .class file or other systems dependency.
classpath=${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/client,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/main,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/mapek,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/scenario_data,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/services,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/domain,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/simulator,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/lib/gson-2.2.4.jar,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/info
## System source code.
sourcepath=${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot/client,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot/main,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot/mapek,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot/scenario_data,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/deltaiot/services,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/domain,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/src/simulator
## System test classpath (compiled test class with dependency).
#test_classpath=${sut-workspace}/
## Define the classpath to compiled test class without dependency.
behavioralmap.sutTestClassCompiledClasspath=${sut-workspace}/
## Used by the nhandle-jpf to delegates libary execution.
native_classpath=${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/client,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/main,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/mapek,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/scenario_data,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/deltaiot/services,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/domain,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/bin/simulator,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/lib/gson-2.2.4.jar,
${sut-workspace}/DeltaIoT/DeltaIoT-source/Simulator2/info
## The main package of the system under analysis.
behavioralmap.sutMainPackageFilter=deltaiot
## List of the packages used to storage the core features of system under analysis.
behavioralmap.sutCoreFeaturePackages=deltaiot,deltaiot.main,deltaiot.mapek,deltaiot.
    ↪ client
## List of the packages used to storage the optional features of system under analysis.
behavioralmap.sutOptionalFeaturePackages=deltaiot.services.*
## List of the classes or packages used to control other classes behavior at runtime.
behavioralmap.sutControllerFeatures=deltaiot.client.Effector
## List of classes whose behavior is controlled by classes defined as controllers.
behavioralmap.sutControlledFeatures=deltaiot.services.Mote
## SUT version.
behavioralmap.sutVersion =1.0.0
## Define the method used in the adaptation loop.
behavioralmap.sutAdaptationLoopMethodName =deltaiot.mapek.FeedbackLoop.execution()
## The Behavioral Map will look for only CD and HL.
behavioralmap.sutIsPublishSubscribeSystem =false
```

APPENDIX B

ACRONYMS

AS - Architectural Smells
ABS - Architectural Bad Smells
AI - Ambiguous Interfaces
API - Application Programming Interface
ATRP - Automated Traffic Routing Problem
BM - Behavioral Map
CIT - Combinatorial Interaction Testing
CT - Combinatorial Testing
CTD - Combinatorial Test Design
$\mathbb{CR}$ - Configuration Rules
CD Cyclic Dependency
CE - Connector Envy
CFA - Control-Flow Analysis
EC - Extraneous Connector
ECA - Event-Condition-Action
DAS - Dynamic Adaptive Systems
DSPL - Dynamic Software Product Line
FM - Feature Model
FODA - Feature Oriented Domain Analysis
HL - Hub-Like Dependency
ICSA - International Conference on Software Architecture
JPF - Java Pathfinder
JVM - Java Virtual Machine
JRE - Java Runtime Environment
JDK - Java Development Kit
MBT - Model-Based Testing

POM - Project Object Model
OM - Oppressed Monitors
OTS - off-the-shelf
SAS - Self-Adaptive Systems
SBFL - State-Based Feedback Loop
SF - Scattered Functionality
SPL - Software Product Lines
SHE - Smart Home Environment
SUT - System Under Test
SWEBOK - Software Engineering Body of Knowledge
UML - Unified Modeling Language
V & V - Validation and Verification

BIBLIOGRAPHY

[1] Oscar Aguayo and Samuel Sepúlveda. Variability management in dynamic software product lines for self-adaptive systems—a systematic mapping. **Applied Sciences**, 12(20):10240, 2022.

[2] Sven Apel and Christian Kästner. An overview of feature-oriented software development. **Journal of Object Technology**, 8(5):49–84, 2009.

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

[4] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In **2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**, pages 13–23. IEEE, 2015.

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

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

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

[8] Davi Monteiro Barbosa, Romulo Gadelha De Moura Lima, Paulo Henrique Mendes Maia, and Evilasio Costa. Lotus@ runtime: A tool for runtime monitoring and verification of self-adaptive systems. In **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**, pages 24–30. IEEE, 2017.

[9] Luciano Baresi and Clément Quinton. 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**, pages 57–63. IEEE Press, 2015.

[10] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Dynamic software product line engineering: a reference framework. **International Journal of Software Engineering and Knowledge Engineering**, 27(02):191–234, 2017.

[11] Len Bass, Paul Clements, and Rick Kazman. **Software architecture in practice**. Addison-Wesley Professional, 2003.

[12] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. **Information systems**, 35(6):615–636, 2010.

[13] Nelly Bencomo, Peter Sawyer, Gordon S Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In **SPLC (2)**, pages 23–32, 2008.

[14] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In **2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems**, pages 10–19. IEEE, 2009.

[15] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run.time. **Computer**, 42(10):22–27, 2009.

[16] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. **Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0**. IEEE Computer Society Press, Washington, DC, USA, 3rd edition, 2014.

[17] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. **Software engineering for self-adaptive systems**, pages 48–70, 2009.

[18] Javier Cámara, Rogerio De Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Testing the robustness of controllers for self-adaptive systems. **Journal of the Brazilian Computer Society**, 20:1–14, 2014.

[19] Javier Cámara, Rogerio De Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness-driven resilience evaluation of self-adaptive software systems. **IEEE Transactions on Dependable and Secure Computing**, 14(1):50–64, 2015.

[20] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. **Journal of Systems and Software**, 91:3–23, 2014.

[21] Nicolás Cardozo and Ivana 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**, pages 108–114, 2020.

[22] Marc Carwehl, Thomas Vogel, Genaína Nunes Rodrigues, and Lars Grunske. Runtime verification of self-adaptive systems with changing requirements. **arXiv preprint arXiv:2303.16530**, 2023.

[23] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. **arXiv preprint arXiv:2002.12543**, 2020.

[24] Betty HC Cheng, Holger Giese, Paola Inverardi, Jeff Magee, Rogério de Lemos, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research road map. In **Dagstuhl Seminar Proceedings**, volume 5525, pages 1–26. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Springer, 2009.

[25] HC al Cheng Betty et al. Software engineering for self-adaptive systems: A research roadmap. **Software Engineering for Self-adaptive Systems**, pages 1–26, 2009.

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

[27] Paul Clements and Linda Northrop. **Software Product Lines: Practices and Patterns**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

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

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

[30] Rogério De Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, et al. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In **Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers**, pages 3–30. Springer, 2017.

[31] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research

roadmap. In **Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers**, pages 1–32. Springer, 2013.

[32] Anind K Dey. Understanding and using context. **Personal and ubiquitous computing**, 5:4–7, 2001.

[33] Jorge Andrés Díaz-Pace, Antonela Tommasel, and Daniela Godoy. Towards anticipation of architectural smells using link prediction techniques. In **2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)**, pages 62–71. IEEE, 2018.

[34] Edilton Lima dos Santos. Stars: Software technology for adaptable and reusable systems. In **Proceedings of the 25th International Systems and Software Product Line Conference (SPLC)**, pages 13–17. ACM, 2021.

[35] Edilton Lima dos Santos, Sophie Fortz, Gilles Perrouin, and Pierre-Yves Schobbens. A vision to identify architectural smells in self-adaptive systems using behavioral maps. In **15th European Conference on Software Architecture (ECSA 2021)**, page 1. CEUR Workshop Proceedings, 2021.

[36] Edilton Lima dos Santos, Pierre-Yves Schobbens, and Gilles Perrouin. Featured scents: Towards assessing architectural smells for self-adaptive systems at runtime. In **19th International Conference on Software Architecture**, pages 71–74. IEEE, 2022.

[37] Erick Barros dos Santos, Rossana MC Andrade, and Ismayle de Sousa Santos. Runtime testing of context-aware variability in adaptive systems. **Information and Software Technology**, 131:106482, 2021.

[38] Benedikt Eberhardinger, Axel Habermaier, and Wolfgang Reif. Toward adaptive, self-aware test automation. In **2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)**, pages 34–37. IEEE, 2017.

[39] Thaddeus Eze, Richard J Anthony, Chris Walshaw, and Alan Soper. The challenge of validation for autonomic and self-managing systems. In **Proceedings of the seventh international conference on autonomic and autonomous systems**, pages 128–133, 2011.

[40] Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In **Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems**, pages 3–10, 2015.

[41] Martin Folwer. Refactoring: Improving the design of existing programs. **Google Scholar Google Scholar Digital Library Digital Library**, 1999.

[42] Francesca Arcelli Fontana, Paris Avgeriou, Ilaria Pigazzini, and Riccardo Roveda. A study on architectural smells prediction. In **2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**, pages 333–337. IEEE, 2019.

[43] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. Arcan: A tool for architectural smells detection. In **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**, pages 282–285. IEEE, 2017.

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

[45] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering**, pages 416–419, 2011.

[46] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**, pages 80–89. IEEE, 2011.

[47] Erik M Fredericks and Betty HC Cheng. Automated generation of adaptive test plans for self-adaptive systems. In **2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**, pages 157–167. IEEE, 2015.

[48] Erik M Fredericks, Andres J Ramirez, and Betty HC Cheng. Towards run-time testing of dynamic adaptive systems. In **2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**, pages 169–174. IEEE, 2013.

[49] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. **IBM systems Journal**, 42(1):5–18, 2003.

[50] SG Ganesh, Tushar Sharma, and Girish Suryanarayana. Towards a principle-based classification of structural design smells. **J. Object Technol.**, 12(2):1–1, 2013.

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

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

[53] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In **Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation**, pages 213–223, 2005.

[54] Arnaud Gotlieb and Bernard Botella. Automated metamorphic testing. In **Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003**, pages 34–40. IEEE, 2003.

[55] D Grove and C Chambers. Ibm research report an assessment of call graph construction algorithms. 2000.

[56] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In **Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, pages 108–124, 1997.

[57] Gabriela Guedes, Carla Silva, Monique Soares, and Jaelson Castro. Variability management in dynamic software product lines: A systematic mapping. In **2015 IX Brazilian Symposium on Components, Architectures and Reuse Software**, pages 90–99. IEEE, 2015.

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

[59] Richard Hamlet. Random testing. **Encyclopedia of software Engineering**, 2:971–978, 1994.

[60] Joachim Hänsel, Thomas Vogel, and Holger Giese. A testing scheme for self-adaptive software systems with architectural runtime models. In **2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops**, pages 134–139. IEEE, 2015.

[61] Salim Hariri, Bithika Khargharia, Houping Chen, Jingmei Yang, Yeliang Zhang, Manish Parashar, and Hua Liu. The autonomic computing paradigm. **Cluster Computing**, 9:5–17, 2006.

[62] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. **IEEE Transactions on Software Engineering**, 30(1):3–16, 2004.

[63] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. **IEEE Transactions on Software Engineering**, 36(2):226–247, 2009.

[64] Henner Heck, Stefan Rudolph, Christian Gruhl, Arno Wacker, Jörg Hähner, Bernhard Sick, and Sven Tomforde. Towards autonomous self-tests at runtime. In **2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)**, pages 98–99. IEEE, 2016.

[65] IBM. An architectural blueprint for autonomic computing. **IBM White Paper**, 31:1–6, 2006.

[66] IBM. **The T. J. Watson Libraries for Analysis (WALA)**. IBM, 2020.

[67] Muhammad Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. Deltaiot: A self-adaptive internet of things exemplar. In **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**, pages 76–82. IEEE, 2017.

[68] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. **IEEE transactions on software engineering**, 37(5):649–678, 2010.

[69] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU-SEI, 1990.

[70] Manpreet Kaur and Rupinder Singh. A review of software testing techniques. **International Journal of Electronic and Electrical Engineering**, 7(5):463–474, 2014.

[71] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. **Computer**, 36(1):41–50, 2003.

[72] Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. An architecture framework for experimentations with self-adaptive cyber-physical systems. In **2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**, pages 93–96. IEEE, 2015.

[73] Konveyor. **TackleTest: Automated Unit and UI Test Generation**. Konveyor, 2023.

[74] Christian Krupitzer, Martin Pfannemüller, Vincent Voss, and Christian Becker. Comparison of approaches for developing self-adaptive systems. 2018.

[75] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. **Pervasive and Mobile Computing**, 17:184–206, 2015.

[76] D Richard Kuhn, Raghu N Kacker, and Yu Lei. **Introduction to combinatorial testing**. CRC press, 2013.

[77] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Behavioral maps: Identifying architectural smells in self-adaptive systems at runtime. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, **Software Architecture**, pages 159–180, Cham, 2022. Springer International Publishing.

[78] Edilton Lima dos Santos, Pierre-Yves Schobbens, Ivan Machado, and Gilles Perrouin. Architectural bad smells for self-adaptive systems: Go runtime! In **Proceedings of the 17th International Working Conference on Variability**

Modelling of Software-Intensive Systems, VaMoS '23, page 85–87, New York, NY, USA, 2023. Association for Computing Machinery.

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

[80] Heng Lu. A context-oriented framework for software testing in pervasive environment. In **29th International Conference on Software Engineering (ICSE'07 Companion)**, pages 77–78. IEEE, 2007.

[81] Chu Luo, Miikka Kuutila, Simon Klakegg, Denzil Ferreira, Huber Flores, Jorge Goncalves, Mika Mäntylä, and Vassilis Kostakos. Testaware: a laboratory-oriented testing tool for mobile context-aware applications. **Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies**, 1(3):1–29, 2017.

[82] Lu Luo. Software testing techniques. **Institute for software research international Carnegie mellon university Pittsburgh, PA**, 15232(1-19):19, 2001.

[83] Frank D Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. **Expert Systems with Applications**, 40(18):7267–7279, 2013.

[84] Phil McMinn. Search-based software testing: Past, present and future. In **2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**, pages 153–163. IEEE, 2011.

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

[86] Zoltán Micskei, Zoltán Szatmári, János Oláh, and István Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In **Agent and Multi-Agent Systems. Technologies and Applications: 6th KES International Conference, KES-AMSTA 2012, Dubrovnik, Croatia, June 25-27, 2012. Proceedings 6**, pages 504–513. Springer, 2012.

[87] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In **Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation**, pages 305–315, 2010.

[88] Benoît Montagu and Thomas Jensen. Trace-based control-flow analysis. In **Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation**, pages 482–496, 2021.

[89] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run. time to support dynamic adaptation. **Computer**, 42(10):44–51, 2009.

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

[91] Freddy Munoz and Benoit Baudry. **Artificial table testing dynamically adaptive systems**. PhD thesis, INRIA, 2009.

[92] Masuma Naqvi. Claims and supporting evidence for self-adaptive systems–a literature review. 2012.

[93] Neo4j. **Neo4j APOC Library**. Neo4j, 2020.

[94] Changhai Nie and Hareton Leung. A survey of combinatorial testing. **ACM Computing Surveys (CSUR)**, 43(2):1–29, 2011.

[95] Dirk Niebuhr and Andreas Rausch. A concept for dynamic wiring of components: correctness in dynamic adaptive systems. In **Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering**, pages 101–102, 2007.

[96] Dirk Niebuhr, Andreas Rausch, Cornel Klein, Jürgen Reichmann, and Reiner Schmid. Achieving dependable component bindings in dynamic adaptive systems - a runtime testing approach. In **2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems**, pages 186–197. IEEE, 2009.

[97] Flavio Oquendo, Jair Leite, and Thais Batista. **Software Architecture in Action**. Springer, 2016.

[98] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In **Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion**, pages 815–816, 2007.

[99] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In **29th International Conference on Software Engineering (ICSE'07)**, pages 75–84. IEEE, 2007.

[100] Lin Padgham, Zhiyong Zhang, John Thangarajah, and Tim Miller. Model-based test oracle generation for automated unit testing of agent systems. **IEEE Transactions on Software Engineering**, 39(9):1230–1244, 2013.

[101] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. 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), pages 33–42. IEEE, 2012.

[102] Gilles Perrouin, Mathieu Acher, Jean-Marc Davril, Axel Legay, and Patrick Heymans. A complexity tale: Web configurators. In **2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE)**, pages 28–31. IEEE, 2016.

[103] Klaus Pohl, Gunter Bockle, and Frank J. vander Linden. **Software Product Line Engineering: Foundations, Principles and Techniques**. Springer-Verlag New York, Inc., 2005.

[104] Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Thomas Schlegel, and Uwe Aßmann. A combined simulation and test case generation strategy for self-adaptive systems. **Journal On Advances in Software**, 7(3&4):686–696, 2014.

[105] Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, and Christian Schuhmayer. Evolution in dynamic software product lines. **Journal of software: evolution and process**, 33(2):e2293, 2021.

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

[107] Andres J Ramirez and Betty HC Cheng. Design patterns for developing dynamically adaptive systems. In **Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems**, pages 49–58. ACM, 2010.

[108] André Reichstaller and Alexander Knapp. Risk-based testing of self-adaptive systems using run-time predictions. In **2018 IEEE 12th international conference on self-adaptive and self-organizing systems (SASO)**, pages 80–89. IEEE, 2018.

[109] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. **ACM transactions on autonomous and adaptive systems (TAAS)**, 4(2):1–42, 2009.

[110] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. In **Proceedings of the 1st International Workshop on Software Refactoring**, pages 1–4, 2016.

[111] Edilton Santos and Ivan Machado. Towards an architecture model for dynamic software product lines engineering. In **IEEE International Conference on Information Reuse and Integration (IRI)**, pages 31–38. IEEE, 2018.

[112] Edilton Lima dos Santos, Gilles Perrouin, and Pierre-Yves Schobbens. Stars: software technology for adaptable and reusable systems phd research project.

In **Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems**, pages 1–2, 2020.

[113] Sergio Segura, Amador Durán, Ana B Sánchez, Daniel Le Berre, Emmanuel Lonca, and Antonio Ruiz-Cortés. Automated metamorphic testing of variability analysis tools. **Software Testing, Verification and Reliability**, 25(2):138–163, 2015.

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

[115] Liwei Shen, Xin Peng, Jindu Liu, and Wenyun Zhao. **Towards Feature-Oriented Variability Reconfiguration in Dynamic Software Product Lines**, pages 52–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[116] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. A systematic review on search based mutation testing. **Information and Software Technology**, 81:19–35, 2017.

[117] Bento R Siqueira, Fabiano C Ferrari, Kathiani E Souza, Valter V Camargo, and Rogério de Lemos. Testing of adaptive and context-aware systems: approaches and challenges. **Software Testing, Verification and Reliability**, 31(7):e1772, 2021.

[118] Bento Rafael Siqueira, Fabiano Cutigi Ferrari, Marcel Akira Serikawa, Ricardo Menotti, and Valter Vieira de Camargo. Characterisation of challenges for testing of adaptive systems. In **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing**, pages 1–10, 2016.

[119] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana 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**, pages 59–66, 2018.

[120] Iuri Santos Souza, Ivan Machado, Carolyn Seaman, Gecynalda Gomes, Christina Chavez, Eduardo Santana de Almeida, and Paulo Masiero. Investigating variability-aware smells in spls: An exploratory study. In **Proceedings of the XXXIII Brazilian Symposium on Software Engineering**, pages 367–376, 2019.

[121] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. **Refactoring for software design smells: managing technical debt**. Morgan Kaufmann, 2014.

[122] Gabriel Tamura, Norha M Villegas, Hausi A Müller, João Pedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, et al. Towards practical runtime verification and validation of self-adaptive software systems. In **Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers**, pages 108–132. Springer, 2013.

[123] TH Tse and Stephen S Yau. Testing context-sensitive middleware-based software applications. In **Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.**, pages 458–466. IEEE, 2004.

[124] Rachel Tzoref-Brill, Saurabh Sinha, Antonio Abu Nassar, Victoria Goldin, and Haim Kermany. Tackletest: A tool for amplifying test generation via type-based combinatorial coverage. In **2022 IEEE Conference on Software Testing, Verification and Validation (ICST)**, pages 444–455. IEEE, 2022.

[125] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In **Proceedings Working IEEE/IFIP Conference on Software Architecture**, pages 45–54. IEEE, 2001.

[126] Emil Vassev, Mike Hinchey, and Paddy Nixon. Automated test case generation of self-managing policies for nasa prototype missions developed with assl. In **2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering**, pages 3–8. IEEE, 2010.

[127] NM Villegas, G Tamura, and HA Müller. Architecting software systems for runtime self-adaptation: Concepts, models, and challenges. In **Managing Trade-Offs in Adaptable Software Architectures**, pages 17–43. Elsevier, 2017.

[128] Thomas Vogel. 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**, pages 101–107, 2018.

[129] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. **Structured testing: A testing methodology using the cyclomatic complexity metric**, volume 500. US Department of Commerce, Technology Administration, National Institute of . . . , 1996.

[130] Kristopher Welsh and Pete Sawyer. Managing testing complexity in dynamically adaptive systems: A model-driven approach. In **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**, pages 290–298. IEEE, 2010.

[131] Danny Weyns. Towards an integrated approach for validating qualities of self-adaptive systems. In **Proceedings of the Ninth International Workshop on Dynamic Analysis**, pages 24–29, 2012.

[132] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges. **Chapter in Handbook of Software Engineering**, page 2, 2017.

[133] Danny Weyns and Jesper Andersson. On the challenges of self-adaptation in systems of systems. In **Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems**, pages 47–51, 2013.

[134] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M Göschka. On patterns for decentralized control in self-adaptive systems. In **Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers**, pages 76–107. Springer, 2013.

[135] W Eric Wong. **Mutation testing for the new century**, volume 24. Springer Science & Business Media, 2001.

[136] Franz Wotawa. Adaptive autonomous systems–from the system's architecture to testing. In **Leveraging Applications of Formal Methods, Verification, and Validation: International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers**, pages 76–90. Springer, 2012.

[137] Jochen Wuttke, Yuriy Brun, Alessandra Gorla, and Jonathan Ramaswamy. Traffic routing for evaluating self-adaptation. In **2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**, pages 27–32. IEEE, 2012.

[138] Chang Xu, Shing Chi Cheung, Xiaoxing Ma, Chun Cao, and Jian Lv. Detecting faults in context-aware adaptation. **Int. J. Softw. Informatics**, 7(1):85–111, 2013.

[139] Lian Yu, Wei Tek Tsai, Yanbing Jiang, and Jerry Gao. Generating test cases for context-aware applications using bigraphs. In **2014 Eighth International Conference on Software Security and Reliability (SERE)**, pages 137–146. IEEE, 2014.