Characterizing Architectural Drifts of Adaptive Systems

Daniel San Martín
DC-UFSCar
São Carlos, Brazil
daniel.santibanez@ufscar.br

Bento Siqueira
DC-UFSCar
São Carlos, Brazil
bento.siqueira@ufscar.br

Valter Vieira de Camargo
DC-UFSCar
São Carlos, Brazil
valtervcamargo@ufscar.br

Fabiano Ferrari
DC-UFSCar
São Carlos, Brazil
fabiano@dc.ufscar.br

***Abstract*—An adaptive system (AS) evaluates its own behavior and changes it when the evaluation indicates that the system is not accomplishing what it is intended to do, or when better functionality or performance is possible. MAPE-K is a reference model that prescribes the adaptation mechanism of ASs by means of high-level abstractions such as Monitors, Analyzers, Planners and Executors and the relationships among them. Since the abstractions and the relationships provided by MAPE-K are generic, other reference models were proposed focusing on providing lower level abstractions to support software engineers in a more suitable way. However, after the analysis of seven representative ASs, we realized the abstractions prescribed by the existing reference models are not properly implemented, thus leading to architectural drifts. Therefore, in this paper we characterized three of these drifts by describing them with a template and showing practical examples. The three architectural drifts of ASs are Scattered Reference Inputs, Mixed Executors and Effectors, and Obscure Alternatives. We expect that by identifying and characterizing these drifts, we can help software architects improve their design and, as a consequence, increase the reliability of this type of systems.**

***Index Terms*—architectural drifts, adaptive system, maintenance**

I. Introduction

An adaptive system (AS) evaluates its own behavior and changes it when the evaluation indicates it is not accomplishing the established goals, or when better functionality or performance is possible. Nowadays, this type of system is critical in many application domains due to their autonomous nature, and due to their efficiency to address complex situations [1].

It is recognized in the software engineering field the importance of structuring a system in such a way that abstractions and concerns become clear and evident in its architecture, since this can severally impact the maintenance activities [2]. MAPE-K is a reference model proposed by IBM [3] that provides the main abstractions for designing ASs. The goal is to motivate software engineers in structuring ASs in such a way that the abstractions become evident and manageable.

According to MAPE-K, ASs can be semantically seen as two subsystems: the Managed and the Managing Subsystems. The first one implements the main functionalities, while the second one implements the logic for adapting the Managed one. MAPE-K is originally presented with four abstractions: (*i*) Monitors sense the Managed Subsystem and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference; (*ii*) Analyzers compare event data against patterns in the knowledge base to diagnose symptoms and stores the symptoms for future reference in the knowledge base, (*iii*) Planners interpret the symptoms and devise a plan; and (*iv*) Executors execute the change in the Managed Subsystem through its effectors [4].

Besides the canonical abstractions prescribed by MAPE-K, there are other low level abstractions equally important for reaching good levels of maintainability. As these abstractions are not evident in MAPE-K, software architects are not aware of them and usually do not consider them when architecting the system [5]. Examples of these abstractions are Reference Input, Measured Output and Alternatives [6] [7]. Reference Inputs represent control objectives (values, thresholds, etc) that must be reached by the system. Measured Outputs consist of indicators of requirements convergence [6], [8]. Alternatives is the set of alternatives for performing a unique task [9].

After analyzing some representative ASs, we have observed that, very often, the implementation of the abstractions (as the MAPE-K as the lower level ones) are scattered and tangled throughout the system. Although there are frameworks, APIs and reference models to assist the development of ASs, most of times they do not guarantee the adherence of the implementation to a reference model [10], [11]. Therefore, the little knowledge about these abstractions and the nonexistence of guidelines about how to design them leads to *architectural drifts*, that occurs when the logical architecture (source code organization) of a system presents differences of the prescribed reference model [12] for that type of system.

In this paper we present the characterization of three drifts we believe being recurrent in ASs. By characterizing these drifts we are not only making evident existing problems; we are also promoting the importance of these abstractions. Our characterization scheme provides a name for the drift, presents the quality attributes impacted, lists the potential reasons for its emergence, explains ways of how to identify them.

We have analyzed seven representative ASs, where four of them were taken from SEAMS conference. We have observed that these drifts occur in all of them. The drifts characterized in this paper are named *Scattered Reference Inputs*, *Obscure Alternatives* and *Mixed Executors and Effectors*. Scattered Reference Inputs occurs when reference input of the managing subsystem are not declared in a unique abstraction. Obscure

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

SANER 2020, London, ON, Canada
Research Papers

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

Alternatives occurs when the alternatives of an AS are not evident. Finally, Mixed Executors and Effectors arises when there is not a clear distinction between executors and effectors. We opted for characterizing drifts involving lower level abstractions because these ones are much less studied and evident in the literature.

This paper shows that although MAPE-K reference model is a well known model for designing the architecture of an AS, it lacks of appropriate guidelines for developers when they need to implement key architectural low-level abstractions, emerging some problems as the drifts we characterized. Also, we present a methodology for discovering architectural drifts that can be used in other domains.

The main contributions of this paper are: *i)* providing a catalog that documents typical drifts of ASs; *ii)* Exploring some abstractions (and ways of implement them) of ASs that are not so well known as the canonical ones from MAPE-K. *iii)* delivering a standardized way of communicating problems when designing adaptive systems.

The work is organized as follows. Section II presents a brief background concepts. Section II-B describes our reference model. In Section III, the three drifts are characterized, and Section IV-C presents a practical example of them. Section V summarizes related work, and Section VI brings a discussion and the conclusion about the drifts.

II. Background

This section summarizes the basic concepts addressed in this paper.

A. Abstractions of Adaptive Systems Architectures

Reference models express a fundamental structural organization schema for software systems, providing a set of predefined abstractions, specifying their responsibilities, and including rules and guidelines for organizing the relationships between them [13]. Software engineers can decide to follow or not these models, or use specific ones according to their needs. Of course, the choice of not using a reference model could impact several quality attributes as the system evolves.

MAPE-K is a high level reference model that presents the main abstractions that exist in the architecture of ASs. It was introduced by Kephart and Chess [14] and IBM [3], being the first proposal of how to architect ASs taking into consideration Control Loops (CLs) [15]. The main abstractions proposed by MAPE-K are Monitors, Analyzers, Planners and Executors [4].

Figure 1 illustrates the schematic view of MAPE-K with its main abstractions for (*i*) monitoring data collected from sensors; (*ii*) analyzing the monitored data; (*iii*) creating plans for the adaptations, and (*iv*) executing the adaptation by means of actuators. All these steps (or actions) share a knowledge base about the system that is being adapted. In addition, an AS could implement more than one CL because each one of them could handle a specific adaptation goal.

Besides to the main abstractions, there are others that are not represented in the MAPE-K reference model, but have been

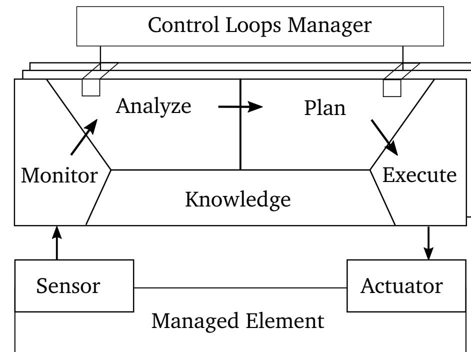

Fig. 1. MAPE-K schematic view (Adapted from [3])

reported by other researchers. These other are in low-level of abstraction and are important when software engineers need to design with more detail the adaptive system architecture [6].

Alternative represents a set of available options of adaptivity that an AS uses for changing the system behavior. For instance, in a self-healing system a failing service could be replaced by one that meets the same characteristics in order to complete successfully the assigned tasks without manual intervention. The main role of a decision is to choose, among a set of possible alternatives, the most suitable one according to the contextual situation [16].

Reference Inputs consists of the concrete and specific set of values, and corresponding types, that are used to specify the state to be achieved and maintained in the managed system by the adaptation mechanism, under changing conditions of system execution [6]. They could be implemented as single reference values, some form of contract, service level objectives, among other possibilities.

Measured Outputs consists of the set of values, and corresponding types, that are measured in the managed system. Naturally, as these measurements must be compared to the Reference Inputs to evaluate whether the desired state has been achieved, it should be possible to find relationships between these inputs and outputs [6].

Reference models, like MAPE-K, are built after a long domain analysis process, combined with knowledge of specialists in that domain. They prescribe certain abstractions that are not straightforward to identify. Most of the times, these abstractions are the fundamental points of maintenance and evolution steps in the system.

B. Architectural Drifts

Architectural drifts are implementation decisions that differ from an existing reference model or planned architecture. A drift occurs when there exist a reference model (or planned architecture) to be followed, but the current implementation of the system presents points which are different from those prescribed in the reference [17]. Architectural drifts not necessarily will result in dramatic problems; they may be there consciously. However, it is advisable to be aware of them, since evolution and maintenance activities can be compromised [12].

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

Some authors link architectural drifts with architectural smells. According to Zimmerman [18], an architectural smell could be defined as the observation or the suspect that something in architecture design and its implementation is no longer adequate under the actual requirements and current constraints for the system; these requirements and constraints may differ from the originally specified ones.

In fact, some architectural smells can cause architectural drifts. Examples are *Implicit Dependencies* and *Design Violations* [19]. The first one occurs when the implementation of a system contains dependencies that are not available in the architectural models; this may cause many liabilities. Developers could create a drift between the desired and the implemented architecture when they add implicit dependencies in the implementation without informing anyone else about these new dependencies. The second one occurs when there is a violation of design policies, such as using relaxed layering instead of strict layering. This should be avoided because different engineers in a project might resolve the same kind of problem with different solutions in an uncontrolled way; this reduces visibility and expressiveness.

To identify architectural drifts, software engineers use architectural conformance checking approaches. There are two main techniques for this: Reflexion Models, and Domain-specific Languages [20]. The Reflexion Models technique compares a high-level model manually created by the architect with a concrete model, extracted automatically from the source code [21]. On the other hand, the Domain-specific Languages technique focuses on architecture conformance provided by software architects to express in a customized syntax the constraints defined by the planned architecture [22].

While it is possible to specify the architecture of a system using a generic vocabulary, it is better to adopt a more specialized one when targeting architectures of a particular application domain [23]. Indeed, architectural patterns guide and focus software architects on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number [2]. Therefore, as architectural drifts affect quality attributes and differ from the architecture originally specified, we argue that architectural drifts of a specific domain should mention concrete terms to help software engineers in the activities that involve the identification and refactoring of them because generic descriptions are vague and do not help software engineers in their tasks.

III. Architectural Drifts of ASs

This section highlights the main results of our efforts on characterizing architectural drifts. Firstly, we provide a brief overview of the methodology we have followed and it consists in five steps. Secondly, we present the characterization of three architectural drifts we were able to find in the ASs.

In order to characterize them, we adopted a uniform template based on the work of Suryanarayana et al. [24].

A. Methodology

The methodology we have followed for characterizing the drifts is described in the following five steps.

1. Collecting Adaptive Systems: The goal of this step was to collect and create a database of ASs for further analysis. Therefore, we set up a software repository with representative ASs. We searched in open repositories such as GitHub and GitLab, research papers that mentioned the location of ASs repositories and gray literature. Most of the collected ASs came from SEAMS conference which is one of the major events in the self-adaptive systems area. We did a fork of all ASs to the research group repository. We filtered the repository according to two rules: clear documentation and all system resources should be available for execution. As a result, the systems shown in Table I were chosen for analysis.

2. Analysis of Literature: The goal of this step was to analyze research literature from a non-systematic review. Besides the main abstractions and their relationships, we also focused on identifying low-level abstractions and their relationships. Thus, we analyzed publications related to reference models of ASs. Particularly, we focused on studies that dealt with the MAPE-K reference model [4] [25] [6]. These studies made a deep analysis of the MAPE-K, identifying and describing other abstractions which enrich the MAPE-K reference model. This happens because MAPE-K delivers just the most canonical and higher level abstractions of ASs, hiding lower-level abstractions.

The most relevant studies we have identified were the ones by Villegas et al. [6], Weyns et al. [11], and IBM [3]. The first one provides the definitions of several abstractions found in ASs. The second one provides a reference model which covers a wide spectrum of MAPE-K perspectives. The third one is the architectural blueprint for autonomic computing. The abstractions are Reference Outputs, Reference Input and Alternatives.

3. Enriching MAPE-K with lower level abstractions: The goal of this step was to complement the MAPE-K reference model with lower level abstractions. After having identified the canonical and also some lower level abstractions of MAPE-K, we elaborated the reference model shown in Figure 2.

It is clear that the MAPE-K reference model is based on the design principle of separation of concerns. However, it just takes into account high-level abstractions and their relationships, leaving to software engineers the implementation of lower-level abstractions which could be implemented in a wrong way, without architectural quality. As a consequence, architectural drifts will affect the evolution of the AS, and hence hardening maintenance tasks.

Figure 2 presents the reference model mapped on the systems that we analyzed. Note that the reference model adds three new abstractions: Alternative, Measured Outputs and Reference Input to the schematic view of MAPE-K.

The CL abstraction contains the four MAPE-K abstractions as usually represented in the literature. The Planner abstraction is not mandatory (multiplicity 0) because several ASs do not require it to perform simple adaptations. Monitor, Analyzer, Planner and Executor abstractions can access the Knowledge abstraction to get some information for using it in their reasoning. Moreover, the Planner accesses the Alternative

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

TABLE I
EXAMPLES OF ADAPTIVE SYSTEMS CONTAINING THE PROPOSED DRIFTS

SYSTEM	DESCRIPTION	DRIFT	LOCATION	SOURCE	LOC
Zanshin-ATM	A system that provides basic banking services, along with managerial services, such as having a bank operator turn the ATM on/off.	SRI	Class: CashDispenser Attribute: cashOnHand Model: model.atm	https://github.com/Advanse-Lab/Zanshin	14.341
ASHYI-EDU	A system that provides virtual learning environment (VLE) with dynamic adaptive planning.	SRI	Class: BeanASHYI Method: isCambioContexto Variable: datos.getContexto()	https://github.com/Advanse-Lab/ASHYI	371.091
mRubis-self-healing*	A system that simulates a marketplace on which users sell or auction items.	SRI, OA	R.I. are declared in the CompArch model (ComponentState metaclass) - Alternatives are implemented in the Plan class	https://github.com/Advanse-Lab/mRUBiS	131.052
TAS*	A system that provides health support to chronic condition sufferers.	OA	Class:TASStart Method: initilize()	https://github.com/Advanse-Lab/TAS	147.072
PhoneAdapter	A mobile system that performs behavioral adaptations according to contextual data and rules.	MEE	Class: MyBroadcastReceiver Method: onReceive	https://github.com/Advanse-Lab/phoneadapter	11.638
AdaSim*	An open-source simulator for the automated traffic routing problem which allows for fast development of solutions to the problem.	MEE	Class: Vehicle Method: setStrategy	https://github.com/Advanse-Lab/adasim	11.111
SAVE*	A system that simulates the recording and manipulation of a video, using an mp4 stream and processing each of the original frames to obtain a compressed version of the stream.	MEE	Class: Encode Method: main	https://github.com/Advanse-Lab/save	670

* Systems taken from the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

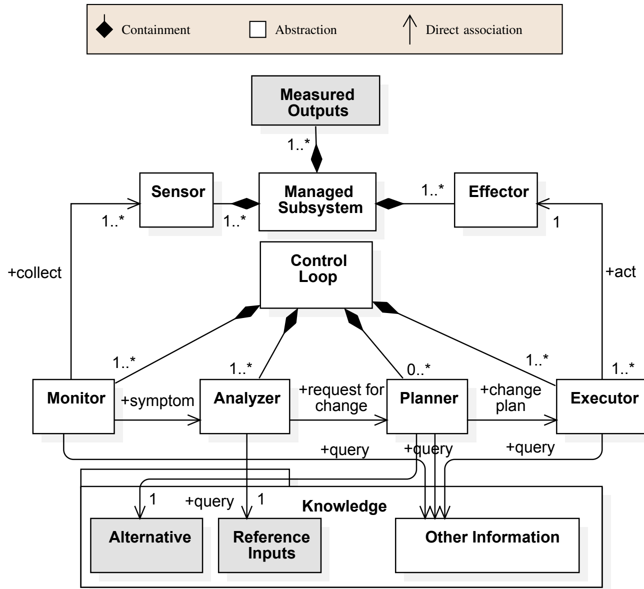

Fig. 2. MAPE-K with lower-level abstractions

abstraction in order to select the best option that fits with the adaptation goal. Analyzers should reason about whether or not there are symptoms of adaptation by taking into account Measured Outputs and Reference Inputs.

Executors must perform the realization of the action plan given by the Planner or Analyzer abstractions through one or more rules by means of the corresponding Effector [7]. This abstraction could also have some kind of intelligence. For instance, it could decide the priority of adaptive rules that will be executed on the managed subsystem and a scheduling schema when there is time constraints [26].

Effectors provide the necessary interfaces to modify the resources or artifacts of the managed system. According to the autonomic blueprint, an effector consists of one or both of the following: A collection of "set" operations that allow the state of the manageable resource to be changed in some way, and a collection of operations that are implemented by autonomic managers that allow the manageable resource to make requests from its manager [3].

If we look at the managed subsystem as a dependency graph, effectors should only have outgoing dependencies to internal components. That means that the efferent coupling is low, so it is easy to maintain them. On the other hand, if we look at the managing subsystem as a dependency graph, executors should only have incoming dependencies so they should be more stable because changing them could affect several planners or analyzers. The relation between executors and effectors must be surjective, that is, every executor corresponds to one effector.

We argue that two of the five abstractions (Alternatives and Reference Inputs) are less recognized in literature and, as a consequence, developers do not pay enough attention to them. It is important to make them visible because there is a great possibility that they receive maintenance tasks. For instance, a self-healing system could need to add new alternatives to manage uncertainties that software architects were not aware

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

in the original design. In the same way, changing or adding new Reference Inputs could be another recurring tasks when threshold values need to be adjusted in order to get a suitable adaptation.

4. Analysis of Systems: The goal of this step was to analyze the systems in two ways: a static analysis with automated tools and a manual analysis for searching the abstractions of the reference model of Figure 2 and mapping them in the ASs. The result of this step was a set of Architectural Drifts.

5. Simulating Maintenance Tasks: The goal of this step was to create several maintenance tasks for analyzing the impact of quality attributes when they are applied in the systems. We raised a list of maintenance tasks for each system such as adding, modifying and removing abstractions that are involved in the architectural drifts. As a result, we characterize the most relevant Architectural Drifts and present them in Subsection III-B. Table II presents the template used in this work to document the drifts.

TABLE II
ARCHITECTURAL DRIFT TEMPLATE

Template Element	Description
Name & description	An intuitive name and a concise description of the architectural drift.
Rationale	Reason/justification of why this is an anomaly in the context of Adaptive Systems.
Potential causes	List of typical reasons for the occurrence of the anomaly.
Impacted quality attributes	Quality attributes impacted negatively, such as modularity, reusability, analysability, modifiability and testability.
Affected architectural abstractions	Architectural abstractions of an adaptive system affected by the architectural anomaly.
Practical considerations	Sometimes, drifts are introduced intentionally either due to constraints (such as language or platform limitations) or to address a larger problem in the overall design.
Identification of the anomaly	How to identify the drift.
Instance of	Whether the anomaly is an instance of a generic one.

B. Three Common Architectural Drifts of ASs

In this section we present the three architectural drifts we have identified and characterized. It is important to emphasize that the focus of these drifts is on maintainability, i.e., the presence of them suggests the conduction of maintenance tasks can be painful. For this, we based our analysis on ISO 25010 standard (https://tinyurl.com/y6e6wru2); modularity, reusability, analyzatility, modifiability and testability.

1) Scattered Reference Inputs (SRI): **Description:** This drift arises when Reference Inputs are not localized/stored in the Knowledge.

Rationale: The lack of a well-modularized module to store Reference Inputs, or their declarations scattered through several modules, makes Analyzers to access different modules, other than relying on a unique and consistent point (i.e. the Knowledge). Besides, the Reference Inputs end up declared in modules that already have other responsibilities. To be more precise, there are four problems:

- Increase on the efferent coupling of Analyzers: Analyzers need to have access to several modules that are not specific of storing the Reference Inputs, increasing the coupling of them with other modules;
- Violation of the Encapsulation Principle: When Reference Inputs are defined into other modules that are responsible for other abstractions, the abstractions are tangled, making these other modules too exposed because they have to be accessed by the Analyzers;
- Decrease of the cohesion of other abstractions: As the other abstractions become tangled with Reference Inputs, their level of cohesion decreases because of afferent coupling, which turns them abstractions with high degree of responsibility;
- Compromising of the reusability: The reusability is also severely impacted. This happens because the reuse of Reference Inputs as a module in other contexts requires the modification of all modules where they are located.

Potential Causes:

- Inadequate architecture analysis: Software architects did not consider the definition of a well defined module to store the reference inputs at the beginning of the architecture design due to tight deadlines, resource constraints or minor performance gains.
- Lack of refactoring: At the beginning of a project, few Reference Inputs were declared, but as software evolves, the number of Reference Inputs could increase so it may be needed to refactor them into a new abstraction. The lack of refactoring may lead to a Scattered Reference Input drift.

Impacted Quality Attributes: Typical maintenance activities are: (*i*) adding Reference Inputs when an AS needs to achieve new adaptation goal; and (*ii*) removing Reference Inputs when some adaptation goals are not desirable anymore. Therefore, the following attributes may be impacted:

- Modularity: The modularity of this abstraction is compromised because its implementation is spread through other modules such as monitors, analyzers, planners and executors, increasing the likelihood of introducing side effects during maintenance tasks.
- Reusability: The reusability of this abstraction is compromised since it is difficult to reuse the Reference Inputs in other contexts, considering that they are not encapsulated in a unique module with a well defined interface.
- Analyzability: The analyzability of this abstraction is affected due to the nature of the drift, more points of failures could be generated by adding or removing Reference Inputs and, as a consequence, the the understanding decreases.
- Modifiability: The modifiabilty of the Managing Subsystem is impacted because changes will take longer, since the time to find the Reference Inputs is higher.

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

Maintenance also becomes risky because it could affect other modules.
- Testability: The testability of this abstraction is impacted because it will be necessary to create different and totally independent test cases, since the Reference Inputs are spread thought other modules. If Reference Inputs were declared in a dedicated module, the test cases will be simpler.

Affected Architectural Abstractions: As the Reference Inputs are scattered, the Analyzer could depend on several modules in order to query them for making an adaptation decision.
Practical Considerations: When there are few reference values to be queried by the analyzer, it may be convenient to store the values in the Analyzer but, in a certain extent, if more values are queried, it is desirable to create an abstraction to store all of them.
Identification of the drift: Once the analyzer is identified, software engineers have to check the rules that triggers an adaptation. These rules are comparisons composed by Reference Inputs and measured outputs. The engineer should analyze if the declaration of all Reference Inputs are stored in a single abstraction, or if they are scattered in several modules.
Instance of: Broken Modularization: When data/methods that should have been localized into a single abstraction are separated and spread across multiple abstractions [27].

2) Obscure Alternatives (OA): **Description:** This drift arises when the set of alternatives of an AS is not implemented as a first class entity.
Rationale: When the Alternative abstraction is not evident in the design of the architecture of the managing subsystem, it means that it was implemented tangled with other abstraction. Consequently, it makes difficult to understand the mechanism of adaptation which may imply raise of maintenance costs. In the MAPE-K reference model, the Analyzer accesses Alternative abstraction for using an adaptation option to reach and maintain a quality level of response of the system according to the environment state. Thus, it is likely that Alternative and Analyzer were implemented as a unique abstraction without an evident difference between them. To be more precise, there are two problems:

- Increasing the size of Analyzers: The main rule of modularization is to decompose abstractions to manageable size. When this rule is violated, it becomes difficult to understand and maintain these modules.
- Increasing the coupling between abstractions: Changing to another approach of adaptation could be cumbersome and risky because that could imply major changes in the logic of the Analyzer in order to support new approaches.

Potential Causes:

- Centralized control: A centralized implementation could be better managed; however, the abstraction will become responsible for a large amount of work and, as a consequence, it could have several points of failures.
- Grouping all functionality together: Often, inexperienced developers tend to group together and provide all related functionality in a single module, without understanding how the Single Responsibility Principle (SRP) should be properly applied.

Impacted Quality Attributes: Typical maintenance activities are: (*ii*) adding a new alternative when an AS needs to achieve new adaptation goal; and (*ii*) modifying an alternative when the purpose of an adaptation needs to change. Therefore, the following attributes may be impacted:

- Modularity: The modularity of this abstraction could break because as alternatives are obscure in the software architecture, adding a new alternative may confuse software maintainers, who would implement it in other abstraction.
- Analyzability: The analyzability of this abstraction is affected because it has noise that makes hard to discern on each alternative, as well as on their purpose regarding the possible adaptations. For instance, when software maintainers need to modify an adaptation alternative, the understanding degree becomes low.
- Modifiability: The modifiability of this abstraction is affected because as the alternatives of adaptation are overlapped, a modification on one alternative may affect others.

Affected Architectural Abstractions: Regarding the Analyzer, strong coupling with Alternative abstraction could limit its capacity of evolving when maintenance or evolution tasks must be performed. Regarding the Alternatives abstraction, adding or removing new alternatives for adapting the managed subsystem could be error-prone tasks since their implementation penetrates the several parts of the analyzer.
Practical Considerations: As we stated before, a centralized control could facilitate the management of the analyzer. However, as long as it grows, there is a trade-off between modularity and size.
Identification of the drift: Software engineers need to identify the Analyzer and the Alternative abstractions. If just the Analyzer abstraction is identified in the current architecture of an AS, then it is likely that the set of alternatives has strongly coupling with the Analyzer.
Instance of: Insufficient Modularization: This drift arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both [27].

3) Mixed Executors and Effectors (MEE): **Description:** This drift occurs when Executors and Effectors are not evident in the architecture of the AS.
Rationale: Executors and Effectors are two abstractions intrinsically connected because the first ones perform structural or behavioral changes on the managed subsystem by means of the second ones. According to MAPE-K, Effectors are implemented in the managed subsystem as touch points for Executors, and the latter are implemented in the managing

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

subsystem. However, it is very common to find the implementation of these abstractions in ASs in a mixed way, without a clear distinction between them [28], thus making difficult the comprehension of the adaptation mechanism. In such cases, it is not clear which parts conform the managed and the managing subsystems. As a result, this could lead to error-prone maintenance activities [29].

Potential Causes:

- Lack of implementation guidelines for the MAPE-K reference model: Despite the fact that MAPE-K shows a scheme of how the main abstractions must communicate among them, it does not provide implementation guidelines; so, very often software engineers end up developing Executors and Effectors in an obscure way, mixing them.
- Lack of knowledge of structural and behavioral properties of each abstraction: Although the main abstractions are well depicted by the MAPE-K reference model, it is possible that software engineers misunderstand the real purpose of each abstraction.

Impacted Quality Attributes: Typical Maintenance Activities are: adding or removing executors or effectors. Therefore, the following attributes may be impacted:

- Modularity: These abstractions are affected because as the AS evolves, the separation of architectural abstractions are getting unclear because the code of Executors becomes tangled with the code of Effectors.
- Reusability: These abstractions are affected because it is not possible to identify the abstractions involved in the drift unequivocally.
- Modifiability: These abstractions are affected because as the functions are overlapped, a single change may affect executors and effectors.
- Testability: These abstractions are affected because the Managed Subsystem cannot be isolated from the Managing Subsystem due to the coupling between them. Hence, testing activities becomes challenging.

Affected Architectural Abstractions: Executors and Effectors, because they cannot be differentiated from each other.

Practical Considerations: If an application contains a large number of decision points encoded in different autonomic elements scattered through the application code, externalizing the self-managing logic away from application objects will relief the burden for future maintainers. There should be no practical considerations for the implementation of this drift.

Identification of the drift: Software engineers must identify the touch points responsible for changing the managing subsystem in order to understand the logical separation of the two subsystems. Once these touch points have been identified, they need to check the degree of coupling of the involved abstractions.

Instance of: The Grand Old Duke of York: This drift occurs when developers could not identify the significance of good abstractions and ignore them even after being suggested by some team members [27].

IV. EXAMPLES

This section presents some Adaptive Systems that contain the drifts we have characterized. Table I lists seven systems and the drifts they contain. The first column shows the name of the system, the second shows a brief description, the third shows the drifts that were identified, the fourth shows the source code artifacts where the drifts are located, the fifth shows the repository and the sixth shows the lines of code.

We can see that three of the ASs present the Scattered Reference Input Declarations drift (ASHYI, Zanshin, TAS), three of them present the the Mixed Executors and Effectors (PhoneAdapter, AdaSim, SAVE) and two of them present the Obscure Alternatives drift (TAS, mRubis).

With the analysis of the source code of these systems, it is clear that developers do not follow naming conventions given by the MAPE-K reference model, so it is not trivial to understand which part of the system conforms with monitors, analyzers, planners and executors. Moreover, in many cases the adaptation mechanism is tangled with the system logic, so it becomes difficult isolate each MAPE-K abstraction.

To present a more detailed example, we have selected two representative systems: Zanshin-ATM and PhoneAdapter. Zanshin-ATM suffers with the presence of the SRI drift, and the PhoneAdapter suffers with the presence of the MEE drift. Subsection IV-A addresses the Zanshin-ATM system, and Subsection IV-C addresses the PhoneAdapter system.

A. Scattered Reference Inputs Example

To exemplify the SRI drift, we use the Zanshin-ATM system. Zanshin [30] is a framework for developing adaptive software and the ATM system uses the Zanshin framework to make itself adaptive. The main goal of ATM is to provide basic banking and managerial services.

Two adaptations scenarios are implemented in this system:

1. Recovering from the malfunction of the ATM printer: In this first scenario, after ATM terminal performing a transaction, it must print a receipt for the customer. If the printer fails, the adaptation strategy is to retry twice the printing operation. If it still fails, then it abort the printing operation.
2. Managing the shortage of cash: In this second scenario, the ATM terminal always must check if it has enough banknotes when a withdraw operation is performed by customers. In case the banknotes available are not enough to serve the customer's request, this task fails and the whole operation is canceled. Therefore, the adaptive system contains preventive actions such as augmenting the number of operators to refill the ATM with cash if dispensers become empty.

In the shortage of cash scenario, the Reference Input of interest is the "total amount of cash available on the ATM dispenser". The measured value corresponds to the amount of money that the customer needs to withdraw from the ATM. Therefore, an adaptation is triggered when the measured value is greater than the Reference Input. Listing 1 presents a snippet of the *CashDispenser* class, where this rule is implemented.

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

```
1  public class CashDispenser {
2
3   private Money cashOnHand;
4  ...
5   public void setInitialCash(Money
        initialCash) {
6     cashOnHand = initialCash;
7   }
8
9   public boolean checkCashOnHand(Money
        amount) {
10    return amount.lessEqual(cashOnHand);
11  }
12 ...
13 }
```

Listing 1. Snipper of CashDispenser class [30]

The *CashDispenser* class implements four methods (two of them are shown in Listing 1: *setInitialCash* and *checkCashOnHand*). The first one (line 5) sets the amount of cash initially on hand, and the second one (line 9) checks if there is enough cash on hand to satisfy a customer's request. The *cahshOnHand* class attribute (line 3) corresponds to the Reference Input, and the variable *amount* of type *Money* (line 9) corresponds to Measured Output. The business rule in line 10 returns true if the dispenser has an amount of cash greater or equal than the customer needs in a withdraw operation. Otherwise, it returns false.

In the Printer Malfunction scenario, the Reference Input of interest is a fixed number of retries. In this case, it was declared in a goal model that includes several adaptation requirements (AR), each of them being composed by the adaptation strategy, the applicability condition, and the resolution condition.

Listing 2 presents a snippet of the goal model in an XMI file. Line 4 specifies the strategy for this scenario, which is retrying the operation every 5 seconds, and the condition is to retry at most twice, as defined in line 8.

```
1   ..
2    <children xsi:type="atm:AR3" ... >
3       <condition xsi:type="..
            SimpleResolutionCondition"/>
4       <strategies
5            xsi:type="..RetryStrategy"
6            time="5000">
7         <condition
8            xsi:type="..
                MaxExecutionsPerSession
                ApplicabilityCondition"
                maxExecutions="2"/>
9       </strategies>
10    </children>
11  ..
```

Listing 2. Snippet of goal model for ATM system [30]

As we can see, these two Reference Inputs were declared in two different places of the system. This makes difficult to understand the mechanism of adaptation, and hence makes harder the maintenance activities involving these Reference Inputs. Adding new Reference Input could be confusing and, in this case, it could affect business rules of the ATM system or other abstractions of the MAPE-K.

A solution to this drift is the implementation of a class that declares all Reference Inputs with their getters and setters. Thus, analyzers can have access to Reference Inputs, and executors could update the values and adjust them whenever it is necessary. This solution is in conformance with the reference model of Figure 2, where Reference Inputs abstractions is clearly identifiable.

B. Obscure Alternatives

To exemplify Obscure Alternatives drift, we use the Tele-Assistance System (TAS) system [9]. TAS provides health support to chronic condition sufferers within the comfort of their homes. TAS uses a combination of sensors embedded in a wearable device and remote services from healthcare, pharmacy and emergency service providers.

TAS takes periodical measurements of the vital parameters of a patient and employs medical service for their analysis. The analysis results may trigger the invocation of a pharmacy service to deliver new medication to the patient, or to change the dose of medication, or the invocation of an alarm service leading, e.g. to an ambulance being dispatched to the patient [9]. This system is considered an AS with self-healing property. That means it has the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent a failure [15].

TAS implements its self-healing property by means of n-versions of its active medical services, so if one active service fail, then it can be replaced by a similar one. Figure 3 presents the *TASStart* class, which implements the *initializeTAS()* method. This method defines all services that will be available in the system. Once the system starts, all the defined services are loaded in a service cache class named *SDClass*.

Figure 3 also shows two more classes: *MainGui* and *ApplicationController*. The first one starts the application and the second one initializes graphical aspects of the system and service quality profiles (performance, costs, preferable service). *TASStart* also executes the system workflow, which runs several cycles according to a modifiable parameter.

Notice that services and its alternatives become obscure because: (*i*) the name of *TASStart* class does not reflect its purpose, nor does the method name *initializeTAS()*; (*ii*) services and their alternatives should be declared isolated from other concerns and in a recognizable class for facilitating maintenance. In this case, the *TASStart* class also implements the execution of the workflow.

C. Mixed Executors and Effectors Example

To exemplify the Mixed Executors and Effectors drift, we use PhoneAdapter [31]. This application uses contextual information to adapt a phone's configuration profile. These profiles are settings that determine a phone's behavior, such as display intensity, ring tone volume and vibration.

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

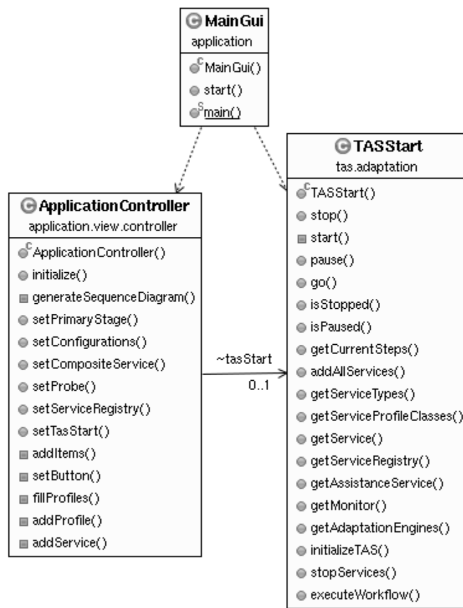

Fig. 3. Class TASStart

Instead of users selecting a profile manually, the application is driven by a set of adaptation rules and each one of them specifies a predicate whose satisfaction automatically triggers the activation of an associated profile. The selected profile prevails until a more suitable one is chosen through the triggering of other rules. Basically, the system is divided in two modules: The *ContextManager* class, and the *Adaptation-Manager* class. The former implements several sensors and monitors to capture context data that is broadcast by means of Android intent objects to all components of the application. The latter filters messages with the new context data to check whether or not the rules are satisfied, and performs changes in the mobile behavior. Listing 3 presents a snippet of the *AdaptationManager* class.

```
1  public class MyBroadcastReceiver {
2     public void onReceive(Context c, Intent
          i) {
3        if(volume>0){
4           mAudioManager.setRingerMode(
               AudioManager.
               RINGER_MODE_NORMAL);
5           mAudioManager.setStreamVolume(
               AudioManager.STREAM_RING,
               volume, AudioManager.
               FLAG_SHOW_UI);
6        }
7        if(vibration==1){
8           mAudioManager.setVibrateSetting(
               AudioManager.
               VIBRATE_TYPE_RINGER,
               AudioManager.
               VIBRATE_SETTING_ON);
9        }
10    }
11 }
```

Listing 3. Snippet of AdaptationManager class

Lines 5 and 9 show two adaptation rules. The first one modifies the volume of the ringtone, and the second one activates the vibration mode.

The *AdaptationManager* class has more than 1000 lines of code, so it is not a trivial task to understand which part of the code corresponds to executors and effectors. Moreover, as there is not a clear distinction between these two abstractions, the Android API code becomes tangled with the custom adaptation rules, and hence difficult to be maintained in case developers need to add or remove new touch points.

Figure 4 presents a possible design for separating effectors and executors by using interfaces. In this case one executor could affect one or more effectors and one effector corresponds to one executor. It is a modular solution that separates these two abstractions in order to identifies them clearly.

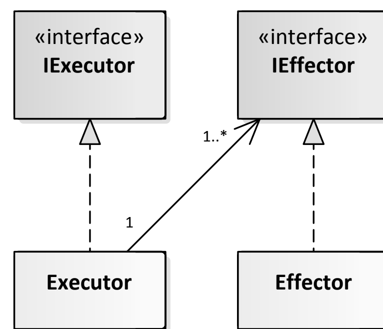

Fig. 4. Separation of executors and effectors

V. Related Work

To the best of our knowledge, there are no studies in the literature focusing architectural drifts in the context of Adaptive System development. Some studies [32], [33], [34] address architectural smells that occur in embedded systems, one study [23] addresses this type of smell in web applications based on MVC (Model-View-Controller), and [35] addresses in the context of software based on Product Lines.

Eliasson et al. [32] performed a secondary study and an exploratory study, obtaining a meta-analysis on the following subjects: Technical Architectural Debt and a characterization of an architectural smell. More specifically, the authors emphasized that there is usually no automatic transformation between architectural models and detailed design. Thus, inconsistencies between architecture models and detailed design may result in architectural technical debt. In the context of embedded systems for cars, the authors characterized the “Misplaced” architectural smell. This smell occurs when there is no specification detailing that a component should be deployed taking into account its communication with other components.

DeAndrade et al. [35] characterized architectural smells in the context of Product Lines through an exploratory study. The authors discussed two ways to identify smells (Resource Model and Variability Management). From this perspective, the authors described five types of architectural

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

smells (namely, Connector Envy; Dispersed Parasite Functionality; Ambiguous interfaces; Connector Adjacent External; and Appeal Concentration). By applying an exploratory study, the authors identified that two architectural smells (Envious Connector, and Ambiguous Interfaces) are introduced during the software development of Product Lines.

Vogelsang et al. [33] performed a study in the context of automotive software. They identified that components with implicit dependencies generate a high future cost in terms of maintenance. In the paper, the authors characterize the "Communal implicit components". These communal implicit components should be analyzed and refactored, due to the increasing maintenance effort to keep them. The authors proposed an automatic component refactoring approach in order to develop a dedicated component layer.

Elizondo et al. [23] characterized six architectural smells that occur in web applications based on the MVC (Model-View-Controller) architectural style. The authors proposed these architectural smells, highlighting when and where they occur. In general, these smells are based on communication between artifacts, *i.e.* Models and Views; Controllers and Models; Controllers and Views. For instance, when there is the necessity to use data in views, usually, developers use data that violate the architectural style.

Regarding Antonino et al.'s work [34], although the authors did not propose new architectural smells, they discussed profiles of software architects in the context of embedded systems - and how such profiles contribute to the occurrence of smells. In this context, the authors only described, for example, the architectural smell Extraneous Connector that consists of two components being deployed in the same container. During the description, the authors discussed how this smell occurs and a possible solution (*i.e.* the creation of a dedicated communication layer).

Comparing our work with other initiatives, we addressed the context of development of adaptive systems. Other studies we described in this section are in other contexts. More specifically, Antonino et al. [34] highlighted an architectural smell that is in the context of our "Scattered Reference Declarations" drift. Vogelang et. al [33] proposed the definition of communal implicit components that is in the same context of the implicit dependencies that we addressed in the "Mixed Executors and Effectors" drift. DeAndrade et al. [35] discussed a problem in the context of the "Obscured Alternatives" architectural drift that we characterized; differently, however, the authors addressed software based on product lines.

VI. Discussion and Conclusions

To the best of our knowledge, this is the first effort in characterizing architectural drifts specific of ASs. Our intention is that our catalog helps in disseminating good design practices regarding ASs. After having analyzed several representative ASs, we have noticed that most of them suffer from bad design practices that can impact maintainability.

As a result, we characterized three drifts. Scattered Reference Input expresses the lack of modularization of Reference Inputs because they are not declared in a Knowledge abstraction. Obscured Alternatives state that the set of alternatives of an AS is not implemented as a first class entities. The Mixed Executors and Effectors indicates that the touch point where the managing subsystem performs adaptation of the managed subsystem does not clearly identify the Executors and Effectors.

Another important aspect is that most of the research initiatives that deal with architectural drifts are domain-independent, i.e., they are applicable to several domains given that they use a specific vocabulary [36], [37], [22]. Nevertheless, while it is possible to specify the architecture of a system using a generic vocabulary, it is better to adopt a more specialized vocabulary when targeting architectures of a particular application domain. Indeed, nowadays researchers are focusing on characterizing drifts that are domain-dependent because it would aid more accurately software engineers when they need to identify drifts of a particular domain.

Also, we see that reference models that are too abstract do not take into account details that might be necessary in the system implementation due to their lack of information. As a consequence, developers could introduce architectural drifts in their designs. Given this scenario, we have augmented the traditional MAPE-K reference model in order to expose some lower-level abstractions.

Although MAPE-K is not mandatory for designing the AS architecture, this work serves as an indicator to software architects whether or not the system follows the MAPE-K model. Of course, the final decision is up to architects who will know the details, contexts and specifics of the system.

Characterizing architectural drifts is a subjective and difficult process for two main reasons. First, there is no standard methodology for finding architectural drifts in practice. Second, it is not straightforward to find a large set of ASs in existing repositories. Although we found systems being characterized as adaptive, most of them were developed for academic purposes. It would be desirable to collect more systems from industry for investigating whether there exist other type of drifts, as well as if there exist drifts that corroborate our catalog of drifts.

By making these drifts evident, we expect software architects can improve the design and implementation of ASs by taking into account these issues when creating new approaches and frameworks, in order to improve architecture quality attributes. Also, We are currently working on a tool that map AS abstractions in the source code, specifies a planned architecture in the ASs context, and performs an architecture conformance checking (ACC) to identify the drifts.

Acknowledgment

This work was funded by the CONICYT PFCHA/DOCTORADO BECAS CHILE/2016 - 72170024. Valter Camargo would like to thank FAPESP (process number 2016/03104−0). We also want to thank professor Jaime Pavlich Mariscal from PUJ University, Colombia for making available ASHYI.

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] R. Laddaga, P. Robertson, and H. Shrobe, "Introduction to self-adaptive software: Applications," in *Proceedings of the 2Nd International Conference on Self-adaptive Software: Applications*, ser. IWSAS'01. Springer-Verlag, 2003, pp. 1–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1754788.1754789

[2] L. Bass, P. Clements, and R. Kazman, "Software architecture in practice third edition written by len bass, paul clements, rick kazman," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 51–52, Feb. 2015, reviewer-Herzog, Jared. [Online]. Available: http://doi.acm.org/10.1145/2693208.2693252

[3] IBM, "An architectural blueprint for autonomic computing," IBM, Technical Report, Jun 2005.

[4] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*. Springer, 2009, pp. 48–70. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_3

[5] A. J. Ramirez and B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10. ACM, 2010, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/1808984.1808990

[6] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas, "A framework for evaluating quality-driven self-adaptive software systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. ACM, 2011, pp. 80–89. [Online]. Available: http://doi.acm.org/10.1145/1988008.1988020

[7] H. Arboleda, A. Paz, M. Jimenez, and G. Tamura, "Development and Instrumentation of a Framework for the Generation and Management of Self-Adaptive EnterpriseApplications," *Ingeniería y Universidad*, vol. 20, pp. 303–333, Dec. 2016. [Online]. Available: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0123-21262016000200004&nrm=iso

[8] S. Souza and P. J. Mylopoulos, "Requirements-based software system adaptation," Technical Report, 2012.

[9] D. Weyns and R. Calinescu, "Tele assistance: A self-adaptive service-based system exemplar," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Apr. 2015, pp. 88–92.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004. [Online]. Available: http://dx.doi.org/10.1109/MC.2004.175

[11] D. Weyns, S. Malek, and J. Andersson, "Forms: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 7, no. 1, pp. 8:1–8:61, May 2012. [Online]. Available: http://doi.acm.org/10.1145/2168260.2168268

[12] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992. [Online]. Available: http://doi.acm.org/10.1145/141874.141884

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[14] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[15] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[16] I. Abdennadher, I. Bouassida Rodriguez, and M. Jmaiel, "A design guideline for adaptation decisions in the autonomic loop," *Procedia Comput. Sci.*, vol. 112, no. C, pp. 270–277, Sep. 2017. [Online]. Available: https://doi.org/10.1016/j.procs.2017.08.242

[17] C. Maffort, M. Valente, M. Bigonha, N. Anquetil, and A. Hora, "Heuristics for discovering architectural violations," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, Oct. 2013, pp. 222–231.

[18] O. Zimmermann, "Architectural refactoring for the cloud: a decision-centric view on cloud migration," *Computing*, vol. 99, no. 2, pp. 129–145, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s00607-016-0520-y

[19] M. A. Babar, A. W. Brown, and I. Mistrik, *Agile Software Architecture: Aligning Agile Processes and Software Architectures*, 1st ed. Morgan Kaufmann Publishers Inc., 2013.

[20] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, Jul. 2009.

[21] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 4, pp. 18–28, Oct. 1995. [Online]. Available: http://doi.acm.org/10.1145/222132.222136

[22] A. d. S. Landi, F. Chagas, B. M. Santos, R. S. Costa, R. Durelli, R. Terra, and V. V. d. Camargo, "Supporting the specification and serialization of planned architectures in architecture-driven modernization context," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, Jul. 2017, pp. 327–336.

[23] P. Velasco-Elizondo, L. Castañeda-Calvillo, A. García-Fernandez, and S. Vazquez-Reyes, "Towards detecting mvc architectural smells," in *Trends and Applications in Software Engineering*, J. Mejia, M. Muñoz, A. Rocha, Y. Quiñonez, and J. Calvo-Manzano, Eds. Springer International Publishing, 2018, pp. 251–260.

[24] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann Publishers Inc., 2014.

[25] R. Hebig, H. Giese, and B. Becker, "Making control loops explicit when architecting self-adaptive systems," in *Proceedings of the Second International Workshop on Self-organizing Architectures*, ser. SOAR '10. New York, NY, USA: ACM, 2010, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/1809036.1809042

[26] A. Farahani, E. Nazemi, and G. Cabri, "A self-healing architecture based on rainbow for industrial usage," *Scalable Computing*, vol. 17, no. 4, pp. 351–368, 2016, cited By 1.

[27] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217303114

[28] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-aware adaptive applications: Fault patterns and their automated identification," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 644–661, Sep. 2010.

[29] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann, "Self-Adaptive Video Encoder: Comparison of Multiple Adaptation Strategies Made Simple," *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017*, pp. 123–128, 2017.

[30] G. Tallabaci and V. E. Silva Souza, "Engineering adaptation with zanshin: An experience report," in *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Apr. 2013, pp. 93–102.

[31] Y. Liu, "Phoneadapter mobile system," 2013. [Online]. Available: http://sccpu2.cse.ust.hk/afchecker/phoneadapter.html

[32] U. Eliasson, A. Martini, R. Kaufmann, and S. Odeh, "Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: A case study," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct. 2015, pp. 33–40.

[33] A. Vogelsang, H. Femmer, and M. Junker, "Characterizing implicit communal components as technical debt in automotive software systems," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Apr. 2016-04, pp. 31–40.

[34] P. Antonino, A. Morgenstern, and T. Kuhn, "Embedded-software architects: It's not only about the software," *IEEE Software*, vol. 33, no. 06, pp. 56–62, Nov. 2016.

[35] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion. ACM, 2014, pp. 12:1–12:6. [Online]. Available: http://doi.acm.org/10.1145/2578128.2578237

[36] D. Garlan, R. T. Monroe, and D. Wile, "Foundations of component-based systems," G. T. Leavens and M. Sitaraman, Eds. New York, NY, USA: Cambridge University Press, 2000, ch. Acme: Architectural Description of Component-based Systems, pp. 47–67. [Online]. Available: http://dl.acm.org/citation.cfm?id=336431.336437

[37] B. Tekinerdogan, "Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices," in *Software Quality Assurance*, I. Mistrik, R. Soley, N. Ali, J. Grundy, and B. Tekinerdogan, Eds. Boston: Morgan Kaufmann, 2016, pp. 221–236.

Authorized licensed use limited to: University of Exeter. Downloaded on May 05,2020 at 22:53:49 UTC from IEEE Xplore. Restrictions apply.