UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REMEDY: ARCHITECTURAL CONFORMANCE CHECKING FOR ADAPTIVE SYSTEMS

DANIEL GUSTAVO SAN MARTÍN SANTIBÁÑEZ

SUPERVISOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos – SP
November/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REMEDY: ARCHITECTURAL CONFORMANCE CHECKING FOR ADAPTIVE SYSTEMS

DANIEL GUSTAVO SAN MARTÍN SANTIBÁÑEZ

Thesis presented to the Computer Science Graduate Program of the Universidade Federal de São Carlos as part of the requisites to obtain the title of Doctor in Computer Science, concentration area: Software Engineering
Supervisor: Prof. Dr. Valter Vieira de Camargo

São Carlos – SP

November/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Daniel Gustavo San Martín Santibáñez, realizada em 12/11/2021.

Comissão Julgadora:

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. Ingrid Oliveira de Nunes (UFRGS)

Prof. Dr. Marco Túlio de Oliveira Valente (UFMG)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

ACKNOWLEDGEMENTS

First of all, I would like to thank Almighty God, the one, who bless me with the opportunity to pursue the journey towards a PhD.

I would like to express my sincere gratitude and very special thanks to Dr. Valter Vieira de Camargo for being no less than a perfect supervisor I could ever possibly imagine. In fact, Valter has always impressed me by exceeding all my expectations. He is a real gentleman and excellent teacher: very supportive, encouraging, understanding, kind, sincere, honest and generous. Throughout my PhD, Valter was not only supporting me to improve my knowledge and writing expertise, but also seemed to have always a better thought and plan for my exposure and future development. Apart from the research, he has given me great advices from his life experience that added a lot of positivity in my personality, which is priceless to me. I will always be very grateful to Valter for making my graduate experience very productive, enjoyable and memorable.

I wish to thank the Universidade Federal de São Carlos for their excellent resources. In particular, thanks to the Departamento de Computação (DC) for providing the research environment and resources, and for their friendly and supporting faculty and staff members including: Prof. Fabiano Ferrari, Prof. Auri Vincenzi, Prof. Daniel Lucrédio, Ivan Rogério da Silva, Nicanor José Costa and Darli José Morcelli. Moreover, I am very grateful and thankful to the Chilean National Agency for Research and Development (ANID) for granting me a very precious PhD scholarship. The opportunity of studying in the Universidade Federal de São Carlos will always be a remarkable experience in my life.

I especially thanks to my wife and daughter for their never-ending love, prayers, encouragements, support and showing confidence on me, without which it would be impossible for me to pursue my studies up to this stage. In addition, thanks to my father, mother and sister for their love, prayers and encouragements throughout my studies. Furthermore, I wish to thank all my colleagues and sincere friends for helping me with their valuable discussions and guidance during my PhD. Last but not least, I would like to thank Simon my lovely dog for accompany me in this journey and having great adventures in leisure times.

Simplicity does not precede complexity, but follows it.

Alan Perlis

RESUMO

Sistemas Adaptativos (SAs) avaliam seu próprio comportamento e são capazes de modificá-lo quando a avaliação indica que ele não está mais atingindo seus objetivos ou quando uma nova funcionalidade e desempenho estão disponíveis. Atualmente esse tipo de sistemas atuam em vários domínios devido à capacidade de lidar com as incertezas de contextos dinâmicos. Apesar de sua relevância, a qualidade da arquitetura encarregada da adaptação não tem sido levada em conta apropriadamente pelos engenheiros de software e em consequência essa falta de atenção pode afetar atributos de qualidade tais como a manutibilidade e evolução. Uma possível explicação desse problema é que os engenheiros de software não são conscientes de modelos de referências para projetar sistemas adaptativos ou seguem parcialmente e isto faz com que surjam desvios arquiteturais, que ocorrem quando a Arquitetura Atual (CA) do sistema desvia-se da Arquitetura Planejada (PA). Apesar que existem muitas abordagens para identificar desvios arquiteturais, eles utilizam um vocabulário genérico para especificar as regras estruturais e de comunicação que não refletem a semântica de um domínio em particular. Em domínios especializados, abstrações específicas são muito importantes e influenciam fortemente como os sistemas devem ser projetados. Portanto, para apoiar as tarefas de Checagem de Conformidade Arquitetural (CCA) em (SAs) propõe-se REMEDY, uma abordagem específica de domínio que permite a especificação da arquitetura adaptativa planejada baseada no modelo de referência MAPE-K, a recuperação da arquitetura adaptativa atual, um processo de checagem de conformidade e a visualização das arquiteturas. Para atingir os objetivos, foram investigadas as ocorrências dos desvios arquiteturais em ASs representativos. Baseados nas descobertas, propõe-se uma Linguagem Específica de Domínio (DSL) que implementa as abstrações canônicas prescritas pelo MAPE-K e outras que não são evidentes no modelo de referência. Além disso, a abordagem fornece regras de domínio pré-configuradas prontas para ser checadas pelos engenheiros de software sem ter que especificá-las manualmente. Foram realizadas duas avaliações: um experimento controlado para avaliar a DSL e uma avaliação qualitativa para avaliar a atividade de checagem arquitetural. A primeira avaliou a produtividade em termos de tempo e erros comparando a DSL com a DCL-KDM, uma CCA genérica. A segunda avaliou a acurácia do processo de checagem de conformidade em dois sistemas. Os resultados mostram que quando os engenheiros de software projetam a parte adaptativa de um SA com a DSL, a produtividade aumenta por sobre o uso de um enfoque genérico. Além disso, a abordagem proposta atingiu 90% de acurácia em termos de precisão e cobertura na hora de identificar os desvios arquiteturais.

Palavras-chave: sistema adaptativo, checagem de conformidade arquitetural, desvío arquitetural

ABSTRACT

Adaptive Systems (ASs) evaluate their own behavior and change it when the evaluation indicates it is not accomplishing the established goals, or when better functionality or performance is possible. Nowadays these kind of systems actuate in several domains due to the capability to deal with uncertainties that came from dynamic contexts. Despite the relevance they are acquiring and according to our findings, the quality of the adaptive architecture have not been properly taken into account by software engineers and consequently this lack of attention may affect quality attributes such as maintenance and evolution. A possible explanation is that software engineers are not aware of reference models to design ASs or they do not implement them completely which makes arise a type of architectural anomaly called architectural drift that occurs when the Current Architecture (CA) deviates from the Planned Architecture (PA). Although there are several approaches to identify architectural drifts they use a generic vocabulary to specify structural and communication rules that do not reflect the semantics of a particular domain. In more specialized domains, specific abstractions become important and strongly influence how systems are designed. In these cases, systems involve components with very specific and specialized roles that end up guiding how the the architecture must be designed. Therefore to support the Architecture Conformance Checking (ACC) process in ASs we propose REMEDY, a domain-specific approach that allows the specification of the planned adaptive architecture based on the *Monitor, Analyzer, Planner, Executor, Knowledge* (MAPE-K) reference model, the recovery of the current adaptive architecture, the conformance checking process and architecture visualizations. To achieve our goals, we investigate the occurrences of architectural drifts in representative ASs. Based on our findings we propose a Domain Specific Language (DSL) that implements the canonical abstractions prescribed by the MAPE-K and others that are not evident in the reference model. Also, our approach provides pre-configured domain rules ready to be checked by software engineers without specifying them from the scratch. We perform two types of evaluation: a controlled experiment to evaluate our DSL and a quality validation of our conformance checking process. In the first one we evaluate productivity in terms of time and errors by using our DSL against to DCL-KDM which is a generic ACC approach. In the second one we evaluate the accuracy of our checking process in two subject systems. The results show that when software engineers design the adaptive part of an AS with our DSL the productivity increased over a generic approach. Also, our approach reached above of 90% of accuracy in terms of precision and recall at time to identify the architectural drifts.

Keywords: adaptive system, architectural conformance checking, architectural drift

LIST OF FIGURES

List of Tables

LIST OF CODE AND ALGORITHMS

CONTENTS

Chapter 1

INTRODUCTION

1.1 Context

After years of maintenance, the architecture of software systems tend to deviate from the intended architecture that was initially planned. A possible way to check if the current implementation is becoming different from the Planned Architecture (PA) is by employing Architecture Conformance Checking (ACC) approaches, whose goal is to detect architectural drifts in existing systems. The motivation is obvious, if the system is deviating from its planned architecture, its quality attributes may not be met anymore.

ACC approaches normally involve the following steps: *i*) specify the PA by using architectural abstractions and making evident the hierarchical compositions and the communication rules among the architectural elements; *ii*) map source-code elements of the system to the architectural elements prescribed in the PA and *iii*) perform the checking/comparison between both (KNODEL; NAAB, 2016; PRUIJT; KÖPPE; WERF, et al., 2017).

Most of the existing ACC approaches are domain-independent, i.e., they deliver canonical and domain-independent architectural abstractions such as components, layers and modules (PASSOS et al., 2010; ROSIK et al., 2011; PRUIJT; KÖPPE; BRINKKEMPER, 2013). So, software architects must work with these abstractions along the whole process by mapping all the source code elements to them. However, in more specialized domains, specific abstractions become very important and strongly influence how systems are designed. In these cases, systems involve components with very specific and specialized roles that end up guiding how the the architecture must be designed.

Adaptive Systems (ASs) is an example of a domain that owns specific abstractions. ASs are able to autonomously cope with disturbances that can show up in the environment, within themselves and in their quality goals (LADDAGA; ROBERTSON; SHROBE, 2003; CHENG et al., 2009; WEYNS; IFTIKHAR; MALEK, et al., 2012). Although this term - *Adaptive System* - transmit the idea that the system is *entirely* adaptive, this is not true. The majority of systems termed as adaptive contain a big core to deal with functional requirements and one or more specific and smaller modules responsible for performing the adaptations. Therefore, the entire system have both a core that can be designed with layers, components and modules and adaptive parts that should be designed with ASs-specific abstractions, like control loops, alternatives, monitors and analyzers.

MAPE-K is a well known reference model for guiding the design of adaptive parts of ASs (HELLERSTEIN et al., 2004; IBM, 2005; BRUN et al., 2009) but it does not present nothing about how to structure the core of the system. It just prescribes the main abstractions that must be used for architecting the adaptive parts as well some implicit access rules between those abstractions (IBM, 2005). Moreover, although MAPE-K has become the de facto reference model for architecting ASs, frequently software engineers do not follow the principles of MAPE-K. That is, many existing ASs present architectural drifts when compared to MAPE-K (RAMIREZ; CHENG, 2010; SERIKAWA et al., 2016; SANTIBANEZ; SIQUEIRA; DE CAMARGO; FERRARI, 2020).

This thesis presents an ACC approach and tooling support for ASs. It is composed of four steps: *i*) the specification of the adaptive part of the AS; *ii*) the mappings of the source-code to be analyzed, with architectural abstractions of AS domain; *iii*) the conformity checking and *iv*) results and graphical visualization of drifts. All steps are novel in that they are tailored to the peculiarities of ACC applied as specific case for ASs.

1.2 Motivations

This thesis was elaborated over two main motivations: *i*) the need of a better comprehension and characterization of design problems that are specific of adaptive systems; *ii*) the absence of an architectural conformance checking approach dedicated to check the conformance between an AS and a AS-specific planned architecture, aligned with MAPE-K concepts.

A poor characterization of architectural design issues in ASs will led to implementations where key abstractions of the domain will end up scattered and tangled among them because of the lack of knowledge of software architects and as a consequence, maintenance and system evolvability will be affected negatively. The first motivation have been already noticed by our research group in a previous work (SERIKAWA et al., 2016) and it was the seed for this thesis. In that research, we found some recurring problems in the design/implementation of monitors in ASs, leading to maintenance, evolution and reuse issues. This evidence pushed us to think about other types of architectural issues that could be found in ASs and not just limited to architectural smells but also other types of architectural problems in other AS abstractions.

Moreover, we realized that the quality of ASs is a topic of interest by research community. Works such as Kaddoum et al. (2010), Ramirez and Cheng (2010), Weyns, Schmerl, et al. (2013), Raibulet, Arcelli Fontana, et al. (2017) among others present guidelines for the evaluation of adaptive systems. Nevertheless, although these approaches identify symptoms that could compromise quality, none of them propose solutions to effectively improve the quality of such as systems. Another aspect that we took in consideration was the fact that researchers in the field of ASs quality just have paid attention to the main MAPE-K abstractions without deepen in others which are equally relevant in ASs domain.

In order to corroborate the initial findings, we performed a systematic mapping to get the state of architectural anomalies in AS, a concept coined by Macia et al. (2012). The results suggested the existence of several types of anomalies in ASs, such as architectural smells, architectural technical debt, anti-patterns and architectural drifts that were reported in journals, conferences and workshops. After a rigorous analysis of the findings of our systematic mapping, we concluded that the majority of them can be classified as architectural drifts. Thus, the second motivation is a direct consequence from the first one - as the specific problems of ASs need a tooling support for being detected.

Indeed, this led us to characterize architectural drifts by analyzing several ASs and as a result we published a paper entitled as *"Characterizing Architectural Drifts of Adaptive Systems "* (SANTIBANEZ; SIQUEIRA; DE CAMARGO; FERRARI, 2020). Finally, in order to design our approach of ACC, we took as a reference the work published by Landi, Santibanez, Santos, Cunha, Durelli, and Camargo (2022) which describes a non-extensible approach, domain-independent for ACC in the Architecture-Driven Modernization (ADM) context. It uses the Knowledge Discovery Metamodel (KDM), a platform and language-independent metamodel, used to represent the PA, Current Architecture (CA) and to performed the conformance checking by comparing these two models. Although it is possible to check the architectural conformance of ASs by using ACC approaches that specify the PA with domain-independent abstractions (TERRA; VALENTE, 2009; MAFFORT et al., 2016), this kind of system and more specifically the part in charge of the adaptations has some particularities; well-known abstractions and domain rules (BARESI; GUINEA, 2012; VELASCO et al., 2018; KRUPITZER; TEMIZER, et al., 2020).

Moreover, nowadays there are many frameworks available for developing ASs that often provide a combination of tools, middleware, development process workflows and component libraries which prescribe the use of the MAPE-K reference architecture. (KRUPITZER; ROTH, et al., 2015). Therefore, the more developers adopt these kind of frameworks to develop ASs, the more implementations will follow the MAPE-K reference model. As systems aged, software architects will need to check architecture conformance in order to maintain the system quality attributes according to the needs. In that sense, our approach fit very well to that purposes because it provides the domain language and expected domain rules to make the ACC process more productive.

Thus we wanted to corroborate if a dedicated/domain specific approach could improve the productivity in this process. Thus our approach adds a set of abstractions (canonical and others that have not been explicitly mentioned in literature) and predefined rules (domain rules) to support the ACC process of ASs as well as a new DSL with custom validators and a new technique to identify and visualize the architectural drifts.

1.3 Goals

Our main goal is to provide a better characterization of architectural problems specific of ASs and also to deliver an approach to facilitate the identification of architectural drifts in ASs. We break the main goal down into the following four subgoals:

Goal 1: *Deliver an ACC approach to support the AS domain.* Although there are several approaches to perform the ACC they are generic, thus we need to design an approach that takes into account the particularities of the AS domain in order to improve the process of identifying architectural drifts and finally increase the productivity of software architects when perform this task. The approach should incorporate domain-specific knowledge rules in order to improve precision on the findings in less time.

Goal 2: *Conduct an empirical research study to catalog architectural drifts of ASs.* 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 we found that in some cases they are not considered at time of architecting the system. Therefore, by analyzing a set of representative ASs we can identify if researchers take into account those low level abstractions and possible architectural drifts that involved them.

Goal 3: *Characterize the state of art of architectural anomalies in AS domain.* The aim here is to unify the scattered knowledge about architectural anomalies found in literature regarding to ASs into one literary source to analyze what has been reported in this area, highlight useful findings, and reflect on what is available. To this purpose, we do not make distinctions between an *adaptive software* and *self-adaptive software*.

Goal 4: *Conduct experimental studies to evaluate the proposed approach.* There are two scopes that can be evaluated: *i*) perform a controlled experiment that provides evidence of improved productivity when software architects use domain-specific ACC approaches, particularly in the research area of AS and *ii*) evaluate the effectiveness of the approach in detecting architectural drifts of ASs.

1.4 Contributions

In summary, the contributions of this thesis are:

- Theoretical Contributions:

- An approach to perform ACC in KDM represented ASs;
- The identification and characterization of three architectural drifts for ASs: Mixed Executor & Effectors, Scattered Reference Inputs and Obscure Alternatives;
- A systematic mapping for identifying architectural anomalies in AS where research evidence of architectural problems were collected, collated and presented by following the B. A. Kitchenham et al. (2002) guidelines;
- A methodology for characterizing architectural drifts which consists in five steps: Collecting ASs; Analysis of Literature; Enriching MAPE-K with lower level abstractions; Analysis of Systems; Simulating Maintenance Tasks;
- Evidences through a controlled experiment showing that productivity of software architects improve when using a DSL of a specific domain rather than a general-purpose DSL;
- An architectural drift template to characterize architectural drifts of ASs;

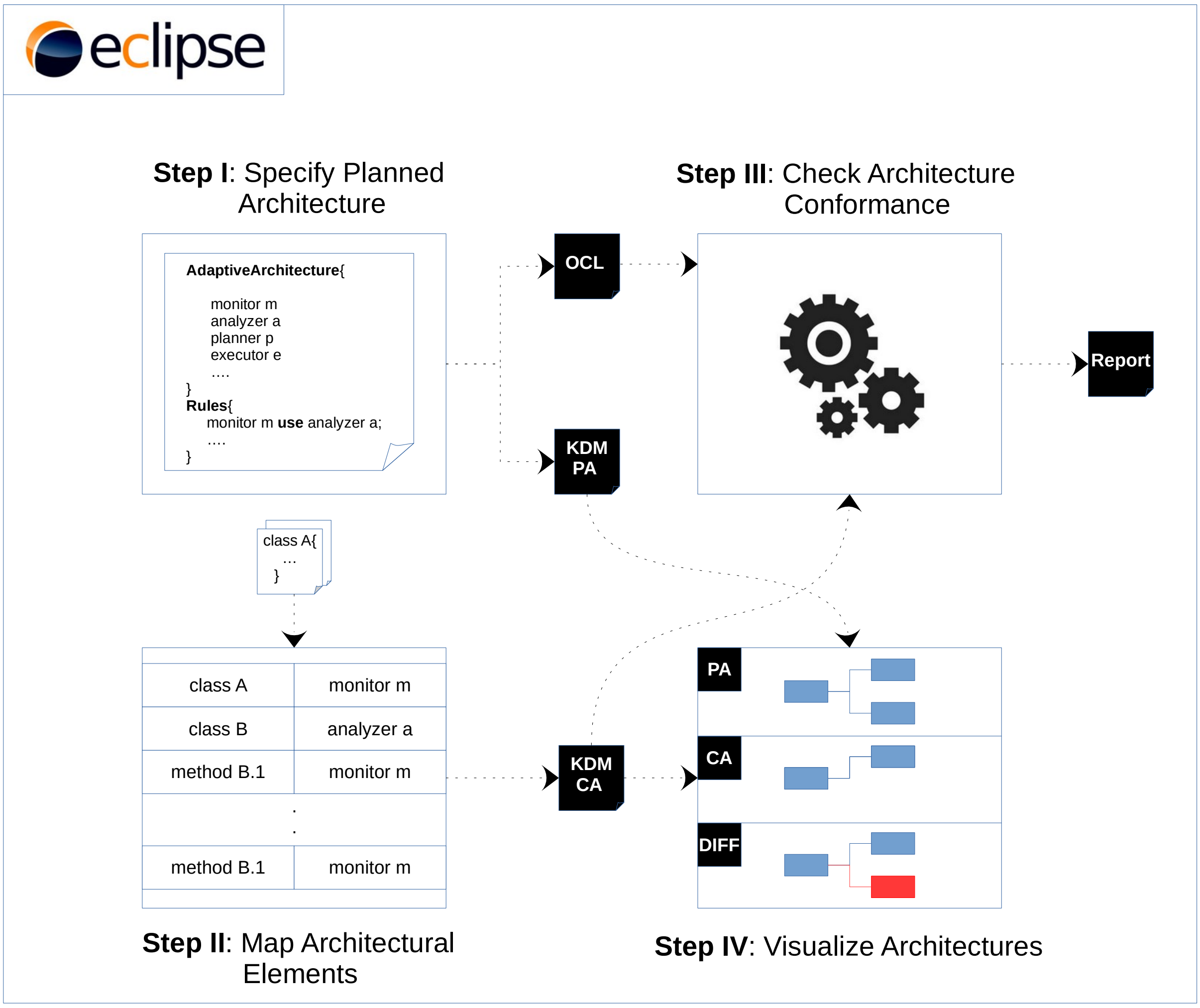
- Technical Contributions:
  - A DSL to specify PAs of ASs;
  - A workbench called REMEDY that supports and guides software architects in the ACC process for ASs.

1.5 Overview of the Approach

Our approach, called REMEDY[1], has been developed since late 2017 and it has suffered several modifications along these years. Some useful ideas were taken from the work publish by Landi, Santibanez, Santos, Cunha, Durelli, and Camargo (2022) but we took the decision of implementing it from the scratch due to two reasons: *i*) Acquire the technical knowledge and know-how about the tools used in the implementation and *ii*) Due to changes in the design which would impact how our approach would identify the architectural drifts in ASs. REMEDY is an ACC workbench for ASs implemented as several Eclipse plugins and it uses KDM models as the main artifacts. The goal is to identify architectural drifts between a PA model and another one that represents the CA of an AS.

Figure 1.1 shows the steps of REMEDY from a software architect point of view. Step I, also called “Specification of Planned Architecture”, aim to create a PA that will be used in next steps of REMEDY. The execution of this step is supported by a DSL called DSL-REMEDY that specify the adaptive part of an AS with a high level of abstraction. At the end of this step, two artifacts are generated; a KDM instance representing the PA and a OCL file with rules to be checked.

[1] https://github.com/dsanmartins/REMEDY


eclipse
Step I: Specify Planned Architecture
Step III: Check Architecture Conformance
OCL
KDM PA
Report
KDM CA
PA
CA
DIFF
Step II: Map Architectural Elements
Step IV: Visualize Architectures

```
AdaptiveArchitecture{

    monitor m
    analyzer a
    planner p
    executor e
    ....
}
Rules{
    monitor m use analyzer a;
    ....
}
```

```
class A{
   ...
}
```

class A	monitor m
class B	analyzer a
method B.1	monitor m
. .	
method B.1	monitor m

Figure 1.1 – A simplified view of steps of REMEDY

Step II, also called "Map Architectural Elements", aim to recovery the CA of the system being analyzed. The execution to generate the CA is supported by a third-party tool called *MoDisco* (BRUNELIERE et al., 2010; BRUNELIÈRE et al., 2014) which extracts information from the source-code and transform it into a KDM model. Then, software architects map the AS abstractions from the PA to code elements that are represented in the model. The output of this step is the CA of the system.

Step III, also called "Check Architecture Conformance", aim to check the conformity of the CA regarding to the PA by using the OCL file with the rules. Step IV, also called "Visualize Architecture", aim to exhibit architectural problems found by REMEDY. At the end of this step, the software architect is capable to visualize which rules passed or not the conformity checking process and a graphical view of the PA, CA and differences between both.

1.6 Publications

We list here published works during the period of doctoral studies, including published in journals, conferences and workshops:

1.6.1 Published

- LANDI, André; SANTIBANEZ, Daniel; SANTOS, Bruno M.; CUNHA, Warteruzannan S.; DURELLI, Rafael S.; CAMARGO, Valter V. de. Architectural Conformance Checking for KDM-Represented Systems. **Journal of Systems and Software**, 2022

- SAN MARTÍN, Daniel; CAMARGO, Valter. A Domain-Specific Language to Specify Planned Architectures of Adaptive Systems. In: 15TH Brazilian Symposium on Software Components, Architectures, and Reuse. Joinville, Brazil: Association for Computing Machinery, 2021. (SBCARS '21), p. 41–50

- SANTIBANEZ, Daniel; SIQUEIRA, B.; DE CAMARGO, V. V.; FERRARI, F. Characterizing Architectural Drifts of Adaptive Systems. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). [s.l.: s.n.], 2020. P. 389–399

- SANTIBANEZ, Daniel; ANGULO, Guisella; MARINHO, Bruno; HONDA, Raphael; CAMARGO, Valter. Specification and use of concern metrics for supporting modularity-oriented modernizations. **Software Quality Journal**, v. 28, n. 3, p. 1087–1111, Sept. 2020

- SANTOS, Bruno; SANTIBANEZ, Daniel; HONDA, Raphael; CAMARGO, Valter Vieira de. Concern Metrics for Modularity-Oriented Modernizations. In: ________. **Quality of Information and Communications Technology**. Cham: Springer International Publishing, 2019. P. 225–238

- SANTOS, Bruno M.; S. LANDI, André de; SANTIBANEZ, Daniel; DURELLI, Rafael S.; CAMARGO, Valter V. de. Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations. **Journal of Systems and Software**, v. 149, p. 285–304, 2019

- SIQUEIRA, Bento; JÚNIOR, Misael; FERRARI, Fabiano; SANTIBANEZ, Daniel; MENOTTI, Ricardo; CAMARGO, Valter V. Experimenting with a Multi-Approach Testing Strategy for Adaptive Systems. In: PROCEEDINGS of the 17th Brazilian Symposium on Software Quality. Curitiba, Brazil: [s.n.], 2018. (SBQS), p. 111–120

- ANGULO, Guisella; SANTIBANEZ, Daniel; SANTOS, Bruno; FERRARI, Fabiano; CAMARGO, Valter V. Vieira. An Approach for Creating KDM2PSM Transformation

Engines in ADM Context: The RUTE-K2J Case. In: PROCEEDINGS of the XII Brazilian Symposium on Software Components, Architectures, and Reuse. Sao Carlos, Brazil: Association for Computing Machinery, 2018. (SBCARS '18), p. 92–101

- SANTIBANEZ, Daniel; V. DE CAMARGO, Valter. Remodularizing Adaptive Systems by Employing Architecture-Driven Modernization Principles. In: VII Workshop de Teses e Dissertações do CBSoft (WTDSoft 2017). [S.l.: s.n.], Sept. 2017

1.7 Structure

The remainder of this thesis is structured as follows:

- Chapter 2 first provides the scientific background that is needed to understand the subsequent chapters;
- Chapter 3 presents a systematic mapping of architectural anomalies in adaptive systems. This provides an overview of types of anomalies reported in literature and the evidences to justify this work;
- Chapter 4 discusses work related to our approach. Section 4.1 covers approaches dealing with non-extensible ACC approaches and Section 4.1.2 with extensible ACC approaches;
- Chapter 5 characterizes architectural drifts of ASs and the process that we follow to identify them. Also, we provide some examples of each drift in real systems;
- Chapter 6 describes in detail our approach of ACC for ASs called REMEDY. It consists in four steps; Specify Planned Architecture, Map Architectural Elements, Check Architecture Conformance and Visualize Architectures;
- Chapter 7 presents a description of the use of REMEDY from the point of view of a software architecture;
- Chapter 8 reports the results of initial experimental evaluation conducted to validate our proposals, which consisted on a controlled experiment to validate DSL-REMEDY in terms of productivity;
- Chapter 9 concludes the thesis. It summarizes what has been done, discusses results, lessons learned and limitations, and gives an outlook on what research could be done in this area in the future.

Chapter 2

Foundations

Before we present our approach, we summarize the state-of-the-art in our chosen area of research. This chapter is divided in four parts: first, in Section 2.1 we introduce concepts related to software architecture. Second, in Section 2.2 we provide a clear definition of concepts about architectural conformance checking. Third, Section 2.3 introduces the foundation of adaptive systems. Fourth, Section 2.4 gives an introduction of the Knowledge Discovery Metamodel and delves in architectural concepts of the metamodel which are important to understand our approach.

2.1 Software Architecture

Software Architecture is the fundamental properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution (MAIER; EMERY; HILLIARD, 2001) . It is a key tool for the software industry because it improves communication between stakeholders, facilitates early design decisions, promotes transferable abstractions of a system and can be used as the basis for system implementation (TAYLOR; MEDVIDOVIC; DASHOFY, 2009).

According to Taylor, Medvidovic, and Dashofy (2009) there are three kind of architectural elements that can be distinguished: processing elements; data elements; connecting elements. These elements are organized according to a design which consists of properties and relationships. Properties and relationships are used to define constraints on architectural elements. Constraints are determined by considerations ranging from basic functional aspects to various non-functional aspects such as economics, performance and reliability.

Bass, Clements, and Kazman (2012) define architecture as a set of software elements characterized by externally visible properties and the relationships existing among them. By "externally visible properties", the authors refer to assumptions other components can make of a component, such as provided services, performance characteristics, fault handling. In our interpretation, assumptions can be intended as contracts when coupled with a complementary number of constraints that ensure their validity. Based on the previous definitions, it is possible conclude that architecture is partially defined through constraints. Architectural design constraints have been often associated to patterns and styles (GIESECKE; HASSELBRING; RIEBISCH, 2007; BASS; CLEMENTS; KAZMAN, 2012).

One of the main issues in software architecture is the erosion that occurs when the implemented architecture of a system diverges from its intended architecture (TAYLOR; MEDVIDOVIC; DASHOFY, 2009; DE SILVA; BALASUBRAMANIAM, 2012). According to Dimech and Balasubramaniam (2013) such degradation may arise due to several factors, including: *i*) unawareness by software developers; *ii*) possibly conflicting requirements that are unforeseen in the early stages; *iii*) technical difficulties that arise during implementation; and *iv*) the pressure of deadlines that are not uncommon in software development.

Some consequences of architecture erosion include failure to meet functional or quality requirements, brittle systems, high maintenance costs and ultimately rapid software aging and obsolescence (DE SILVA; BALASUBRAMANIAM, 2012). Architecture erosion is a well-recognized problem and a number of approaches have been proposed to prevent, minimize or repair erosion, mainly based on the concept of architecture conformance (KNODEL; POPESCU, 2007).

Software architecture might be specified in different forms and notations depending on the purpose they serve. Caracciolo, Lungu, and Nierstrasz (2015) state that there are two groups of modeling languages to specify software architecture; the first one are Architecture Definition Languages (ADLs), used to describe an architecture in terms of properties and relations. The second one are modeling languages that provide dedicated constructs for the definition of constraints. For instance, Garlan (1995) presents *Aesop* that enables users to declare topological invariants (i.e., allowed dependencies) as part of the definition of an architectural style. Such an invariant can be used to perform a structural check of the model and verify that entities of a certain type are only connected to other entities of a certain type through predefined ports.

Moriconi and Riemenschneider (1997) present *SADL* that supports the declaration of first-order logic predicates for the definition of similar invariants. Luckham et al. (1995) developed *Rapide* that offers support for behavioral constraints that can be used to define run-time invariants (e.g., message values, invocation sequences, abstract state). Garlan, Monroe, and Wile (2000a) developed *ACME* that relies on a separated constraint language called *Armani*. This language allows the writing of first-order logic predicates and can be used to define type constraints (like in previously mentioned languages) and heuristics (i.e.,numerical thresholds for limiting the size of specific parts of the model). Feiler and Gluch (2012) propose *REAL*, a language to verify type checking invariants and constraints on the graph structure.

OCL OMG (2005) is a language for defining constraints in UML models. UML is used in order to describe a system in terms of entities and relationships. Entities can be annotated (e.g., stereotypes, fields, methods) and relationships can be either static or dynamic (e.g., class diagrams, sequence diagrams). OCL provides a set of functions that can be used to navigate the graph structure of a model and create assertions on identified elements. Assertions can be defined to constraint property values, entity types and relationship cardinalities. OCL is a very expressive language which has been taken as inspirational source in several other approaches for defining architectural constraints in models based on *Meta-Object Facility* (MOF) metamodels (MEDVIDOVIC, 2006).

2.2 Architectural Conformance Checking

Architecture Conformance Checking (ACC) is one of the main activities in software quality control. ACC goals is to reveal the differences between the intended architecture or Planned Architecture (PA) and its real implementation (KNODEL; POPESCU, 2007) or Current Architecture (CA). It reveals the relations and constraints foreseen by the PA that were violated by the system's implementation. Figure 2.1 illustrates a PA of a java system, used as example in the work of Pruijt, Köppe, Werf, et al. (2017). The architecture is composed of two modules, *ModuleA* and *ModuleB* and each one of them has two submodules. The classes in the submodules are related via associations, showing for instance that an instance of *Class1* may know upmost one instance of Class2. The dependency arrows (the dashed arrows) show that *ModuleA1* is allowed to use *ModuleB1* and that *ModuleA2* is allowed to use *ModuleB* (PRUIJT; KÖPPE; WERF, et al., 2017).

The full set of relationships rules are the following: *i*) *ModuleA1* is allowed to use *ModuleB1*; *ii*) *ModuleA2* is allowed to use *ModuleB*, so also both sub modules, *ModuleB1* and *ModuleB2*; *iii*) *ModuleA1* is not allowed to use *ModuleB2*; *iv*) The submodules of ModuleA are allowed to use each other; *v*) The submodules of *ModuleB* are allowed to use each other. Notice that in this case, the first three rules are explicitly visible in the diagram, while the last two are implicit.

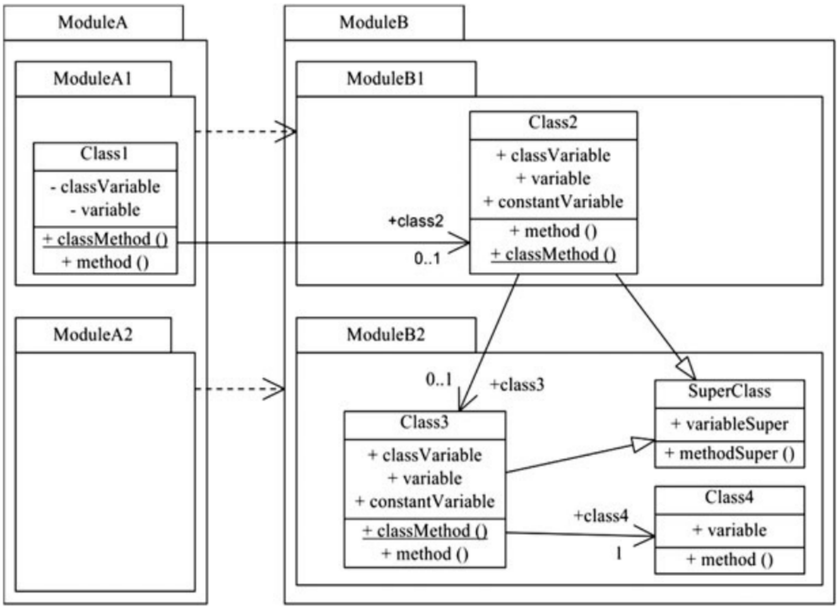
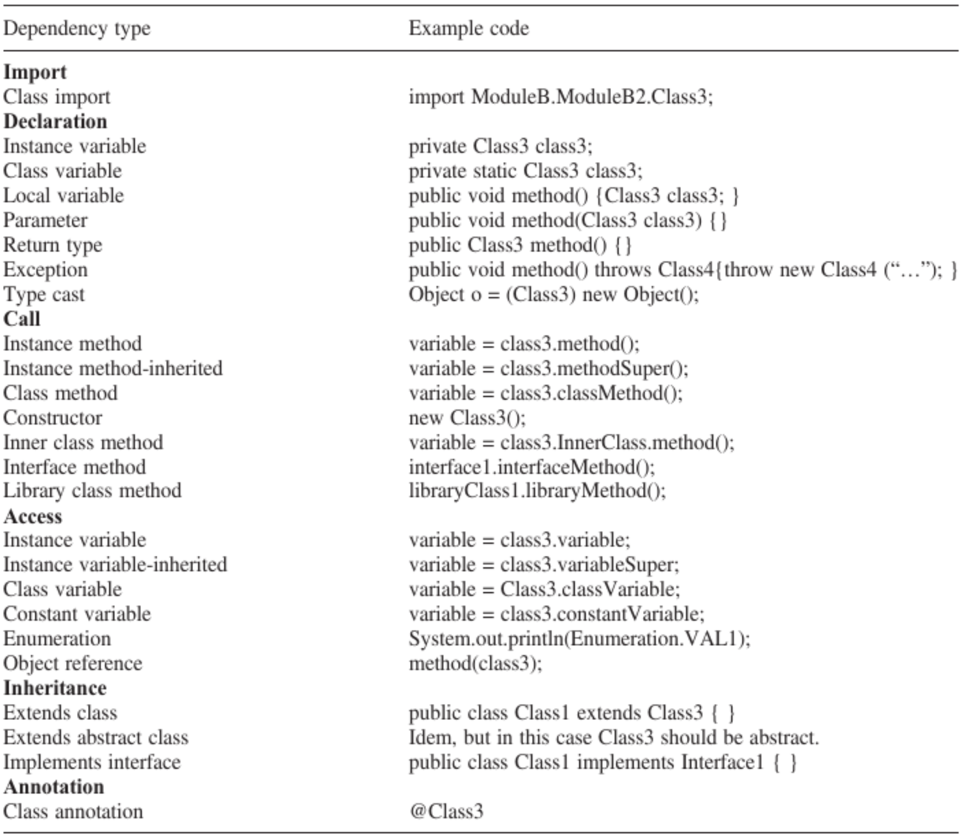


Figure 2.1 – Example of a planned architecture (From (PRUIJT; KÖPPE; WERF, et al., 2017)

In order to check whether the source code of the system conforms the PA authors executed several tools that support ACC. The dependency types checked by the tools were divided into six groups: *Import*, *Declaration*, *Call*, *Access*, *Inheritance* and *Annotation*. Figure 2.2 shows the complete dependency types with code example. Each code example shows a code construct that, if programmed within *Class1*, would violate the intended architecture of Figure 2.1. This is because the code construct includes a dependency to an element of *ModuleB2*, while the intended architecture does not allow *ModuleA1* to use *ModuleB2*.



Dependency type	Example code
Import	
Class import	import ModuleB.ModuleB2.Class3;
Declaration	
Instance variable	private Class3 class3;
Class variable	private static Class3 class3;
Local variable	public void method() {Class3 class3; }
Parameter	public void method(Class3 class3) { }
Return type	public Class3 method() { }
Exception	public void method() throws Class4{throw new Class4 ("…"); }
Type cast	Object o = (Class3) new Object();
Call	
Instance method	variable = class3.method();
Instance method-inherited	variable = class3.methodSuper();
Class method	variable = class3.classMethod();
Constructor	new Class3();
Inner class method	variable = class3.InnerClass.method();
Interface method	interface1.interfaceMethod();
Library class method	libraryClass1.libraryMethod();
Access	
Instance variable	variable = class3.variable;
Instance variable-inherited	variable = class3.variableSuper;
Class variable	variable = Class3.classVariable;
Constant variable	variable = class3.constantVariable;
Enumeration	System.out.println(Enumeration.VAL1);
Object reference	method(class3);
Inheritance	
Extends class	public class Class1 extends Class3 { }
Extends abstract class	Idem, but in this case Class3 should be abstract.
Implements interface	public class Class1 implements Interface1 { }
Annotation	
Class annotation	@Class3

Figure 2.2 – Dependency types in Pruijt, Köppe, Werf, et al. (2017) example

According to Caracciolo, Lungu, and Nierstrasz (2015) the specification of the PA could be done mainly by techniques that fall into two categories; *i*) A Domain Specific Language (DSL) which is designed to formulate concepts in a form and notation that is familiar to the user. DSLs typically do not assume any kind of specific technical knowledge and are mostly declarative and *ii*) A General Purpose Language (GPL) where commonly an extension of the programming language has been developed to support architecture specification. Also, authors categorize ACC tools as non-extensible and extensible. The first one are specialized and not extensible off-the-shelf tools while the second one are off-the-shelf tools based on a plugin architecture where the new functionality is partially adaptable by operators Caracciolo, Lungu, and Nierstrasz (2015).

Figure 2.3 illustrates the vision of conformance checking in the broader context of the software development lifecycle. The model and the code can change independently and the conformance checking operation detects any incompatibilities between the two descriptions as violations of the conformance rules. As a result of the ACC, it is possible to get three kinds of information. The first one is the relationships that were specified as “allowed” in the PA that are implemented in the current system. These relationships are called “convergences”, and they show that the implementation is compatible with the PA. The second one is the relationships that were specified as “not allowed” in the PA, but they are present in the current system. These relationships are called “divergence” or “architectural drifts”, and they reveal that the implementation is not compatible with the PA. The third one is the relationships that were not specified in the PA, but they are present in the current system. These relationships are called “absences”, and they show that the relations in the implementation were not found in the PA (DIMECH; BALASUBRAMANIAM, 2013).

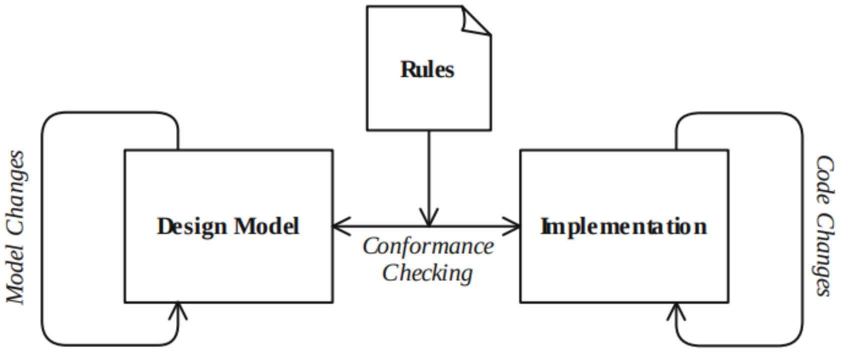


Figure 2.3 – Overview of Conformance between Model and Implementation (DIMECH; BALASUBRAMANIAM, 2013)

There are two ways of conducting ACC processes (KNODEL; POPESCU, 2007; MURPHY; NOTKIN; SULLIVAN, 1995). The first one is the static verification in which the source code is compared with the PA. The second one is the dynamic verification in which characteristics of the running system is compared with the PA. Moreover, the two main static techniques for performing ACC are Reflection Models and Compliance Relations Rules (KNODEL; POPESCU, 2007; MURPHY; NOTKIN; SULLIVAN, 1995).

Reflection model is a technique that supports the use of a high-level system model as an eyeglass to see the source code model. Usually, this technique is applied when there is a few or none information about the system and its architecture (MURPHY; NOTKIN; SULLIVAN, 1995). The Compliance Relations Rules specify constraints between the architectural elements. These constraints can allow, prohibit, or impose the relations between the elements (KNODEL; POPESCU, 2007).

Several conformance checking tools have been analyzed in literature. For instance, de Silva and Balasubramaniam (2012) proposes a taxonomy to categorize existing techniques and approaches. Pruijt, Köppe, and Brinkkemper (2013) and Passos et al. (2010) compare multiple tools by evaluating their capabilities through an experiment. In both studies the authors conclude that existing tools offer complementary features and none of them can be considered as a perfect replacement for all the others.

Caracciolo, Lungu, and Nierstrasz (2014) performed a survey to discover which tools practitioners use to test architectural constraints. Table 2.1 shows an overview of the most visible conformance testing tools found in literature that accept textual specification as input. We include REMEDY, the approach that was developed in the context of this thesis. Solutions as DCL , inCode.Rules and REMEDY verify constraints on relationships between classes and modules such as access, declaration and extension. Some languages such as SOUL, LogEn and SCL focus more on structural properties of classes and methods. Notice that inCode.Rules also detects code smells such as God class and Data class.

	DCL	**TamDera**	**inCode.Rules**	**SOUL**	**LogEn**	**SCL**	**ArchFace**	**ArchJava**	**Classycle**	**NDepend/CQLinq**	**Semmle.QL**	**REMEDY**
constraint types												
- relationships	✓	✓	✓	✓			✓	✓	✓			✓
- elements				✓	✓	✓				✓	✓	✓
- code smells			✓									
detected RM relations												
- convergence	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- absences	✓	✓		✓		✓	✓	✓		✓		✓
- divergences	✓	✓	✓		✓				✓	✓	✓	✓
DSL extensibility												
- new predicates				✓	✓							
programming languages	[J: Java; S: Smalltalk; P: Prolog, D: Datalog; C: C++]											
- analyzed system	J	J	J	J/S	J	J/C	J	J	J	.net	J	J
- tool implem.	J	P	J	S	D	J	J	J	J	J	.net	J/Xtex

Table 2.1 – Comparison among conformance checking tools based on a textual DSL. Adapted from Caracciolo, Lungu, and Nierstrasz (2014)

Most of these solutions, with exception of Terra and Valente (2009) (DCL), Gurgel et al. (2014) (TamDera), Kramer (2012) (NDepend/CQLinq) and REMEDY, are only able to detect one of the following violations: absences or divergences (MURPHY; NOTKIN; SULLIVAN, 1995). Only two of the reviewed solutions offer support for language-level extension (i.e., SOUL (MENS; WUYTS; D'HONDT, 1999) and LogEn (EICHBERG et al., 2008)). Both are logic programming languages in which new predicates can be defined by composing existing predicates.

The general lack of support for extensibility limits the expressiveness of the solution. Almost all techniques, with the exception of ArchFace (UBAYASHI; NOMURA; TAMAI, 2010) and ArchJava (ALDRICH; CHAMBERS; NOTKIN, 2002), assume that architectural constraints are specified in a separated text file. ArchFace and ArchJava require the user to define constraints directly in the source code by using special constructs that are checked at compile time.

According to Caracciolo, Lungu, and Nierstrasz (2014), there are three requirements that ACC should fulfill; extensibility, usability and multifaceted modeling. The first one is the capability of adding new language constructs in order to model different architectures. The second one is the perception of language simplicity and it is intuitively enough to communicate the right message to the stakeholders involved in the architecting phase, but shall also enable formality so to drive analysis and other automatic tasks. The third one is about if the language can describe several architectural viewpoint.

2.3 Adaptive Systems

Adaptive Systems or Self-Adaptive Systems are systems that deal with uncertainties by reconfiguring their structure or adjust their behavior during operation (WEYNS, 2018). Examples of uncertainties are dynamic allocation of resources and evolving user requirements. Architecture-based self-adaptation separates the concerns of the Managed System that is subject to adaptation and the the concerns of the Managing System that contains the adaptation logic. The Managing System realizes a control mechanism that monitors and adapts the Managed System to achieve the adaptation goals (BRUN et al., 2009).

Another term used by researchers to describe this kind of systems is *autonomic computing systems* (HORN, 2001). According to Kephart and Chess (2003), autonomic computing exposes one or more *self-** adaptation properties such as self-configuration, self-optimization, self-healing and self-protection. Systems with self-configuration capabilities reconfigure themselves automatically, based on high level policies that specify evolution goals, as well as reconfiguration symptoms and strategies.

Systems with self-optimizing property adapt themselves to improve non-functional properties according to business goals and changing environmental situations. Systems with self-healing property can automatically detects, diagnoses, and repairs localized software and hardware problems. Systems with self-protection property automatically defend against malicious attacks or cascading failures. They use early warning to anticipate and prevent system wide failures (KEPHART; CHESS, 2003; MULLER; VILLEGAS, 2014).

Many researchers use the terms autonomic and self-adaptive interchangeably. However, according to Salehie and Tahvildari (2009) there are some similarities but also some differences between adaptive systems and autonomic computing. In their view, the term autonomic refers to a broader context, handling all layers of the system's architecture (from applications to hardware), whereas self-adaptive has less coverage — constrained mostly to applications and middleware — and, thus, falling under the umbrella of autonomic computing.

Figure 2.4 shows the conceptual architecture of an AS. The Managing System observes the Managed System and the Environment through monitors. If it identifies changes on the Managed System then adaptations are triggered to maintain the Managed System's goals via effectors.

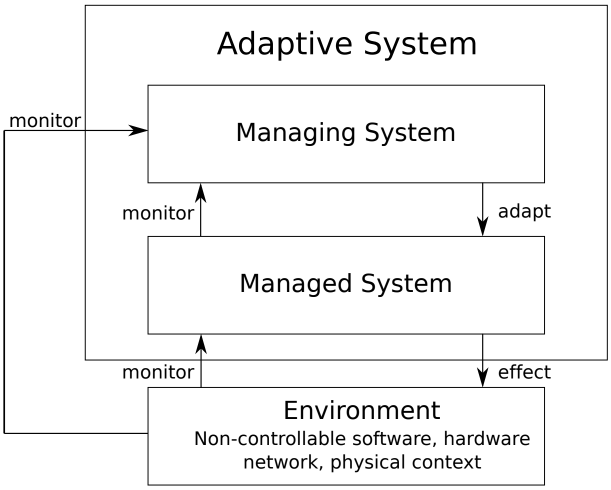


Figure 2.4 – Conceptual Architecture of an AS

2.3.1 Feedback Control Theory and Feedback Loops

The heart of an adaptive system is an abstraction called Feedback Control Loop which is a concept that comes from the field of Control Theory (DOYLE; FRANCIS; TANNENBAUM, 1991). Figure 2.5 depicts a simplified view of a control system. The *reference input* is “the desired value of the measured output” while the *measured output* is a “a measurable characteristic of the target system” (HELLERSTEIN et al., 2004). In other words, the *reference input* is the system requirements and the *measured output* is what is being monitored by the system. For instance, consider a (simplified view of a) robot that must road in parallel to a wall with a certain distance D_1. While it is moving, a monitor captures data from a sensor according to its polling rate. The data is the measure of the distance between the wall and its current position D_0 because, due to mechanical parts, in some points the distance measured could be greater or lesser than the required distance. In this example, D_1 is the reference input, whereas the current distance D_0 , which can be read from the robot sensor, is the measured output.

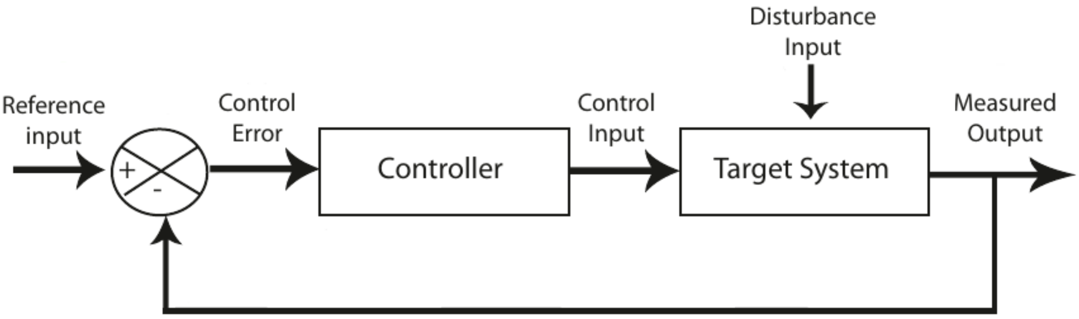


Figure 2.5 – Classical block diagram of a feedback control system

Given this information, the controller “computes values of the control input based on current and past values of control error”. The control error is “the difference between the reference input and the measured output”, while the control input is “a parameter that affects the behavior of the target system and can be adjusted dynamically”. Back to the example, the control error E can be calculated as $E = D_1 - D_0$, leading to a straightforward definition for the control input: if $E < 0$, the *controller* (a servo controller that controls a servo motor) should move the wheels in direction to the wall to get back to the desired distance. Analogously, if $E > 0$, the *controller* should move the wheels in direction away from the wall. The idea is to keep D_0 as close as possible to D_1 at all times.

Finally, the disturbance input “are factors that affect the measured output but for which there is no governing control input”. In other words, these are taken from the context in which the system executes. Neither the system nor the controller have any control over these values. For the robot, the inclination of the road or wheels wear are examples of *disturbance inputs*, as they can have an influence on the measured distance D_0.

In more complex systems, such as information systems that usually have multiple inputs and multiple outputs, it is uncommon producing models with high degree of formality due to human resources and time constraints. For these reasons, adaptive systems can be considered a simplified view of control systems. Thus, the generic mechanism for adaptation in software systems is the feedback control loop.

Feedback control loops provide the generic mechanism for adaptation and typically involves four key activities: collect, analyze, decide, and act. Sensors collect data from the executing system and its context about its current state. The accumulated data are then cleaned, filtered, and pruned and, finally, stored for future reference to portray an accurate model of past and current states. The diagnosis then analyzes the data to infer trends and identify symptoms. Subsequently, the planning attempts to predict the future to decide on how to act on the executing system and its context through actuators or effectors (BRUN et al., 2009).

A well-kown example of a feedback control loop is the the autonomic computing MAPE-K reference model (HORN, 2001; KEPHART; CHESS, 2003), whose acronym stands for the four activities that it performs: monitor, analyze, plan and execute. If adaptive systems need to evaluate their behavior and act accordingly, they must have some kind of feedback loop among their components, even if implicit or hidden in the system’s architecture.

Figure 2.6 shows the MAPE-K loop (monitor, analyzer, planner, executor, and knowledge base). Monitors collect, aggregate and filter information from the environment and the target system (i.e., the system to be evolved), and send this information in the form of symptoms to the next element in the loop. Analyzers correlate the symptoms received from monitors to decide about the need for adapting the system. Based on business policies, planners define the maintenance activities to be executed to adapt or evolve the system. Executors implement the set of activities defined by planners. The knowledge base enables the information flow along the loop, and provides persistence for historical information and policies required to correlate complex situations.

Sensors and effectors are endpoints that expose the state and control operations of managed elements in the system. Sensors allow to gather information from both the environment and

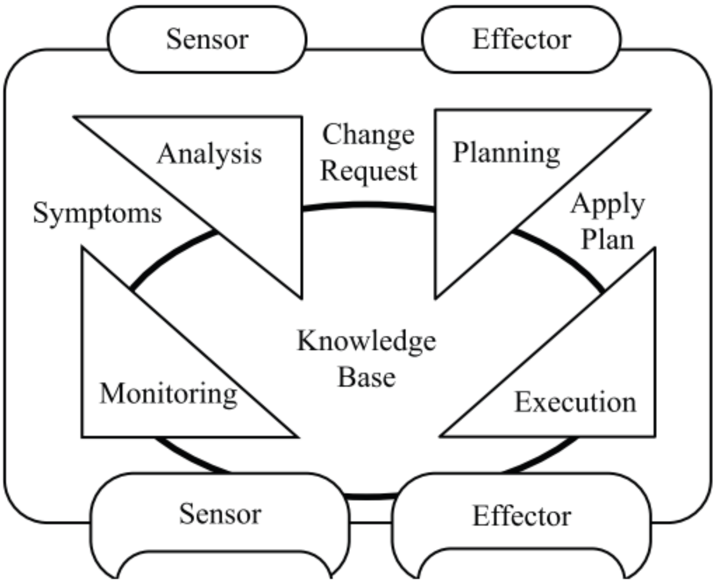


Figure 2.6 – The MAPE-K feedback control loop

other Managing Systems. Effectors provide the interfaces to implement the control actions that evolve the managed element. Managed elements can be either system components or other managing systems.

Researchers have expressed the need to make these feedback loops first-class citizens in the design of adaptive systems (WEYNS; MALEK; ANDERSSON, 2010; VOGEL; GIESE, 2014). For instance, Brun et al. (2009) notice that "while [some] research projects realized feedback systems, the actual feedback loops were hidden or abstracted. [. . .] With the proliferation of adaptive software systems it is imperative to develop theories, methods and tools around feedback loops."

Cheng et al. (2009) declare that "Even though control engineering as well as feedback found in nature are not targeting software systems, mining the rich experiences of these fields and applying principles and findings to software-intensive adaptive systems is a most worthwhile and promising avenue of research for self-adaptive systems. We further strongly believe that self-adaptive systems must be based on this feedback principle."

Therefore, feedback control loops are a fundamental architectural element in the design of control systems, whereby the output to be controlled is compared to a desired reference value and their difference is used to compute corrective control action (DOYLE; FRANCIS; TANNENBAUM, 1991). In other words, measurements of a system's output are used to achieve externally specified goals by adjusting parameters that in some way affect indicators that these goals are being achieved. For this reason, feedback loops are also called closed loops and are present in some form in almost any system that is considered automatic, such as an automobile cruise control or an industrial control system (HELLERSTEIN et al., 2004).

2.3.2 Lower-level Abstractions of MAPE-K

MAPE-K reference model only depicts canonical abstractions of an AS but there are others, equally important that software architects must pay attention for a good architectural quality and we called them as “low level abstractions”. As these abstractions are not evident in MAPE-K, we found that in some cases software architects are not aware of them and usually do not consider them when architecting the system (RAMIREZ; CHENG, 2010). The schematic view of MAPE-K that is used in this thesis is considered as a base reference model is described on Chapter 5.

Normally, a system that is considered adaptive is composed of the Managed Subsytem, which is the biggest base part, and one or more Managing Subsystems, which are modules responsible for performing the adaptations. Notice that MAPE-K is much more devoted to design the adaptation parts than the base system itself. MAPE-K of Figure 5.1 contains the conventional known abstractions and also three lower level abstractions presented by other works (VILLEGAS et al., 2011; WEYNS; IFTIKHAR; SÖDERLUND, 2013; ARBOLEDA et al., 2016; ABDENNADHER; BOUASSIDA RODRIGUEZ; JMAIEL, 2017).

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 (ABDENNADHER; BOUASSIDA RODRIGUEZ; JMAIEL, 2017). A common strategy to implement it, is using N-version programming defined as the independent generation of N functionally equivalent programs from the same initial specification (CHEN; AVIZIENIS, 1995; PSAIER; DUSTDAR, 2011).

For instance, a typical example occurs when a cluster of virtual machines in a cloud computing environment needs to reach a certain goal conditioned by Service Level Agreements SLAs. In this case, an autonomous self-optimizing system is required to identify overload servers and migrated them to other servers to prevent violation of service level agreement (NAJAFIZADEGAN; NAZEMI; KHAJEHVAND, 2021).

Reference Inputs (requirements) consist 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 (VILLEGAS et al., 2011). They could be implemented as single reference values, some form of contract, service level objectives, among other possibilities.

Measured Outputs (indicators) 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 (VILLEGAS et al., 2011).

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.

2.4 Knowledge Discovery Metamodel

The Knowledge Discovery Metamodel (KDM) is a metamodel part of the Architecture-Driven Modernization (ADM) initiative whose intention is to promote industry consensus on the modernization of the existing software system. The initiative combines reengineering concepts, Model-Driven Architecture (MDA) principles and standard metamodels. Also, ADM introduces several modernization standards besides KDM such as the Abstract Syntax Tree Metamodel (ASTM) and Structured Metrics Metamodel (SMM) (PÉREZ-CASTILLO; DE GUZMÁN; PIATTINI, 2011; OMG, 2017).

KDM is able to represent all the characteristics of software systems in a unique metamodel (PÉREZ-CASTILLO; DE GUZMÁN; PIATTINI, 2011). A schematic representation of KDM can be seen in Figure 2.7. It is divided into four layers that are further divided into packages. Each package is an internal metamodel, concentrating on specific aspects of the software. Thus, there are packages for representing a wide spectrum of systems abstractions, from low-level details like source-code (`Code` package) and run-time actions (`Action` package) to high-level details like User Interface (`UI` package), Business Rules (`Conceptual` package) and Architectural View (`Structure` package) (OMG, 2009, 2016).

The modernization process starts by reverse engineering a system into a KDM instance that, by its turn, is analyzed/mined to search for problems. Next, a set of refactorings and optimizations are performed to obtain a refactored and improved KDM instance (DURELLI, R. S. et al., 2017; DURELLI, R. et al., 2014). The process is completed with the generation of the modernized system. According to Pérez-Castillo, De Guzmán, and Piattini (2011), ADM can support many kinds of modernization scenarios such as: platform migration, language to language conversion, application improvement and architectural revitalization (ULRICH; NEWCOMB, 2010).

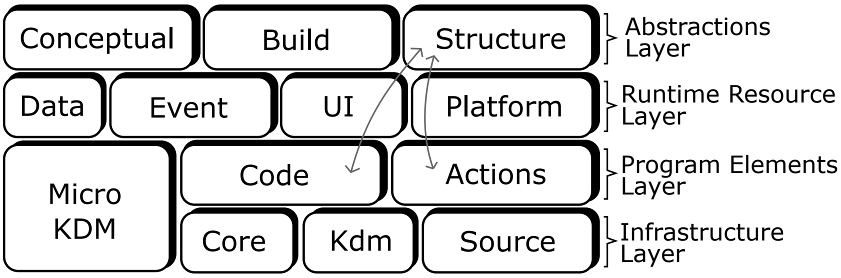


Figure 2.7 – The four KDM layers and packages from (PÉREZ-CASTILLO; DE GUZMÁN; PIATTINI, 2011)

It is important to mention that although KDM is divided into layers, its packages can communicate with each other. These inter-package communications are a key point in KDM since it allows mapping higher-level abstractions to lower-level ones. These communications between the packages are schematically represented in Figure 2.7 by the two arrows from `Structure` to `Code` and `Action` packages. These three packages are the most important packages in the context of our research.

Code Package contains all the metaclasses for modeling the source code static structure. For example, `ClassUnit` metaclass represents classes and `InterfaceUnit` metaclass represents interfaces. The Code package has a total of 90 metaclasses and all the abstract elements for representing the source code (OMG, 2016). The Action Package defines metaclasses to represent behavioral units. Examples of these behaviors are: declarations (`Reads`, `Creates`, etc.), operators (`Writes`, `Addresses`, etc), and flow conditions (`Flow`, `TrueFlow`, etc.). When generating a KDM instance, it is assumed that each element of the Action package corresponds to a behavior in a programming language.

The Structure Package is one of the most important ones as it is able to represent the logical architecture of a software system. Figure 2.8 shows in the left part, the Structure package metaclasses in gray and other important related metaclasses in white. In the right part, there is a schematic representation of a Structure package instance. Structure package provides five metaclasses for representing architectural elements: `Subsystem`, `Component`, `SoftwareSystem`, `ArchitectureView` and `Layer`. Besides, by means of the self-relationship of the `AbstractStructureElement`, it is possible to create a hierarchy among these elements. For instance, it is possible to create an architecture having a Software System with two subsystems, which include two layers each, where each layer can include two components.

This package also provides an important means for specifying mappings between higher-level concepts to lower-level ones. This can be seen like an abstract-concrete mapping and this is done by an attribute named "`implementation`" (OMG, 2016), represented by the relationship between the AbstractSctructureElement and KDMEntity metaclasses. Notice that KDMEntity metaclass belongs to the Core package, which is a central KDM package that provides base metaclasses for the other packages. `KDMEntity` is one of the most important metaclasses, since all the other KDM metaclasses are direct or indirect subclasses of it. Thus, all KDM metaclasses are KDM Entities.

`AggregatedRelationship` is another important metaclass herein because its role is to capture relationships among architectural abstractions. It is a kind of relationship that can group other primitive relationships within it. This is being represented in the Figure by the 0..n association between the `AggregatedRelationship` metaclass and the `KDMRelationship` metaclass. In KDM, every relationship type is represented by a metaclass, examples of primitive relationships are method calls (`Calls` metaclass), object instantiation (`Creates` metaclass) and implements relationships (`Implements` metaclass). Each `AggregatedRelationship` involves two KDM Entities, the source (*from* property) and target (*to* property), as can be seen in

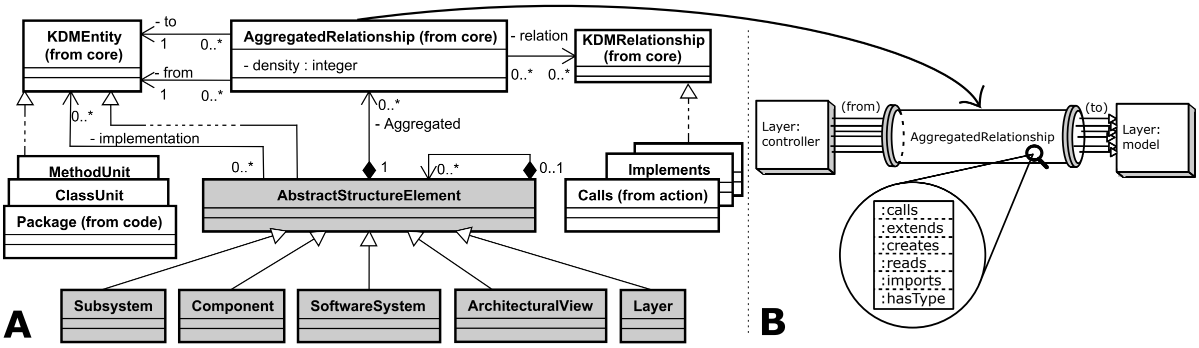


Figure 2.8 – A - Structure Package Class Diagram from (OMG, 2017); B - Schematic example of a Structure Package Instance

part A of Figure 2.8.

Since all the architectural elements are KDM Entities (due to the inheritance), it is possible to represent relationships between these architectural elements employing the `AggregatedRelationship`, which is schematically shown in Figure 2.8 Part **B**. In the example, we have a relationship between the layers Controller and Model. The cylinder between the two layers represents an instance of the `AggregatedRelationship` metaclass. The controller layer represents the source (*from*) of the relationship, and the model layer represents the target (*to*) of the relationship. An aggregated relationship incorporates primitive relationships inside itself. Primitive relationships are “actions” or structural dependencies that are also represented as KDM metaclasses. In Figure 2.8 Part **B**, they are represented by the set of arrows that connects the two layers through the `AggregatedRelationship` instance. Every `AggregatedRelationship` has a density, which represents the number of primitive relationships inside it. In this example, the density is six since it involves six relationship instances (calls, extends, creates, reads, imports, and hasType).

An important point here is regarding the types of relationships presented in KDM. Some of them have cannonical names that makes easy to understand what they really are in source code, such as: calls, extends, imports, etc. However, there are some other terms that ask for an additional explanation. The terms are:

- HasType. This type of relationship occurs when a source code element has the type of another source code element;
- UsesType. This type of relationship occurs when there is a line of code that makes a data conversion.

2.5 Chapter Summary

In this chapter we introduced the main theoretical concepts that define the context of this thesis. First, we give an overview of software architecture, its main concepts and ways to specify it. As the software evolves, it grows in complexity, became less maintainable over time and as a consequence, problems begin to arise. One of the main problems is the architecture erosion that occurs when the intended architecture tends to drift away. In order to correct it software architects apply ACC techniques to detect divergences when a CA is not conforming to the PA. We also reviewed existing tools and compare some characteristics of them with our approach.

Second, we present adaptive systems and the main abstraction that enables adaptivity in systems. MAPE-K is a well-known reference model that provides the main abstractions for designing ASs. The goal is to motivate software engineers in structuring ASs in such away that the abstractions become evident and manageable. Besides to the main abstractions, there are three that are not represented in the MAPE-K reference model, but have been 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 (VILLEGAS et al., 2011).

Finally, in this chapter we present ADM with focusing on KDM which is an important part of our approach. KDM enables the representation of the architectural viewpoint of an Adaptive System (AS), language and platform independent, and through the `AggregatedRelationship` metaclass which represent a relationship between two KDM entities, it is possible to identify architectural drifts in systems.

Chapter 3

Systematic Mapping: Architectural Anomalies in Adaptive Systems

This chapter presents a systematic mapping of architectural anomalies in adaptive systems. The questions to be answered by this revision are: What are the architectural anomalies found in ASs?; Are there architectural anomalies specific of ASs? If so, what are the main characteristics of them? and what approaches have been proposed to detect Architectural Anomalies in Adaptive Systems?.

3.1 Introduction

Architectural anomalies are defined as the implementation of unintended design decisions (PERRY; WOLF, 1992). Anomalies encompass a wide range of issues such as architectural smells (GARCIA et al., 2009), architectural violations and drifts (S. LANDI et al., 2017) and architectural antipatterns (BROWN et al., 1998). Each one of them has their own particularities, but there is a consensus in the research area that they impact negatively quality attributes such as modularity, reusability, analisability, modifiability and testability (BASS; CLEMENTS; KAZMAN, 2012).

In order to systematically manage architectural anomalies in ASs, it is necessary to have a clear and thorough understanding on the state of the art of anomalies reported in ASs. It is unclear what types of architectural anomalies are recurrent in ASs and if there are specific ones on this domain. Answering these questions would help researchers to advance the state of the art and practitioners to appraise and select techniques for dealing with them in their application context.

In this chapter, we report the results of a systematic mapping (SM) study for examining the concept of architectural anomalies in ASs. The SM was executed by considering three phases: *i*) planning, *ii*) conducting and *iv)* reporting and discussion of the results.

3.2 Planning

In this phase we have defined the protocol. This protocol contains: (*i*) the research questions, (*ii*) the search strategy, (*iii*) the inclusion and exclusion criteria and (*iv*) the data extraction and synthesis method.

The main objective of this SM is to categorize the types of architectural anomalies in ASs. The motivation for realizing a SM comes from the work published by Raibulet, Arcelli Fontana, et al. (2017), where authors identify general guidelines for the evaluation of self-adaptive systems, independent of their type, application domain, or implementation details. Particularly, we focus on findings about software adaptability at the architectural level, where metrics have been proposed to capture various aspects of an AS such as performance, design issues and architectural characteristics at design time.

Thus the motivation comes up to know what kind of design issues affect AS, if they are specific and how software engineers identify them. Therefore aiming at finding all primary studies for the understanding and summarizing of evidence about architectural anomalies, the following Research Questions (RQs) were established:

- **RQ$_1$**: What are the architectural anomalies found in ASs?
- **RQ$_2$**: Are there architectural anomalies specific of ASs? If so, what are the main characteristics of them?
- **RQ$_3$**: What approaches have been proposed to detect Architectural Anomalies in Adaptive Systems?

To define the search string, we used PICOC (Population, Intervention, Comparison, Outcome and Context) approach, adapted by (KITCHENHAM, B. et al., 2009). The definition of the PICOC elements are as follows:

- **Population:** Refers to specific Software Engineering role, category of software engineer, an application area or an industry group. In our work, we defined population as the literature about software systems that use/address properties of ASs and architectural anomalies;
- **Intervention:** Refers to a software methodology, tool, technology, or procedure. In our work, we defined intervention as the set of primary studies that found/mentioned architectural anomalies;
- **Comparison:** Not applied in this work, but we performed a snowballing forward;
- **Outcomes of relevance:** Refer to factors of importance to practitioners such as improved reliability, reduced production costs, and reduced time to market. In our work, the out-

comes were the characterization of the current state of the art about architectural anomalies related to ASs;

- **Context:** Refers to which context the comparison come from (e.g. academia or industry), the participants involved in the study (e.g. practitioners, academics, consultants, students) and the tasks being performed (e.g. small scale, large scale). In our work, in general, the context relates to practitioners and researchers who work with ASs.

Table 3.1 presents the complete set of keywords, synonyms and logic operators that were executed in digital libraries. Also, each set of keywords and synonyms belongs to a scope: the population or the intervention.

Keywords and synonyms of each scope have been associated with an OR operand and between the two scopes have been associated with an AND operand. Moreover, we included other type of systems such as context-aware, cyber-physical, mobile, embedded, multi-agent and autonomic because they also can be classified as ASs according to others SMs of the research area (MATALONGA; RODRIGUES; TRAVASSOS, 2017).

We have used the search string on the following digital libraries: ACM (portal.acm.org), IEEE (ieeexplore.ieee.org), Scopus (scopus.com) and Google Scholar (scholar.google.com).

In order to determine which primary studies are relevant to answer our research questions, we have applied a set of inclusion and exclusion criteria. The inclusion criteria are:

I1- Studies that mention or discuss architectural anomalies in adaptive systems, present a catalog of architectural anomalies or explore their impact on any facet of these type of systems;

I2- Studies presenting or using a technique aimed to the identification of architectural anomalies in adaptive systems;

Exclusion criteria utilized were:

E1- Duplicate papers or extensions of already included papers, in order to avoid possible threats to conclusion validity;

E2- Papers that are not available, as we cannot inspect them;

E3- Secondary or tertiary studies (e.g. systematic literature reviews, surveys among others);

E4- Studies in the form of editorials and tutorial, short papers, and poster, as they are deemed to not provide the required level of detail and information;

E5- Studies that have not been published in English language, as their analysis would result to be too time consuming.

Keyword	OP	**Synonyms**	OP	**Scope**
adaptive	OR	self-adaptive	OR	Population
		self-adaptiveness	OR	
		self-awareness	OR	
		self-configuring	OR	
		self-healing	OR	
		self-managed	OR	
		self-managing	OR	
		self-optimizing	OR	
		self-protecting	OR	
contex-aware	OR	context-awareness	OR	
cyber-physical		–	OR	
embedded		–	OR	
mobile		–	OR	
multi-agent		–	OR	
autonomic		–		
AND				
architectural anomaly	OR	architecture anti-pattern	OR	Intervention
		architecture antipattern	OR	
		architectural antipattern	OR	
		architectural bad smell	OR	
		architectural defect	OR	
		architectural flaw	OR	
		architectural smell	OR	
		architecture defect	OR	
		architectural debt	OR	
		architectural technical debt	OR	
		architectural drift	OR	
		architecture drift	OR	
		architectural violation	OR	
		architecture violation		

Table 3.1 – Keywords and search string

We have created data extraction forms to record information obtained by the researchers from the primary studies. The form for data extraction provides some standard information and the following fields: *(i)* name of the anomalies, *(ii)* causes of the anomaly, *(iii)* characteristic of the anomaly, *(iv)* context of the study and type of systems affected by the anomaly and *(v)* title, authors, journal and publication details.

3.3 Conducting

During this phase, the generic search string of Table 3.1 was adapted according to the specificity of each digital library. The searches were performed in four digital libraries, as suggested in Chen et al. (CHEN; BABAR; ZHANG, 2010), which are listed in Table 3.2.

Figure 3.1 presents the distribution of papers retrieved for each database after applying the inclusion and exclusion criterion. Notice that we performed two iterations; the first one was carried out on November 2019 and the second one on March 2021. On the first iteration stage 1, **398** papers were returned from the digital libraries. After applying the exclusion criteria E1, **288** papers were stayed and **110** papers were left.

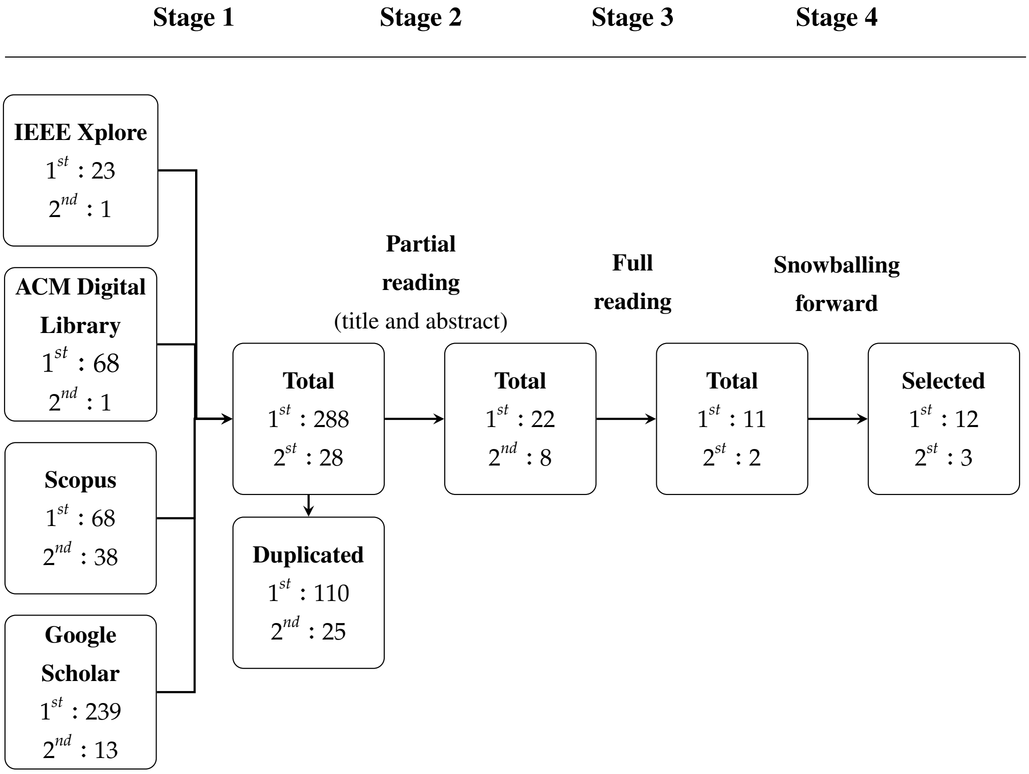


Figure 3.1 – Selection of papers

On the first iteration Stage 2, by reading title and abstract and applying exclusion criterion E2, E3, E4 and E5, **22** papers were selected. On the first iteration Stage 3, after reading full text and applying inclusion criterion I1 and I2 **11** stayed and **11** papers were left. Finally, on the first iteration Stage 4, in order to avoid lifting out relevant studies, we applied the “snowballing forward” technique to find more potentially relevant studies by checking other works that reference each selected study. After that, just one more paper was added to the final set, thus in the third stage the number of papers for data extraction were **12**. The second iteration followed the same process as the first iteration and as a result three papers were added to our systematic mapping.

Table 3.2 summarizes the number of primary Studies Obtained (SO), the number of primary Studies Included (SI) in our SM, the Index Rate (IR), and the Precision Rate (PR) of each selected source. For instance, 24 studies were recovered from ACM Digital Library and 6 studies were included in our SM. The precision rate is 25% (i.e., $\frac{6}{24}$) and the index rate is 46% (i.e., $\frac{6}{13}$). Google Scholar and Scopus showed the best index rate, since 92.33% and 61.53% of all included studies were indexed by these publication databases. In addition, Scopus and IEEE Xplore showed the best precision rate, because 7.54% and 13.04% of the primary studies recovered by these publication databases were included in our SM.

Digital library	SO	SI	IR	PR
IEEExplore	24	6	46%	25%
ACM Digital Library	69	4	30.07%	5.79%
Scopus	106	8	61.53%	7.54%
Google Scholar	252	12	92.33%	4.76%
Total	**451**	**30**		

Table 3.2 – Electronic databases searched

SO: Number of primary studies obtained;
SI: Number of primary studies included;
IR: Ratio between included primary studies of a database and the total of included primary studies in the SLR;
PR: Ratio between the total of included studies of a database and the total of obtained primary studies by this database.

Table 3.3 shows the selected publications of our SM with an ID for reference purposes. In addition, Figure 3.2 summarizes the type of publication; 20% were published in journals, $46,6\%$ were published in conferences and $33,3\%$ were published in workshops. A brief analysis of the data revealed that the studies were published in different venues of publication but it draws attention that none of them were published in the Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). An explanation for it could be that researchers are focusing on the treatment of the uncertainty according to the trends of interest in the research community (WEYNS, 2018).

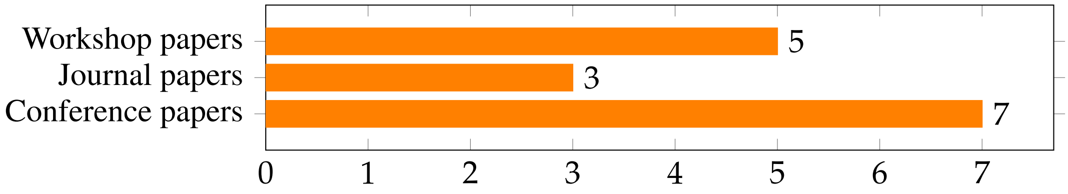


Figure 3.2 – Number of papers selected from workshops, journals and conferences

3.4 Reporting

In this section, we present our synthesized observations to each research question. In order to visualize the number of architectural anomalies found in literature we created a bubble chart in Figure 3.3. It shows the distribution of studies according to the publication year, the type of architectural anomaly and the frequency. The set of selected papers is not large and this could be explained because the research topic has not been deeply investigated.

By observing Table 3.3, most of the papers retrieved came from industry practitioners in the area of embedded system. This is aligned with the work of Weyns, Malek, and Andersson (2012), where authors state that the three main application domains for which self-adaptation has been used are services based systems, robotics and embedded systems. Thus it seems just

ID	Author	Title	Source	Year
S01	Seo et al.	Exploring the Role of Software Architecture in Dynamic and Fault Tolerant Pervasive Systems	Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments	2007
S02	Eliasson et al.	Identifying and Visualizing Architectural Debt and Its Efficiency Interest in the Automotive Domain : A Case Study	Workshop on Managing Technical Debt	2015
S03	Vogel-Heuser et al.	Evolution of software in automated production systems: Challenges and research directions	Journal of Systems and Software	2015
S04	Ampatzoglou et al.	The Perception of Technical Debt in the Embedded Systems Domain : An Industrial Case Study	Workshop on Managing Technical Debt	2016
S05	Oliveira et al.	Embedded-Software Architects It's Not Only about the Software	IEEE Software	2016
S06	Bagheri et al.	Software architectural principles in contemporary mobile software: from conception to practice	Journal of Systems and Software	2016
S07	Mera-Gómez et al.	Elasticity Debt : A Debt-Aware Approach to Reason About Elasticity Decisions in the Cloud	IEEE/ACM International Conference on Utility and Cloud Computing	2016
S08	Serikawa et al.	Towards the Characterization of Monitor Smells in Adaptive Systems	Brazilian Symposium on Components, Architectures and Software Reuse	2016
S09*	Vogelsang et al.	Characterizing implicit communal components as technical debt in automotive software systems	Working IEEE/IFIP Conference on Software Architecture	2016
S10	Mera-Gómez et al.	A Debt-Aware Learning Approach for Resource Adaptations in Cloud Elasticity Management	International Conference on Service-Oriented Computing	2017
S11	Pelliccione et al.	Automotive Architecture Framework: The experience of Volvo Cars	Journal of Software Architecture	2017
S12	Vogel-Heuser et al.	Adapting the concept of technical debt to software of automated Production Systems focusing on fault handling, mode of operation and safety aspects	International Federation of Automatic Control	2017
S13†	San-Martín et al.	Characterizing Architectural Drifts of Adaptive Systems	International Conference on Software Analysis, Evolution and Reengineering	2020
S14*	Raibulet et al.	SAS vs. NSAS: Analysis and Comparison of Self-Adaptive Systems and Non-Self-Adaptive Systems based on Smells and Patterns	International Conference on Evaluation of Novel Approaches to Software Engineering	2020
S15	Raibulet et al.	A preliminary analysis of self-adaptive systems according to different issues	Software Quality Journal	2020

* Paper retrieved by snowballing forward † Paper produced by this PhD thesis

Table 3.3 – Selected publications

in the last years academics have been paid attention to architectural anomalies in ASs but in isolated way. As we stated before, there was no works publicated in main conferences and journals of this area, such as SEAMS and TAAS[1].

Based on these 15 primary studies we have identified 27 architectural anomalies (without duplicates) in ASs. Table 3.4 shows the complete set of architectural anomalies identified on each paper and domain of the study. Although, paper **S12** presents architectural anomalies, authors describe them superficially and in a mixed way so it is not clear the conditions where

[1] Transactions on Autonomous and Adaptive Systems

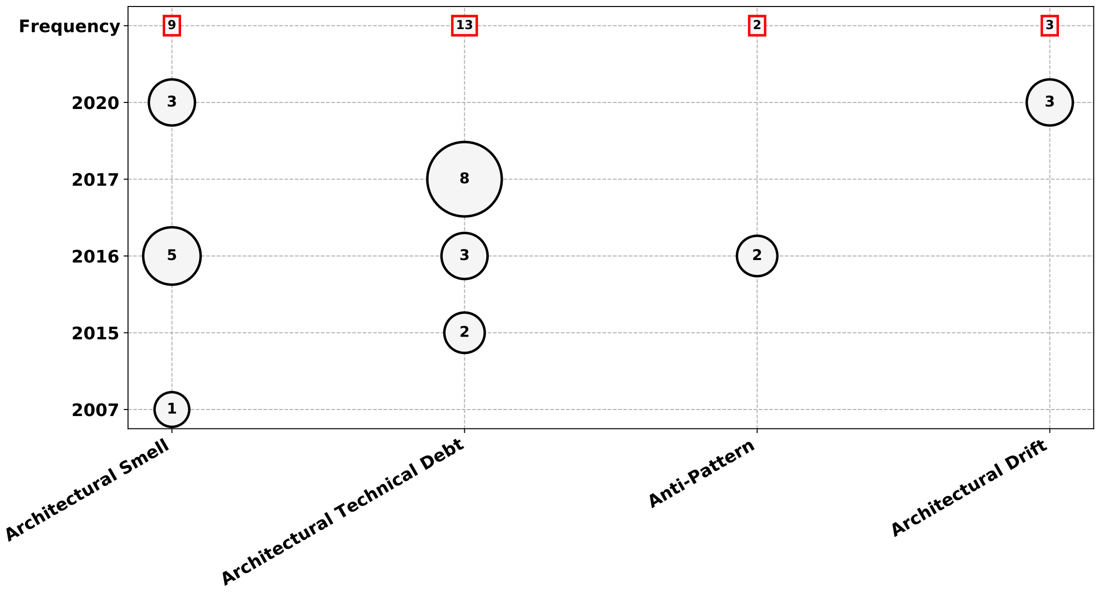


Figure 3.3 – Distribution of architectural anomalies in primary studies

they appear in real systems.

RQ$_1$: *What are the architectural anomalies found in ASs?*

As some authors did not specify clearly the type of architectural reported in their papers, we have classified the anomalies according to conventional categories found in literature. In the following we outline the anomalies found in our SM; Architectural Smell, Architectural Technical Debt, Anti-Patterns and Architectural Drifts. The last category is developed in Chapter 5 because corresponds to a contribution of this thesis.

- **Architectural Smell**

  Seo et al. (2007) proposed a dynamic software architecture for pervasive systems that are mission critical and have stringent fault tolerance requirements. The application scenario is composed of sensors, gateways, hubs and mobile devices. Sensors monitor the environment around them, communicate their status to one another and to the gateways. Gateways manage and coordinate sensors. Hubs are used to evaluate and visualize the sensor data for the users, as well as provide an interface through which the user can send control commands to the various sensors and gateways in the system. Mobile devices are used by users to make decisions.

  The reason to propose the new architecture came to the fact that architecture components can fail due to two problems; *i*) finite batteries lives and *ii*) the wireless is susceptible to permanent or temporary disconnections. Hence the software system should support autonomic fail-over: if a host providing a given service fails, then another host in the system should quickly be enabled as a new provider of the same service (SEO et al., 2007).

ID	Architectural Anomaly	Type	Domain
S01	1. Coupled Components	Smell	Embedded Systems
S02	1. Misplaced Logical Components	ATD	Automotive
S03	1. Violation of architectural constraints	ATD	Manufacturing
S04	1. Violation of architectural constraints	ATD	Embedded Systems
S05	1. Extraneous Connector; 2. Scattered Functionality.	Smell	Automotive
S06	1. Inconsistent hierarchical (de)composition; 2. Extensibility limitations; 3. Omission of architectural concepts; 4. Broadcast receiver's connector envy.	AP AP Smell Smell	Mobile
S07	1. Elasticity Debt	ATD	Cloud Systems
S08	1. Obscure Monitor; 2. Oppressed Monitors.	Smell Smell	Mobile
S09	1. Implicit communal components	ATD	Automotive
S10	1. Elasticity Debt	ATD	Cloud Systems
S11	1. Violation of architectural constraints	ATD	Automotive
S12	1. Violation of the five layer module model for modularization. 2. No hierarchy in automatic mode; 3. Inappropriate modularity; 4. Suboptimal hierarchy in the field of fault handling; 5. Poor architecture of safety-related parts of the software; 6. Structure of code does not allow to update alarm numbers.	ATD	Machinery and Plant Manufacturing
S13	1. Scattered Reference Inputs 2. Obscure Alternatives 3.Mixed Executors and Effectors	Drift Drift Drift	Several Domains
S14	1. Unstable Dependency 2. Hub-Like Dependency 3. Cyclic Dependency	Smell Smell Smell	Several Domains
S15	1. Unstable Dependency 2. Hub-Like Dependency 3. Cyclic Dependency	Smell Smell Smell	Several Domains

Table 3.4 – Architectural anomalies per paper

As these are embedded systems, in nature they are resource-constrained so the existing approaches for providing fault tolerance on traditional desktop platforms are often inefficient in this domain (SEO et al., 2007). Moreover, many times the implementation of an advanced fail-over support (i.e., replication, synchronization, and recovery) is coupled with the application logic and as a consequence, it generates difficulties to engineers when performing maintenance tasks.

Garcia et al. (2009) define an architectural smell as "*a commonly (although not always intentionally) used architectural decision that negatively impacts system quality*". They may be caused by applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity (GARCIA et al., 2009).

Therefore we classified this issue as an architectural smell and named it as Coupled Components, because the decision of designing the fail-over system coupled with the application logic affects the maintenance quality attribute.

Vogelsang, Femmer, and Junker (2016) affirms that computer scientists' knowledge of embedded-systems concepts such as controllers and actuators is usually limited. So, the role of embedded-software architect is often played by engineers from other fields who have a limited education in software architecture. Thus, as consultancy they observed that two recurrent problems arose with embedded-software architects; *i*) Incompleteness and Inconsistencies Due to Missing Traceability and *ii*) Architectural Smell.

The first one points out that embedded-software architects neglect the fundamental traceability that should exist between the architecture drivers and architecture design. The second one points out that design decisions negatively impacted systems quality (VOGELSANG; FEMMER; JUNKER, 2016).

Two architectural smells are reported; *i*) Extraneous Connector and *ii*) Scattered Functionality. The first one occurs when two connectors of different types connect a pair of components (GARCIA et al., 2009). The second one occurs when multiple components realize the same high-level concern and some of them are responsible for orthogonal concerns (GARCIA et al., 2009).

Figure 3.4 shows the AS. It is composed of two sensors: a wheel rotation speed and a front ultrasonic distance sensors. The data is requested by the Vehicle Sensor Processor component which computes some measures by using the obstacle distance parameter from the Cruise Control component. The data is sent to the CAN bus and the Transmission Control Unit uses it to perform an adaptation such as slow down the vehicle.

The Extraneous Connector occurs because of the additional direct connection between the Vehicle Sensor Processor and the Cruise Control. This architectural smell also implies a deployment constraint because the vehicle speed control and cruise must be deployed on the same ECU.

Bagheri et al. (2016) argue that the ADL-like manifest of Android app does not allow the specification of key elements that are typically found in traditional ADLs such as connectors, configurations resulting from interconnections between components and connectors and required interfaces of components or connectors. As a consequence, the omission of these architectural concepts hinder architectural analysis and understandability of an app. As mobile apps can also be adaptive systems (SERIKAWA et al., 2016; SIQUEIRA; JÚNIOR; FERRARI; SANTIBANEZ; MENOTTI; CAMARGO, 2018), the rigid manifest cannot allow the specification of MAPE-K abstractions making difficult the system understanding.

Another architectural smell reported by (BAGHERI et al., 2016) is the Connector Envy (GARCIA et al., 2009). Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. These broadcasts are sent when an event of interest occurs. The problem is they poorly separate concerns, resulting in deficiencies that affect maintainability and effi-

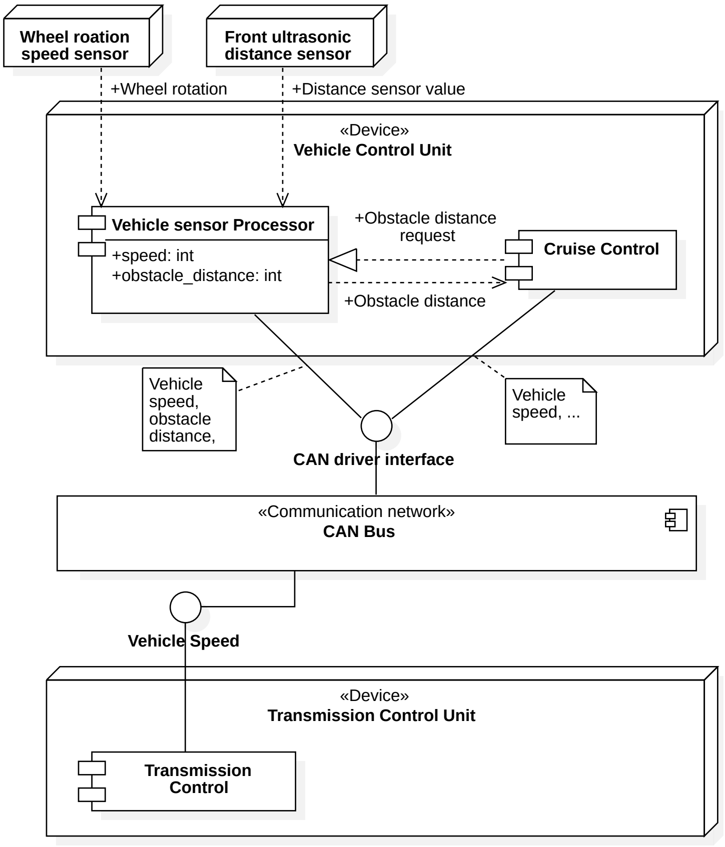


Figure 3.4 – Extraneous Connector smell

ciency (BAGHERI et al., 2016). This poor separation of concerns stems from the fact that Broadcast Receivers tend to include application-specific logic, which are the responsibility of components, and distribute Intents to other components.

Serikawa et al. (2016) report two architectural smells; Obscure Monitor and Oppressed Monitors. The first one occurs when a monitor is not implemented as a first-class entity, making the source-code of this facet tangled with the source-code of other components. In object-oriented systems, that means there is not a class or interface that represent the monitor, but a set of lines of code inside arbitrary classes or methods. The second one occurs when a set of monitors exhibits the following three main characteristics: *i*) they are independent from each other concerning the data manipulated; *ii*) they have the same polling rate and *iii*) the execution order of the monitors is predetermined in compilation time and remain unmodified in runtime.

Raibulet, Fontana, and Carettoni (2020a) and Raibulet, Fontana, and Carettoni (2020b) report the same three architectural smells; *i*) Unstable Dependency, *ii*) Hub-Like Dependency and *iii*) Cyclic Dependency. According to authors, the first one occurs due to the fact that ASs must adapt and change their structure and/or behavior based on changing internal or external system variables; this means that the various classes and packages of the system have to be able to maintain the highest possible degree of flexibility.

The second one occurs due to the fact that ASs operate in dynamic environments so they must interact with several components or controllers and they need to be able to analyze and adapt changes in the surrounding environment simultaneously to ensure the quality of the provided services. For the third one authors do not give an explanation.

- **Architectural Technical Debt (ATD)**

  The concept of technical debt is a metaphor used to encapsulate numerous software quality problems that if they are not addressed in time they could get worse in future (ERNST et al., 2015). Eliasson et al. (2015) present the Misplaced Logical Components ATD. This ATD occurs when Logical Components (LCs) are deployed on different Electronic Control Units (ECU) than the ones that were intended by the architects, resulting in non-allowed dependencies between different domains. As a consequence, this violation may result in a change in the amount of communication over the network and have an impact on efficiency.

  Vogel-Heuser, Fay, et al. (2015) present the problems and challenges that faces automated production systems (aPS). This kind of systems are comprised of mechanical parts, electrical and electronic parts (automation hardware) and software, all closely interwoven, and thus represent a special class of mechatronic systems (VOGEL-HEUSER; FAY, et al., 2015). For instance, the change of a feedback controller in the control model could result in characteristics of system inputs (e.g., limits on the value or change rates), that the employed mechanical actuactor might not be able to satisfy. A similar problem arises if a sensor is replaced in the mechanical system that has different quality characteristics, which affect the quality of a feedback controller of the software.

  A typical architectural debt in this domain is the non-compliance between architectural guidelines and the system architecture. Particularly, the violation of the Operator Controller Module (OCM) pattern, which defines concrete layers and interfaces between them for different parts of the embedded software – feedback controllers, hard real-time communication and reconfiguration of the feedback controllers, and a soft real-time layer.

  Ampatzoglou et al. (2016) state that a typical ATD in the embedded-systems domain are architectural debts. This occurs when engineers do not follow the guidelines of the initial architecture so they end up implementing new requirements or performing maintenance tasks that do not conform the planned architecture. A possible cause of the debt occur because some runtime quality attributes are given a higher priority than maintainability. Specifically, the embedded systems domain prioritizes reliability, functionality, and performance against maintainability.

  Mera-Gómez et al. (2016, 2017) present the elasticity debt in cloud systems. It occurs when there is a valuation of the gap between an optimal and an actual adaptation decision. The cause is the difficulty in predicting resource demand, coarse computing resource granularity, elapsed time between computing resources are acquired and when they are effectively ready to be used.

Vogelsang, Femmer, and Junker (2016) present the Implicit Communal Components ATD. It occurs when developers (re)use any signal that is available on the bus system to implement or adapt a feature, regardless of the origin of that signal. In many cases, the signals a developer (re)uses originate from the implementation of another feature, however, the developer is usually not aware of that.

Pelliccione et al. (2017) state the existence of a gap between prescriptive and descriptive architecture that causes an architecture degradation in Volvo cars. This degradation might show up in two different ways: *i*) architectural drift when the descriptive architecture includes changes that are not included in, encompassed by, or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture's design decisions; *ii*) architectural erosion is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture.

Vogel-Heuser and Neumann (2017) report a technical debt classification based on (LI; AVGERIOU; LIANG, 2015) that shows six architectural debts in aPS; Violation of the five layer module model for modularization; No hierarchy in automatic mode; Inappropriate modularity; Suboptimal hierarchy in the field of fault handling; Poor architecture of safety-related parts of the software; Structure of code does no allow to update alarm numbers.

- **Anti-Patterns**

Bagheri et al. (2016) report two anti-patterns: Inconsistent hierarchical (de) composition and Extensibility limitations. The first one is about that Android provides constructs with arbitrarily different structures, behaviors, and semantics at different levels of hierarchy. Similarly, the rules of composition at one level are completely different from another level. Software-architecture research has advocated for recursive rules of (de)composition, whereby a component can be decomposed into lower-level components of the same type and vice versa (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). One benefit of such an approach, is that the same set of composition rules can be applied at different levels of hierarchy.

The second one points out that Android provides limited support for building other component types that do not fit the semantics of Android's predefined components. Connectors in Android face similar extensibility limitations. They lack the ability to build configurations corresponding to certain topologies. For instance, Android lacks support for isolating groups of components into their own layers where each layer is separated by a different event bus. Such a design provides loose coupling and separation of concerns at a granularity above Android components (BAGHERI et al., 2016).

The architectural anomalies extracted from the selected papers belong to different domains but the prevalent is embedded systems. Even though automotive systems are relevant in this SM a closer look shows that the anomalies found in this domain occurs in embedded systems because this kind of systems are composed of embedded subsystems. It is important to point out that some reported adaptive systems were developed by different specialists with different architectural viewpoint.

For instance, in the developing of adaptive systems of aPS and automotive domains were involved software engineers, mechanical engineers and electrical engineers where each one of them tend to focus on isolated parts of the architecture design. As a consequence, the whole architecture specification is often incomplete and inconsistent, which results in issues such as architectural anomalies (VOGELSANG; FEMMER; JUNKER, 2016).

Another matter that draws attention is the fact that almost none of the selected works report the abstractions involved where the anomalies are present in the system. We think this information is important because if the anomaly always occurs in the same set of abstractions with few variations, software architects can document it and effectively address it by means of refactoring techniques/tools.

RQ$_2$: Are there architectural anomalies specific of ASs? If so, what are the main characteristics of them?

According to our findings we conclude that there are architectural anomalies reported in literature that are specific for adaptive systems, but they can be generalized with existing others. As we state in RQ$_1$, there are several domains where ASs can actuate but few works reported with details where the anomalies occurs. Moreover, none of the works explicitly recognize the MAPE-K as a key abstraction that performs a logical or structural adaptation in the whole system, with the exception of the studies presented by Serikawa et al. (2016), Santibanez, Siqueira, de Camargo, and Ferrari (2020), Raibulet, Fontana, and Carettoni (2020a) and Raibulet, Fontana, and Carettoni (2020b). Thus, it could indicate that in most cases software architects are not awareness of the abstractions involved in the system adaptation or maybe they do not recognize them as a first-class citizen abstractions so end up implemented hidden in the system architecture.

The two most reported types of architectural anomalies are ATD and Architectural Smells. Nevertheless, regarding to ATD it is very common that authors refer it as a violation of architectural rules, which is a generic term for architectural drifts (NORD et al., 2012). Regarding to Architectural Smells authors report well-known smells cataloged by Garcia et al. (2009) and Fontana et al. (2016) such as Unstable Dependency, Hub-Like Dependency, Cycle Dependency, Extraneous Connector and Scattered Functionality. The main characteristic of these smells is that they typify modularization problems. Indeed, according to the classification of Suryanarayana, Samarthyam, and Sharma (2014) most of them fit in the modularization design principle scheme.

In spite of an Architectural Smell is defined as “a commonly (although not always intentionally) used architectural decision that negatively impacts system quality”, in literature we can find other definitions. For instance, Zimmermann (2015) states that an Architectural Smell are suspicions or indications that something in the architecture is no longer adequate under the current requirements and constraints, which might differ from the original ones. This definition is very close to the definition of an Architectural Drift. Moreover, smells can occurs when due to the non-adherence to best practices and process, violation of design principles and inappropriate use of patterns among others (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Therefore, we can conclude that researchers use already known architectural anomalies to characterize in a generic way problems found in the architecture of ASs. Thus the lack of a catalog of architectural anomalies specific of ASs without using terms of the AS domain add more difficulties when software architects need to identify anomalies because generics one do not capture the essence of the problem that exist in a specific domain (VELASCO et al., 2018).

RQ$_3$: What approaches have been proposed to detect Architectural Anomalies in Adaptive Systems?

In order to detect Architectural Anomalies researchers use automated tools for this purpose because manual techniques are human-centric, tedious, time-consuming, and error prone. These techniques require a great human effort, therefore not effective for detecting smells in large systems (PEREIRA DOS REIS et al., 2021). Nevertheless, most of the works reported by our SM do not specify if they used automated tools or manual techniques.

Particularly, Raibulet, Fontana, and Carettoni (2020b) inform that they used Arcan to collect Architectural Smells. This tool can detect *i*) Cyclic Dependency (detected on classes and packages) refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem’s dependency structure (RAIBULET; FONTANA; CARETTONI, 2020b); *ii*) Unstable Dependency: (detected on packages) describes a subsystem (component) that depends on other subsystems that are less stable than itself, according to the Instability metric value (RAIBULET; FONTANA; CARETTONI, 2020b); and *iii*) Hub-Like Dependency: (detected on classes) this smell arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions (RAIBULET; FONTANA; CARETTONI, 2020b).

3.5 Chapter Summary

This chapter presented a systematic mapping of architectural anomalies of ASs. From 15 selected works we extracted 27 architectural anomalies and most of them reported from embedded and automotive domains. Although researchers have been worried of detecting architectural anomalies and knowing their implications at mid long-term they still report them in a generic way without taking into account ASs domain particularities.

A possible cause is that stakeholders are unconscious about ASs abstractions or they do not identify them as first-class citizen with the exception of sensors and actuators, so the implementation of the adaptive part of the system ends up scattered and tangled with the business logic. As a consequence, software architects do not use specific techniques/tools for identifying architectural anomalies of ASs, but tools that identify common generic anomalies.

Therefore, there is a lack of approaches that effectively identify architectural anomalies in adaptive systems that take into account the particularities of the domain. In spite of the SM reported ATD (most of them related to constraint violations), they are not comprehensive enough. So we see an opportunity of research that can help researchers and practitioners interested in software architecture and dealed with adaptive systems to characterize and identify architectural drifts.

Chapter 4
RELATED WORKS

In this chapter, we survey the state of the art in supporting architecture conformance checking by reviewing existing approaches in terms of their main characteristics and comparing them to our proposal.

4.1 Architecture Conformance Approaches

In this section we review ACC approaches that are related with our work. According to our non-systematic review, the approaches can be divided in two main groups; non-extensible and extensible. The approaches categorized in the first group can not be extended with new concepts and abstractions to the language. On the other hand, the approaches categorized in the second group allow extensions of new concepts and abstractions to the language.

Regardless of whether an approach is categorized as non-extensible or extensible, it can hold characteristics that characterize it as a generic or domain-specific. Generic approaches use a vocabulary that is capable to model software architectures in a wide range of contexts because the atomic entities of the language are semantically correct on those contexts.

Regarding to Domain-Specific approaches, they use a specialized vocabulary that belongs to a specific domain so the semantic of atomic entities of the language would not make sense in other ones. Another key characteristic is that they hold the domain knowledge in terms of rules that govern the domain. Thus, software architects could enable of disable them according to certain particularities without to create them from the scratch.

Extensible approaches have slightly stronger relationship with our approach because they can be extended with Adaptive System (AS) abstractions but the domain rules must be created by software architects in any case. Our approach is considered as non-extensible domain-specific for ASs

4.1.1 Non-Extensible Approaches

A variety of approaches on ACC have been developed in order to detect architectural drifts. Once identified, the aim is to refactor the CA to reconcile it as was initially designed. The main characteristic of generic approaches is the use of a generic vocabulary to describe architectural abstractions such as entity, layer, module and subsystem. Moreover, some techniques use vocabulary that relies on source-code concepts such as packages and classes to denote components or modules for specifying the intended or planned architecture (BRUNET et al., 2012; GURGEL et al., 2014).

One of the first approaches developed for detecting architectural violations is reflexion models (HOLT et al., 2000). This technique compare two models; a high-level model which contains entities and relations between these entities and a low-level model commonly created from source code and represented as a call graph. A mapping has to be performed between the two models where entities from the high-level model have to be assigned to elements of the source model, typically using regular expressions. Based on these models and the mapping, a software reflexion model is computed to determine where the high-level model does (not) comply with the source model (HOLT et al., 2000; MURPHY; NOTKIN; SULLIVAN, 1995).

Some tools that have been implemented with this approach were presented by Brunet et al. (2012) and Herold et al. (2015). The first one uses *Visual diff* tool[1] and manual inspection to detect architectural drifts. It detects violations produced by method calls, field access, generalization, exceptions, returned types and received parameters. The second one presents *JITTAC*[2] that performs comparisons between models and detect violations related to accesses and calls of packages, classes and methods in Java.

Dependency structure matrix (DSM) is based on a square matrix, which the intersections among lines and columns denotes a relation between components (classes) in an object-oriented system (BALDWIN; CLARK, 2000; SANGAL et al., 2005). It visualizes in matrix how much a component is associated with another one. It also supports the grouping of components into modules, which facilitates analysis among component relations, allowing architects to work with DSM in a hierarchical way, which they could use to analyze the architecture in different abstraction levels. DSM can also be used in conjunction with architectural rules. An implementation of this approach is *Lattix Architect*[3], which has a graphical interface showing the dependencies in a system through a DSM. Architects can define new constraints in which they allow or forbid dependencies between different types of entities (e.g., interfaces, classes, packages). Developers can identify rule violations and cycles by visually navigating the reverse engineered DSM.

Other approaches are based on queries for detecting architectural drifts such as *CQLink*[4] and

[1] https://w3.cs.jmu.edu/bernstdh/web/common/tools/diff.php

[2] http://actool.sourceforge.net/

[3] http://lattix.com

[4] http://www.ndepend.com

Semmle .QL[5]. They are based on a SQL-inspired syntax where a query consists of a conditional select statement and code entities that corresponds to types, methods, namespaces among others which are compared with specific values. This technique works only at source code, which is a low-level abstraction so it lacks of adequate support for the representation of high-level concepts, with the consequence that it may be difficult to understand the architecture.

ArchLint (MAFFORT et al., 2016) is a well-known approach to verify the architectural conformance that uses some heuristics implemented as SQL queries. The technique requires a control version system (code history) and a high-level document modeling the system. Then, it combines static code analysis, change history analysis, and a set of heuristics for detecting absences and divergences to classify architectural dependencies or the lack of them.

On the other hand, there are solutions that uses a textual notation that are capable of checking rules specified in a dedicated DSL. *DCL 2.0* (ROCHA et al., 2017), *DCL-KDM* (S. LANDI et al., 2017), *TamDera* (GURGEL et al., 2014) and InCode.Rules (MARINESCU; GANEA, 2010) are designed to define constraints on code dependencies (e.g., accesses, declarations, extensions).

InCode.Rules can also be used to identify classes affected by specific design flaws (e.g., god class, data class). DCL 2.0 allows the user to define rules as single statements with a clearly defined syntactical structure. TamDera allows the user to define hierarchical concepts, which slightly improves the modularity of the specification. InCode.Rules supports rule composition: each rule can be used to define an exception to another rule. DCL-KDM (S. LANDI et al., 2017) is an extesion of DCL 2.0 and supports architects in the specification and serialization of PAs to be used in ADM-based projects. Particularly, the specification and serialization of PAs with the original version of KDM modles and Structure Package.

Textual notation tools are characterized by high usability and a well defined strict specification language. The authors of DCL 2.0 claim that their language is more usable than other logic inspired alternatives. Those are supposedly based on a more complex and heavyweight notation and offer poor performance. Other researchers recognize the difficulty that typical users encounter when approaching solutions that require a basic understanding of logic programming (LOZANO; MENS; KELLENS, 2015). Rocha et al. (2017) also compare DCL 2.0 to alternative solutions based on refection models and dependency structure matrices, stating that their language is more expressive, reusable and handles a wider set of constraint types. These type of languages are declarative and do not require any specific programming skill and rules can be defined by using the constructs offered by the supported notation.

Velasco et al. (2018) present an initial effort to characterize a set of architectural smells relevant to the Model-View-Controller (MVC) architectural style. These smells correspond to access among layers that must be forbidden according to the MVC architectural style. Table 4.1 describes a categorization of smells relevant to the MVC architectural style. Notice that these smells can be treated as architectural drifts because all of them are access violations among

[5] https://semmle.com/

layers.

Name	Description
Model includes View's computations and/or data	Happens when the Model contains presentation of data of end-user requests (e.g. HTML code).
Model includes Controller's computations and/or data	Happens when the Model has direct access to variables that represent the end-user's request (i.e. direct access to $_GET, $_POST variables).
View includes Model's computations and/or data	Happens when the Controller has domain logic (e.g. code of DB queries).
View includes Controller's computations and/or data	Happens when the View has direct access to variables that represent the end-user's request (i.e. direct access to $_GET, $_POST variables.).
Controller includes View's computations and/or data	Happens when the Model contains presentation of data of end-user requests (e.g. HTML code).
Controller includes Model's computations and/or data	Happens when the Controller has domain logic (e.g. code of DB queries).

Table 4.1 – Categorisation of smells relevant to the MVC architectural style (VELASCO et al., 2018)

In order to detect these violations a static analysis tool (*PHP_CodeSniffer*) was implemented that "sniffs" PHP code files according to a set of rules defined in a coding standard document. It works by tokenising the contents of a code file into building blocks. These are then validated through the use of text analysis to check a variety of aspects against the coding standard in question. In this context, a coding standard can be seen as a set of conventions regulating how code must be written. These conventions often include formatting, naming, and common idioms. Multiple coding standards can be used within *PHP_CodeSniffer*. After the analysis process, *PHP_CodeSniffer* outputs a list of violations found, with corresponding error messages and line numbers.

4.1.2 Extensible Approaches

In this section we discuss extensible approaches for architecture conformance checking. As we state before, this kind of approaches is a little bit more related to ours because it is possible to create new domain abstractions as AS abstractions. The main advantage of this type of approaches is that they are inherently extensible within the boundaries set by the underlying language model. Users can define new concepts by declaring and combining facts and predicates.

This form of extensibility allows developers to adapt the notation to the specific vocabulary required to describe their architecture. Nevertheless, according to Caracciolo, Lungu, and Nierstrasz (2015) the usability is compromised because these kind of languages involves programming capabilities which typically go beyond the skills possessed by average software engineers.

Languages like *SOUL* (MENS; WUYTS; D'HONDT, 1999), *LogEn* (EICHBERG et al., 2008) and SCL (HOU; HOOVER, 2006) are examples of solutions that can be used for conformance checking. *SOUL* is a Prolog-inspired internal DSL implemented in Smalltalk. A set of predefined high-level predicates can be used to create architectural rules or define new predicates. Pre-defined predicates are evaluated using dedicated analyzers. The representation of the target architecture can be enriched by adding new facts to the fact base.

LogEn is an internal DSL implemented in DataLog, a subset of Prolog. Rules and generic predicates are conceptually specified in the same way as in SOUL. Facts are automatically extracted from the source code using a static analyzer. Source code entities can be grouped in logical sets (called ensembles) programmatically using a dedicated predicate or declaratively using specific annotations in the analyzed code. *SCL* is an external DSL inspired by OCL. The language is used to define first-order logic formulas that can be automatically evaluated against the source code of a target system. Users can express structural constraints in a declarative and language-independent notation using pre-defined functions and predicates.

Caracciolo, Lungu, and Nierstrasz (2015) propose a solution that that aims at combining the practical utility of existing compliance checking tools with some characteristics of Architectural Description Languages (ADLs). Architectural rules can be specified through a domain specific language (DSL) and automatically verified through external off-the-shelf analysis tools. The DSL can be further extended as new concerns, and consequently tools, are supported. The logical steps required to evaluate user-defined rules are encoded in purpose built tool adapters. The authors claim most of the tools that performs architectural conformance checking are at best used to provide basic support information during manual tasks. Developers still heavily rely on manual reviews as a means to check architectural conformance. The use of manual techniques does not scale and entails additional costs that could be minimized by automating parts of the process and using existing solutions and technologies. Moreover, according to their observations many tools provide insufficient documentation material so practitioners end up discarding them because they do not fulfill with industry requirements and the adoption curve is steep.

The approach consists of two subsystems; *Dictõ* and *Probõ*. The first one is a DSL for the specification of architectural rules. The language aims at supporting software architects in formalizing and testing prescriptive assertions on functional and non-functional aspects of a software system. The second one is a tool coordination framework that verifies rules written with Dicto using third-party tools. Supported tools and analyzers are managed through custom crafted adapters (CARACCIOLO; LUNGU; NIERSTRASZ, 2015).

In Listing 4.1, lines $1-4$ show the mapping between the symbolic entities used in rules and the corresponding concrete entities present in the system. In this example, symbolic entities are *Test*, *View*, *Model* and *Controller*. They have a type and properties. Concrete entities are name of packages or classes of a system and wildcards can be used to select multiple entities. Line 5 shows a rule where all subject entities *can only* depend on the object entities.

The rules written in Dictõ are validated by Probõ and normalized to generate predicates, which are evaluated though third-party tools with custom adapters. The adapters are assigned to predicates according to a set of pre-defined syntactic matching criteria specified in the adaptor class. They are responsible of generating a test specification that, once executed, produces sufficient information to evaluate the predicates they are assigned to (CARACCIOLO; LUNGU; NIERSTRASZ, 2015). Listing 4.2 shows the rules created by Probõ with the input of Dictõ.

```
Test = Package with name:"com.app.Test"
View = Package with name:"com.app.View"
Model = Package with name:"com.app.Model"
Controller = Package with name:"com.app.Controller"
Test, View can only depend on Model, Controller
```

Listing 4.1 – Mappings between symbolic and concrete entities (CARACCIOLO; LUNGU; NIERSTRASZ, 2015)

```
depend-on(com.app.Test,com.app.View)
depend-on(com.app.Test,com.app.Controller)
depend-on(com.app.Test,com.app.Test)
depend-on(com.app.Test,com.app.Model)
```

Listing 4.2 – Predicates generated by Probõ (CARACCIOLO; LUNGU; NIERSTRASZ, 2015)

Rules are essentially predicates related to a variable number of subjects through the use of modal verbs (e.g., must, can). Particularly, this approach support *must* and *cannot* with its variations *only can* and *can only*. If one of predicates fails then Probõ can conclude that the original rule is not correctly enforced in the target system.

4.2 Comparison

Table 4.2 presents a comparison among the most representative approaches described in this chapter. It includes the approach, abstractions that can be represented by the approach, how is represented the PA and the CA, what mechanisms are used to detect the drifts and type of drift detected by the approach.

Although we do not find in literature approaches/techniques of ACC used in AS domain, it is clear that non-extensible and extensible approaches can be used for that purpose. Regarding to non-extensible approaches they use generic terms to specify the system architecture where some of them use concepts near to source-code (low-level of abstractions) and others use concepts such as subsystems, components and layers (high-level of abstraction). Also, the majority of them perform the conformance checking in systems developed in Java.

The exception is the work presented by Landi, Santibanez, Santos, Cunha, Durelli, and Camargo (2022) that present Arch-KDM an ACC that uses DCL-KDM to specify the Planned Architecture (PA). The Current Architecture (CA) is obtained by mapping subsystems, components and layers to souce-code elements. In both cases a Knowledge Discovery Metamodel (KDM) model instance is generated; one for the PA and another for the CA. These instances are compared and as a result a list of drifts is presented to the software architect. As the conformance checking is performed in the KDM instance, which can represent the architectural

Approach	Abstractions	PA representation	CA representation	Drift Detection	Type of Drifts
(BRUNET et al., 2012)	Packages, Classes, Methods	Model from Documentation	System Code	Visual Diff Tool & Manual Inspection (Reflexion Model)	Method Calls, Field Access, Generalization, Catched and Thrown Exceptions, Returned Types and Received Parameters
(GURGEL et al., 2014)	An abstraction called *Concept*. It represents Classes, Interfaces, Methods and Fields	First-Order Logic Formula	System Code in Knowledge Base	Prolog Queries	Method Calls, Creates, Declares, Extends, Implements, Exception, Handling, Dependency
(HOLT et al., 2000)	Subsystems, Modules and Entities	System hierarchy with Subsystems and Modules	System hierarchy with entities (files)	PBS tools, lift	Unexpected Dependencies & Gratuitous dependencies
(MURPHY; NOTKIN; SULLIVAN, 1995)	Modules and Calls among them	High-Level Model	System Code	Reflexion Model	Convergence, Divergence and Absence of Calls
(HEROLD et al., 2015)	Packages, Classes, Methods	Package Model	System Code Model	Reflexion Model	Architecturally misplaced software units & Architecturally divergent callbacks
(MAFFORT et al., 2016)	Packages, Classes	–	Component Model	SQL Queries	method calls, variable declaration, inhertance, exceptions among others
(ROCHA et al., 2017)	Component	Textual/DSL	–	Dependency Checker in Source Code	method calls, reading/writing field, creates, extends, implements, interfaces, annotations, exceptions, handles and derives
(S. LANDI et al., 2017)	SubSystems, Modules, Layers	Textual/DSL creates a KDM instance	KDM instance	Comparisons between the PA and the CA models	method calls, reading/writing field, creates, extends, implements, interfaces, annotations, exceptions, handles and derives

Table 4.2 – Comparison of approaches of ACC

viewpoint of the system in a way that is language and platform independent, in theory the approach could be applied to systems developed in any programming language and software architects would only need the parser to transform the source-code into a KDM model.

Regarding to extensible approaches all of them are language dependent so they cannot be reusable in systems developed with different programming languages. As we stated before, although these kind of language provides flexibility because they allows the creation of new concepts and abstractions, they cannot be extensible in terms of supporting preconceived domain rules due to the dependency with the new created abstractions. Moreover, the usability is impaired because software engineers when approaching this kind of language, most of the time have to get out of their comfort zone and learn a new programming paradigm such as declarative or logical language (LOZANO; MENS; KELLENS, 2015).

4.3 Chapter Summary

In this chapter we depicted the main approaches currently available for architecture conformance in a non-systematic way. These approaches can be categorized according to the techniques implemented for detecting architectural drifts; constrained, tool-specific languages with support for simple and moderately complex predicates, formal languages that support the specification of rules as first-order predicates, where its extensibility is limited to the boundaries inherited from the underlying logic formalism. Some approaches are used to define contracts directly within source-code while others can be used to define complex architectural rules but the use is limited to developers with specific technical skills.

Neither the approaches above mentioned uses a domain vocabulary nor prescribe domain rules for detecting architectural drifts in ASs. As it was observed by Boutekkouk (2021), generic approaches fail to model characteristics that are domain specific so there are two directions: *i*) The use of domain specific ACC instead of general purpose ACC and *ii*) Using extensible ACC approaches and frameworks to generate domain-specific ACC. Moreover, although extensible approaches can be used to support AS domain vocabulary, they are unable to check predefined domain rules without specifying them in order to reduce efforts of specifying the PA and at the end, improve productivity.

In the remainder of the thesis, we investigate architectural anomalies of AS. Based on our observations, we design REMEDY, an intuitive and executable DSL which can be used to specify the adaptive part of an AS architecture. Also, it checks architecture conformance by predefined domain rules and custom rules.

Chapter 5
Architectural Drifts of Adaptive Systems

In this Chapter we present the characterization of three drifts that are recurrent in some ASs; Scattered Reference Inputs (SRI), Obscure Alternatives (OA) and Mixed Executors and Effectors (MEE). 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.

5.1 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 the 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. (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

5.1.1 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:

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)

Table 5.1 – Examples of Adaptive Systems containing the Proposed Drifts

clear documentation and all system resources should be available for execution. As a result, the systems shown in Table 5.1 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 (BRUN et al., 2009) (HEBIG; GIESE; BECKER, 2010) (VILLEGAS et al., 2011). 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. (VILLEGAS et al., 2011), Weyns et al. (WEYNS; MALEK; ANDERSSON, 2012), and IBM (IBM, 2005). 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 5.1.

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 5.1 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.

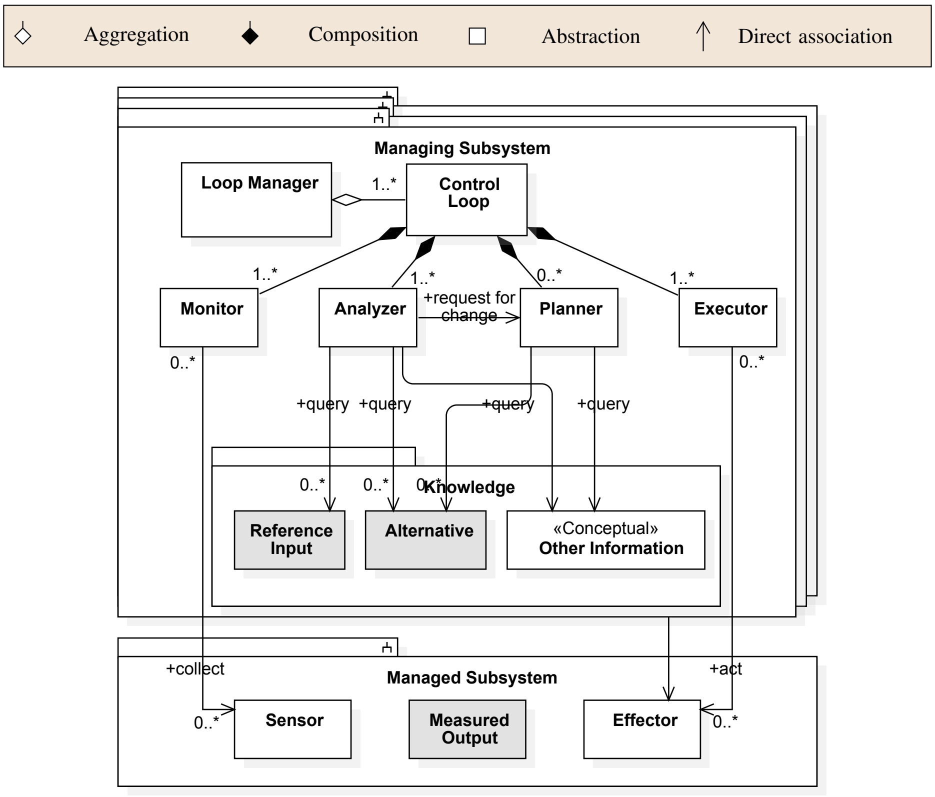


Figure 5.1 – MAPE-K enriched with lower-level abstractions

Normally, a system that is considered adaptive is composed of the Managed Subsytem, which is the bigger base part, and one or more Managing Subsystems, which are modules responsible for performing the adaptations. Notice that MAPE-K is much more devoted to design the adaptation parts than the base system itself. Control Loops (CLs) could be declared in an existing Managing Subsystem or in Loop Managers. The latter occurs when two or more Control Loops (not necessarily deployed in the same location) need some kind of interaction for achieving the adaptation goal.

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 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 (ARBOLEDA et al., 2016). 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 (FARAHANI; NAZEMI; CABRI, 2016).

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 (IBM, 2005).

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 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 and dynamic analysis with automated tools and a manual analysis for searching the abstractions of the reference model of Figure 5.1 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 5.1.2. Table 5.2 presents the template used in this work to document the drifts.

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.

Table 5.2 – Architectural drift template

5.1.2 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.

5.1.2.1 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. 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 (SHARMA; SPINELLIS, 2018).

5.1.2.2 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; (*ii*) modifying an alternative when the purpose of an adaptation needs to change and *iii*) changing an alternative by a new one when it corresponds. 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 (SHARMA; SPINELLIS, 2018).

5.1.2.3 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 one perform structural or behavioral changes on the managed subsystem by means of the second one. According to MAPE-K, Effectors are implemented in the managed subsystem as touch points for Executors, and the latter are implemented in the managing 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 (SAMA et al., 2010), 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 (MAGGIO et al., 2017).

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 reuse executors and effectors in isolation in other contexts when they are mixed.

- 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 (SHARMA; SPINELLIS, 2018).

5.2 Examples

This section presents some Adaptive Systems that contain the drifts we have characterized. Table 5.1 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 5.2.1 addresses the Zanshin-ATM system, and Subsection 5.2.3 addresses the PhoneAdapter system.

5.2.1 Scattered Reference Inputs Example

To exemplify the SRI drift, we use the Zanshin-ATM system. Zanshin (TALLABACI; SILVA SOUZA, 2013) 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 5.1 presents a snippet of the *CashDispenser* class, where this rule is implemented.

```
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 5.1 – Snippet of CashDispenser class (TALLABACI; SILVA SOUZA, 2013)

The *CashDispenser* class implements four methods (two of them are shown in Listing 5.1: *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 5.2 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  ..
12
```

Listing 5.2 – Snippet of goal model for ATM system (TALLABACI; SILVA SOUZA, 2013)

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 5.1, where Reference Inputs abstractions is clearly identifiable.

5.2.2 Obscure Alternatives Example

To exemplify Obscure Alternatives drift, we use the TeleAssistance System (TAS) system (WEYNS; CALINESCU, 2015). 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 (WEYNS; CALINESCU, 2015). 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 (SALEHIE; TAHVILDARI, 2009).

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 5.2 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 5.2 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.

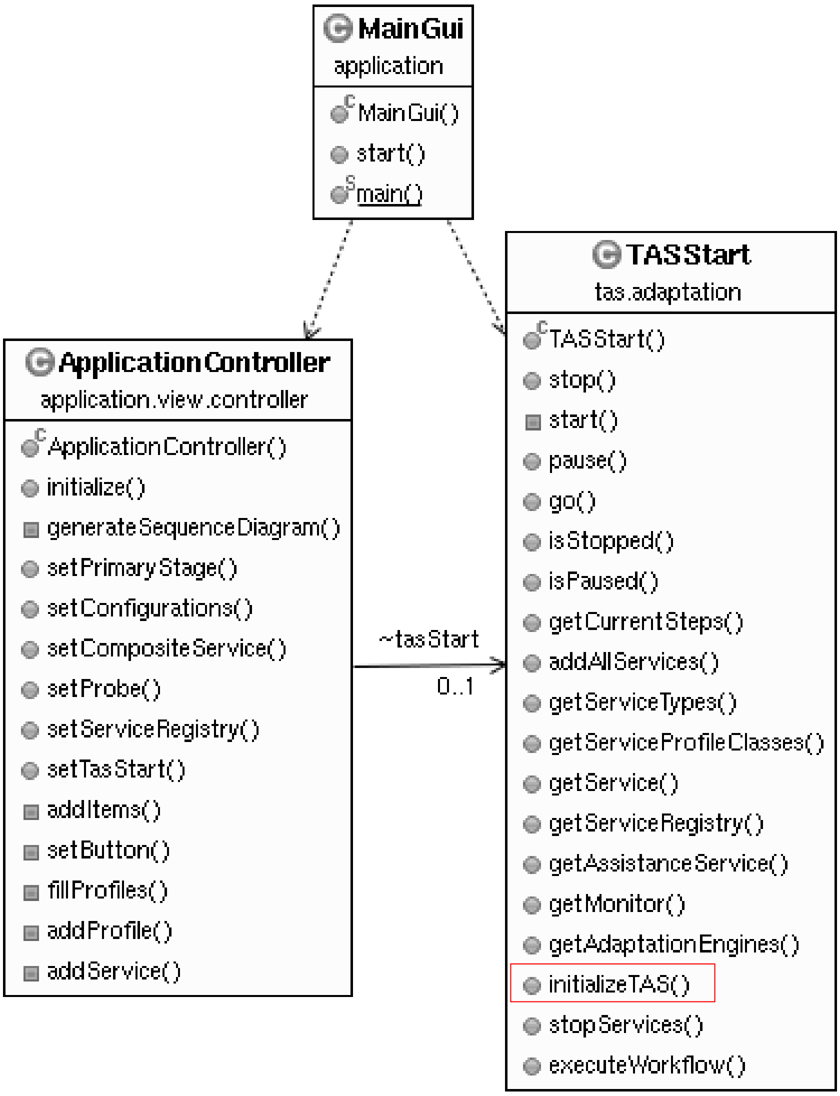


Figure 5.2 – Class TASStart

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.

5.2.3 Mixed Executors and Effectors Example

To exemplify the Mixed Executors and Effectors drift, we use PhoneAdapter (LIU, 2013). 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.

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 *AdaptationManager* 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 5.3 presents a snippet of the *AdaptationManager* class.

```
 1  public class AdaptationManager{
 2
 3    private AudioManager mAudioManager;
 4    ..
 5    public class MyBroadcastReceiver extends BroadcastReceiver{
 6      public void onReceive(Context c, Intent i) {
 7        if(volume>0){
 8           mAudioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
 9           mAudioManager.setStreamVolume(AudioManager.STREAM_RING,
10                      volume,AudioManager.FLAG_SHOW_UI);
11         }
12         if(vibration==1){
13           mAudioManager.setVibrateSetting(
14                      AudioManager.VIBRATE_TYPE_RINGER,
15                      AudioManager.VIBRATE_SETTING_ON);
16          }
17  ..
18  }
```

Listing 5.3 – Snippet of AdaptationManager class of PhoneAdapter

Lines 9 and 13 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 5.3 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.

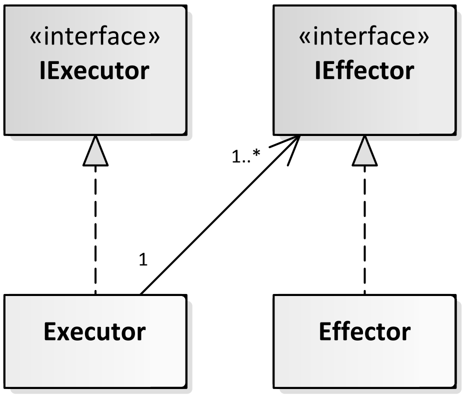


Figure 5.3 – Separation of executors and effectors

By separating the abstractions from the implementations, placing each in different subysystems, allows the flexibility to provide new implementations that could completely replace the existing implementation. For instance, new effector implementation can be created and places in the Managed Subsystem allowing to remove existing effector implementations without modifying the existing system.

5.3 Chapter Summary

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 (GARLAN; MONROE; WILE, 2000b; TEKINERDOGAN, 2016). 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.

Chapter 6

REMEDY: Approach for Conformance Checking

In this chapter we present our steps towards an ACC approach called REMEDY. Our approach works with two kinds of abstractions: i) conventional architectural abstractions and ii) ASs-specific ones. That is, besides conventional abstractions already used by other approaches (layers, components, modules, etc), it also delivers specific abstractions for better addressing ASs-specific parts of the system, such as monitors, analyzers, planners and executors. Moreover, REMEDY enables software architects can map source-code elements of the system to the abstract elements declared in the DSL and provides a mechanism for comparing the Planned Architecture (PA) with the Current Architecture (CA) for showing the found drifts.

6.1 An Adaptive Robotic System

In this section we present an adaptive robotic system that is used as an example across the chapter to exemplify the application of Remedy. This system aims at monitoring indoor environments following walls. It uses a light sensor for following the walls, trying to keep a constant distance of them. The effort of the robot to keep a constant distance from walls is controlled by a first control loop which makes adjustments in the percentage of turning the wheels and in the velocity. Besides, there is a slower control loop over the first one which analyze the adjustment parameters and change them to improve the performance of the robot. The goal is to make the robot to move as straight as possible.

Figure 6.1 shows schematically the Planned Architecture for the system. All blocks/rectangles/packages (as the bigger as the smaller ones) represent instances of architectural abstractions available in our DSL. By the stereotypes it is possible to see the names of the abstractions available in the DSL. In the upper part of the figure there is the Managing Subsystem, called *adaptationManager*. In the lower part there is the Managed Subsystem, called *Environment Guard Robot*. For simplicity reasons, the Managed part is not detailed, but usually this is much bigger than the Managing part. As can be seen, the Managing aggregates just one Loop Manger *loopManager*. The abstractions *ManagingSubSystem* and *ManagedSubsystem* are “default” abstractions, as they must be presented in any specification.

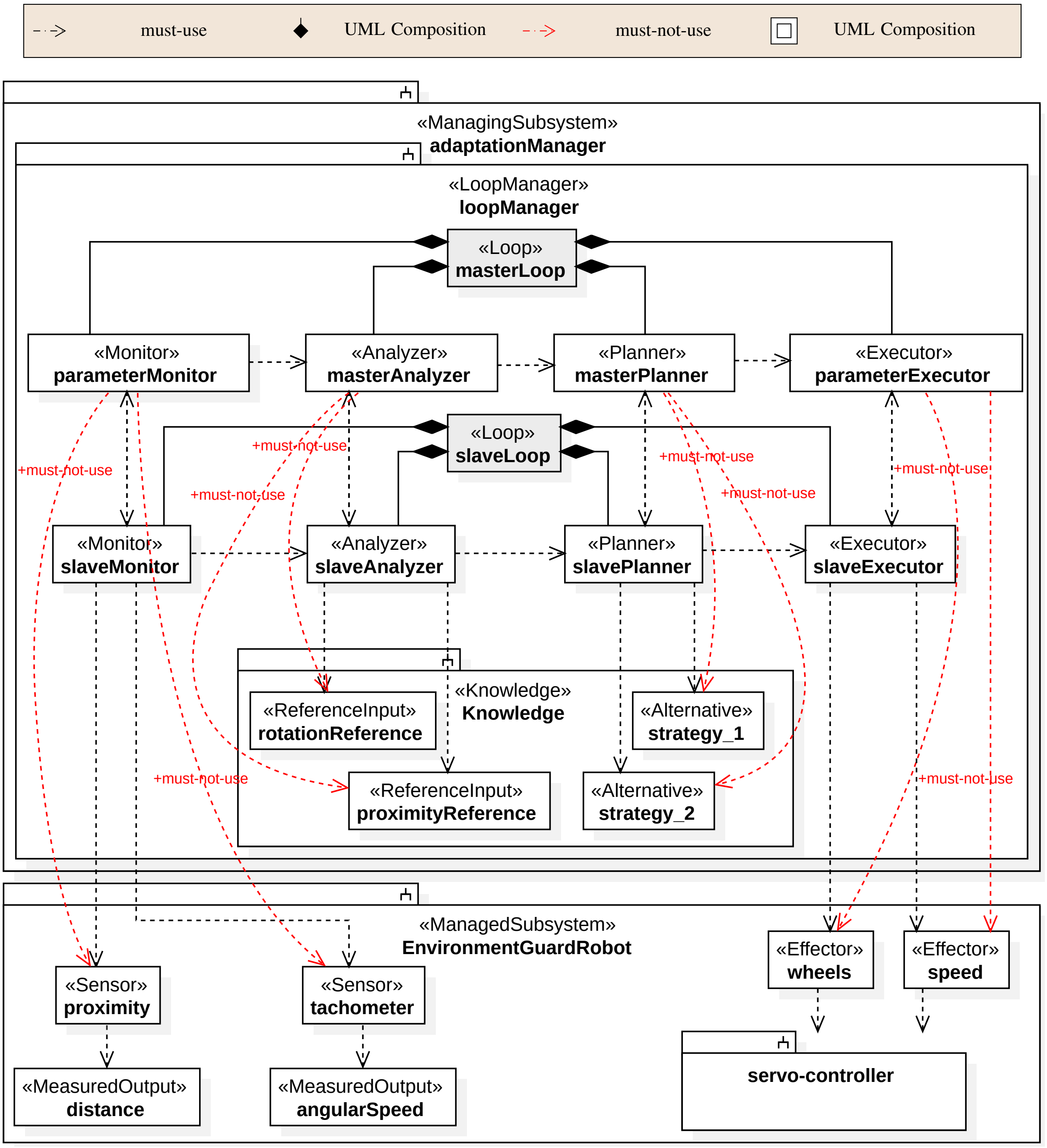


Figure 6.1 – Adapted UML to represent the Robotic System

In this case, the *LoopManager* aggregates two Control Loops, *masterLoop* and *slaveLoop*. Each one of the Control Loops was designed to have four abstractions; Monitor (*parameterMonitor, slaveMonitor*), Analyzer (*masterAnalyzer, slaveAnalyzer*) , Planner (*masterPlanner, slavePlanner*) and Executor (*parameterExecutor, slaveExecutor*). In addition, *slaveLoop* also contains a Knowledge abstraction called *knowledge* that is composed of two ReferenceInputs (*proximityReference, rotationReference* and two Alternatives *(strategy_1, strategy_2*), which represent different strategies of adaptation.

The Managed is composed of two Sensors (*proximity, tachometer*), two effectors; *wheels* and *speed*, two MeasuredOutput; (*distance, angularSpeed*) and a generic component called servo-controller which is not detailed in the specification of the DSL.

There are two types of relations that can be identified in this figure. One type is represented by the arrows among the elements - *communication rules*. The another type is represented by the hierarchical compositions among them - *structural relations*. The last one occurs when element/abstraction is within another one. Figure 6.1 shows 26 rules of type *must-use*, 8 rules of type *must-not-use* and when a communication rule is not present it means that the relationship is forbidden.

6.2 The REMEDY Approach

REMEDY approach involves four main steps A, B, C and D as can be seen in Figure 6.2. The step A is when software architects create/specify the Planned Architecture (PA) using DSL-REMEDY - this normally happens (but not always) at the early stages of life cycle. The specification process can be divided in three mutually-dependent substeps: *i*) Specify Structure of Managing Elements - When software architects specify (Step A.1) the adaptive elements that must exist in the system (monitor, analyzer, planner, executor, etc); *ii*) Specify Structure of Managed Elements - when the core elements are specified (Step A.2) employing common architectural abstractions (layers, components, modules, etc) and *iii*) Specify Communication Rules (Step A.3).

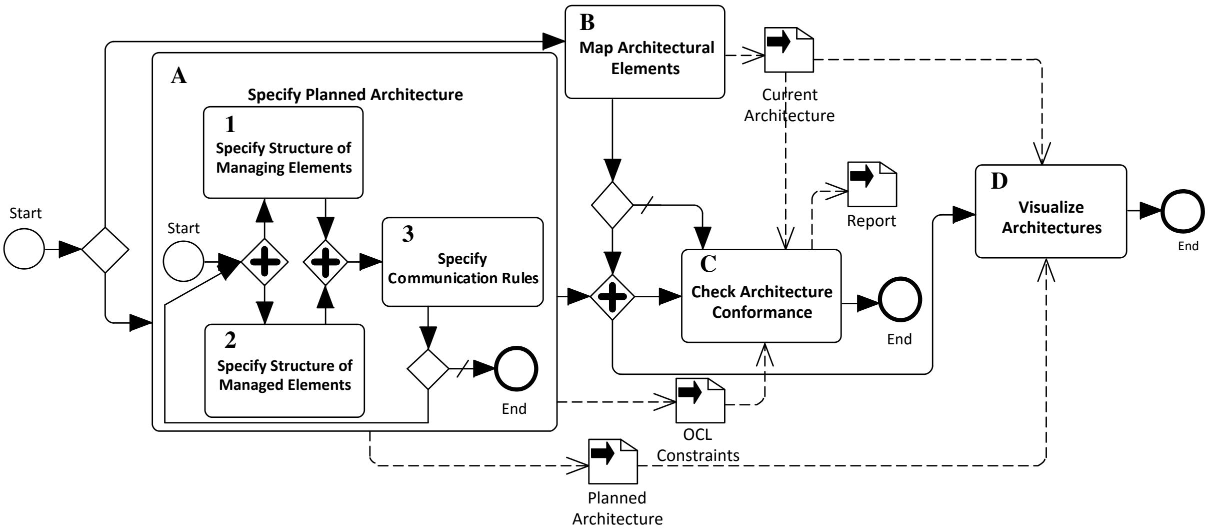

B
Map Architectural Elements
Current Architecture
A
Specify Planned Architecture
1
Specify Structure of Managing Elements
Start
Start
3
Specify Communication Rules
2
Specify Structure of Managed Elements
End
Report
C
Check Architecture Conformance
End
D
Visualize Architectures
End
OCL Constraints
Planned Architecture

Figure 6.2 – REMEDY approach

Step B aims at obtaining a representation of the Current Architecture (CA) of the system. Here software architects must map the architectural elements declared in the specification (monitors, analyzers, layers, components, modules, etc.) to the source code elements (variables, methods, etc.). This is the moment in which architects inform how the abstract elements are materialized in the source code. Step C is automatic and it has the responsibility of identifying the drifts by comparing the PA with the CA. Internally, both the PA and CA are represented as KDM model instances (ULRICH; NEWCOMB, 2010). Finally, in Step D software architects can visualize graphically the PA, CA and the differences between them. REMEDY was implemented as an eclipse plugin workbench which provides all user interfaces to support the mentioned steps.

6.2.1 Specify the Planned Architecture

In this section we detail the substeps to specify a PA with our textual DSL. It is important to highlight that all the subsets that must be performed by a software architect are done in the same specification file. For a better comprehension we explain each substep in each subsection below.

6.2.1.1 Specify the Structure of Managing Elements

In this substep software architects must start the elaboration of the planned architecture by specifying the elements of the Managing Subsystem. Listing 6.1 shows the specification that corresponds to the Managing Subsystem of Figure 6.1 that was performed with DLS-REMEDY. The way the abstractions are composed follow a Java syntax/style which defines the structural rules.

```
Architecture EnvironmentGuardRobot-PlannedArchitecture {
  Managing adaptationManager {
    LoopManager loopManager {

      Loop masterLoop withDomainRules{
        Monitor parameterMonitor;
        Analyzer masterAnalyzer;
        Planner masterPlanner;
        Executor parameterExecutor;
      }

      Loop slaveLoop withDomainRules{
        Monitor slaveMonitor;
        Analyzer slaveAnalyzer;
        Planner slavePlanner;
        Executor slaveExecutor;
        Knowledge knowledge {
           ReferenceInput proximityReference;
           ReferenceInput rotationReference;
           Alternative strategy_1;
           Alternative strategy_2;
...
}
```

Listing 6.1 – Managing subsystem of the PA

An abstraction can be composed of others or solely, then the nomenclature to specify it follows:

`[AS_Abstraction] [ID] [{..}]; | [AS_Abstraction] [ID];`

Where `[AS_Abstraction]` is a MAPE-K abstraction and `[ID]` is a unique string that identifies the abstraction in the whole specification. Inside the brackets architects must specify the abstractions that compound the abstraction of higher hierarchy. We intentionally implemented the unique abstraction ID strategy to simplify the access to them at time of specifying communication rules.

In Line 1 one must inform a name that identifies this PA. In this example, there is just one Managing subsystem (Line 2) and just one Loop Manager (line 3). A Loop Manager can manage one or more Loops and in this case there are two of them specified in the PA (lines 5 and 12). Loops can holds several MAPE-K abstractions and the *masterLoop*, which is the Loop that coordinates the *slaveLoop*, has four; a *parameterMonitor*, a *masterAnalyzer*, a *masterPlanner* and a *parameterExecutor* (lines 6-9).

As the robot has constrained resources, this specification enables the decentralization of Loops to a better scalability with respect to communication and computation. For instance, the *masterLoop* could be deployed in a remote server and the *slaveLoop* in the robot. Thus, the communication overhead is limited and the computational burden is spread over the two nodes (WEYNS; IFTIKHAR; SÖDERLUND, 2013). The *slaveLoop*, is composed of 5 MAPE-K abstractions. It adds a Knowledge called *knowledge* in line 17. This abstraction is composed of four abstractions (lines 18-21); *proximityReference*, *rotationReference*, *strategy_1* and *strategy_2*. The first two are of type *ReferenceInput* and the second two are of type *Alternative*. The *proximityReference* holds a value that indicates the distance between the robot and the wall. The *rotationReference* holds a value that indicates the angular speed of wheels. Notice that Loops are specified with the keyword **withDomainRules** (lines 5 and 12) to activate domain rules of ASs which is explained in subsection 6.2.1.3.

6.2.1.2 Specify the Structure of Managed Elements

In this substep software architects must specify elements of the Managed subsystem (also called system layer (GARLAN; CHENG, et al., 2004), managed resources (IBM, 2005), base-level subsystem (WEYNS; MALEK; ANDERSSON, 2012), target system (HELLERSTEIN et al., 2004)) that deals with the domain functionality. Managing subsystems have communication with the Managed subsystem by means of touchpoints which commonly are implemented by sensors and effectors (IBM, 2005). DSL-REMEDY provides these abstractions to be specified in the Managed subsystem section. Also, it is possible to specify Measured Outputs that corresponds to the measures in the Managed subsystem.

Listing 6.2 shows the specification of the Managed subsystem of the PA. Line 25 declares a Managed subsystem which in this case is composed of two Sensors (*proximity* and *tachometer*),

```
23 Architecture EnvironmentGuardRobot-PlannedArchitecture {
24 ..
25   Managed environmentGuardRobot {
26      Sensor proximity;
27      Sensor tachometer;
28      Effector wheels;
29      Effector speed;
30      MeasuredOutput distance;
31      MeasuredOutput angularSpeed;
32      Component servo-controller;
33   }
34 }
35 ..
```

Listing 6.2 – Managed subsystem of the PA

lines 26 – 27, two Effectors (*wheels* and *speed*)), lines 28 – 29, two MeasuredOutput (*distance* and *angularSpeed*) lines 30-31 and one generic component in line 32. At the writing of this thesis is possible to specify generic abstractions such as subsystems, components and layers with our DSL but implementations related to them such as code generators and validators we leave them as part of a future work (see Section 9.4.1).

6.2.1.3 Specify Communication Rules

In DSL-REMEDY, we opt for having two types of coarse-grained communication rules: *must-use* and *must-not-use*. Fine-grained access rules are not covered by our DSL because types of dependencies among Adaptive System (AS) abstractions is limited and the majority of abstractions are not compoundable in major abstractions. Therefore in order to avoid language verbosity we do not include keywords that control accesses in a fine-grained level.

The *must-use* access means that an abstraction *A* must use an abstraction *B*. The dependency type can be a callable method, objects creation, implementation of interfaces, class hierarchy, exceptions, imports, casting, assignments, value initializer and type bindings. The *must-not-use* means that an abstraction *A* must not access an abstraction *B*.

Table 6.1 presents all the rules allowed by the DSL that can be written by software architects. Abstractions of each column *must* or *must not* use abstractions of each row. Notice that they can be connected if they belong to the same level of abstraction. The only exceptions are Monitor to Sensor, Executor to Effector, Analyzer to ReferenceInput, Analyzer to Alternative, Planner to Alternative and Sensor to MeasuredOutput.

REMEDY implements twenty domain rules that can be activated or deactivated by software architects with the aids of a workbench. These rules were obtained by means of the analysis of the MAPE-K reference model. For instance, planners must not use monitors and vice-versa. Table 6.2 presents the complete set of domain rules where $\longrightarrow$ represents the *must-use* accesses and $\not\longrightarrow$ represents the *must-not-use* accesses. When an AS does not conform its domain rules we will refer to this concept as domain drift.

As we state before, the keyword *withDomainRules* must be declared in Loops to enable domain rules. Thus, the rules will affect all abstractions that match with a specific rule. For

Loop Manager	Loop	Monitor	Analyzer	Planner	Executor	Knowledge	Sensor	
✓								**Loop Manager**
	✓							**Loop**
		✓	✓	✓	✓	✓		**Monitor**
		✓	✓	✓	✓	✓		**Analyzer**
		✓	✓	✓	✓	✓		**Planner**
		✓	✓	✓	✓	✓		**Executor**
		✓	✓	✓	✓			**Knowledge**
		✓						**Sensor**
					✓			**Effector**
			✓					**Reference Input**
			✓	✓				**Alternative**
							✓	**Measured Output**
✓: It means that an abstraction *must-use* or *must-not-use* other.								

Table 6.1 – The access rules allowed by the DSL

Monitor	Analyzer	Planner	Executor	Knowledge	
	↛	↛	↛	↛	**Monitor**
⟶		↛	↛	↛	**Analyzer**
↛	⟶		↛	↛	**Planner**
↛	↛	⟶		↛	**Executor**
⟶	⟶	⟶	⟶		**Knowledge**

Table 6.2 – Domain rules of ASs

instance, if a Loop has two monitors and the rule *Monitor* ↛ *Planner* is activated then a constraint for each monitor will be generated to be checked in the CA. Listing 6.3 shows an example of a *Loop* with domain rules enabled for its four abstractions.

```
Loop masterLoop withDomainRules{
  Monitor parameterMonitor;
  Analyzer masterAnalyzer;
  Planner masterPlanner;
  Executor parameterExecutor;
}
```

Listing 6.3 – Example of a Loop with domain rules activated

Listing 6.4 shows the communication rules of Figure 6.1. Lines 41 and 42 enable the communication between *masterLoop*, *slaveLoop* and vice-versa. The DSL automatically checks whether there are rules that connect both abstractions or not. If it detects the absence of rules connecting them then one or more errors will raise at design time and PA will not generate its output. On the other hand, if software architects specify that a Loop must not use another one then all rules that connects both abstractions will not take into account in the generated output of the PA. Also, it is possible to interconnect LoopManagers which follow the same rules as Loops.

Lines 43 − 47 specify all rules related with Monitors and their accesses. Lines 48 − 53

```
40 Rules{
41   loop masterLoop must-use loop slaveLoop;
42   loop slaveLoop must-use loop masterLoop;
43   monitor parameterMonitor must-use monitor slaveMonitor;
44   monitor slaveMonitor must-use monitor parameterMonitor;
45   monitor slaveMonitor must-use sensor proximity;
46   monitor parameterMonitor must-not-use sensor proximity;
47   monitor slaveMonitor must-use sensor tachometer;
48   analyzer masterAnalyzer must-use analyzer slaveAnalyzer;
49   analyzer slaveAnalyzer must-use analyzer masterAnalyzer;
50   analyzer slaveAnalyzer must-use reference-input proximityReference;
51   analyzer slaveAnalyzer must-use reference-input rotationReference;
52   analyzer masterAnalyzer must-not-use reference-input proximityReference;
53   analyzer masterAnalyzer must-not-use reference-input rotationReference;
54   planner masterPlanner must-use planner slavePlanner;
55   planner slavePlanner must-use planner masterPlanner;
56   planner slavePlanner must-use alternative strategy_1;
57   planner slavePlanner must-use alternative strategy_2;
58   planner masterPlanner must-not-use alternative strategy_1;
59   planner masterPlanner must-not-use alternative strategy_2;
60   executor parameterExecutor must-use executor slaveExecutor;
61   executor slaveExecutor must-use executor parameterExecutor;
62   executor slaveExecutor must-use effector wheels;
63   executor slaveExecutor must-use effector speed;
64   executor parameterExecutor must-not-use effector wheels;
65   executor parameterExecutor must-not-use effector speed;
66   sensor proximity must-use measured-output distance;
67   sensor tachometer must-use measured-output angularSpeed;
68   effector wheels must-use Servo-Controller;
69   effector speed must-use Servo-Controller;
70 }
```

Listing 6.4 – Communication rules of the PA

specify all rules related with Analyzers and their accesses. Lines 54 – 59 specify all rules related with Planners and their accesses. Lines 60 – 65 specify all rules related with Executors and their accesses and finally lines 66 – 69 specify all rules related with Sensors, Effectors and their accesses.

6.2.1.4 DSL-REMEDY Validators

Despite the DSL-REMEDY can load domain rules they are not mandatory if a software architect writes a custom rule that violates a specific domain rule. Thus it is important the implementation of custom validators to check additional constraints in the DSL which cannot be done at parsing time. DSL-REMEDY implements three types of validators. The first one checks that abstractions do not access themselves. For instance, the rule *monitor monitor_1 must-use monitor monitor_1* is forbidden. Listing 6.5 shows an example of this validator where line 1 checks the monitor is not null, line 2 checks if a monitor is accessing itself and line 3 raise the error in the corresponding line where the rule was specified in the DSL. The same implementation is valid for the other abstractions so we do not include them in the document.

```
1 if (dslRuleMonitor.monitor2 !== null)
2   if (dslRuleMonitor.monitor == dslRuleMonitor.monitor2)
3     error("Check a monitor does not have dependency with itself",
      ↪ SasDslPackage.eINSTANCE.DSLRuleMonitor_Monitor2, DUPLICATE_MONITOR_ACCESS)
```

Listing 6.5 – Checking if an abstraction depend on itself

The second one checks that rules can not be duplicated. Listing 6.6 shows an example of this validator and in order to implement it, a *Hash* data structure is created for each abstraction to disallow duplicate communication rules. Line 1 implements a for loop that iterates over the elements of the *Hash* structure. In line 3, if there are duplicates then line 5 raise errors in the corresponding lines where of duplicated rules.

```
1  for (entry:multiMapRuleMonitor2Monitor.asMap.entrySet) {
2      val duplicates = entry.value
3    if (duplicates.size > 1){
4      for (d:duplicates)
5        error("Duplicated rule",d, SasDslPackage.eINSTANCE.DSLRuleMonitor_Monitor2,
     ↪ DUPLICATE_RULES)
6    }
7  }
```

Listing 6.6 – Checking if a communication rule is duplicated

The third one checks if communicating rules are violating domain constraints. Listing 6.7 shows an example of this kind of validator that checks if a monitor must use a planner. Line 1 get the domain rule written in the DSL and line 5 verifies if the domain rule is deactivated through the computational support. If the rule is presented but deactivated then a warning is raised.

```
1    var dslDomain = dslRuleMonitor.monitor.eContainer.eContents.filter(DSLDomainRule).toList
2    if (!dslDomain.isEmpty) {
3      val queryClass = new QueryClass(MainView.getDatabaseUrl())
4      val rule = queryClass.ruleIsActive("Monitor","Planner");
5      if (Boolean.valueOf(rule.get(1)))
6        if (dslRuleMonitor.planner !== null && dslRuleMonitor.access.equals("must-use"))
7          warning("The rule is violating the domain rule number  " + rule.get(0),
     ↪ SasDslPackage.eINSTANCE.DSLRuleMonitor_Planner)
8    }
```

Listing 6.7 – Checking if a rule is violating a domain rule

6.2.1.5 Outputs of DSL-REMEDY

In this subsection we detailed the two outputs of DSL-REMEDY: The AS planned architecture and an OCL file generated automatically by taking into account the description language. The AS planned architecture is an instance of the KDM metamodel. This instance can represent several viewpoints of a system and its main characteristic is that the model is language and platform independent which makes REMEDY capable of identifying architectural drifts in architectures that were implemented in several programming languages. In order to generate the KDM model we use an existing tool called MoDisco that provides APIs for manipulating some model generators called discoverers in Java (BRUNELIÈRE et al., 2014).

The KDM generated by MoDisco includes two models: one for representing the source-code of the system and the other one for representing the inventory (physical resources of the system such as system files). Thus, REMEDY implements several algorithms for creating the

Structure Model (Architectural viewpoint of the system) in the KDM instance previously generated by MoDisco.

The *Structure Model* is created according to the mappings performed by software architects which is composed of the abstractions and their relationships. One of the major challenge in terms of implementation was to generate the relationships among the AS abstractions automatically. These relationships are represented by the *AggregatedRelationship* metaclass of KDM and takes into account the relationships among source-code elements (OMG, 2016, pp. 30–33).

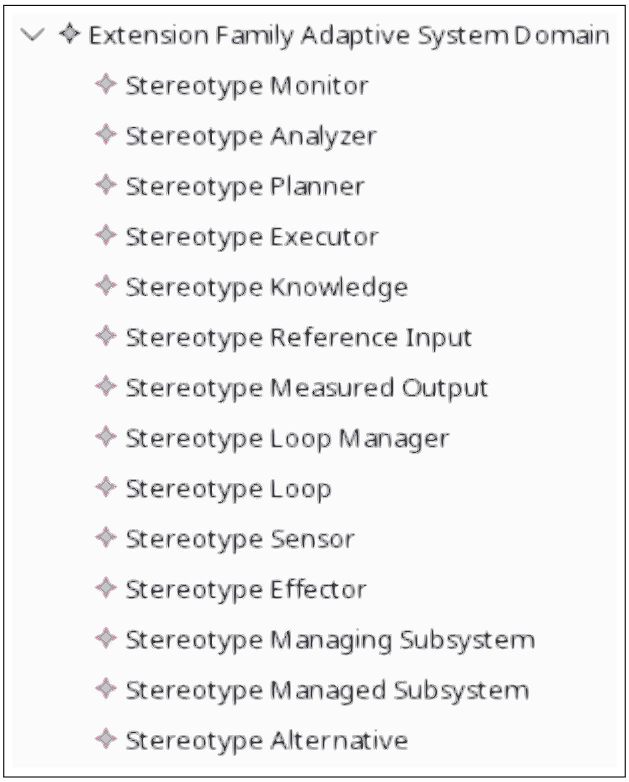
Listing 6.8 shows a brief example of a *structure model* with *AggregatedRelationShip*. In this case the architecture is composed by two abstractions; *parameterMonitor* (line 2) and *masterAnalyzer* (line 12. The first one access the second one by means of an *aggregatedRelationship* (line 7). *parameterMonitor* is implemented by one code element, method *C()* (line 3). On the other hand, *masterAnalyzer* is implemented by two code elements, methods *A()* and *B()* (line 13).

```
 1 <model name="EnvironmentGuardRobot-PlannedArchitecture"
       ↪ xsi:type="structure:StructureModel">
 2   <structureElement name="parameterMonitor" xsi:type="structure:Component"
 3       implementation="//path/to/codeElement/C()"
 4       outAggregated="//path/to/aggregated"
 5       inAggregated=""
 6       stereotype="/0/@extension.0/@stereotype.1">
 7       <aggregated from="//path/to/parameterMonitor" to="//path/to/masterAnalyzer"
 8           relation="Call Call
 9           density="2"
10       />
11   </structureElement>
12   <structureElement name="masterAnalyzer" xsi:type="structure:Component"
13       implementation="//path/to/codeElement/A() //path/to/codeElement/B()"
14       outAggregated=""
15       inAggregated="//path/to/outAggregated"
16       stereotype="/0/@extension.0/@stereotype.0"
17   />
18 </model>
```

Listing 6.8 – Example of an AggregatedRelationship

Also, *parameterMonitor* is indicating an outgoing relationship (line 4) that references its aggregated relationship (line 7). This relationship describes the from and to, the type of the relation which, in this case, are two method calls ($C() \longrightarrow A()$ and $B() \longrightarrow A()$) and the density of the relationship (number of relationships between two abstractions). Finally, *masterAnalyzer* must declare a reference for the incoming relationship (line 15) to validate the relationship. Notice that both abstractions of Listing 6.8 have stereotypes in order to extend the KDM metamodel by means of a lightweight mechanism extension . Figure 6.3 shows all the stereotypes defined in the PA and the CA to support AS abstractions. These stereotypes help us to perform the ACC without depending on the syntactic specification of the architectural elements.

Listing 6.9 shows a brief snippet of the OCL file generated from the PA of Listings 6.1, 6.2 and 6.4 but with domain rules activated for *loop_2*. DSL-REMEDY was built with the specification of a grammar written in EBNF (extended backus-naur form) (see A.1) and implemented



Extension Family Adaptive System Domain
- Stereotype Monitor
- Stereotype Analyzer
- Stereotype Planner
- Stereotype Executor
- Stereotype Knowledge
- Stereotype Reference Input
- Stereotype Measured Output
- Stereotype Loop Manager
- Stereotype Loop
- Stereotype Sensor
- Stereotype Effector
- Stereotype Managing Subsystem
- Stereotype Managed Subsystem
- Stereotype Alternative

Figure 6.3 – KDM lightweight extension to support AS abstractions

```
package structure
-- Check the existence of AS abstractions --
  context StructureModel
  inv exist_parameterMonitor: Component.allInstances()->exists(c|
    ↪ c.name='parameterMonitor' and c.stereotype->asSequence()->first().name = 'Monitor')
-- Check structural rules of AS --
  context StructureModel
  inv composite_parameterMonitor: Component.allInstances()->select(c|
    ↪ c.name='parameterMonitor' and c.stereotype->asSequence()->first().name =
    ↪ 'Monitor')->
    ↪ exists(d|d.oclContainer().oclAsType(Component).name='masterLoop' and
    ↪ d.oclContainer().oclAsType(Component).stereotype->asSequence()->first().name =
    ↪ 'Loop')
-- Check communication rules of AS --
  context StructureModel
  inv not_access_parameterMonitor_proximity: not
    ↪ AggregatedRelationship.allInstances()->exists(c| c.from.name='parameterMonitor'
    ↪ and c.to.name='proximity')
-- Domain rules --
  context StructureModel
  inv domain_not_access_slaveMonitor_slavePlanner: not
    ↪ AggregatedRelationship.allInstances()->exists(c| c.from.name='slaveMonitor' and
    ↪ c.to.name='slavePlanner')
endpackage
```

Listing 6.9 – Snippet of OCL constraints file

in *Xtext* which allowed us to generate the constraints from the PA. Due to space restrictions, it shows one rule per section because the whole file has more than 400 LoC.

Notice that the OCL file is composed of four sections. The first one verifies the existence of the declared abstractions. The second one verifies the structural rules. The third one verifies the communication rules and the fourth one verifies domain rules that were activated by software

```
package br.ufscar.advanse.adapters;
import java.util.concurrent.ThreadLocalRandom;
public final class BluetoothAdapter{
    String address;
    public void getScanMode(){int min =0; ..}
    ...
}
```

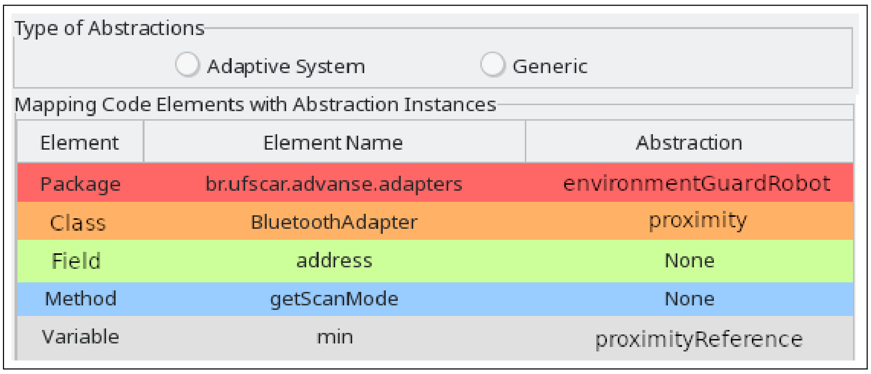
Listing 6.10 – Snippet of source code of an AS

architects.

The first rule (lines $3-4$), verifies the existence of the Monitor *parameterMonitor*. It checks if exist an element with name *parameterMonitor* and if the stereotype corresponds to a Monitor. The second rule (lines $6-8$), verifies the composition of *parameterMonitor* in *loop_1*. It checks if *parameterMonitor* is contained in *loop_1*. The third rule (lines $10-11$), verifies the forbidden access from *parameterMonitor* to *proximity*. It will check that there is no *AggregatedRelationship* between *parameterMonitor* and *proximity*. The fourth rule (lines $13-14$), verifies the domain rule *Monitor* $\not\longrightarrow$ *Planner*.

6.2.2 Mapping Architectural Elements to Source Code

The mapping of the architectural elements, declared in the DSL (monitors, analyzers, etc), to source code must be performed by a software architect who has the knowledge for recognizing them. Figure 6.4 shows the user interface provided by REMEDY to perform the mapping. It reflects code elements of the currently open class file such as package name, class name, fields class, methods and variables. In this case, it is reflecting the content of Listing 6.10.



Type of Abstractions

○ Adaptive System ○ Generic

Mapping Code Elements with Abstraction Instances

Element	Element Name	Abstraction
Package	br.ufscar.advanse.adapters	environmentGuardRobot
Class	BluetoothAdapter	proximity
Field	address	None
Method	getScanMode	None
Variable	min	proximityReference

Figure 6.4 – Computational support for mapping abstractions

The fist column shows the code element, the second column the name of the code element and third column the mapping. Notice that the types of code element are highlighted with different colors for a better understanding. Also, the variables of a method (gray color) are represented just below of the method where they belong. For instance, variables *min* and *randomNum* belong to the method *getScanMode*. Thus, software architects should map code elements with the abstraction that implements the AS architecture by picking them in the combobox. All mappings are stored in a embedded database, ready to be processed when software architects need to apply the mappings in the CA. Figure 6.5 shows the relational data model to store the information about architectural elements, code elements and their mappings.

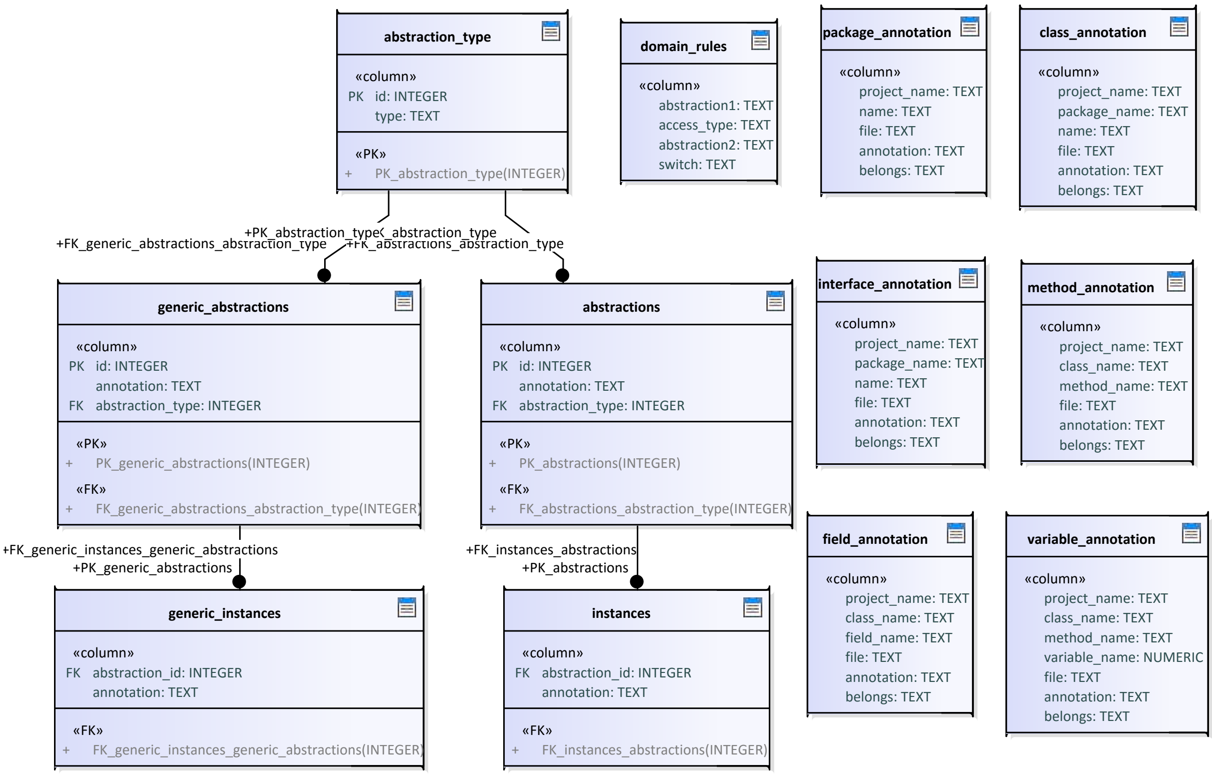


Figure 6.5 – Relational data model for mappings

The *abstraction_type* table stores information about the type of abstractions that handles REMEDY (AS domain and generic abstractions). Generic abstractions and their instances (elements) are stored in *generic_abstracions* and *generic_instances* respectively while AS abstractions and their instances are stored in *abstractions* and *instances* tables. The *domain_rules* table stores the predefined domain rules of REMEDY. The *package_annotation*, *class_annotation*, *interface_annotation* , *method_annotation*, *field_annotation*, *variable_annotation* tables store the mappings according to the source-code element type.

In our example, it was mapped the package of the class with *environmentGuardRobot*, the class with *proximity* and the variable *min* with *proximityReference*. The abstraction names shown in the comboboxes of the Abstraction column are taken from the PA. Algorithm 6.11 shows, in a high-level of abstraction, the logic to create the Structure Model in the KDM of the CA.

In Line 2, the method *createStructureElementFromTree*() creates the *Structure Model* into the KDM with the AS elements that were mapped previously. The architectural representation follows the same hierarchy as was implemented in source code. For instance, if a class and a field belonging to the same class were mapped with a *Knowledge* and a *ReferenceInput* respectively then our algorithm will create a component of type *Knowledge* and inside of it a

Algorithm 6.11 Algorithm for creating structure package of CA

Input: *KDM CA (kdmResource)*
Output: *KDM CA with an instance of the Structure Model*

```
1: for all packageMappings, classMappings, methodMappings, fieldMappings, variableMappings do
2:     kdmResource ← createStructureElementFromTree()
3:     kdmResource ← createImplementations()
4:     kdmResource ← createRelationShips()
5: end for
6: return kdmResource
```

ReferenceInput component. To do so, we use the information stored in the embedded database and BaseX[1], a xquery engine to manage xml based files.

Listing A.2 shows the method with details. In Line 3 the method *createImplementations*() adds all source code elements that contributes to the implementation of an architectural element. For instance, it is possible to map two classes with the same abstraction and in that case both will implement the mapped abstraction. To do so, we use *Query/View/Transformation operational (QVT-o)*, to perform the changes in the KDM CA. Listing A.3 shows the transformation file with details.

In Line 4 the method *createRelationShips*() adds the relationships among AS elements. REMEDY is capable of identifying 13 types of relationships and as we explained before, they are created in the *relation* attribute of the *aggregated* metaclass. Thus, for every package, class, method, field and variable mapped to an AS element the method identifies the relationships and add them to a component or subsystem according to the case. Listing A.4 shows the transformation file with details.

6.2.3 Checking Architecture Conformance

In this step, REMEDY performs the ACC. From the user's point of view, this step begins when a software architect press a button. Thus internal mechanisms of the workbench executes several algorithms in order to the OCL constraints be applied in the CA. Algorithm 6.12 shows the checking process in a high level of abstraction.

Line 1 initializes the Pivot OCL engine that evaluates constraints in MOF (Meta-Object Facility) models. Line 2 creates an interface to use the OCL engine. Line 3 gets the KDM instance and store it in memory. Line 4 gets the OCL constraints and store them in memory. Line 5 creates a HashMap to store the constraints in order to invoke them by key. Lines $6-14$ gets all objects of *Constraint* type and put them in the HashMap.

Line 15 gets the structure model of the KDM instance. Line 16 iterates over all constraints of the HashMap. Line 17 evaluates each constraint against the structure model of the KDM instance. This is a boolean value that returns false whether the constraint is not satisfied in the CA and true otherwise. Lines $18-27$ verifies the type of constraint that was checked for stored

[1] https://basex.org/

Algorithm 6.12 Algorithm for checking architectural constraints

Input: *KDM CA, OCL constraintFile*

```
initOCL()
ocl ← OCL.newInstance
kdmResource ← resourceSet.getResource(CA)
oclResource ← ocl.parse(constraintFile)
constraintMap ← HashMap < String, ExpressionInOCL >
for all object ∈ oclResource do
    if object instanceof Constraint then
        constraint ← object as Constraint
        expression ← ocl.getSpecification(constraint)
        if expression not null then
            constraintMap.put(expression.name, expression)
        end if
    end if
end for
structureModel ← kdmResource.getStructure()
for all object ∈ constraintMap do
    check ← ocl.evaluate(structureModel, object)
    if object.key == "exist" then
        database.insertExistence(check, object)
    else if object.key == "composite" then
        database.insertComposite(check, object)
    else if object.key == "access" then
        database.insertAccess(check, object)
    else if object.key == "domain" then
        database.insertDomain(check, object)
    end if
end for
```

them in specific tables of the embedded database. The output of the checking algorithm is a set of identified drifts (existence of the abstractions, structural rules, communication rules and domain rules) that is stored in a embedded database of our workbench. Listing A.5 shows the implementation with more details.

Figure 6.6 shows the relational data model to store the architectural drifts identified by REMEDY which is composed of 11 tables.

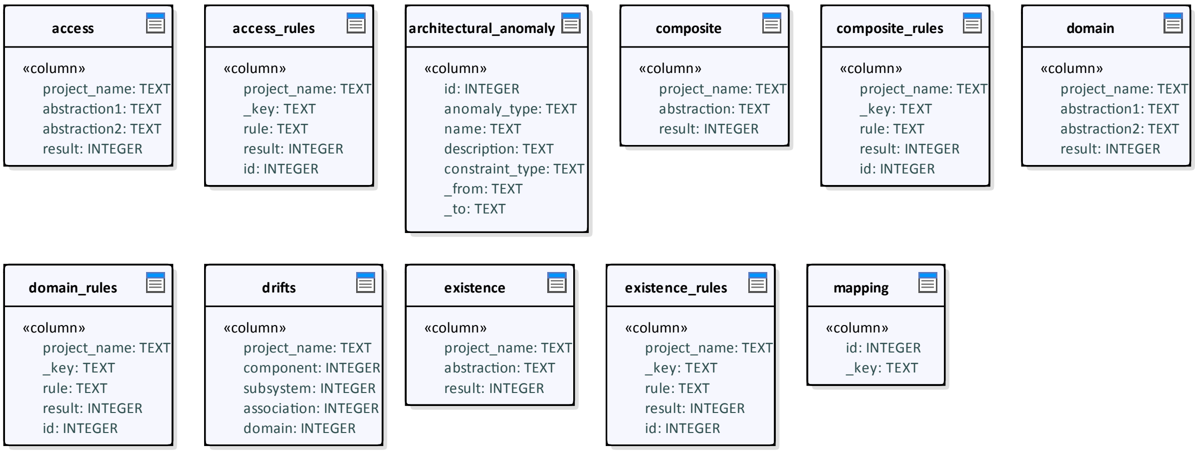


Figure 6.6 – Relational data model for architectural drifts

Tables with prefix *access_** store rules of type must-use/must-not-use and the result of applying them. Similarly, tables with prefix *composite_** store rules related to abstractions composition, tables with prefix *existence_** store rules related to the existence of the abstraction and tables with prefix *domain_** store domain rules and the corresponding result after applying them in the CA. The *drifts*, *architectural_anomaly* and *mapping* tables are used to consolidate data which aid in a report generation.

6.2.4 Architecture Visualizations

The visualization of architectures is an alternative step where software architects can visualize graphically the PA, CA and the differences between both. This is possible due to model transformations that is performed over the KDM instances and the result is used by a third party tool called *PlantUML* [2] to visualize the architectures. The visualization of drifts is possible due to another third party tool called *Eclipse EMF Compare* [3] which compares the PA and the CA (as UML Package diagrams), compute differences and as a result a new model is created. This model passes through the same process of transformation to be visualized graphically.

6.2.4.1 Model Transformation

The transformation from KDM to UML takes into account just the structure model of KDM. The model is transformed into a UML Package Diagram. Listing 6.13 shows the main parts of the QVT-o transformation file that takes as an input a KDM instance and return as a result a UML instance. Line 4 and Line 8 (mapping *Model2Model()*) transform Subsystems and Components of the Structure Package of KDM into UML Packages. In our case the Structure Package is composed only of subsystems and components. Subsystems (the Managing and Managed abstractions) can only be composed of Components (others AS abstractions) and a Component can only be composed of Components.

Thus, in line 13 (mapping *kdmSubsystem2UmlPackage()*) the code transform components that belong to a subsystem into UML Packages and it calls the mapping *kdmSubsystem2UmlPackage()* to recursively, if is the case, transform the composition of components into UML packages. Similarly, in line 21 it is done the same procedure if a Component is present in the high-level of hierarchy of the model. As in our representation only components have relationships with other components it is necessary to implement a mapping that transforms the *AggregatedRelationship* instances into UML Dependencies.

Line 27 shows the mapping to transform the relationships of Structure Package of KDM into UML. In UML a dependency is composed by a *client - supplier* relationship so attribute *from* is mapped to attribute *client* and attribute *to* is mapped to attribute *supplier*. Notice that we uses the special keyword *late* in order to obtain a valid reference of the object to be transformed. A

[2] https://plantuml.com/

[3] https://www.eclipse.org/emf/compare/

UML profile also is created to maintain a reference to the type of AS abstraction in the UML Package.

Once the UML model is created, it is transformed again by using a specific API called *structurizr* [4] to convert the UML model into a DSL for being interpreted by PlantUML.

```
mapping StructureModel :: Model2Model():Model when {self.name = modelName}{
    name :="UML Package Diagram";
    self.structureElement -> select(c | c.oclIsKindOf(Subsystem))[Subsystem] ->
    ↪ forEach(p){
        result.nestedPackage += p->map kdmSubsystem2UmlPackage();
    };

    self.structureElement -> select(c | c.oclIsKindOf(Component))[Component] ->
    ↪ forEach(p){
        result.nestedPackage += p->map kdmComponent2UmlPackage();
    }
}
mapping Subsystem:: kdmSubsystem2UmlPackage(): uml::Package {
    name := self.name;
    nestedPackage += self.structureElement -> select(c |
    ↪ c.oclIsKindOf(Component))[Component] ->  map kdmComponent2UmlPackage();
    self -> any(c| c.aggregated ->notEmpty()) -> aggregated -> forEach(t) {

        result.packagedElement += t-> map aggregated2dependency();
    };
}
mapping Component:: kdmComponent2UmlPackage(): uml::Package {
    name := self.name;
    nestedPackage += self.structureElement -> select(c |
    ↪ c.oclIsKindOf(Component))[Component] ->  map kdmComponent2UmlPackage();
    self -> any(c| c.aggregated ->notEmpty()) -> aggregated -> forEach(t){

        result.packagedElement += t-> map aggregated2dependency();
    };
}
mapping  AggregatedRelationship::aggregated2dependency():uml::Dependency {
    client := self._from.late resolve(uml::Package);
    supplier := self.to.late resolve(uml::Package);

}
```

Listing 6.13 – A QVT-o model transformation: from KDM to UML

The complete source code to transform from UML to Structurizr is available in Listing A.6. The DSL generated is interpreted by PlantUML which is a textual notation to construct UML diagrams. Listing A.7 shows the PlantUML code of PA of the Robotic System presented in Figure 6.1.

6.3 Compute Differences Between the PA and CA

The computation of differences between the PA and CA is performed by a third-party plugin called EMF Compare. This plugin allows to recognize elements of a EMF model that were

[4] https://structurizr.com/

deleted, added, changed or moved when it is compared with another EMF model. In our case we identify architectural elements that are not presented in the PA when it is compared with the CA. Algorithm 6.14 shows the comparison of the PA and CA by means of the UML Package representation. Line 1 initialize the plugin. In line 2 the scope of comparisons is determined and in our case the comparisons always begin from the root nodes of both models. Line 3 computes the comparisons. Line 4 gets the differences between both models. Line 5 iterates over the differences. If the difference is an *AS element* (line 6) and it is marked as "DELETE" then the element is added into a list (line 8).

Algorithm 6.14 Algorithm for comparing the PA and CA

Input: *UML Package PA (resourceSet1), UML Package CA (resourceSet2), PlantUML PA File (file)*

```
 1: initEMFCompare()
 2: scope ← rangeOfComparison(resourceSet1, resourceSet2)
 3: comparison ← compare(scope)
 4: differences ← comparison.getDifferences()
 5: for all difference ∈ differences do
 6:     if difference is an ASElement then
 7:         if difference is marked as DELETE then
 8:             listDeletedASElement ← difference
 9:         end if
10:     else if difference is a Dependency then
11:         if difference is marked as DELETE and difference is marked as ADD then
12:             do nothing
13:         else if difference is marked as DELETE then
14:             listDependency ← difference
15:         end if
16:     end if
17: end for
18: for all ASElement ∈ listDeletedASElement do
19:     change the color of the AS element in file to gray
20: end for
21: for all Dependency ∈ listDependency do
22:     change the color of the dependency in file to red
23: end for
```

if the difference is a *Dependency* (line 10) and it is marked as "DELETE" and "ADD", nothing is done because it means that the element was preserved in the CA. Otherwise if the difference is just marked as "DELETE" then it means that this element does not appear in the CA and therefore is added into a list (line 14).

Finally, all elements that belong to the list *listDeletedASElement* are searched in the PlantUML PA file to change their color to gray (line 19). Similarly, all elements that belong to the list *istDependency* are searched in the PlantUML PA file to change their color to red (line 22).

6.4 Quality of the Checking Process

In this section, we present and discuss results on applying the approach proposed in this thesis in two systems. The main goal is to demonstrate the applicability of our approach in development contexts through an analysis of the accuracy of architectural drifts identification.

6.4.1 Definition of the Evaluation

Goal: The goal is to evaluate the effectiveness of the REMEDY in detecting architectural drifts of ASs. Therefore, we are interested in the effectiveness of the Step C of our approach, detailed in Section 6.2.3, where OCL constraints are applied in the CA in order to detect the drifts. As REMEDY is a solution for detecting ASs drifts, we are interested in evaluating three points: *i*) the detection of violations of structural rules; *ii)* detection of violations of communication rules and *iii)* the detection of violations of domain rules.

Object of Study: The algorithms of REMEDY that perform the checking process.

Perspective: The perspective is from the point of view of software architects and researchers i.e. they would like to know if REMEDY is capable of detecting architectural drifts of ASs.

Quality Focus: The main effect studied in the evaluation is the accuracy of REMEDY in detecting the drifts. The focus is on Precision and Recall as follow:

$$Precision = \frac{TP}{TP + FP} \quad , \quad Recall = \frac{TP}{TP + FN}$$

Where TP (True Positives) are the number of drifts correctly identifies and FP, (False Positives) the number of drifts incorrectly identifies and $falseNegatives$ (False Negatives) are drifts which were not identified but should have been. Thus precision is the fraction of relevant drifts among the retrieved drifts while recall is the fraction of the total amount of relevant drifts that were actually retrieved.

6.4.2 Planning

In this study, we focus on the accuracy of ACC support provided by REMEDY, which we scoped in three main questions:

RQ1 : Assuming that the PA and CA specifications are correct, does REMEDY achieve high levels of accuracy when detecting architectural drifts regarding to structural rules ?

RQ2 : Assuming that the PA and CA specifications are correct, does REMEDY achieve high levels of accuracy when detecting architectural drifts regarding to communication rules?

RQ3 : Assuming that the PA and CA specifications are correct, does REMEDY achieve achieve high levels of accuracy when detecting architectural drifts regarding to domain rules?

Therefore, to answer the three questions we collected the drifts detected by REMEDY. The focus is to verify if the checking algorithm is able to identify drifts among the AS abstractions (MAPE-K) abstractions and lower-level AS abstractions).

6.4.2.1 Context Selection

To check the accuracy of our approach we have used two ASs: The first one is UNDERSEA (GERASIMOU et al., 2017), a simulated UUV (unmanned underwater vehicles) exemplar built on top of the open-source middleware MOOS-IvP2, a widely used platform for the implementation of autonomous applications on UUVs.

UNDERSEA was designed and implemented following MAPE-K and comprises a simulated managed system (UUV) and its controller. It has four adaptation scenarios that the controller must handle by adjusting: *a*) the UUV speed and *b*) the sensor configuration (on/off). Figure 6.7 shows the architecture of UNDERSEA.

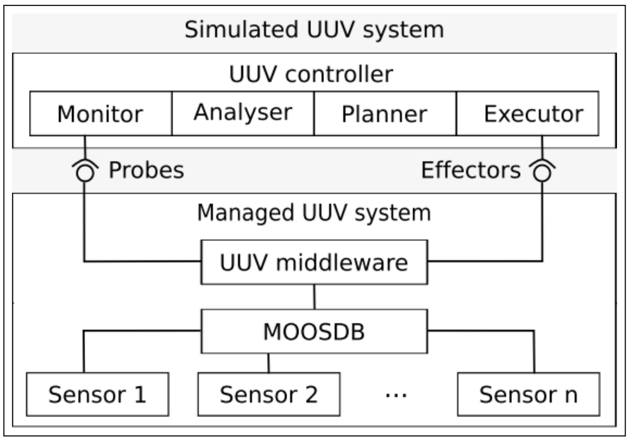


Figure 6.7 – High-level UNDERSEA architecture (From (GERASIMOU et al., 2017))

The UUV controller runs a MAPE loop, selects the desired vehicle speed and sensors configuration, and communicates its decision to the managed UUV system through Effectors. The managed UUV receives this decision through the UUV middleware and enforces the MOOS-IvP[5] to adapt the behavior of the UUV system by realizing the new configuration on sensors. UNDERSEA has 3078 LoC which can be considered a small size project so it was possible to map all abstractions in a short period of time.

The second one is TAS (WEYNS; CALINESCU, 2015), a reference implementation of a Tele Assistance System (TAS) that 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 (WEYNS; CALINESCU, 2015). Figure 6.8 shows the TAS architecture with abstractions that enables the adaptation mechanism. As it can be seen, the platform offers *probes* that monitor

[5] An open source middleware for engineering autonomous applications on unmanned marine vehicles `https://tinyurl.com/ya9n4mtv`

the service-based systems (SBSs), e.g., the WorkflowProbe monitors the start and completion of the workflow executions and individual service invocations. Also, ReSeP offers *effectors* that enbale runtime manipulation of the SBS architecture and parameters. An adaptation engine can use these probes and effectors to track SBS behavior and adapt the SBS dynamically. Table 6.3 shows information on the size of both systems.

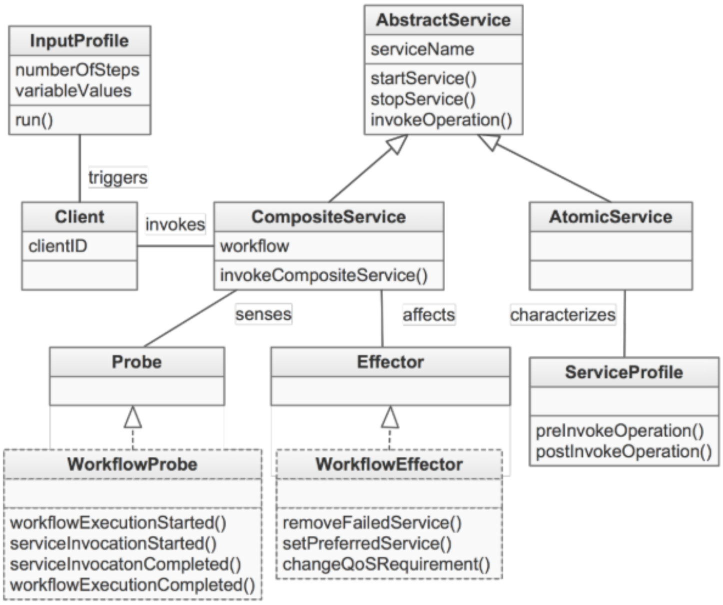


Figure 6.8 – Tele Assistance Service Based System

	Undersea	TAS
LOC	3,104	36,071
Packages	12	31
Components	44	103

Table 6.3 – Target systems used in the evaluation

6.4.3 Operation

6.4.3.1 Preparation

Undersea System

Listing 6.15 shows the PA used to identify the architectural drifts in the CA of UNDERSEA. The specification is composed of two subsystems, the Managing and Managed subsystems (lines 2 and 14), one control loop (line 3), the canonical abstractions (lines 4-8), two Reference Input (lines $9-10$) that represents the value of the UUV speed and the number of sensor configuration.

The Managed subsystem is composed of a Sensor (line 15), an Effector (line 16), two Measured Outputs (lines $17-18$ and four communication rules (lines $20-23$) were specified in section Rules. Notice that the keyword *withDomainRules* is declared in the loop so this assure that the twenty domain rules are enabled to generate OCL rules that will be checked in the CA.

Table 6.4 shows the oracle of UNDERSEA which is composed of two columns: Relationships and Type. The first one is the relationship that we discovered in the CA and the second one is the type of the relationships; a convergence (C), a divergence (D) or an absence (A). We found 29 relationships where 24 of them are absences compared to the PA and 5 of them are convergences.

```
 1 Architecture Undersea {
 2   Managing managing_1 {
 3     Loop loop_1 withDomainRules{
 4       Monitor monitor_1;
 5       Analyzer analyzer_1;
 6       Planner planner_1;
 7       Executor executor_1;
 8       Knowledge knowledge_1 {
 9          ReferenceInput rinput_1;
10          ReferenceInput rinput_2;
11       }
12     }
13   }
14   Managed managed_1 {
15      Sensor sensor_1;
16      Effector effector_1;
17      MeasuredOutput moutput_1;
18      MeasuredOutput moutput_2;
19   }
20 }
21 Rules{
22   monitor monitor_1 must-use sensor sensor_1;
23   analyzer analyzer_1 must-use reference-input rinput_1;
24   analyzer analyzer_1 must-use reference-input rinput_2;
25   sensor sensor_1 must-use measured-output moutput_1;
26   sensor sensor_1 must-use measured-output moutput_2;
27   executor executor_1 must-use effector effector_1;
28 }
```

Listing 6.15 – PA of UNDERSEA

TAS System

Listing 6.16 shows the PA uses to identify the architectural drifts in the CA of TAS. The specification is composed of two subsystems, the Managing and Managed subsystems (lines 2 and 14), one control loop (line 3), the canonical abstractions (lines 4-7), a Reference Input (line 8) that represents the value of the maximum response time of a service and an Alternative (line 9) that corresponds to a set of adaptation alternatives to replace a failed service. Notice that in this case we do not specify a Planner abstraction because according to our source-code analysis it was not considered as part of the implementation.

The Managed subsystem is composed of a Sensor (line 14), an Effector (line 15), a Measured Outputs (lines 16 and five communication rules (lines $20-24$) were specified in section Rules. In the same way as we done for UNDERSEA, the keyword *withDomainRules* is declared

Relationships	Type
[1] **monitor** monitor_1 use **monitor** monitor_1;	A
[2] **analyzer** analyzer_1 use **analyzer** analyzer_1;	A
[3] **planner** planner_1 use **planner** planner_1;	A
[4] **executor** executor_1 use **executor** executor_1;	A
[5] **monitor** monitor_1 use **knowledge** knowledge_1;	C
[6] **analyzer** analyzer_1 use **knowledge** knowledge_1;	C
[7] **planner** planner_1 use **knowledge** knowledge_1;	C
[8] **executor** executor_1 use **knowledge** knowledge_1;	C
[9] **loop** loop_1 use **monitor** monitor_1;	A
[10] **loop** loop_1 use **analyzer** analyzer_1;	A
[11] **loop** loop_1 use **planner** planner_1;	A
[12] **loop** loop_1 use **executor** executor_1;	A
[13] **loop** loop_1 use **sensor** sensor_1;	A
[14] **loop** loop_1 use **effector** effector_1;	A
[15] **knowledge** knowledge_1 use **sensor** sensor_1;	A
[16] **planner** planner_1 use **reference-input** rinput_1;	A
[17] **effector** effector_1 use **executor** executor_1;	A
[18] **planner** planner_1 use **reference-input** rinput_2;	A
[19] **loop** loop_1 use **knowledge** knowledge_1;	A
[20] **loop** loop_1 use **measured-output** moutput_2;	A
[21] **sensor** sensor_1 use **measured-output** moutput_2;	C
[22] **analyzer** analyzer_1 use **measured-output** moutput_2;	A
[23] **analyzer** analyzer_1 use **sensor** sensor_1;	A
[24] **planner** planner_1 use **reference-input** rinput_2;	A
[25] **knowledge** knowledge_1 use **knowledge** knowledge_1;	A
[26] **knowledge** knowledge_1 use **sensor** sensor_1;	A
[27] **sensor** sensor_1 use **knowledge** knowledge_1;	A
[28] **planner** planner_1 use **measured-output** measured-output_1;	A
[29] **effector** effector_1 use **sensor** sensor_1;	A

Table 6.4 – Oracle for UNDERSEA

in the loop to assure that the twenty domain rules are enabled to generate the OCL rules that will be checked in the CA.

Table 6.5 shows the oracle of TAS which is similar in structure to the oracle of UNDERSEA. We found 19 relationships where 15 of them are absences compared to the PA and 2 of them are convergences and 2 of them are divergences. In this system the Managing subsystem abstraction was mapped to a class called *MyAdaptationEngine* which realizes the monitoring and effectoring, however the analyzer is not composing the Managing but we found it implemented by the method *invokeServiceOperation* in the *CompositeService* class that was mapped as the Managed subsystem.

```
Architecture TasArchitecture {
  Managing managing_1 {
    Loop loop_1 withDomainRules{
      Monitor monitor_1;
      Analyzer analyzer_1;
      Executor executor_1;
      Knowledge knowledge_1 {
         ReferenceInput rinput_1;
         Alternative alternative_1;
      }
    }
  }
  Managed managed_1 {
     Sensor sensor_1;
     Effector effector_1;
     MeasuredOutput moutput_1;
  }
}
Rules{
  monitor monitor_1 must-use sensor sensor_1;
  analyzer analyzer_1 must-use reference-input rinput_1;
  sensor sensor_1 must-use measured-output moutput_1;
  executor executor_1 must-use effector effector_1;
  analyzer analyzer_1 must-use alternative alternative_1;
}
```

Listing 6.16 – PA of TAS

Relationships	Type
[1] **monitor** monitor_1 use **managing_1** managing_1;	A
[2] **managing** managing_1 use **monitor** monitor_1;	A
[3] **alternative** alternative_1 use **executor** executor_1;	A
[4] **executor** executor_1 use **effector** effector_1;	C
[5] **executor** executor_1 use **managed** managed_1;	A
[6] **effector** effector_1 use **managed** managed_1;	A
[7] **managed** managed_1 use **analyzer** analyzer_1;	A
[8] **managed** managed_1 use **measured-output** moutput_1;	A
[9] **analyzer** analyzer_1 use **measured-output** moutput_1;	A
[10] **analyzer** analyzer_1 use **analyzer** analyzer_1;	A
[11] **analyzer** analyzer_1 use **reference-input** rinput_1;	C
[12] **managed** managed_1 use **managed** managed_1;	A
[13] **managed** managed_1 use **knowledge** knowledge_1;	A
[14] **managed** managed_1 use **monitor** monitor_1;	A
[15] **managed** managed_1 use **reference-input** rinput_1;	A
[16] **managed** managed_1 use **reference-input** rinput_1;	A
[17] **analyzer** analyzer_1 use **alternative** alternative_1;	D
[18] **managing** managing_1 use **effector** effector_1;	A
[19] **analyzer** analyzer_1 use **knowledge** knowledge_1;	D

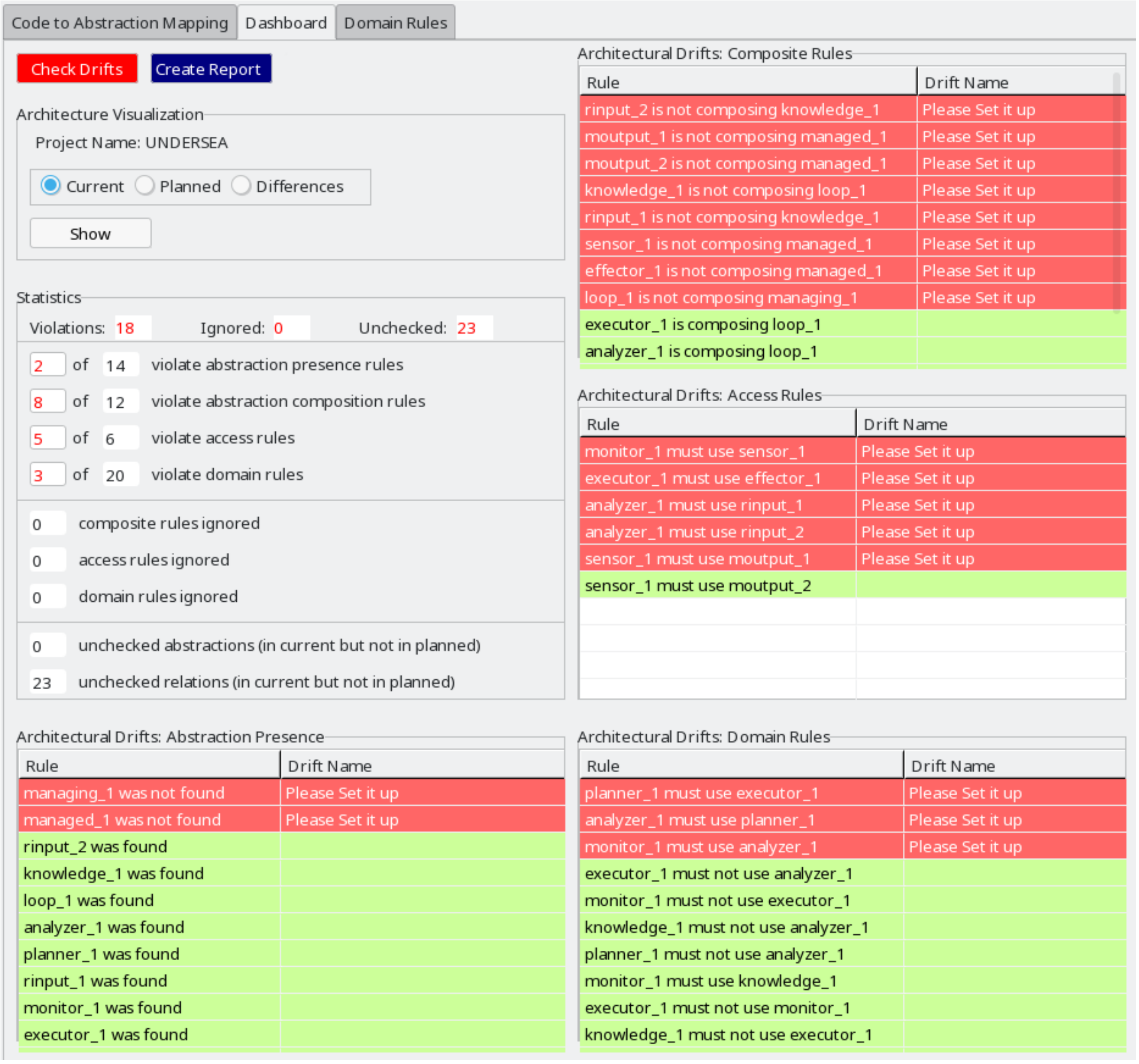
Table 6.5 – Oracle for TAS

6.4.3.2 Execution & Data Validation

Undersea System

The execution consisted of performing the domain drift detection. Figure 6.9 shows the REMEDY Dashboard with all the statistics after the checking process. REMEDY found 18 violations; 2 related to abstraction presence, 8 related to composite rules, 5 related to access rules and 3 related to domain rules. Notice that there are 23 unchecked relations or absences (relations that exist in current but not in planned), but our oracle contains 24 absences relationships.

After a close look we realized that the relationship number 27 of our oracle was not generated by REMEDY. The cause of this missing was a lack of information on the KDM instance of the CA because although the *Sensor* class in the *parseReply* method, makes a static call to the *Knowledge* class, the Modisco plugin did not generate properly the method call. Nevertheless, as in our PA we do not specify an access rule from sensor to knowledge, the relationship is not considered in the calculation of the metrics.



Architecture Visualization

Project Name: UNDERSEA

● Current ○ Planned ○ Differences

Statistics

Violations: 18 Ignored: 0 Unchecked: 23

2 of 14 violate abstraction presence rules

8 of 12 violate abstraction composition rules

5 of 6 violate access rules

3 of 20 violate domain rules

0 composite rules ignored

0 access rules ignored

0 domain rules ignored

0 unchecked abstractions (in current but not in planned)

23 unchecked relations (in current but not in planned)

Architectural Drifts: Composite Rules

Rule	Drift Name
rinput_2 is not composing knowledge_1	Please Set it up
moutput_1 is not composing managed_1	Please Set it up
moutput_2 is not composing managed_1	Please Set it up
knowledge_1 is not composing loop_1	Please Set it up
rinput_1 is not composing knowledge_1	Please Set it up
sensor_1 is not composing managed_1	Please Set it up
effector_1 is not composing managed_1	Please Set it up
loop_1 is not composing managing_1	Please Set it up
executor_1 is composing loop_1	
analyzer_1 is composing loop_1	

Architectural Drifts: Access Rules

Rule	Drift Name
monitor_1 must use sensor_1	Please Set it up
executor_1 must use effector_1	Please Set it up
analyzer_1 must use rinput_1	Please Set it up
analyzer_1 must use rinput_2	Please Set it up
sensor_1 must use moutput_1	Please Set it up
sensor_1 must use moutput_2	

Architectural Drifts: Abstraction Presence

Rule	Drift Name
managing_1 was not found	Please Set it up
managed_1 was not found	Please Set it up
rinput_2 was found	
knowledge_1 was found	
loop_1 was found	
analyzer_1 was found	
planner_1 was found	
rinput_1 was found	
monitor_1 was found	
executor_1 was found	

Architectural Drifts: Domain Rules

Rule	Drift Name
planner_1 must use executor_1	Please Set it up
analyzer_1 must use planner_1	Please Set it up
monitor_1 must use analyzer_1	Please Set it up
executor_1 must not use analyzer_1	
monitor_1 must not use executor_1	
knowledge_1 must not use analyzer_1	
planner_1 must not use analyzer_1	
monitor_1 must use knowledge_1	
executor_1 must not use monitor_1	
knowledge_1 must not use executor_1	

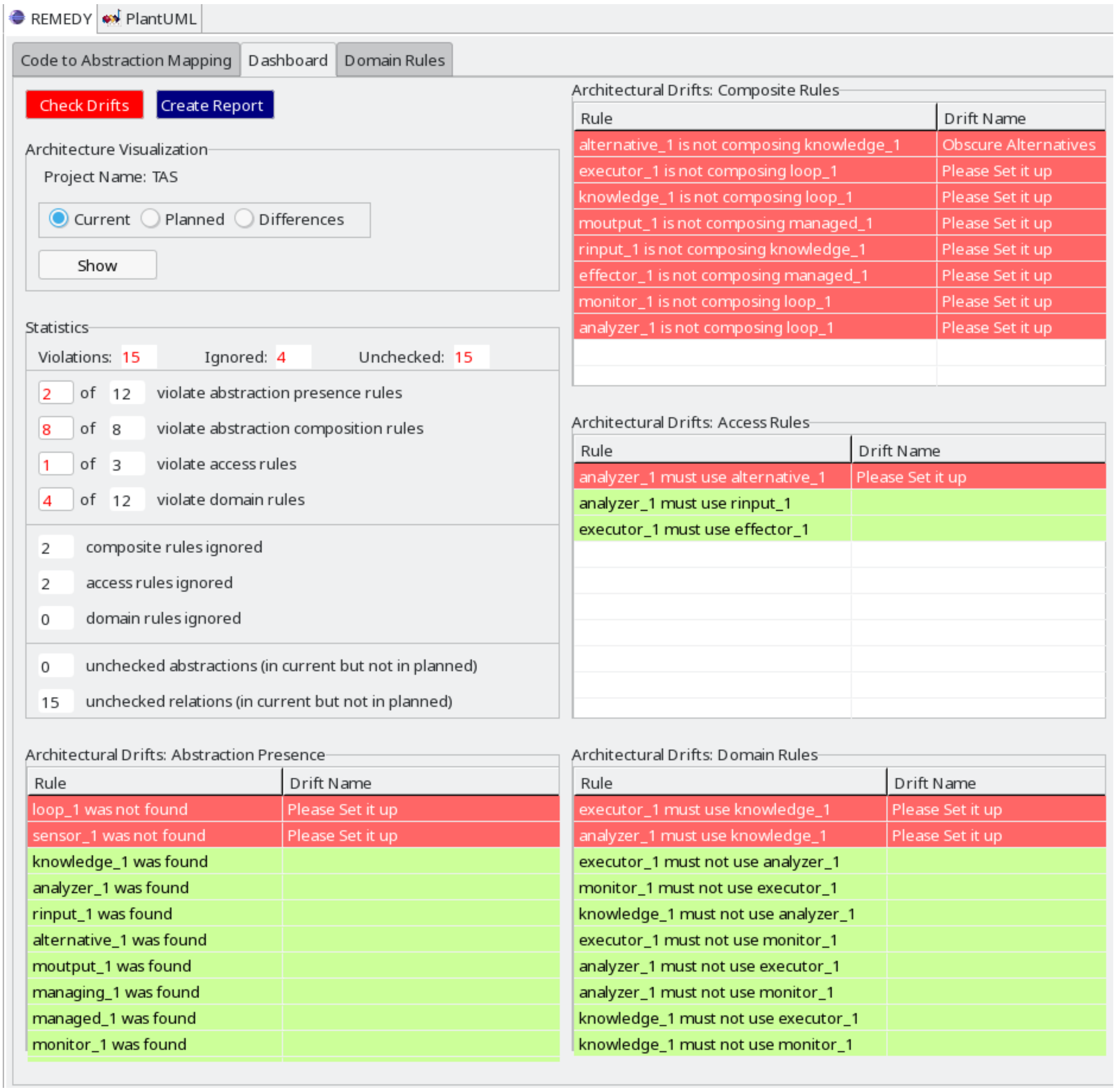
Figure 6.9 – Architecture Conformance Checking on UNDERSEA

TAS System

Figure 6.10 shows the REMEDY Dashboard with all the statistics after the checking process.

REMEDY found 15 violations; 2 related to abstraction presence, 8 related to composite rules, 1 related to access rules and 4 related to domain rules. Herein REMEDY found 15 absences.

In this case, REMEDY failed in identifying correctly two rules of our oracle; rule 17 and rule 19. Similarly to the previous system, there is a lack of information in the generated KDM CA. Particularly the problem occurs due to MoDisco cannot represent the relationship *HasValue* correctly. *HasValue* is a specific meta-model element that represents semantic relation between a data element and its initialization element, which can be a data element or an action element (OMG, 2016). Thus for rules 17 and 19 the elements that implement the architectural abstractions are being communicated with this type of relationship but the KDM instance fail to provide the destination of the relationship. We also report this problem in Landi, Santibanez, Santos, Cunha, Durelli, and Camargo (2022).



Architecture Visualization

Project Name: TAS

● Current ○ Planned ○ Differences

Statistics

Violations: 15 Ignored: 4 Unchecked: 15

2 of 12 violate abstraction presence rules

8 of 8 violate abstraction composition rules

1 of 3 violate access rules

4 of 12 violate domain rules

2 composite rules ignored

2 access rules ignored

0 domain rules ignored

0 unchecked abstractions (in current but not in planned)

15 unchecked relations (in current but not in planned)

Architectural Drifts: Composite Rules

Rule	Drift Name
alternative_1 is not composing knowledge_1	Obscure Alternatives
executor_1 is not composing loop_1	Please Set it up
knowledge_1 is not composing loop_1	Please Set it up
moutput_1 is not composing managed_1	Please Set it up
rinput_1 is not composing knowledge_1	Please Set it up
effector_1 is not composing managed_1	Please Set it up
monitor_1 is not composing loop_1	Please Set it up
analyzer_1 is not composing loop_1	Please Set it up

Architectural Drifts: Access Rules

Rule	Drift Name
analyzer_1 must use alternative_1	Please Set it up
analyzer_1 must use rinput_1	
executor_1 must use effector_1	

Architectural Drifts: Abstraction Presence

Rule	Drift Name
loop_1 was not found	Please Set it up
sensor_1 was not found	Please Set it up
knowledge_1 was found	
analyzer_1 was found	
rinput_1 was found	
alternative_1 was found	
moutput_1 was found	
managing_1 was found	
managed_1 was found	
monitor_1 was found	

Architectural Drifts: Domain Rules

Rule	Drift Name
executor_1 must use knowledge_1	Please Set it up
analyzer_1 must use knowledge_1	Please Set it up
executor_1 must not use analyzer_1	
monitor_1 must not use executor_1	
knowledge_1 must not use analyzer_1	
executor_1 must not use monitor_1	
analyzer_1 must not use executor_1	
analyzer_1 must not use monitor_1	
knowledge_1 must not use executor_1	
knowledge_1 must not use monitor_1	

Figure 6.10 – Architecture Conformance Checking on TAS

6.4.4 Analysis

The goal here is to determine if REMEDY is able to identify structural, communication and domain rules with high accuracy. Thus we calculate the precision and recall metrics to our conformance checking process for UNDERSEA and TAS systems. Table 6.6 and Table 6.7 show these two metrics and as it can be seen for both systems, the precision and recall reached 100% for structural rules. Regarding to communicatoin and domain rules, the values are slightly different but with good values.

$$
\begin{array}{ll}
Precision_{structural} = \frac{26}{26} = 1 \longrightarrow 100\% & Recall = \frac{26}{26} = 1 \longrightarrow 100\% \\
Precision_{communication} = \frac{6}{6} = 1 \longrightarrow 100\% & Recall = \frac{6}{6} = 1 \longrightarrow 100\% \\
Precision_{domain} = \frac{20}{20} = 1 \longrightarrow 100\% & Recall = \frac{20}{20} = 1 \longrightarrow 100\%
\end{array}
$$

Table 6.6 – Precision and Recall for UNDERSEA

$$
\begin{array}{ll}
Precision_{structural} = \frac{20}{20} = 1 \longrightarrow 100\% & Recall = \frac{20}{20} = 1 \longrightarrow 100\% \\
Precision_{communication} = \frac{2}{2} = 1 \longrightarrow 100\% & Recall = \frac{2}{3} = 0.66 \longrightarrow 67\% \\
Precision_{domain} = \frac{19}{19} = 1 \longrightarrow 100\% & Recall = \frac{19}{20} = 0.95 \longrightarrow 95\%
\end{array}
$$

Table 6.7 – Precision and Recall for TAS

Therefore, the answer to **RQ1**, **RQ2** and **RQ3** is that REMEDY achieve high levels of accuracy in the detection of architectural drifts of AS when the information of the KDM CA is correct and complete. Also, it is important that the domain expert can map accurately the AS abstractions in source-code in order to REMEDY creates the KDM CA with trusted information to checked by the OCL rules from the PA. We are aware that more examples will be needed to identify other possible flaws on the completeness of the KDM generated by MoDisco plugin.

6.4.5 Threats of Validity

We must state at least three threats of the reported evaluation. First, even though we rely on a simulation but representative system of SEAMS community, we cannot claim that our approach will provide equivalent accuracy rates in other ASs, as it usually happens in empirical studies of software engineering which can be classified as an external validity. Second, we relied on the PA and CA were correctly specified but in order to minimize any bias a deep analysis were performed on these artifacts by the authors. As typical in human-based classifications, our results might be affected by some degree of subjectivity also known as construct validity.

6.5 Chapter Summary

We have presented REMEDY, an ACC approach whose focus is to identify drifts in Adaptive Systems. As our approach involves a domain-specific part, software architects do not need to create rules for checking dependencies stated by the MAPE-K reference model, just inform which of them need to be checked. Although generic ACC approaches have the advantage of being applied in a vast set of systems, there some points in favor of domain-dependent ACC approaches:

- When working with specific domains, during the Mapping step, generic approaches ask architects to map the generic elements declared in the PA (like components) to more specific ones (like monitors). This is not so straightforward because there are a smaller number of generic components than the number of specific ones;
- Besides, domain-specific elements have pre-defined behavior that is lost when this mapping is done;
- ACC domain-independent used for specifying the PA is not able to incorporate domain-specific rules, forcing architects to write rules which are very common and canonical of the domain. This waste of time could be avoided;
- Considering that the architects know the ASs domain, they can be much more precise using a language which delivers the canonical abstractions of the domain, as they already know the behavior of them.

Regarding when and how to use REMEDY, there are some considerations. *i*): The DSL can be used at the early stages of development for creating the PA of a new system. In this case, the PA guide the development process but, only when there are some source code available, the ACC process could be performed; *ii*) The checking process can also be conducted for existing systems whose PA was not previously specified with our DSL. However, in this case, the PA specification must be created and, many times, the problem is to recuperate/remember the architectural decisions established in the beginning of the development. But, if documents are describing the PA or experience architects, this is quite straightforward; *iii*) The frequency of the checking process can vary. Once the PA specification and Mapping are done, it can be triggered for every commit or a monthly checking process according to software architects need.

Chapter 7

TOOLING SUPPORT: REMEDY

In this chapter we provide an overall description of a prototype workbench called REMEDY. Its source code is available in GitHub[1] and it has been developed for more than four years. We also provide a detail of each one of its components. Section 7.1 presents the using of the REMEDY through its user interfaces from the software architect's point of view. Section 7.2 presents the architecture of REMEDY.

7.1 Remedy Workbench

In this section we present the user interfaces of REMEDY that support software architects in performing the ACC process which is depicted in detail in Chapter 6. Recalling our process, it is conformed by three main steps. In the first step software architects specify structural and communication rules of the architecture of an Adaptive System (AS) by means of a textual DSL. In the second step software architects map source code elements to architectural elements of ASs and finally in the third step the conformance checking is performed to find architectural drifts.

Figure 7.1 shows the Eclipse Platform with some projects available in the Project Explorer part. The ACC process starts by selecting a project and opening a pop-up menu by clicking on the menu called “REMEDY: Architectural Checking for Adaptive Concepts”. This will change the Eclipse perspective to enable all the tools the software architect needs to perform the ACC process. Also, the project explorer is updated with some artifacts that are necessary to aid in the ACC and later in this section we discuss them.

We exemplify REMEDY by means of an AS called PhoneAdapter (SAMA et al., 2010). It characterizes itself as “adaptive system” because it uses context information to adapt the mobile phone profile. Phone profile is a configuration that determine the phone's behavior. This configuration may involve the display intensity, ring tone volume, vibration mode, Bluetooth discovery and others. Due to its adaptability capacity, the application automatically changes

[1] https://github.com/dsanmartins/REMEDY/tree/developing

between profiles based on rules previously set. The selected profile prevails until a more suitable one is activated through other rules.

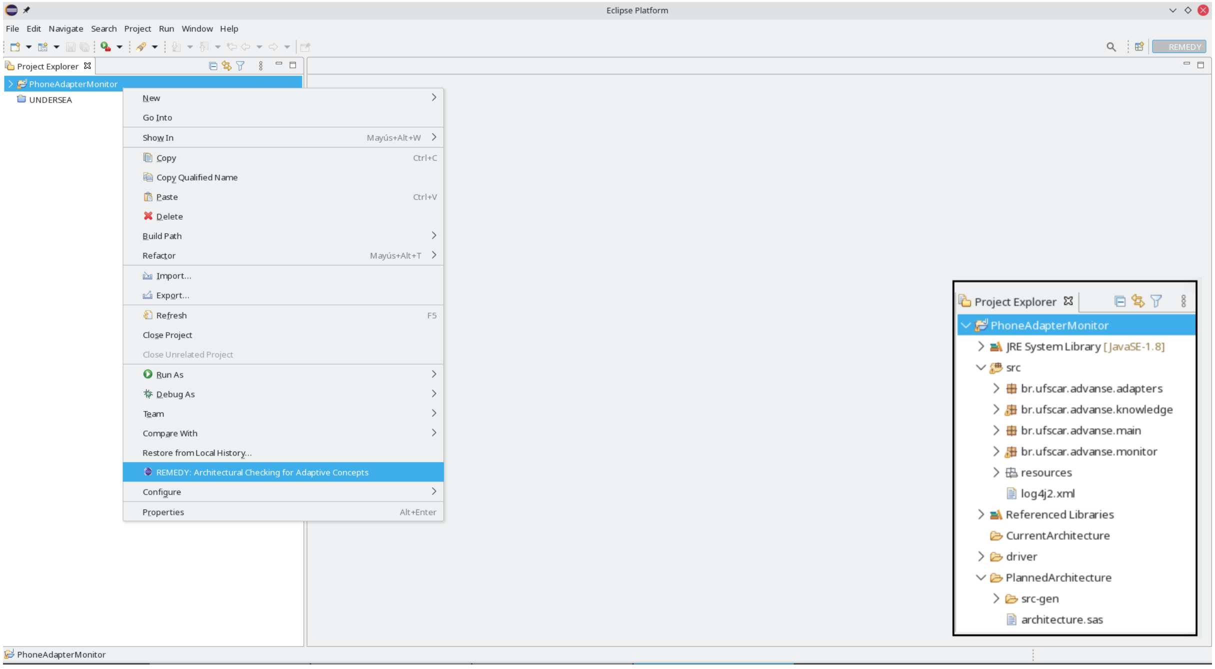
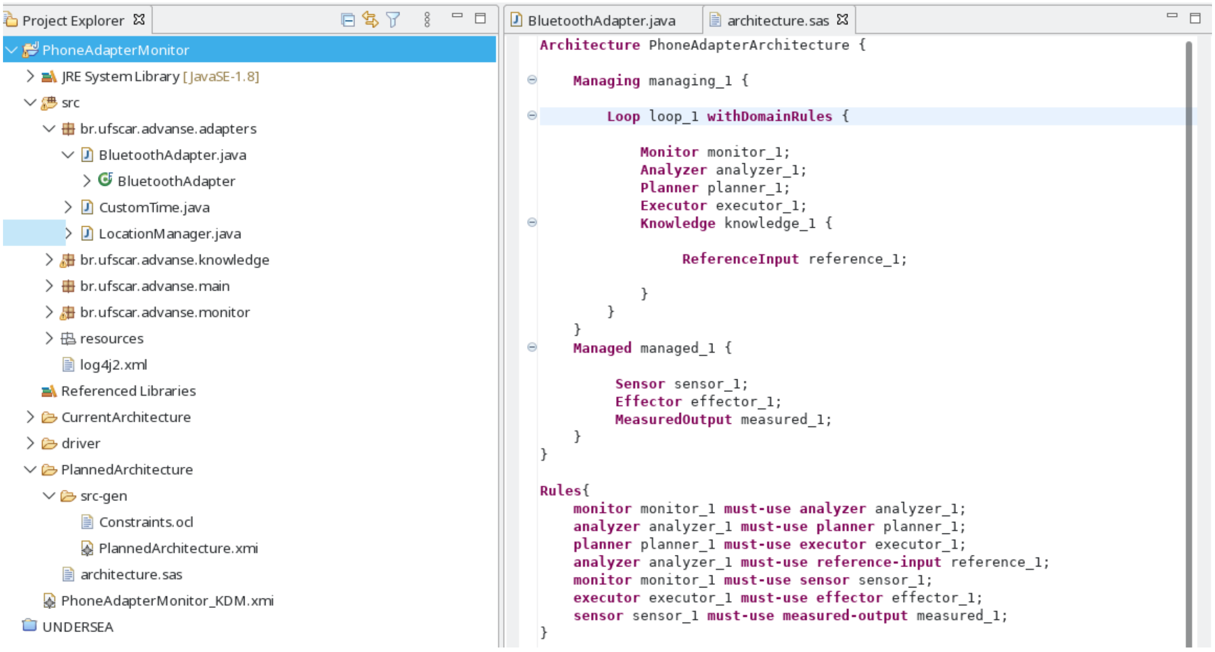


Figure 7.1 – Pop-up menu to setup the ACC process

In order to proceed with the ACC, it is necessary to have the specification of the Planned Architecture (PA). Therefore, if this specification already exists, the checking process can move on. If it does not exist, one must firstly create it. To make easier the process of creating/elaborating the planned architecture, REMEDY provides an exemplar of the file called *architecture.sas* that can be modified according to the needs of the software architect. The editor we have developed for specifying the PA provides all functionalities expected by an editor such as copy and paste, automatic indentation and code auto-completion.

Figure 7.2 shows an hypothetical PA for PhoneAdapter that is used through this section for exemplification purposes. As it can be seen, this PA is composed of structural rules, communication rules and domain rules. Structural Rules belong to the content of *Architecture* section, Communication Rules belong to the content of *Rules* section and Domain Rules are enabled by the special keyword *withDomainRules*. Once the architectural specification is completed the software architect can proceed to recover the Current Architecture (CA) by mapping AS elements to source code elements.

Notice that our PA specifies twelve architectural elements of AS. For each element, our DSL will create an ocl rule to check the existence of the element and the composition of it if correspond. Thus, regarding to structural rules DSL-REMEDY will create twelve ocl rules of existence and ten ocl rules of composition (the managing and the managed abstractions are not composable in a major abstraction). Regarding to communication rules, for each one of them DSL-REMEDY will create an ocl rule which includes rules specified by the software architect and domain rules. In our example there are twenty seven communication rules where seven are specified in the PA and twenty come from the domain rules. As the rule “`monitor_1 must-use analyzer analyzer_1;`” was specified but also domain rules were enabled, REMEDY will remove the duplicated rule. Therefore, the PA will create a total quantity of forty eight ocl rules to be checked in a CA.



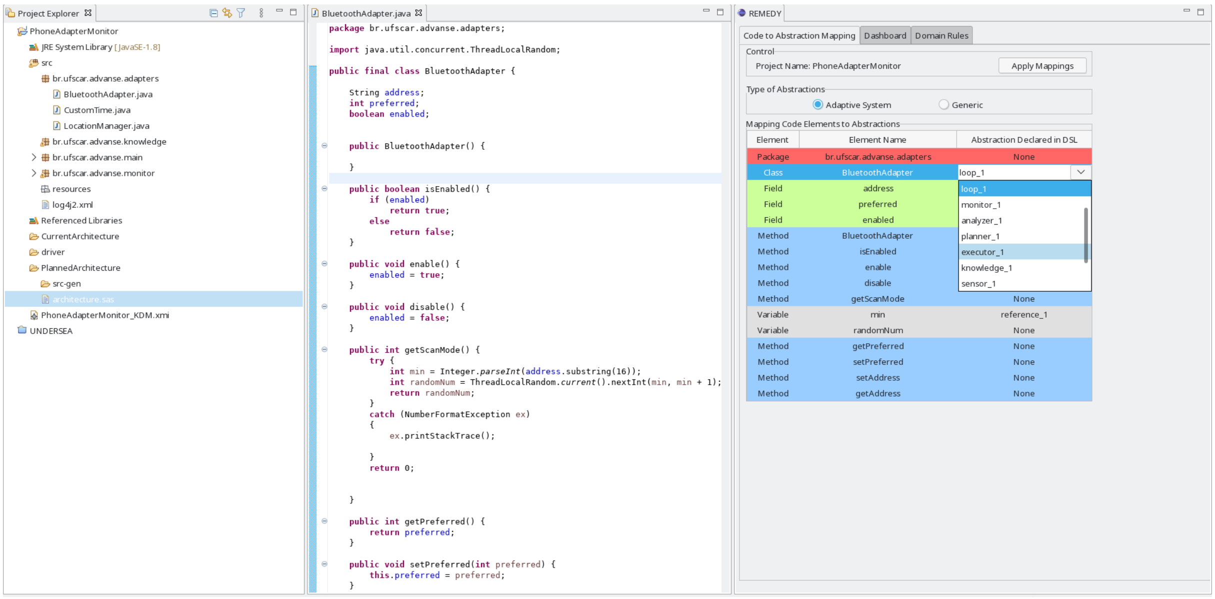
```
Architecture PhoneAdapterArchitecture {

    Managing managing_1 {

        Loop loop_1 withDomainRules {

            Monitor monitor_1;
            Analyzer analyzer_1;
            Planner planner_1;
            Executor executor_1;
            Knowledge knowledge_1 {

                ReferenceInput reference_1;

            }
        }
    }
    Managed managed_1 {

        Sensor sensor_1;
        Effector effector_1;
        MeasuredOutput measured_1;
    }
}

Rules{
    monitor monitor_1 must-use analyzer analyzer_1;
    analyzer analyzer_1 must-use planner planner_1;
    planner planner_1 must-use executor executor_1;
    analyzer analyzer_1 must-use reference-input reference_1;
    monitor monitor_1 must-use sensor sensor_1;
    executor executor_1 must-use effector effector_1;
    sensor sensor_1 must-use measured-output measured_1;
}
```

Figure 7.2 – Specify the planned architecture through the editor

Figure 7.3 shows the main window of REMEDY which is composed of three tabs: Code to Abstraction Mapping, the Dashboard and Domain Rules. Particularly, Figure 7.3 is showing the first one where software architects map the architectural abstractions declared in the DSL to souce code elements of the system under analysis. REMEDY provides two types of architectural elements; generic types (components, layers, modules) and AS types (domain abstractions) that can be accessed by clicking on the radio button according to the preferences. These elements are taken from the specification of the PA to fill comboboxes of the column Abstraction in the grid.

In order to perform the mappings software architects must open each class file of the eclipse project, thus the content of the class such as package name, class name, fields, methods and variables will appear on the grid where each element is corresponding to one row. Once the mapping is completed, software architects must press the “Apply Mappings” button to recovery the CA. For instance, as can be seen the *min* variable was mapped to the architectural element *reference_1* that is a Reference Input abstraction.

At this point, the software architect has created the PA and the CA so the next step is the execution of the conformance checking. Figure 7.4 depicts the dashboard of REMEDY that is conformed by nine sections: Architecture Visualizations, Statistics, Architectural Drifts (Abstraction Presence, Composite/Structural Rules, Access/Communication Rules, Domain Rules), Architectural Rules Ignored Architectural Rules and two pie charts (Valid Rules and All Rules).



```
PhoneAdapterMonitor
    JRE System Library [JavaSE-1.8]
    src
        br.ufscar.advanse.adapters
            BluetoothAdapter.java
            CustomTime.java
            LocationManager.java
        br.ufscar.advanse.knowledge
      > br.ufscar.advanse.main
      > br.ufscar.advanse.monitor
        resources
        log4j2.xml
    Referenced Libraries
    CurrentArchitecture
    driver
    PlannedArchitecture
        src-gen
        architecture.sas
    PhoneAdapterMonitor_KDM.xmi
UNDERSEA
```

```java
package br.ufscar.advanse.adapters;

import java.util.concurrent.ThreadLocalRandom;

public final class BluetoothAdapter {

    String address;
    int preferred;
    boolean enabled;


    public BluetoothAdapter() {

    }

    public boolean isEnabled() {
        if (enabled)
            return true;
        else
            return false;
    }

    public void enable() {
        enabled = true;
    }

    public void disable() {
        enabled = false;
    }

    public int getScanMode() {
        try {
            int min = Integer.parseInt(address.substring(16));
            int randomNum = ThreadLocalRandom.current().nextInt(min, min + 1);
            return randomNum;
        }
        catch (NumberFormatException ex)
        {
            ex.printStackTrace();

        }
        return 0;


    }

    public int getPreferred() {
        return preferred;
    }

    public void setPreferred(int preferred) {
        this.preferred = preferred;
    }
```

Control

Project Name: PhoneAdapterMonitor

Type of Abstractions

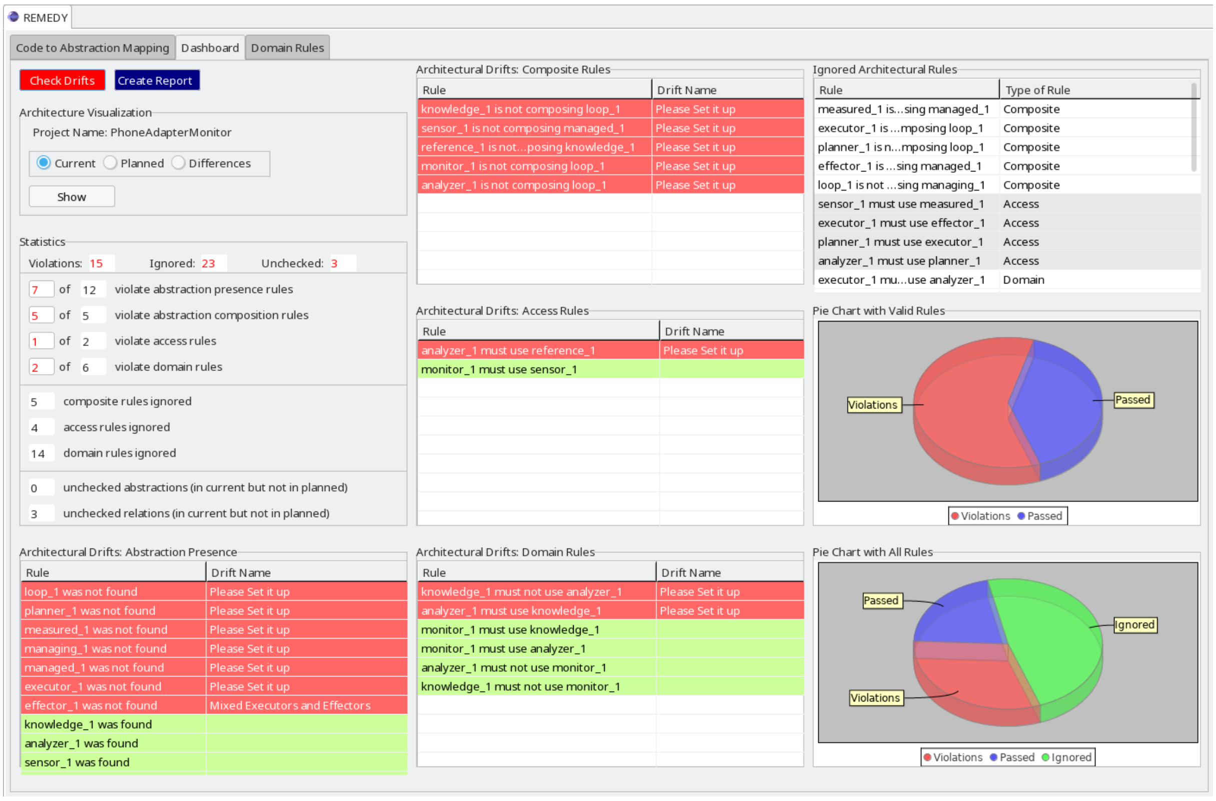
● Adaptive System ○ Generic

Mapping Code Elements to Abstractions

Element	Element Name	Abstraction Declared in DSL
Package	br.ufscar.advanse.adapters	None
Class	BluetoothAdapter	loop_1
Field	address	
Field	preferred	
Field	enabled	
Method	BluetoothAdapter	
Method	isEnabled	
Method	enable	
Method	disable	
Method	getScanMode	None
Variable	min	reference_1
Variable	randomNum	None
Method	getPreferred	None
Method	setPreferred	None
Method	setAddress	None
Method	getAddress	None

loop_1
monitor_1
analyzer_1
planner_1
executor_1
knowledge_1
sensor_1

Figure 7.3 – Mapping architectural elements to code elements



Architecture Visualization

Project Name: PhoneAdapterMonitor

● Current ○ Planned ○ Differences

Statistics

Violations: 15 Ignored: 23 Unchecked: 3

7 of 12 violate abstraction presence rules

5 of 5 violate abstraction composition rules

1 of 2 violate access rules

2 of 6 violate domain rules

5 composite rules ignored

4 access rules ignored

14 domain rules ignored

0 unchecked abstractions (in current but not in planned)

3 unchecked relations (in current but not in planned)

Architectural Drifts: Composite Rules

Rule	Drift Name
knowledge_1 is not composing loop_1	Please Set it up
sensor_1 is not composing managed_1	Please Set it up
reference_1 is not...posing knowledge_1	Please Set it up
monitor_1 is not composing loop_1	Please Set it up
analyzer_1 is not composing loop_1	Please Set it up

Ignored Architectural Rules

Rule	Type of Rule
measured_1 is...sing managed_1	Composite
executor_1 is ...mposing loop_1	Composite
planner_1 is n...mposing loop_1	Composite
effector_1 is ...sing managed_1	Composite
loop_1 is not ...sing managing_1	Composite
sensor_1 must use measured_1	Access
executor_1 must use effector_1	Access
planner_1 must use executor_1	Access
analyzer_1 must use planner_1	Access
executor_1 mu...use analyzer_1	Domain

Architectural Drifts: Access Rules

Rule	Drift Name
analyzer_1 must use reference_1	Please Set it up
monitor_1 must use sensor_1	

Pie Chart with Valid Rules

Architectural Drifts: Abstraction Presence

Rule	Drift Name
loop_1 was not found	Please Set it up
planner_1 was not found	Please Set it up
measured_1 was not found	Please Set it up
managing_1 was not found	Please Set it up
managed_1 was not found	Please Set it up
executor_1 was not found	Please Set it up
effector_1 was not found	Mixed Executors and Effectors
knowledge_1 was found	
analyzer_1 was found	
sensor_1 was found	

Architectural Drifts: Domain Rules

Rule	Drift Name
knowledge_1 must not use analyzer_1	Please Set it up
analyzer_1 must use knowledge_1	Please Set it up
monitor_1 must use knowledge_1	
monitor_1 must use analyzer_1	
analyzer_1 must not use monitor_1	
knowledge_1 must not use monitor_1	

Pie Chart with All Rules

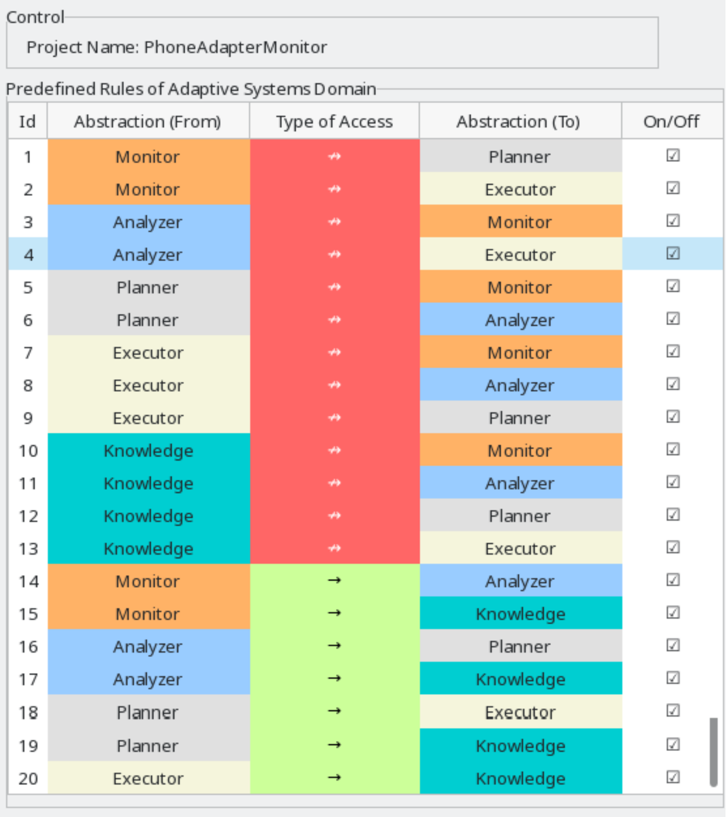
Figure 7.4 – Dashboard of REMEDY

The execution of the ACC begins when the button *Check Drifts* is pressed (top left of dashboard). The information does not take a long time in appear but it depends of the quantity of rules to be checked. In our example, from the total quantity of forty eight ocl rules, twenty five were considered valid rules and twenty three were ignored because there are architectural elements that were not found in the CA (see Abstraction Presence grid in the dashboard). From valid rules, effectively checked by REMEDY (25), 15 of them are violated in the CA: (7 abstraction presence, 5 composition rules, 1 access rule and 2 domain rules).

Although rules of the Ignored Architectural Rules grid can be interpreted as violated rules, the intention of this separation is to visualize what are the valid rules that were effectively checked. Also, it has technical implications because as these rules are not validated by REMEDY, REMEDY is not overloaded with rules that beforehand are known as violated. The Unchecked statistic means the quantity of abstractions and relations that are presented in the CA but not in the PA. For instance, in our example all abstractions of the CA were defined in the PA, that is why the number is zero. On the other hand, there are three relations that exist in CA but not in PA. Pie Charts provide a graphical view of the information already shown in Statistic section.

All grids that show drifts have two columns; column *Rule* shows the rule that have been checked and column *Drift Name* shows a name that identify the drift and should be assigned manually by the software architect. Rows in red color indicate that these rules does not passed the checking while rules in green rows passed the checking. For instance, in the Abstraction Presence grid, *effector_1* does not passed the checking (it was not found in the CA) so it could indicate the presence of Mixed Executors and Effectors drift.

Although the keyword *withDomainRules* enables the generator of domain rules, it only occurs at DSL level. Software architects must specify which domain rules should be generated by the DSL by means of a UI support. Figure 7.5 shows the domain rules interface which is a grid that contains five columns.



Control

Project Name: PhoneAdapterMonitor

Predefined Rules of Adaptive Systems Domain

Id	Abstraction (From)	Type of Access	Abstraction (To)	On/Off
1	Monitor	↛	Planner	☑
2	Monitor	↛	Executor	☑
3	Analyzer	↛	Monitor	☑
4	Analyzer	↛	Executor	☑
5	Planner	↛	Monitor	☑
6	Planner	↛	Analyzer	☑
7	Executor	↛	Monitor	☑
8	Executor	↛	Analyzer	☑
9	Executor	↛	Planner	☑
10	Knowledge	↛	Monitor	☑
11	Knowledge	↛	Analyzer	☑
12	Knowledge	↛	Planner	☑
13	Knowledge	↛	Executor	☑
14	Monitor	→	Analyzer	☑
15	Monitor	→	Knowledge	☑
16	Analyzer	→	Planner	☑
17	Analyzer	→	Knowledge	☑
18	Planner	→	Executor	☑
19	Planner	→	Knowledge	☑
20	Executor	→	Knowledge	☑

Figure 7.5 – Domain rules interface

The first column holds the ID of the rule, the second and fourth columns holds the canonical abstractions of MAPE-K (monitor, analyzer, planner, executor and knowledge) where the second represents the caller (from) and the fourth the callee (to). The third column holds the type of access (must-use or must-not-use) and the fifth column holds a switch to enable or disable the rule.

In our example, the twenty domain rules were enabled but just six of them were checked by REMEDY because, as several abstractions do not exist in the CA, they are ignored. From that universe, two domain rules not passed the checking while four passed. Regarding to the pie charts *Valid Rules* shows the percentages of violated rules and passed rules while *All Rules* shows the percentages of violated, passed and ignored rules.

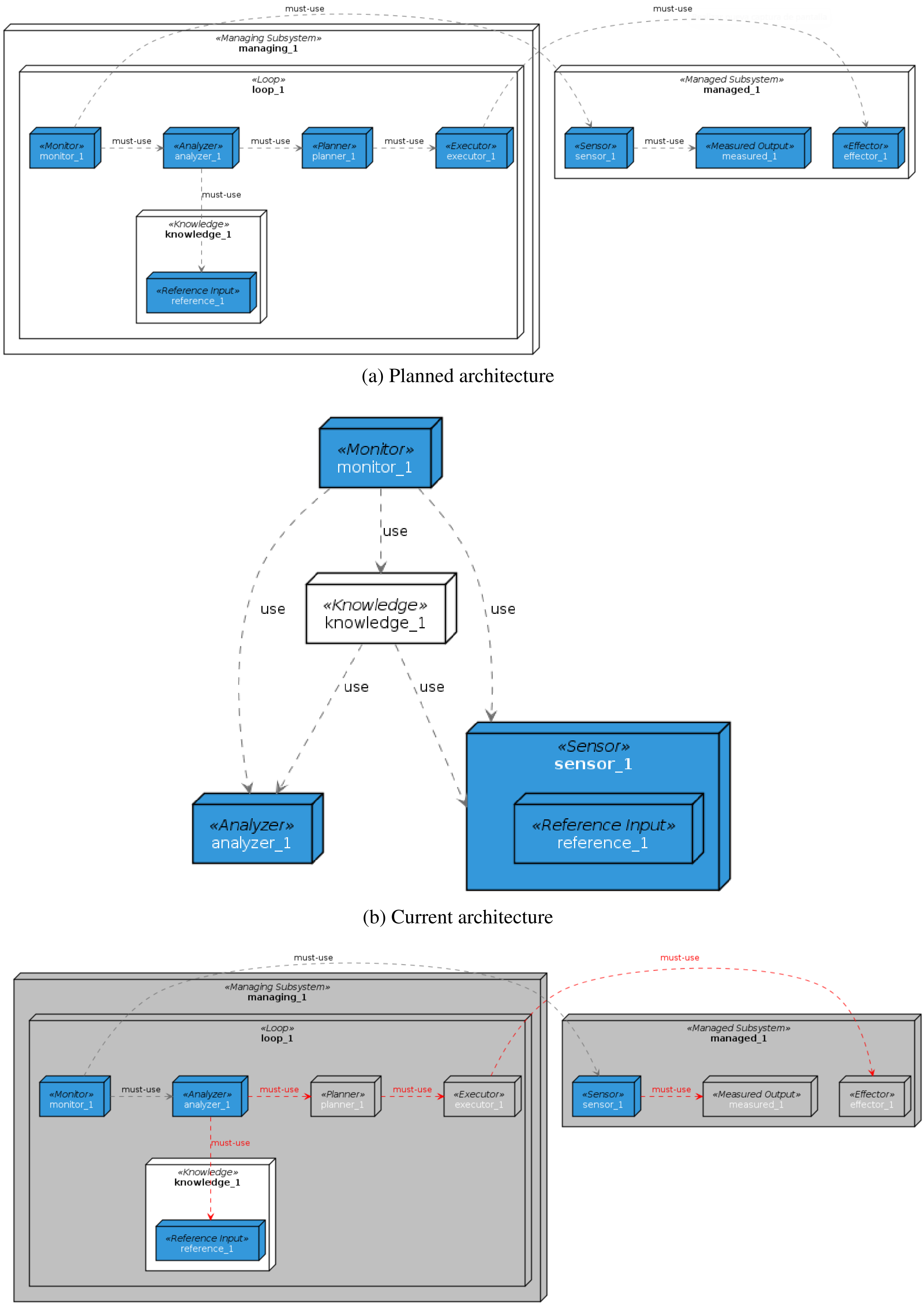
Software architects also have the possibility of viewing graphically the PA, CA and the difference between both by clicking the radio buttons of the Architecture Visualization section in the dashboard. Figure 7.6a shows the PA according to the specification shown in Figure 7.2. The access rules that were specified are the following:

```
[1] monitor monitor_1 must-use analyzer analyzer_1; (*)
[2] analyzer analyzer_1 must-use planner planner_1;
[3] planner planner_1 must-use executor executor_1;
[4] analyzer analyzer_1 must-use reference-input reference_1;
[5] monitor monitor_1 must-use sensor sensor_1;
[6] executor executor_1 must-use effector effector_1;
[7] sensor sensor_1 must-use measured-output measured_1;
```

Regarding to the twenty domain rules created by REMEDY, it just checked the following rules:

```
[8] knowledge knowledge_1 must-not-use analyzer analyzer_1;
[9] analyzer analyzer_1 must-use knowledge knowledge_1;
[10] monitor monitor_1 must-use knowledge knowledge_1;
[11] knowledge knowledge_1 must-not-use monitor monitor_1;
[12] monitor monitor_1 must-use analyzer analyzer_1; (*)
[13] analyzer analyzer_1 must-not-use monitor monitor_1;
```

As we stated before, there is one rule duplicated marked with an (*) on the rules above. In this case just the domain rule is considered while the other one is disregarded. Figure 7.6b shows the CA that was created after performing the mappings. REMEDY recovers automatically the architectural relationships by identifying the relationships among source code elements mapped to AS abstractions. In our example, REMEDY created five architectural relationships for the CA and are the following:



(a) Planned architecture

(b) Current architecture

(c) Differences between the PA and the CA

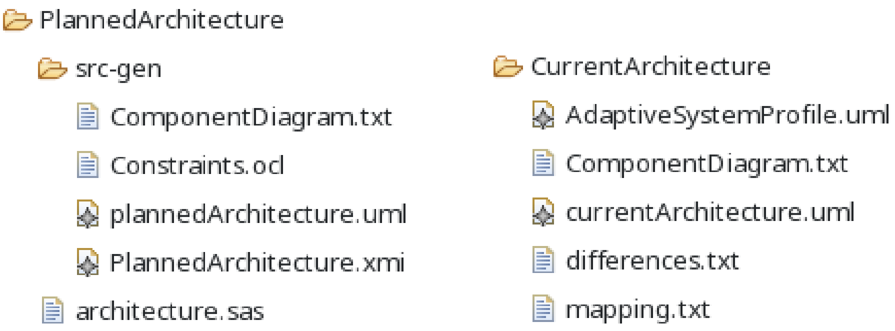
Figure 7.6 – The planned and current architectures and the differences

```
[A] monitor monitor_1 use analyzer analyzer_1;
[B] knowledge knowledge_1 use analyzer analyzer_1;
[C] knowledge knowledge_1 use sensor sensor_1;
[D] monitor monitor_1 use knowledge knowledge_1;
[E] monitor monitor_1 use sensor sensor_1;
```

Figure 7.6c shows the difference between the PA and the CA which is computed by taking as a reference the PA so that is the reason Figure 7.6c shows the PA but with different colors. The comparison between both models is performed by a third-party tool after applied model transformations as we explained in Section 6.2.4. In this case, rule [12] (PA) has its equivalent to rule [A] (CA), rule [5] (PA) has its equivalent to rule [E], rule [10](PA) has its equivalent to rule [D] (CA) and rules [11](PA), [13](PA) also are in conformance with the (CA), so all of them passed the conformance checking. Rules [4], [8] and [9] have not an equivalent in the CA so they did not pass the checking. Rules [2], [3], [6] and [7] were ignored because the architectural elements of the CA do not exist in PA so they appear in the Ignored Architectural Rules grid.

7.1.1 Artifacts Generated by REMEDY

In this subsection we detail the artifacts generated by REMEDY to support the ACC process. Thus, Figure 7.7 shows the folders and files produced by REMEDY that are generated on each step of the process.



```
PlannedArchitecture
    src-gen
        ComponentDiagram.txt
        Constraints.ocl
        plannedArchitecture.uml
        PlannedArchitecture.xmi
    architecture.sas
```

(a) Artifacts of planned architecture

```
CurrentArchitecture
    AdaptiveSystemProfile.uml
    ComponentDiagram.txt
    currentArchitecture.uml
    differences.txt
    mapping.txt
```

(b) Artifacts of current architecture

Figure 7.7 – Artifacts of REMEDY

Figure 7.7a depicts several files that support the specification of a PA. As we state before, REMEDY provides a template to guide software architects in the specification of a PA named as *architecture.sas*. Once the specification is complete, two files are generated from the specification; *Constraint.ocl* and *PlannedArchitecture.xmi*. The first one contains all rules to be checked in the CA and the second one is a Knowledge Discovery Metamodel (KDM) instance that represents the PA. Files named as *plannedArchitecture.uml* and *ComponentDiagram.txt* are created after the execution of the visualization of the PA. The first one is the result of model transformation from KDM to UML and the second one is the result of transformation from the UML model to a plantUML file.

Figure 7.7b depicts the folder and files that support the visualization of the CA. After performing the mappings, a KDM instance with the architectural representation of the AS is generated in the root of the eclipse project. When the software architect needs to visualize the CA and the drifts, the KDM CA is transformed into a UML file named as *currentArchitecture.uml*. The transformation uses two files to this purpose *AdaptiveSystemProfile.uml* and *mapping.txt* that adds profile information to the *currentArchitecture.uml* file. *ComponentDiagram.txt* is the result of the transformation from the UML file to a plantUML file. Also, *differences.txt* is a plantUML file that shows graphically the drifts found by REMEDY.

7.2 Architecture of REMEDY

Figure 7.8 gives an overview of the REMEDY workbench. It is composed of seven plugins with dependencies implemented in OSGi to be deployed in the eclipse framework.

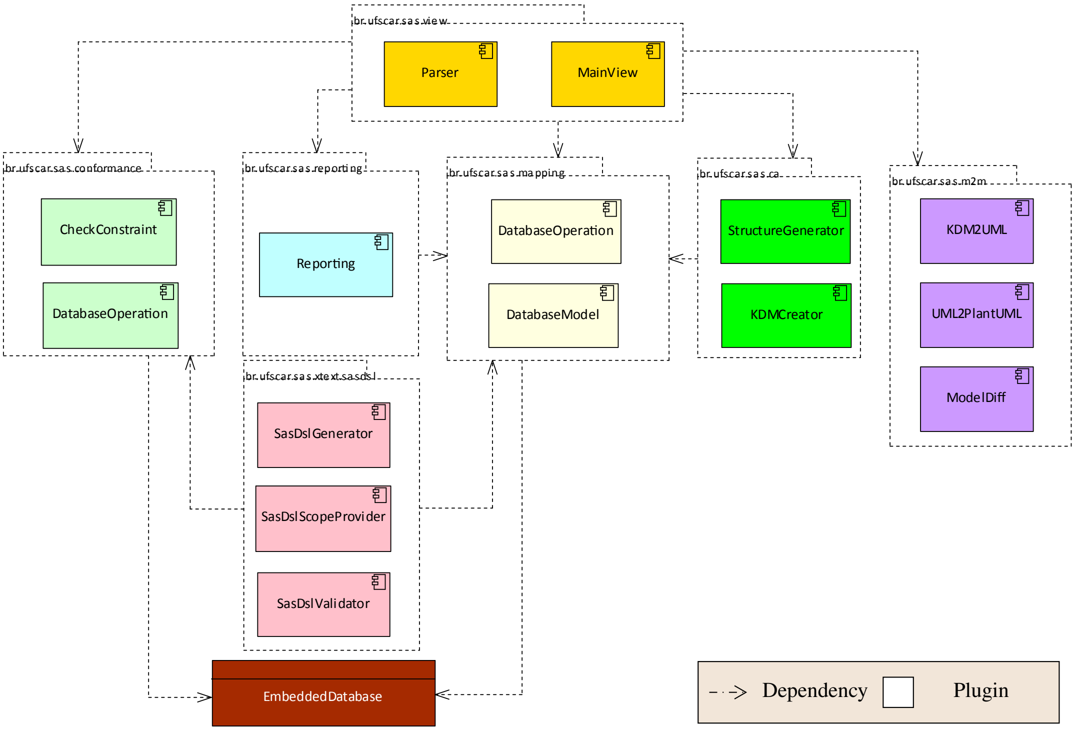


Figure 7.8 – Overview of the REMEDY workbench

First of all, when software architects need to specify the PA, the plugin *br.ufscar.sas.xtext.sasdsl* is activated to deliver a set of functionalities for that purpose such as a text editor, validators and generators. Component *SasDslValidator* validates the specification as we described in previous chapter. Component *SasDslScopeProvider* enables local and global scoping of our language such as cross-reference resolution. Component *SasDslGenerator* generates two artifacts, the OCL file which contains the architectural rules to be applied in the CA and a KDM instance to represent the architectural model. The architectural elements specified by the software architect are stored in an embedded database through the *br.ufscar.sas.conformance* plugin.

Once the specification of the PA is completed, the architect needs to generate the CA with the mappings that relate architectural elements with source-code elements. The mappings are performed by means of a UI implemented in SWT (Standard Widget Toolkit) and activated by the *MainView* component. To process the mappings, firstly the *KDMCreator* component creates the KDM CA from source code by using MoDisco plugin. The *StructureGenerator* component implements the methods for creating the Structure Package of KDM, based on the mappings. It uses a combination of a Xquery engine and qvt-o transformations to specify the architectural elements and their relationships.

The visualization of the PA and CA is performed by the *KDM2UML*, *UML2PlantUML* and *ModelDiff* components. The first one transforms a Structure Package into a UML Package. The second one transforms the UML Package into a PlantUML file and the third one computes the difference between both models (CA and PA) to show graphically the architectural drifts. The ACC process is performed by the *CheckConstraint* component and *Reporting* component is in charge of creating a word document with the information of the process.

DatabaseOperation and *DatabaseModel* are components to handle the connection and sql queries with the embedded database. In the same way, *DatabaseOperation* from *br.ufscar.sas.-conformance* plugin handles the connection and sql queries related to *CheckConstraint* component, that involves domain rules and DSL validations.

7.3 Chapter Summary

In this chapter, we described a prototype workbench that has been created as part of our research in order to help software architects with the architecture conformance checking process for adaptive systems. That is possible due to the workbench implements concepts of adaptive system domain based on the MAPE-K reference model.

The workbench, which as the the process described in Chapter 6 is also called *REMEDY*, is composed of four different views: the specification of the PA, the mapping of code elements with AS abstractions, domain rules and a dashboard. The specification of the PA is based on *Xtext*, a framework for development of programming languages and domain-specific languages. This PA is written in a file called architecture.sas which generates the KDM PA and OCL constraints, as we describe in Section 6.2.1.5.

The mapping of code elements with AS abstractions is a grid with comboboxes preloaded with AS abstractions already specified in the PA. It generates the KDM CA with an instance of the Structure Package of the KDM metamodel. To do so, it uses two technologies, XQuery and qvt-o both of them to handle xmi resources to perform operations over it such as queries and model transformation. Domain rules is a grid that shows 20 predefined rules where software architects can activate/deactivate the rules according to their needs. This grid works together with our DSL by using a special keyword (*withDomainRules*) in the specification of the PA to enable the generation of the domain rules automatically without the need of specify them in the rules section of the DSL.

Finally, the dashboard enables the ACC and shows the results of it in two ways; textually by showing quantitatively the statistics and the rules passed or violated according to the color, graphically by means of pie charts.

Chapter 8
EVALUATION

In this chapter we present a controlled experiment that aims at evaluating empirically whether software engineers improve their productivity when they specify planned architectures of the adaptive part of an Adaptive System (AS) by means of a domain-specific approach (DSL-REMEDY) against to a generic approach (DCL-KDM).

8.1 Controlled Experiment

In this section we present a controlled experiment for the empirical evaluation of DSL-REMEDY, the DSL that aids software architects to specify Planned Architectures (PAs) of ASs, against a generic ACC approach. The goal is to provide experimental evidence that the use of a domain-specific approach improves software architects productivity by reducing the efforts at time to specify the adaptive architecture. We follow the experiment process suggested by Wohlin et al. (2012) that includes the following steps: scoping, setting, planning, analysis/discussion and threats of validity.

8.1.1 Scoping

One of the claimed benefits of using domain-specific ACC approaches is the improvement of productivity because some structural and communication rules from the domain are already known and do not need to be specified by the architects (VELASCO et al., 2018). Nevertheless, at best of our knowledge, no controlled experiments have been conducted that provide evidence of improved productivity when software architects use domain-specific ACC approaches, particularly in the research area of ASs.

Hence, the contribution of this section is to present a controlled experiment that compares two different approaches/tools for ACC. The first is DCL-KDM (S. LANDI et al., 2017) a generic ACC approach and the second is DSL-REMEDY, a domain-specific ACC approach for ASs. The goal of our experiment is put in the following statement:

Analyze the architecture specification of ASs with ACC tools † **for the purpose** of evaluating two different tools **with respect to their** productivity **from the point of view of** researchers **in the context of** final-year undergraduate students in computer engineering.
† We are referring to the adaptive part of an AS.

In this context, *productivity* relates to cost in time, quantity of errors and work effort required to specify the architecture of the adaptive part of an AS. The experiment was carried out in the context of a PhD study, involving students of the Computer Engineering Bachelor program in a prestigious university in Chile.

8.1.2 Setting

The experiment was performed in one week at the end of the second semester of 2020. The subjects of the experiment were 24 final-year undergraduate students that had been taken software engineering, domain-specific language and software architecture courses. The experiment consisted of three activities with four hours of work each day:

1. **A training session** was given covering the theoretical topics of AS, software architecture and DSLs and a more practical topic teaching REMEDY and DCL-KDM. Also, they filled a form that was used to profile the students for dividing them in two groups of 12 students each one. Some of the questions asked to them were about programming skills, industry experience and if they attended some courses of the bachelor's program related with the experiment;
2. **A pilot experiment** was conducted to familiarize with the artifacts used in the real experiment. Students signed a consent letter and the two groups perform two architectural specification of ASs by using REMEDY and DCL-KDM. With the provided information by the pilot, we analyzed if the given time to complete the activity was optimal, if students understood experiment instructions and if the tools were used correctly.
3. **The experiment** was then conducted with the same format as the pilot but different AS architecture specifications.

REMEDY can be downloaded from this repository `https://tinyurl.com/y34jyeut` while DCL-KDM from this one `10.5281/zenodo.5136838`.

8.1.3 Planning

Subsequently, we discuss experiment design, hypotheses, and independent and dependent variables.

8.1.3.1 Experimental Design

Table 8.1 presents the experiment design. Students were separated in two balanced groups according to their profile and each group performed two different assignments for specifying

ASs. Both assignments are hypothetical, specified in UML notation, but they had the same level of difficulty and were designed by taking into account well known patterns of AS (WEYNS; IFTIKHAR; SÖDERLUND, 2013). Thus it involved the creation of AS abstractions and their communication rules.

Group	First Task	Second Task
G1	S-I (DCL-KDM)	S-II (REMEDY)
G2	S-II (REMEDY)	S-I (DCL-KDM)

Table 8.1 – Experiment design

In the first task, Group 1 specified S-I by using DCL-KDM and Group 2 specified S-II by using REMEDY. In the second task, Group 1 specified S-II by using REMEDY and Group 1 specified S-I by using DCL-KDM. The artifacts used in the controlled experiment such as forms and architectural specifications are shown in Appendix B.4.

8.1.3.2 Hypotheses Formulation

The research goal of this experiment is to compare the use of DCL-KDM and REMEDY regarding productivity in terms of time to complete an AS specification and the number of errors made by subjects. Also, we want to know if there are differences of perception of effort when subjects perform architectural specifications with both tools. Therefore, the research goal can be refined in 3 sub-goals that map to a set of hypotheses. In particular, each sub-goal maps to a null hypothesis to be tested, and an alternative hypothesis in favor and to be accepted if the null hypothesis is rejected. We formulate 3 null hypotheses (H_0) and three alternative hypotheses (H_α):

- H_{01}: There is no difference in time to complete an architectural specification of AS by using DCL-KDM or REMEDY.

$$H_{01} : \mu_{time_{DCL-KDM}} = \mu_{time_{REMEDY}} \tag{8.1}$$

$$H_{\alpha 1} : \mu_{time_{DCL-KDM}} > \mu_{time_{REMEDY}} \tag{8.2}$$

- H_{02}: There is no difference on errors when specifying the architecture of an AS with DCL-KDM or REMEDY.

$$H_{02} : \mu_{errors_{DCL-KDM}} = \mu_{errors_{REMEDY}} \tag{8.3}$$

$$H_{\alpha 2} : \mu_{errors_{DCL-KDM}} > \mu_{errors_{REMEDY}} \tag{8.4}$$

- H_{03}: There is no difference on effort when specifying the architecture of an AS with DCL-KDM or REMEDY.

$$H_{03} : \mu_{effort_{DCL-KDM}} = \mu_{effort_{REMEDY}} \tag{8.5}$$

$$H_{\alpha 3} : \mu_{effort_{DCL-KDM}} > \mu_{effort_{REMEDY}} \tag{8.6}$$

8.1.3.3 Independent and Dependent Variables

Each hypothesis requires the definition of a set of independent and dependent variables, and a selection of proper metrics to measure the dependent variables.

• *Independent Variables:* Independent variables are variables in the experiment that can be manipulated and controlled. In our experiment, there are two independent variables:

- Techniques: The treatment used by a subject to solve an assignment. This variable is the factor of the experiment that is changed to observe the effect on the dependent variables. The two possible values of this factor are DCL-KDM and REMEDY;

- Specifications: The problem to be solved by the subject (specifications S-I and S-II). Since the specifications have similar numbers of abstractions, the specification is not considered as a factor but as a fixed variable.

• *Dependent Variables:* Dependent variables are variables that we want to study to see the effect of different treatments. For each hypothesis, we defined the corresponding dependent variables:

- Time: The time in minutes to complete an architectural specification of AS;

- Errors: Number of errors found after finish the architectural specification;

- Effort: Likert-type scale from 1 to 4 that denotes the perception of effort of subjects, where 1 means easy to use and 4 very difficult.

8.1.4 Analysis & Discussion

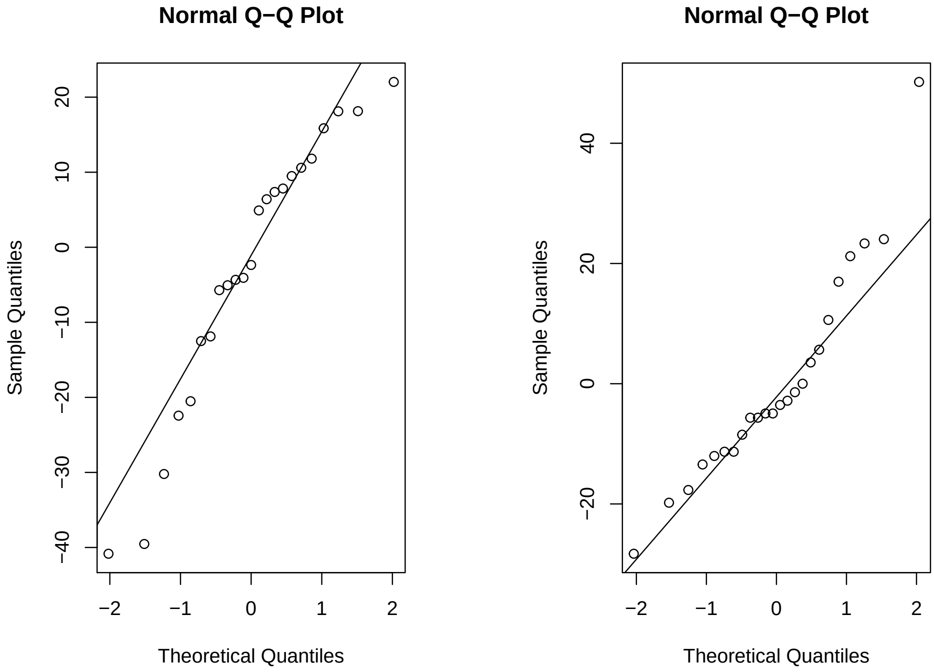
8.1.4.1 Analysis

In total, 24 subjects provided usable data for paired comparison of time in minutes, number of errors and effort. Table 8.2 depicts the mean and standard deviation for the factor Tool on Time on Error and on Effort. The values show that there is a difference between the means but to know if they are sufficiently different we applied an analysis of variance test.

	Time		Error		Effort	
Tool	**mean**	**sd**	**mean**	**sd**	**mean**	**sd**
DCL-KDM	64.75	18.43024	4.791	3.106	2.83	0.637
REMEDY	45.75	16.67920	1.291	1.122	1.70	0.624

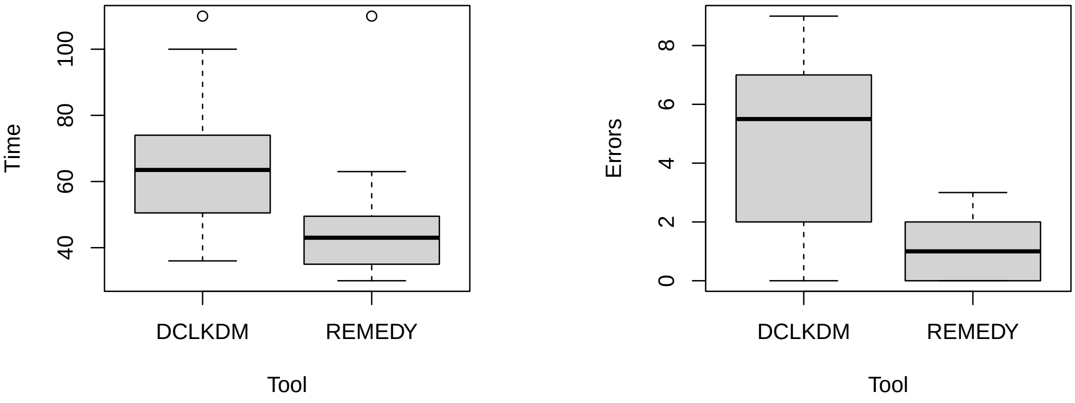
Table 8.2 – Mean and Sd of Time, Error and Effort

As we have 1 factor with 2 levels within-subjects (repeated measures), we applied the paired sample t-test. Also, in order to mitigate carryover effects which introduces biases in the results we used a full counterbalancing strategy, as is depicted in Table 8.1. After applying a t-test on Orders, we got a p-value of 0.9174 that suggests our results do not have an order effect where order itself could cause differences of performance. The Shapiro-Wilk normality test of residuals for subjects and subjects tools were 0.1119 and 0.06314 respectively indicating normality. Indeed, Figure 8.1 shows graphically by means of Q-Q plots that the data is normally distributed. Figure 8.2 - (a) shows the boxplot of time per tools and it seems there is differences on average when subjects performed with different tools. The t-test paired samples Time on Tools gave us a p-value of 0.00088 that means there is a significant difference between both approaches.



(a) Residuals on subjects (b) Residuals on subjects/tool

Figure 8.1 – Q-Q plots



(a) Time per tools (b) Errors per tools

Figure 8.2 – Boxplots for all measurements

Errors are a count response and often do not satisfy the assumption of normality for anovas.

In our case the error data for DCL-KDM fits on a Poisson distribution with a p-value of $6.890398e-06$ by using the goodness–of–fit tests. On the other hand, errors of REMEDY did not fit so we chose to apply a non-parametric analysis, the wilcoxon signed-rank test. Figure 8.2 - (b) shows the boxplot of errors per tools and again it seems there is differences on average when subjects performed with different tools. Thus the p-value after applying wilcoxon signed-rank test was $5.722e^{-6}$ that means there is significant differences on Errors by using DCL-KDM or REMEDY.

Effort is an ordinal likert-scale response from 1 to 4 and this psychometric scale rarely satisfies the conditions for anova so we used again the non-parametric analysis wilcoxon signed-rank test. The p-value was $9.537e-7$ which indicates that there is differences on Effort when subjects use DCL-KDM or REMEDY. Table 8.2 shows the mean and standard deviation for the factor Tool on Effort. On average, the perception of subjects is that DCL-KDM is more difficult to use than REMEDY for specifying the architecture of ASs.

Therefore, based on the statistical analysis every null hypothesis (H_0) is rejected with a significance level (α) of 0.05.

8.1.4.2 Discussion

The descriptive analysis shows that there is a clear improvement for the dependent variables when subjects use REMEDY compared to DCL-KDM. This is confirmed by the statistical tests. On average, time is about 30% lower with REMEDY compared to DCL-KDM. Closer examination of the specifications performed with REMEDY reveals that subjects made use of predefined domain-specific rules support when needed so this could explain the saving time. Also, subjects that performed the specifications with DCL-KDM wrote some abstractions with different types. The solutions with both tools can be obtained in the following url `10.5281/zenodo.5136838`.

For instance, DCL-KDM allows three types of abstractions; subsystem, layer and component, where subsystem and layer can be composed of subsystems, layers and components. Thus the choice decision among component, layer and subsystem for an AS abstraction could affect the productivity at time to do the specification.

With regard to errors, Figure 8.3 shows a bar chart with the number of errors made by subjects. When subjects used DCL-KDM the total number of errors was 115, where 46 correspond to structural rules (SR) and 69 to communication rules (CR). On the other hand, when subjects used REMEDY the total number of errors was 31 and all of them correspond to communication rules.

The main characteristic of DCL-KDM is the capability of specifying compound abstractions deeply. For instance, a subsystem can hold layers that in turn can hold other subsystems and layers that in turn can hold components. On the other hand, the composition of abstractions in REMEDY is well-known due to AS domain restrictions. Therefore in copy paste operations or when the specification contains several lines of code that makes difficult to read and understand it, subjects are more likely to make mistakes when specifying structural rules. Moreover, we

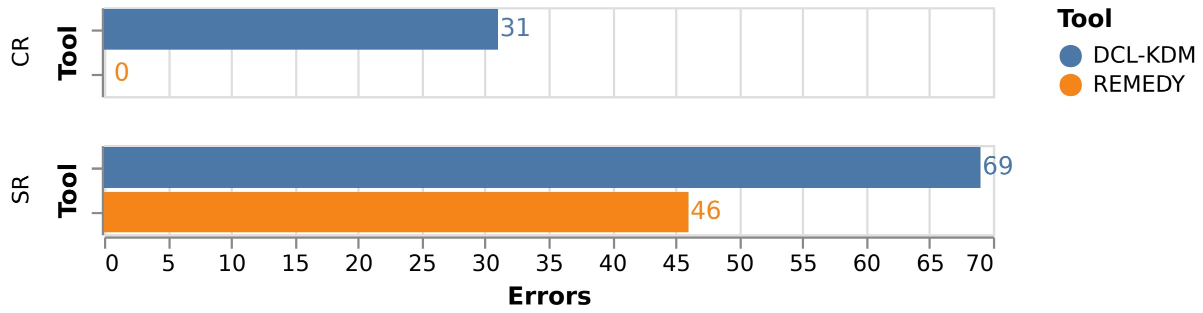


Figure 8.3 – Errors found with both tools

found different types of error of communication rules in DCL-KDM such as duplicate rules and circular rules due to lack of validation.

Regarding errors made with REMEDY all of them are related to communication rules of low-level abstractions. A possible answer of why this happens is because subjects made copy and paste of the communication rules where were involved reference input abstractions of different control loops. Indeed, this was a bug because when the rule that allows access between two control loops was absent, REMEDY did not validate accesses of abstractions of one control loop to another with low-level abstractions. The bug was corrected in the next version of REMEDY.

Finally, concerning effort the overall score was average for REMEDY and difficult for DCL-KDM. We believe the results are strongly influenced by the type of DSL. REMEDY uses an appropriate language for the AS domain and the way it constructs the structural rules is identical to the MAPE-K reference model which contrast with DCL-KDM. Although, the results could be predictable we were able to demonstrate it scientifically.

8.1.5 Threats of Validity

8.1.5.1 Threats to Internal Validity

Internal validity is the extent to which independent variables are responsible for the effects seen to the dependent variables. Due to global pandemic of SARS-CoV-2, the controlled experiment was carried out online and as a consequence the instructor could not verify in situ if there was any kind of interaction among subjects about how to specify the architectures.

To reduce this threat, before experiment activities begin we explained to them that it was not a competition so there was not any kind of rewards for finishing in less time or having an optimal architecture. After having analyzed their specifications (pilot and experiment) we determine that exists heterogeneity in architecture specifications, so if there was any type of interaction among subjects it did not affect the experiment.

8.1.5.2 Threats to Construct Validity

Construct validity is the degree to which the operationalization of measures in the study represent the constructs in the real world. We have seen one type of such threats; Inadequate preoperational explication of constructs. Although researchers delivered all concepts involved in the experiment to subjects, maybe the depth of contents have not been optimal for the understanding of some subjects, this was due to some restrictions of the number of online sessions and the time duration of each online session.

To reduce this threat, a subject profile was performed and contrasted with the academic history of each participant. Moreover, a researcher of this study was in charge of at least one course that subjects needed to attend to participate in this experiment so in a certain way we already known subjects competencies.

8.1.5.3 Threats to External Validity and Conclusion Validity

External validity is the degree to which findings of a study can be generalized to other subject populations and settings. Conclusion validity concerns generalizing the result of the experiment to the concept or theory behind the experiment. Due to practical restrictions, we deal with students of an undergraduate program in Computer Engineering as subjects for our study. Although students do not represent expert software engineers, they are the next generation of software professionals (KITCHENHAM, B. A. et al., 2002). Also, it is possible the specifications does not exists in real world applications. To mitigate this threat, the architectural specifications were designed considering patterns based on influential papers of the area.

Finally, there is a threat concerning the reliability of time measure. We have asked the subjects to set the starting and ending time. In this sense we could have had a problem because a subject could forget to mark the time. To mitigate this, we use an online program to chronometer the time. Each subject shared the chronometer so the instructor was aware of the time spent.

8.2 Chapter Summary

In this chapter, we conducted a controlled experiment to evaluate empirically the productivity of software architects by using DSL-REMEDY to perform the specification of PAs. It shows that the use of a specific DSL such as REMEDY improves productivity at time of specifying the adaptive part of an AS, by reducing the effort, time to complete the specification and reducing the number of errors.

Chapter 9

CONCLUSION

This chapter concludes the thesis with retrospective view on the goals and how they have been achieved, and further provides a general outlook of the work done on each step of our approach. The chapter is organized as follow. Section 9.1 summarizes the main contributions and validation results, and Section 9.2 outlines benefits gained and scientific findings made. Section 9.3 summarizes assumptions and limitations of our contributions. Section 9.4 suggests future work.

9.1 Contributions

Architecture erosion is usually the result of modifications to a system that disregard its fundamental architecture rules and as a consequence occurs a deviation from the intended software architecture. Adaptive Systems (ASs) are not exempted from suffer this problem, and as we shown in Chapter 3 by means of a systematic mapping, researchers are concerned about architectural anomalies in this kind of systems but more studies are needed in order to characterize and report them.

Moreover, besides the canonical abstractions prescribed by MAPE-K, we shown in Chapter 5 that there are others equally important for reaching good levels of maintainability (we called them as low level abstractions). 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 (RAMIREZ; CHENG, 2010). 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 for that type of system (PERRY; WOLF, 1992). In order to control the architecture deviation a process of architecture restoration must be applied which commonly involves two strategies; architecture recovery and architecture reconciliation.

Up to the time of writing this thesis we have not found researches about Architecture Conformance Checking (ACC) of ASs that takes into account the Knowledge Discovery Metamodel (KDM) metamodel, so this work will contribute with new knowledge in three research fields:

AS, ACC and Architecture-Driven Modernization (ADM). Thus by delivering an approach and tool this work will support software architects in detecting architectural drifts that could correct them through model refactorings and conform the system to the intended architecture.

In this thesis, we presented an approach to identify architectural drifts in ASs and to support software architects in the process of ACC, leading the following main contributions:

- **An approach to perform ACC in KDM-represented ASs** - We designed a process of ACC for ASs that consists in four steps; *i*) Specify Planned Architecture; *ii*) Map Architectural Elements; *iii*) Check Architecture Conformance and *iv*) Visualize Architectures. In the first one software architects specify the planned architecture through a DSL called DLS-REMEDY. The specification involves some sub-steps; the specification of managing elements and managed elements with AS abstractions. We demonstrate in Chapter 8 that by using a domain vocabulary of AS the productivity improves so the use of DSL-REMEDY over generic approaches is well justified. Another important activity that software architects should perform is the specification of communication rules by using the AS elements defined in previous sub-steps. These rules enable or forbid the access of abstractions to others and their specification is aided by several custom validators. Our DSL not only takes into account the main abstractions prescribed by the MAPE-K reference model but also low level abstractions (Chapter 5). The output is twofold, a file with OCL queries to be applied on any KDM-represented AS for checking conformance and a KDM representation of the Planned Architecture (PA) to visualize the architecture graphically;

  In the second one, software architects recover the Current Architecture (CA) after perform mappings to source-code elements. The mappings enable the creation of the architecture model of the KDM-represented AS where the ACC is realized by means of the OCL file from the PA. In the third one, the ACC is executed to show the drifts found by the process and finally in the fourth one software architects can visualize the CA, PA and drifts graphically. As we reported in Chapter 3, we did not find approaches of ACC for ASs so this contribution adds new knowledge in the state of art;

- **Characterization of three drifts of ASs** - In Chapter 5 we reported three architectural drifts of ASs named as *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 Alternatives occurs when the alternatives of an AS are not evident. Mixed Executors and Effectors arises when there is not a clear distinction between executors and effectors. We analyzed seven representative ASs and observed that these drifts occur in all of them which leads to conclude 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. Moreover, another contribution is that we presented a well defined methodology for discovering architectural drifts that can be used in other domains and a template to characterize them;

- **Systematic Mapping** - In Chapter 3 we developed a systematic mapping of architectural anomalies in ASs. To the best of our knowledge this is the first study that concern about this topic. We find that most of the anomalies were reported in the domain of automotive systems but as we shown it can occur in other domains where ASs actuate. This can motivate other researchers in performing works on the area.

9.2 Lesson Learned

Assessed from a broader viewpoint, there are two scientific findings made by this thesis, which we are going to point out below.

- **Availability of Adaptive Systems** - At the beginning of this research, one of the main challenges of this project was related to the availability of ASs. Although the Software Engineering for Adaptive and Self-Managing Systems (SEAMS) conference maintains an open repository with software artifacts, most of them are academic investigations so it may not reflect the reality of what occur on industry and practitioners that work with this kind of systems while the others are partially published because copyright restrictions. We also did a research in open repositories such as gitHub and gitLab, but most of them were little systems with poor documentation so it was impracticable the installation and operation;

- **Software Architects Expertise** - Another challenge we needed to overcome was the fact that architectural elements of an AS can be implemented in several ways. For instance, some implementations use specific frameworks for that purpose while others are implemented in ad-hoc manner (from the scratch) which difficult the automated recognition of such elements. We spent several months trying to find a solution, from the technical point of view, that could be integrated to our approach without success. At the end, we realized that more studies are needed to cope this issue and probably a new thesis could deep on it, by developing machine learning algorithms to identify the AS architectural abstractions. Therefore, we leave the recovery of the CA on hands of software architects that have the expertise to recognize the architectural abstractions in source code by means of mappings.

9.3 Limitations of the approach

While our approach provides modeling AS abstractions, a systematic process and a workbench which can aid software architects in specifying and recovering adaptive systems and performing ACC, this assistance is also limited in several aspects. The following list provides a summary of limitations that we have identified for the approach:

- **Specification of Generic Architecture Abstractions** - Our approach just takes into account the specification and mappings of the adaptive part of ASs so at the moment of writing this thesis, software architects are not able to specify the whole system architecture in terms of generic components, modules and layers. This is a limitation because domain abstractions could interact with generic abstractions with the consequence that those relationships cannot be specified and therefore unable to check their conformance;
- **Centralization** - An analogous issue comes from the fact that REMEDY centralizes the specification of control loops, whereas some systems like socio-technical systems, are composed of independent self-organized agents which collaborate towards a resulting adaptive system. Self-organization is the focus of conferences such as the International Conference on Autonomic Computing and Self-Organizing Systems. Our based approach could be used to specify independent software-based agents, but DSL-REMEDY do not provide any mechanism to specify non-centralized feedback loops. Further investigation is required to analyze the relation of our work with proposals on self-organized adaptive systems. Initial ideas can be obtained from the work published by Alvares de Oliveira, Sharrock, and Ledoux (2012) and Weyns, Schmerl, et al. (2013);
- **MoDisco Dependency** - MoDisco tool is not capable of generating the complete KDM instance of an AS when specific meta-model elements need to be invoked. For instance, we found that the meta-class HasValue (type of relationship) which represents semantic relation between a data element (objects or primitive variables) and its initialization element (data or one or more programming language statements), is not modeled completely and a as a consequence, the ACC process will not identify drifts where the meta-class be involved;
- **Fine-grained Rules Specification** - Although our approach uses a coarse-grained rules specification that is functional to our purposes with *must-use* and *must-not-use* keywords, in some cases software architects could need more control on checking the type of relationship between the abstractions, thus non-trivial modifications must be done in the DSL to support this feature and also the code generator should be capable of generating the type of relationships supported by KDM;
- **Experiments** - This thesis reports on experiments with exemplars based on scenarios of AS specification to evaluate if software architects (undergraduate students) perform better

when using dedicated DSL of AS rather than generic ones. However, many other kinds of experiments are needed in order to provide a more complete validation of the approach;

Surveys with practitioners can evaluate the proposed systematic approach and modeling language, whereas the use of real applications instead of simulations would make for a stronger case for the DSL's effectiveness. Moreover, full-fledged case studies with industrial partners would be advised before taking the results of this research to industrial settings.

9.4 Future Work

The previous section highlighted several limitations of our proposal, all of which could be considered an opportunity for future work. Nevertheless the next steps will be concentrated in two areas; an integration with the Arch-KDM tool and add recommendation capabilities for architectural refactorings. These are explained in the following next subsections.

9.4.1 ARCH-KDM Integration

In literature, there are several approaches to specify systems in a generic way by using abstractions such as layers, components, modules and interfaces (TERRA; VALENTE, 2009; JAMSHIDI et al., 2013). Thus to our purpose, we have started the integration of ARCH-KDM developed by Landi et al. (S. LANDI et al., 2017) with our DSL. This approach also relies in KDM models for checking the architectural conformance so it fits very well to our intentions. In order to achieve the full integration three activities should be performed: *i*) Integrate ARCH-KDM grammar into REMEDY with custom validations; *ii*) implement code generation of the PA in a KDM instance and OCL constraints and; *iii*) add the generic abstractions of ARCH-KDM into REMEDY to be mapped in the CA.

Up to the writing of this thesis, the first activity was completed in 50%, that means the grammar of ARCH-KDM was integrated to REMEDY but custom validations must be written and unit testing need to be performed. Listing 9.1 shows an example where a Managed subsystem is specified with a Sensor *proximity* in the Subsystem *sensorsSubsystem* (Line 5).

In the second activity we will have to modify the generator template that is implemented with the Xtend programming language. This template is conformed by two major parts; the fist one implements the Managing subsystem and the second one the Managed subsystem.

```
1     ..
2 Managed environmentGuardRobot {
3    ..
4    subSystem sensorsSubsystem;
5    Sensor proximity , inSubSystem: sensorsSubsystem;
6    ..
7 }
8 ..
```

Listing 9.1 – Example of a generic abstraction in the PA

Listing 9.2 shows the template of the Managed subsystem implemented in Xtend. For every *sensor* (Line 4), *effector* (Line 9) and *measuredOuput* (Line 14) element specified in the DSL a *structuredElement* is created in the KDM instance that represents the PA.

```
1  ..
2  «FOR arch : architectureDefinition.managed»
3    <structureElement xsi:type="structure:Subsystem" name="«arch.name»" stereotype="/0/
     ↪ @extension.0/@stereotype.12">
4      «FOR sensor : arch.sensor»
5          <structureElement xsi:type="structure:Component" name="«sensor.name»"
     ↪    stereotype="/0/@extension.0/@stereotype.9" «outAggregatedPath.get(sensor.name)»
     ↪   «inAggregatedPath.get(sensor.name)»>
6              «aggregatedPath.get(sensor.name)»
7          </structureElement>
8      «ENDFOR»
9      «FOR effector : arch.effector»
10       <structureElement xsi:type="structure:Component" name="«effector.name»"
     ↪ stereotype="/0/@extension.0/@stereotype.10" «inAggregatedPath.get(effector.name)»>
11
12         </structureElement>
13     «ENDFOR»
14     «FOR measuredOutput : arch.measuredOutput»
15       <structureElement xsi:type="structure:Component" name="«measuredOutput.name»"
     ↪ stereotype="/0/@extension.0/@stereotype.6" «inAggregatedPath.get(measuredOutput.
     ↪ name)»>
16
17         </structureElement>
18     «ENDFOR»
19   </structureElement>
20 «ENDFOR»
21 ..
```

Listing 9.2 – Template of the Managed subsystem in Xtend

To extend it with generic architectural elements of ARCH-KDM, we need to modify the Xtend template by implementing other *FOR* cycles as we did for AS elements to create new structure elements. Something we need to validate is that the relationships among architectural elements are correct, because as they are based on XPath paths and must be created dynamically when the DSL is modified, paths could be pointing to wrong abstractions or abstractions that do not exist. Thus some programming mechanisms need to be constructed to validate this aspect, specially when generic elements include AS elements as shows Listing 9.1 in Line 5.

In the same way, the OCL Xtend template must be modified to include the new constraints that will take into account generic elements of ARCH-KDM.

9.4.2 Refactoring Recommendations

The goal of a refactoring is to improve a certain quality while preserving others (FOWLER, 1999). For instance, code refactoring is a technique for restructuring code to make it more maintainable without changing its observable behavior. Code refactorings work on machine-readable entities such as packages, classes and methods; hence, they can leverage data structures from compiler construction such as abstract syntax trees (ZIMMERMANN, 2015). Architectural refactorings deal with architecture documentation and the manifestation of the architecture in the code and run-time artifacts. Architectural refactorings pertain to components and connectors (modeled, sketched, or represented implicitly in code), design decision logs (which come as structured or unstructured text) planning artifacts such as work items in project management tools (ZIMMERMANN, 2015).

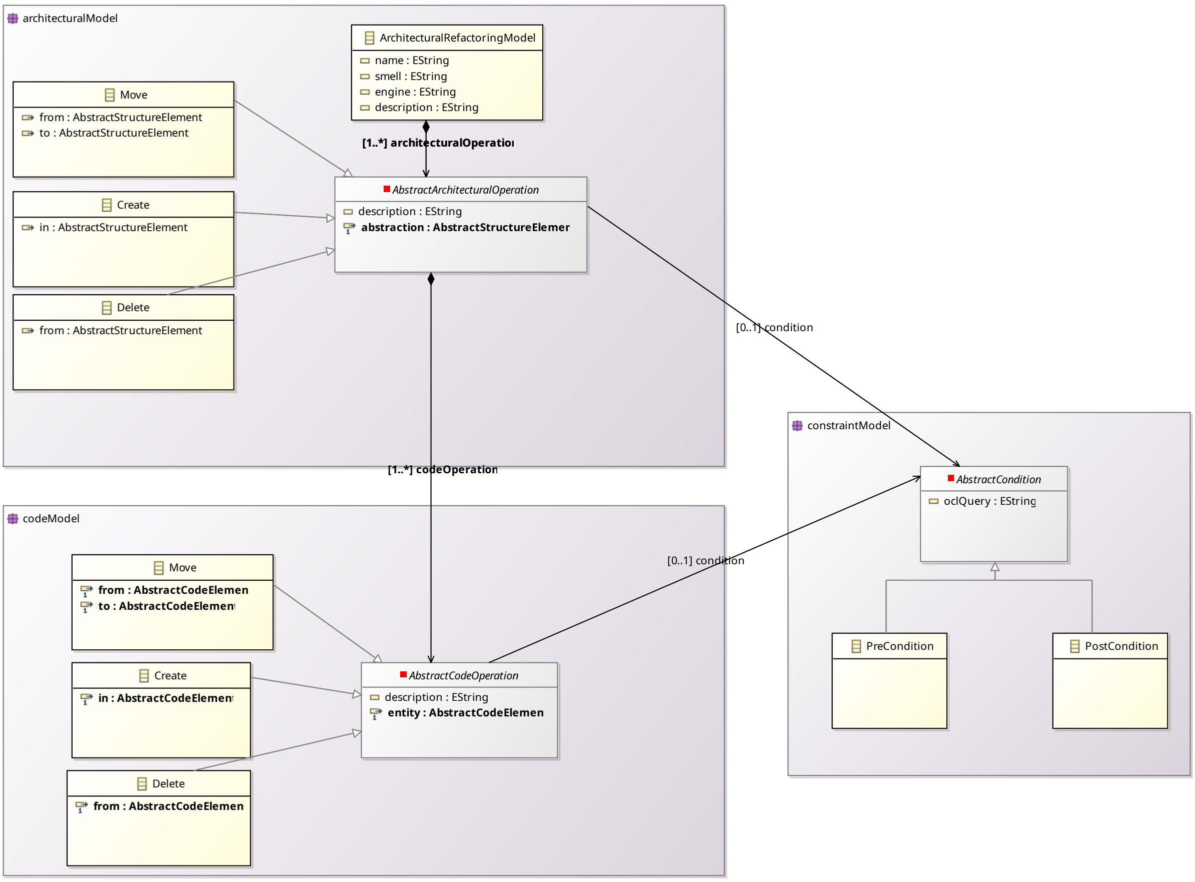


Figure 9.1 – Refactoring metamodel

According to Lin et al. (2016), architectural refactorings can contain hundreds of steps and experienced developers could carry them out over several weeks. Therefore, developers need to explore a correct sequence of refactorings steps among many more incorrect alternatives. Thus, we propose a refactoring metamodel to support architectural refactorings activities for ASs. The main idea behind this is to provide a standardization of the refactorings to automate the process. Several different instances of this meta-model would be recommended to the software architect so it will pick one of them according to which best fit on the current situation.

Figure 9.1 shows the refactoring meta-model which is composed of three packages; a constraint model, an architectural model and a code model. The first one defines three meta-classes; the *AbstractCondition*, *PreCondition* and *PostCondition* whose intention is to establish a pre and post condition through ocl queries to verify if it is possible to perform the refactorization and validate the refactorization after its execution.

The second one aim at perform the architectural refactoring. It is composed of four meta-classes; *ArchitecturalRefactoringModel* which provides a description of the refactoring such as the name, type of smell that attends, transformation engine used to perform the refactoring and a description. *AbstractArchitecturalOperation* which represent the refactoring operation through three concrete classes *Move*, *Create* and *Delete*.

The third one is similar to the second one but at source-code level. This package is necessary because the refactorings performed in high level of abstraction must be propagated to low level of abstractions. Thus, the package is composed of four meta-classes; *AbstractCodeOperation*, *Move*, *Create* and *Delete*.

REFERENCES

ABDENNADHER, Imen; BOUASSIDA RODRIGUEZ, Ismael; JMAIEL, Mohamed. A Design Guideline for Adaptation Decisions in the Autonomic Loop. **Procedia Computer Science**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 112, n. 100, p. 270–277, Sept. 2017. Cit. on p. 39.

ALDRICH, Jonathan; CHAMBERS, Craig; NOTKIN, David. ArchJava: Connecting Software Architecture to Implementation. In: PROCEEDINGS of the 24th International Conference on Software Engineering. Orlando, Florida: Association for Computing Machinery, 2002. (ICSE '02), p. 187–197. Cit. on p. 34.

ALVARES DE OLIVEIRA, Frederico; SHARROCK, Remi; LEDOUX, Thomas. Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing. In: ________. **Coordination Models and Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 29–43. Cit. on p. 136.

AMPATZOGLOU, A. et al. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD). [S.l.: s.n.], 2016. P. 9–16. Cit. on p. 56.

ARBOLEDA, Hugo et al. Development and Instrumentation of a Framework for the Generation and Management of Self-Adaptive EnterpriseApplications. en. **Ingeniería y Universidad**, scieloco, v. 20, p. 303–333, Dec. 2016. Cit. on pp. 39, 72.

BAGHERI, Hamid et al. Software architectural principles in contemporary mobile software: from conception to practice. **Journal of Systems and Software**, v. 119, p. 31–44, 2016. Cit. on pp. 54, 55, 57.

BALDWIN, Carliss Y.; CLARK, Kim B. **Design Rules: The Power of Modularity**. [S.l.]: The MIT Press, Mar. 2000. DOI: 10.7551/mitpress/2366.001.0001. Cit. on p. 62.

BARESI, Luciano; GUINEA, Sam. Architectural Styles for Adaptive Systems: A Tutorial. In: 2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems. [S.l.: s.n.], 2012. P. 237–238. Cit. on p. 23.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice**. 3rd. [S.l.]: Addison-Wesley Professional, 2012. Cit. on pp. 29, 30, 45.

BOUTEKKOUK, Fateh. Architecture Description Languages Taxonomies Review: A Special Focus on Self-Adaptive Distributed Embedded Systems. **Int. J. Technol. Diffus.**, IGI Global, USA, v. 12, n. 1, p. 53–74, Jan. 2021. Cit. on p. 68.

BROWN, William H. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1998. Cit. on p. 45.

BRUN, Yuriy et al. Engineering self-adaptive systems through feedback loops. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, 5525 LNCS, p. 48–70, 2009. Cit. on pp. 22, 35, 37, 38, 70.

BRUNELIERE, Hugo et al. MoDisco: a generic and extensible framework for model driven reverse engineering. In: PROCEEDINGS of the IEEE/ACM international conference on Automated software engineering. Antwerp, Belgium: ACM, 2010. (ASE '10), p. 173–174. Cit. on p. 26.

BRUNELIÈRE, Hugo et al. MoDisco: A model driven reverse engineering framework. **Information and Software Technology**, v. 56, n. 8, p. 20, 2014. Cit. on pp. 26, 93.

BRUNET, João et al. On the Evolutionary Nature of Architectural Violations. In: 2012 19th Working Conference on Reverse Engineering. [S.l.: s.n.], 2012. P. 257–266. Cit. on pp. 62, 67.

CARACCIOLO, Andrea; LUNGU, Mircea Filip; NIERSTRASZ, Oscar. A Unified Approach to Architecture Conformance Checking. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. [S.l.: s.n.], 2015. P. 41–50. Cit. on pp. 30, 32, 64, 65, 66.

________. How Do Software Architects Specify and Validate Quality Requirements? In: ________. **Software Architecture**. Cham: Springer International Publishing, 2014. P. 374–389. Cit. on p. 34.

CHEN, Lianipng; BABAR, Muhammad Ali; ZHANG, He. Towards an Evidence-based Understanding of Electronic Data Sources. In: PROCEEDINGS of the 14th International Conference on Evaluation and Assessment in Software Engineering. UK: BCS Learning & Development Ltd., 2010. (EASE'10), p. 135–138. Cit. on p. 48.

CHEN, Liming; AVIZIENIS, A. N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELlABlLlTY OF SOFTWARE OPERATlON. In: TWENTY-FIFTH International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. [S.l.: s.n.], 1995. P. 113–. Cit. on p. 39.

CHENG, Betty H C et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. **Software Engineering for Self-Adaptive Systems**, p. 1–26, 2009. Cit. on pp. 21, 38.

DE SILVA, Lakshitha; BALASUBRAMANIAM, Dharini. Controlling software architecture erosion: A survey. **Journal of Systems and Software**, v. 85, n. 1, p. 132–151, 2012. Cit. on pp. 30, 33.

DIMECH, Claire; BALASUBRAMANIAM, Dharini. Maintaining Architectural Conformance during Software Development: A Practical Approach. In: ________. **Software Architecture**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 208–223. Cit. on pp. 30, 33.

DOYLE, John Comstock; FRANCIS, Bruce A.; TANNENBAUM, Allen R. **Feedback Control Theory**. [S.l.]: Prentice Hall Professional Technical Reference, 1991. Cit. on pp. 36, 38.

DURELLI, R.S. et al. KDM-RE: A Model-Driven Refactoring Tool for KDM. In: WORKSHOP on Software Visualization, Maintenance, and Evolution (VEM), 2014. [S.l.: s.n.], 2014. P. 1. Cit. on p. 40.

DURELLI, Rafael S. et al. Improving the Structure of KDM Instances via Refactorings: An Experimental Study Using KDM-RE. In: PROCEEDINGS of the 31st Brazilian Symposium on Software Engineering. [S.l.: s.n.], 2017. P. 174–183. Cit. on p. 40.

EICHBERG, Michael et al. Defining and continuous checking of structural program dependencies. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. [S.l.: s.n.], 2008. P. 391–400. DOI: 10.1145/1368088.1368142. Cit. on pp. 34, 64.

ELIASSON, U. et al. 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). [S.l.: s.n.], 2015. P. 33–40. Cit. on p. 56.

ERNST, Neil A. et al. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In: PROCEEDINGS of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 50–60. Cit. on p. 56.

FARAHANI, A.; NAZEMI, E.; CABRI, G. A Self-healing architecture based on RAINBOW for industrial usage. **Scalable Computing**, v. 17, n. 4, p. 351–368, 2016. Cit. on p. 72.

FEILER, Peter H.; GLUCH, David P. **Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language**. 1st. [S.l.]: Addison-Wesley Professional, 2012. Cit. on p. 30.

FONTANA, F. A. et al. Automatic Detection of Instability Architectural Smells. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). [S.l.: s.n.], Oct. 2016. P. 433–437. Cit. on p. 58.

FOWLER, Martin. **Refactoring - Improving the Design of Existing Code**. [S.l.]: Addison-Wesley, 1999. (Addison Wesley object technology series). Cit. on p. 138.

GARCIA, Joshua et al. Identifying Architectural Bad Smells. In: LECTURE Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). [S.l.: s.n.], 2009. 5581 LNCS. P. 146–162. Cit. on pp. 45, 53, 54, 58.

GARLAN, David. **An introduction to the Aesop system**. [S.l.], July 1995. Cit. on p. 30.

GARLAN, David; CHENG, Shang-Wen, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 10, p. 46–54, Oct. 2004. Cit. on p. 89.

GARLAN, David; MONROE, Robert T.; WILE, David. Acme: Architectural Description of Component-Based Systems. In: FOUNDATIONS of Component-Based Systems. USA: Cambridge University Press, 2000. P. 47–67. Cit. on p. 30.

________. Foundations of Component-based Systems. In: LEAVENS, Gary T.; SITARAMAN, Murali (Eds.). New York, NY, USA: Cambridge University Press, 2000. Acme: Architectural Description of Component-based Systems, p. 47–67. Available from: <http://dl.acm.org/citation.cfm?id=336431.336437>. Cit. on p. 83.

GERASIMOU, S. et al. UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles. In: 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). [S.l.: s.n.], 2017. P. 83–89. Cit. on p. 104.

GIESECKE, Simon; HASSELBRING, Wilhelm; RIEBISCH, Matthias. Classifying architectural constraints as a basis for software quality assessment. **Advanced Engineering Informatics**, v. 21, n. 2, p. 169–179, 2007. Cit. on p. 30.

GURGEL, Alessandro et al. Blending and Reusing Rules for Architectural Degradation Prevention. In: PROCEEDINGS of the 13th International Conference on Modularity. Lugano, Switzerland: Association for Computing Machinery, 2014. (MODULARITY ’14), p. 61–72. Cit. on pp. 34, 62, 63, 67.

HEBIG, Regina; GIESE, Holger; BECKER, Basil. Making control loops explicit when architecting self-adaptive systems. **Proceeding of the second international workshop on Self-organizing architectures - SOAR ’10**, p. 21, 2010. Cit. on p. 70.

HELLERSTEIN, Joseph L. et al. **Feedback Control of Computing Systems**. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004. Cit. on pp. 22, 36, 38, 89.

HEROLD, Sebastian et al. Detection of violation causes in reflexion models. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). [S.l.: s.n.], 2015. P. 565–569. Cit. on pp. 62, 67.

HOLT, R. C. et al. Architectural Repair of Open Source Software. In: PROCEEDINGS IWPC 2000. 8th International Workshop on Program Comprehension. Los Alamitos, CA, USA: IEEE Computer Society, Jan. 2000. P. 48. Cit. on pp. 62, 67.

HORN, Paul. **Autonomic Computing: IBM’s Perspective on the State of Information Technology**. [S.l.], 2001. Cit. on pp. 35, 37.

HOU, D.; HOOVER, H.J. Using SCL to specify and check design intent in source code. **IEEE Transactions on Software Engineering**, v. 32, n. 6, p. 404–423, 2006. DOI: 10.1109/TSE.2006.60. Cit. on p. 64.

IBM. **An Architectural Blueprint for Autonomic Computing**. [S.l.], June 2005. Cit. on pp. 22, 70, 72, 89.

JAMSHIDI, P. et al. A Framework for Classifying and Comparing Architecture-centric Software Evolution Research. In: 2013 17th European Conference on Software Maintenance and Reengineering. [S.l.: s.n.], 2013. P. 305–314. Cit. on p. 137.

KADDOUM, Elsy et al. Criteria for the Evaluation of Self-* Systems. In: PROCEEDINGS of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. Cape Town, South Africa: Association for Computing Machinery, 2010. (SEAMS '10), p. 29–38. Cit. on p. 22.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, v. 36, n. 1, p. 41–50, 2003. Cit. on pp. 35, 37.

KITCHENHAM, B. A. et al. Preliminary guidelines for empirical research in software engineering. **IEEE Transactions on Software Engineering**, v. 28, n. 8, p. 721–734, 2002. Cit. on pp. 25, 132.

KITCHENHAM, Barbara et al. Systematic literature reviews in software engineering – A systematic literature review. **Information and Software Technology**, v. 51, n. 1, p. 7–15, 2009. Special Section - Most Cited Articles in 2002 and Regular Research Papers. Cit. on p. 46.

KNODEL, Jens; NAAB, Matthias. How to Perform the Architecture Compliance Check (ACC)? In: PRAGMATIC Evaluation of Software Architectures. Cham: Springer International Publishing, 2016. P. 83–94. Cit. on p. 21.

KNODEL, Jens; POPESCU, Daniel. A Comparison of Static Architecture Compliance Checking Approaches. In: 2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07). [S.l.: s.n.], 2007. P. 12–12. Cit. on pp. 30, 31, 33.

KRAMER, Jeff. Whither Software Architecture? (Keynote). In: PROCEEDINGS of the 34th International Conference on Software Engineering. Zurich, Switzerland: IEEE Press, 2012. (ICSE '12), p. 963. Cit. on p. 34.

KRUPITZER, Christian; ROTH, Felix Maximilian, et al. A survey on engineering approaches for self-adaptive systems. **Pervasive Mob. Comput.**, v. 17, p. 184–206, 2015. Cit. on p. 23.

KRUPITZER, Christian; TEMIZER, Timur, et al. An Overview of Design Patterns for Self-Adaptive Systems in the Context of the Internet of Things. **IEEE Access**, v. 8, p. 187384–187399, 2020. Cit. on p. 23.

LADDAGA, Robert; ROBERTSON, Paul; SHROBE, Howie. Introduction to Self-adaptive Software: Applications. In: PROCEEDINGS of the 2Nd International Conference on Self-adaptive Software: Applications. [S.l.]: Springer-Verlag, 2003. (IWSAS'01), p. 1–5. Cit. on p. 21.

LANDI, André; SANTIBANEZ, Daniel; SANTOS, Bruno M.; CUNHA, Warteruzannan S.; DURELLI, Rafael S.; CAMARGO, Valter V. de. Architectural Conformance Checking for KDM-Represented Systems. **Journal of Systems and Software**, 2022. Cit. on pp. 23, 25, 27, 66, 110.

LI, Zengyang; AVGERIOU, Paris; LIANG, Peng. A systematic mapping study on technical debt and its management. **Journal of Systems and Software**, v. 101, p. 193–220, 2015. Cit. on p. 57.

LIN, Yun et al. Interactive and Guided Architectural Refactoring with Search-based Recommendation. In: PROCEEDINGS of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Seattle, WA, USA: ACM, 2016. (FSE 2016), p. 535–546. Cit. on p. 139.

LIU, Yepang. **PhoneAdapter Mobile System**. [S.l.: s.n.], 2013. Available from: <`http://sccpu2.cse.ust.hk/afchecker/phoneadapter.html`>. Cit. on p. 82.

LOZANO, Angela; MENS, Kim; KELLENS, Andy. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. **Science of Computer Programming**, v. 105, p. 73–91, 2015. Cit. on pp. 63, 67.

LUCKHAM, D.C. et al. Specification and analysis of system architecture using Rapide. **IEEE Transactions on Software Engineering**, v. 21, n. 4, p. 336–354, 1995. Cit. on p. 30.

MACIA, I. et al. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In: 2012 16th European Conference on Software Maintenance and Reengineering. [S.l.: s.n.], Mar. 2012. P. 277–286. Cit. on p. 23.

MAFFORT, Cristiano et al. Mining architectural violations from version history. **Empirical Software Engineering**, v. 21, n. 3, p. 854–895, Jan. 2016. Cit. on pp. 23, 63, 67.

MAGGIO, Martina et al. 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**, p. 123–128, 2017. Cit. on p. 77.

MAIER, M.W.; EMERY, D.; HILLIARD, R. Software architecture: introducing IEEE Standard 1471. **Computer**, v. 34, n. 4, p. 107–109, 2001. Cit. on p. 29.

MARINESCU, Radu; GANEA, George. inCode.Rules: An agile approach for defining and checking architectural constraints. In: PROCEEDINGS of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing. [S.l.: s.n.], 2010. P. 305–312. Cit. on p. 63.

MATALONGA, Santiago; RODRIGUES, Felyppe; TRAVASSOS, Guilherme Horta. Characterizing testing methods for context-aware software systems: Results from a quasi-systematic literature review. **Journal of Systems and Software**, v. 131, p. 1–21, 2017. Cit. on p. 47.

MEDVIDOVIC, Nenad. Moving Architectural Description from Under the Technology Lamppost. In: 32ND EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06). [S.l.: s.n.], 2006. P. 2–3. Cit. on p. 31.

MENS, K.; WUYTS, R.; D'HONDT, T. Declaratively codifying software architectures using virtual software classifications. In: PROCEEDINGS Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275). [S.l.: s.n.], 1999. P. 33–45. Cit. on pp. 34, 64.

MERA-GÓMEZ, Carlos et al. A Debt-Aware Learning Approach for Resource Adaptations in Cloud Elasticity Management. In: SERVICE-ORIENTED Computing. Cham: Springer International Publishing, 2017. P. 367–382. Cit. on p. 56.

MERA-GÓMEZ, Carlos et al. Elasticity Debt: A Debt-Aware Approach to Reason about Elasticity Decisions in the Cloud. In: PROCEEDINGS of the 9th International Conference on Utility and Cloud Computing. Shanghai, China: Association for Computing Machinery, 2016. (UCC '16), p. 79–88. Cit. on p. 56.

MORICONI, M.; RIEMENSCHNEIDER, R. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. In: cit. on p. 30.

MULLER, Hausi; VILLEGAS, Norha. Runtime Evolution of Highly Dynamic Software. In: MENS, Tom; SEREBRENIK, Alexander; CLEVE, Anthony (Eds.). **Evolving Software Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. v. 234. chap. 8, p. 229–264. Cit. on p. 35.

MURPHY, Gail C.; NOTKIN, David; SULLIVAN, Kevin. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In: PROCEEDINGS of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering. Washington, D.C., USA: Association for Computing Machinery, 1995. (SIGSOFT '95), p. 18–28. Cit. on pp. 33, 34, 62, 67.

NAJAFIZADEGAN, Negin; NAZEMI, Eslam; KHAJEHVAND, Vahid. An Autonomous Model for Self-optimizing Virtual Machine Selection by Learning Automata in Cloud Environment. **Software: Practice and Experience**, v. 51, n. 6, p. 1352–1386, 2021. Cit. on p. 39.

NORD, R. L. et al. In Search of a Metric for Managing Architectural Technical Debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. [S.l.: s.n.], 2012. P. 91–100. Cit. on p. 58.

OMG. **Architecture-Driven Modernization**. [S.l.: s.n.], 2017. http://adm.omg.org/. Cit. on pp. 40, 42.

________. **Architecture-Driven Modernization Standards Roadmap**. v. 8. [S.l.: s.n.], 2009. P. 423. Available in `http://adm.omg.org/`. Cit. on p. 40.

________. **Knowledge Discovery Meta-model (KDM)**. [S.l.: s.n.], 2016. Available in `http://www.omg.org/technology/kdm/`, specification available in `http://www.omg.org/spec/KDM/`. Cit. on pp. 40, 41, 94, 110.

OMG. **Object Constraint Language Specification, version 2.0**. Ed. by OMG. [S.l.], May 2005. Available from: <`http://fparreiras/papers/OCLSpec.pdf`>. Cit. on p. 30.

PASSOS, L. et al. Static Architecture-Conformance Checking: An Illustrative Overview. **IEEE Software**, v. 27, n. 5, p. 82–89, Sept. 2010. Cit. on pp. 21, 33.

PELLICCIONE, Patrizio et al. Automotive Architecture Framework: The experience of Volvo Cars. **Journal of Systems Architecture**, v. 77, p. 83–100, 2017. Cit. on p. 57.

PEREIRA DOS REIS, José et al. Code Smells Detection and Visualization: A Systematic Literature Review. **Archives of Computational Methods in Engineering**, Mar. 2021. Cit. on p. 59.

PÉREZ-CASTILLO, Ricardo; DE GUZMÁN, Ignacio García Rodríguez; PIATTINI, Mario. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. **Computer Standards and Interfaces**, Elsevier B.V., v. 33, n. 6, p. 519–532, 2011. Cit. on p. 40.

PERRY, Dewayne E.; WOLF, Alexander L. Foundations for the Study of Software Architecture. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 17, n. 4, p. 40–52, Oct. 1992. Cit. on pp. 45, 133.

PRUIJT, Leo; KÖPPE, Christian; BRINKKEMPER, Sjaak. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In: 2013 IEEE International Conference on Software Maintenance. [S.l.: s.n.], 2013. P. 220–229. Cit. on pp. 21, 33.

PRUIJT, Leo; KÖPPE, Christian; WERF, Jan Martijn van der, et al. The accuracy of dependency analysis in static architecture compliance checking. **Software: Practice and Experience**, v. 47, n. 2, p. 273–309, 2017. Cit. on pp. 21, 31, 32.

PSAIER, Harald; DUSTDAR, Schahram. A survey on self-healing systems: approaches and systems. **Computing**, v. 91, n. 1, p. 43–73, Jan. 2011. Cit. on p. 39.

RAIBULET, C.; ARCELLI FONTANA, F., et al. Chapter 13 - An Overview on Quality Evaluation of Self-Adaptive Systems. In: MISTRIK, Ivan et al. (Eds.). **Managing Trade-Offs in Adaptable Software Architectures**. Boston: Morgan Kaufmann, 2017. P. 325–352. Cit. on pp. 22, 46.

RAIBULET, Claudia; FONTANA, Francesca; CARETTONI, Simone. A preliminary analysis of self-adaptive systems according to different issues. **Software Quality Journal**, v. 28, n. 3, p. 1213–1243, Sept. 2020. Cit. on pp. 55, 58.

________. SAS vs. NSAS: Analysis and Comparison of Self-Adaptive Systems and Non-Self-Adaptive Systems based on Smells and Patterns. In: PROCEEDINGS of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, [s.l.]: SciTePress, 2020. P. 490–497. Cit. on pp. 55, 58, 59.

RAMIREZ, Andres J; CHENG, Betty H C. Design patterns for developing dynamically adaptive systems. **Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems**, p. 49–58, 2010. Cit. on pp. 22, 39, 133.

ROCHA, Henrique et al. DCL 2.0: modular and reusable specification of architectural constraints. **Journal of the Brazilian Computer Society**, v. 23, n. 1, p. 12, Aug. 2017. Cit. on pp. 63, 67.

ROSIK, Jacek et al. Assessing Architectural Drift in Commercial Software Development: A Case Study. **Software: Practice and Experience**, John Wiley &amp; Sons, Inc., USA, v. 41, n. 1, p. 63–86, Jan. 2011. Cit. on p. 21.

S. LANDI, A. d. et al. Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). [S.l.: s.n.], July 2017. v. 1, p. 327–336. Cit. on pp. 45, 63, 67, 125, 137.

SALEHIE, Mazeiar; TAHVILDARI, Ladan. Self-adaptive software: Landscape and research challenges. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, v. 4, n. 2, p. 1–42, 2009. Cit. on pp. 35, 80.

SAMA, M. et al. Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification. **IEEE Transactions on Software Engineering**, v. 36, n. 5, p. 644–661, Sept. 2010. Cit. on pp. 77, 113.

SAN MARTÍN, Daniel; CAMARGO, Valter. A Domain-Specific Language to Specify Planned Architectures of Adaptive Systems. In: 15TH Brazilian Symposium on Software Components, Architectures, and Reuse. Joinville, Brazil: Association for Computing Machinery, 2021. (SBCARS '21), p. 41–50. Cit. on p. 27.

SANGAL, Neeraj et al. Using Dependency Models to Manage Complex Software Architecture. **ACM SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 10, p. 167–176, Oct. 2005. Cit. on p. 62.

SANTIBANEZ, Daniel; SIQUEIRA, B.; DE CAMARGO, V. V.; FERRARI, F. Characterizing Architectural Drifts of Adaptive Systems. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). [S.l.: s.n.], 2020. P. 389–399. Cit. on pp. 22, 23, 27, 58.

SEO, C. et al. Exploring the Role of Software Architecture in Dynamic and Fault Tolerant Pervasive Systems. In: FIRST International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (SEPCASE '07). [S.l.: s.n.], 2007. P. 9–9. Cit. on pp. 52, 53.

SERIKAWA, Marcel A. et al. Towards Monitor Smells in Adaptive Systems. In: X Brazilian Symposium on Components, Architectures, and Reuse (SBCARS) 2016, Maringa, PR, Brazil, September 19-23, 2015. [S.l.: s.n.], 2016. P. 52–60. Cit. on pp. 22, 54, 55, 58.

SHARMA, Tushar; SPINELLIS, Diomidis. A survey on software smells. **Journal of Systems and Software**, v. 138, p. 158–173, 2018. Cit. on pp. 75, 76, 78.

SURYANARAYANA, Girish; SAMARTHYAM, Ganesh; SHARMA, Tushar. **Refactoring for Software Design Smells: Managing Technical Debt**. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. Cit. on pp. 58, 59, 69.

TALLABACI, G.; SILVA SOUZA, V. E. Engineering adaptation with Zanshin: An experience report. In: 2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). [S.l.: s.n.], Apr. 2013. P. 93–102. Cit. on pp. 78, 79, 80.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. **Software Architecture: Foundations, Theory, and Practice**. [S.l.]: Wiley Publishing, 2009. Cit. on pp. 29, 30, 57.

TEKINERDOGAN, Bedir. Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices. In: MISTRIK, Ivan et al. (Eds.). **Software Quality Assurance**. Boston: Morgan Kaufmann, 2016. P. 221–236. Cit. on p. 83.

TERRA, Ricardo; VALENTE, Marco Tulio. A Dependency Constraint Language to Manage Object-Oriented Software Architectures. **Software: Practice and Experience**, John Wiley &amp; Sons, Inc., USA, v. 39, n. 12, p. 1073–1094, Aug. 2009. Cit. on pp. 23, 34, 137.

UBAYASHI, Naoyasu; NOMURA, Jun; TAMAI, Tetsuo. Archface: a contract place where architectural design and code meet together. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. [S.l.: s.n.], 2010. v. 1, p. 75–84. Cit. on p. 34.

ULRICH, William M.; NEWCOMB, Philip. **Information Systems Transformation: Architecture-Driven Modernization Case Studies**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. Cit. on pp. 40, 88.

VELASCO, Perla et al. Towards Detecting MVC Architectural Smells. In: ________. **Trends and Applications in Software Engineering**. Cham: Springer International Publishing, 2018. P. 251–260. Cit. on pp. 23, 59, 63, 64, 125.

VILLEGAS, Norha M. et al. 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. Waikiki, Honolulu, HI, USA: ACM, 2011. (SEAMS '11), p. 80–89. Cit. on pp. 39, 43, 70.

VOGEL, Thomas; GIESE, Holger. Model-Driven Engineering of Self-Adaptive Software with EUREMA. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, New York, NY, USA, v. 8, n. 4, 18:1–18:33, Jan. 2014. Cit. on p. 38.

VOGEL-HEUSER, Birgit; FAY, Alexander, et al. Evolution of software in automated production systems: Challenges and research directions. **Journal of Systems and Software**, v. 110, p. 54–84, 2015. Cit. on p. 56.

VOGEL-HEUSER, Birgit; NEUMANN, Eva-Maria. Adapting the concept of technical debt to software of automated Production Systems focusing on fault handling, mode of operation and safety aspects. **IFAC-PapersOnLine**, v. 50, n. 1, p. 5887–5894, 2017. 20th IFAC World Congress. Cit. on p. 57.

VOGELSANG, A.; FEMMER, H.; JUNKER, M. Characterizing Implicit Communal Components as Technical Debt in Automotive Software Systems. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). [S.l.: s.n.], 2016. P. 31–40. Cit. on pp. 54, 57, 58.

WEYNS, D. Engineering Self-Adaptive Software Systems – An Organized Tour. In: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W). [S.l.: s.n.], 2018. P. 1–2. Cit. on pp. 35, 50.

WEYNS, Danny; CALINESCU, Radu. 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. [S.l.: s.n.], 2015. P. 88–92. Cit. on pp. 80, 104.

WEYNS, Danny; IFTIKHAR, M Usman; SÖDERLUND, Joakim. Do External Feedback Loops Improve the Design of Self-adaptive Systems?: A Controlled Experiment, p. 3–12, 2013. Cit. on pp. 39, 89, 127.

WEYNS, Danny; IFTIKHAR, M. Usman; MALEK, Sam, et al. Claims and supporting evidence for self-adaptive systems: A literature study. In: 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). [S.l.: s.n.], 2012. P. 89–98. Cit. on p. 21.

WEYNS, Danny; MALEK, Sam; ANDERSSON, Jesper. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, New York, NY, USA, v. 7, n. 1, 8:1–8:61, May 2012. Cit. on pp. 50, 70, 89.

________. On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future. In: PROCEEDINGS of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. Cape Town, South Africa: Association for Computing Machinery, 2010. (SEAMS '10), p. 84–93. Cit. on p. 38.

WEYNS, Danny; SCHMERL, Bradley, et al. 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**. Ed. by Rogério de Lemos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 76–107. Cit. on pp. 22, 136.

WOHLIN, Claes et al. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2012. Cit. on p. 125.

ZIMMERMANN, O. Architectural Refactoring: A Task-Centric View on Software Evolution. **IEEE Software**, v. 32, n. 2, p. 26–29, Mar. 2015. Cit. on pp. 59, 139.

Acronyms

ACC Architecture Conformance Checking. 9, 21, 22, 23, 24, 25, 28, 31, 32, 33, 34, 43, 61, 66, 68, 85, 103, 113, 114, 116, 120, 125, 133, 134, 136

ADL Architecture Definition Languages. 30

ADM Architecture-Driven Modernization. 23, 40, 43, 134

AS Adaptive System. 9, 21, 22, 23, 24, 25, 26, 28, 43, 61, 64, 66, 68, 90, 103, 104, 111, 113, 114, 115, 118, 121, 125, 132, 133, 134, 135, 136, 137, 139

CA Current Architecture. 9, 23, 25, 26, 31, 43, 66, 85, 103, 105, 106, 107, 109, 110, 111, 114, 115, 116, 117, 118, 120, 121, 134, 135

DSL Domain Specific Language. 9, 23, 25, 32, 85, 113, 132, 134, 136

GPL General Purpose Language. 32

KDM Knowledge Discovery Metamodel. 23, 25, 40, 43, 66, 67, 120, 121, 133, 136

MAPE-K *Monitor, Analyzer, Planner, Executor, Knowledge*. 9, 22, 23, 104, 133

MOF *Meta-Object Facility*. 31

PA Planned Architecture. 9, 21, 23, 25, 26, 31, 32, 33, 43, 66, 68, 85, 88, 89, 103, 105, 106, 107, 109, 111, 114, 115, 117, 118, 120, 121, 125, 132, 134

SEAMS Software Engineering for Adaptive and Self-Managing Systems. 135

Appendix A
Technical Aspects of REMEDY

In this Appendix we deep in technical of REMEDY such as its grammar and the mechanisms to generate the CA and to enable the architecture visualizations.

A.1 REMEDY EBNF grammar

```
⟨ArchitectureDefinition⟩  ⊨  Architecture ⟨ID⟩ { ⟨DSLManaging+⟩ ⟨DSLManaged+⟩ }
                              Rules { ⟨DSLRules*⟩ }
              ⟨DSLRules⟩  ⊨  ⟨DSLRuleController⟩ | ⟨DSLRuleMonitor⟩ | ⟨DSLRuleAnalyzer⟩ |
                              ⟨DSLRulePlanner⟩ | ⟨DSLRuleExecutor⟩ | ⟨DSLRuleMO⟩ |
                              ⟨DSLRuleMController⟩ | ⟨DSLRuleKnowledge⟩
    ⟨DSLRuleMController⟩  ⊨  loopManager ⟨DSLManagerController⟩ (must-use | must-not-use)
                              loopManager ⟨DSLManagerController⟩;
     ⟨DSLRuleController⟩  ⊨  loop ⟨DSLController⟩ (must-use | must-not-use)
                              loop ⟨DSLController⟩;
        ⟨DSLRuleMonitor⟩  ⊨  monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              monitor ⟨DSLMonitor⟩; |
                              monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              analyzer ⟨DSLAnalyzer⟩; |
                              monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              planner ⟨DSLPlanner⟩; |
                              monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              executor ⟨DSLExecutor⟩; |
                              monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              knowledge ⟨DSLKnowledge⟩; |
                              monitor ⟨DSLMonitor⟩ (must-use | must-not-use)
                              sensor ⟨DSLSensor⟩;
```

```
⟨DSLRuleAnalyzer⟩  ⊨  analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      monitor ⟨DSLMonitor⟩; |
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      analyzer ⟨DSLAnalyzer⟩; |
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      planner ⟨DSLPlanner⟩; |
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      executor ⟨DSLExecutor⟩; |
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      knowledge ⟨DSLKnowledge⟩; |
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      reference-input ⟨DSLReferenceInput⟩;
                      analyzer ⟨DSLAnalyzer⟩ (must-use | must-not-use)
                      alternative ⟨DSLAlternative⟩;
 ⟨DSLRulePlanner⟩  ⊨  planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      monitor ⟨DSLMonitor⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      analyzer ⟨DSLAnalyzer⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      planner ⟨DSLPlanner⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      executor ⟨DSLExecutor⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      knowledge ⟨DSLKnowledge⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      effector ⟨DSLEffector⟩;
⟨DSLRuleExecutor⟩  ⊨  planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      monitor ⟨DSLMonitor⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      analyzer ⟨DSLAnalyzer⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      planner ⟨DSLPlanner⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      executor ⟨DSLExecutor⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      knowledge ⟨DSLKnowledge⟩; |
                      planner ⟨DSLPlanner⟩ (must-use | must-not-use)
                      effector ⟨DSLEffector⟩;
```

```
     ⟨DSLRuleKnowledge⟩  ⊨  knowledge ⟨DSLKnowledge⟩ (must-use | must-not-use)
                            monitor ⟨DSLMonitor⟩; |
                            knowledge ⟨DSLKnowledge⟩ (must-use | must-not-use)
                            analyzer ⟨DSLAnalyzer⟩; |
                            knowledge ⟨DSLKnowledge⟩ (must-use | must-not-use)
                            planner ⟨DSLPlanner⟩; |
                            knowledge ⟨DSLKnowledge⟩ (must-use | must-not-use)
                            executor ⟨DSLExecutor⟩;
            ⟨DSLRuleMO⟩  ⊨  sensor ⟨DSLSensor⟩ (must-use | must-not-use)
                            measured ⟨DSLMeasuredOutput⟩;
          ⟨DSLManaging⟩  ⊨  Managing ⟨ID⟩ { ⟨DSLManagerController*⟩ ⟨DSLController*⟩ }
           ⟨DSLManaged⟩  ⊨  Managed ⟨ID⟩ { ⟨DCLStructureElement*⟩ ⟨DSLSensor*⟩ ⟨DSLEffector*⟩
                            ⟨DSLMeasuredOutput*⟩ }
⟨DSLManagerController⟩  ⊨  LoopManager ⟨ID⟩ { ⟨DSLController*⟩ }
        ⟨DSLController⟩  ⊨  Loop ⟨ID⟩ (⟨DSLDomainRule⟩)? { ⟨DSLMonitor*⟩ ⟨DSLAnalyzer*⟩
                            ⟨DSLPlanner*⟩ ⟨DSLExecutor*⟩ ⟨DSLKnowledge*⟩ }
        ⟨DSLDomainRule⟩  ⊨  (ID | withDomainRules)
           ⟨DSLMonitor⟩  ⊨  Monitor ⟨ID⟩ ;
          ⟨DSLAnalyzer⟩  ⊨  Analyzer ⟨ID⟩ ;
           ⟨DSLPlanner⟩  ⊨  Planner ⟨ID⟩ ;
          ⟨DSLExecutor⟩  ⊨  Executor ⟨ID⟩ ;
         ⟨DSLKnowledge⟩  ⊨  Knowledge ⟨ID⟩ { ⟨DSLReferenceInput*⟩ ⟨DSLAlternative*⟩ }
            ⟨DSLSensor⟩  ⊨  Sensor ⟨ID⟩ ;
          ⟨DSLEffector⟩  ⊨  Effector ⟨ID⟩ ;
    ⟨DSLReferenceInput⟩  ⊨  ReferenceInput ⟨ID⟩ ;
    ⟨DSLMeasuredOutput⟩  ⊨  MeasuredOutput ⟨ID⟩ ;
       ⟨DSLAlternative⟩  ⊨  Alternative ⟨ID⟩ ;
```

Listing A.1 – EBNF grammar of REMEDY

A.2 Codes to Create the Current Architecture

```

public void createStructureElementFromTree(Manager baseXManager, String path_) throws
  ↪ Exception{
  List<String> memory1 = new ArrayList<String>();
 List<String> memory2 = new ArrayList<String>();
 List<String> rootList = tree.getChildren("roots");
 memory1.addAll(rootList);
 memory2.addAll(rootList);
 while (!memory1.isEmpty()){
  String child = memory1.get(0);
  rootList = tree.getChildren(child);
  Set<String> set = new HashSet<>(rootList);
  rootList.clear();
  rootList.addAll(set);
  if (!rootList.isEmpty()){
   for (int i = rootList.size()-1; i >= 0; i--){
    if (i == rootList.size()-1 ){
     String parent = memory1.remove(0);
     if (memory2.contains(new String(parent))){
      memory2.remove(new String(parent));
      this.createStructureElement(baseXManager, null, arent, path_);
     }
     memory1.add(0,rootList.get(i));
    }
    else
     memory1.add(0,rootList.get(i));
    this.createStructureElement(baseXManager, child, memory1.get(0),path_);
   }
  }
  else
   memory1.remove(0);
 }
 for (String memory: memory2)
  this.createStructureElement(baseXManager, null, memory, path_);
}
```

Listing A.2 – Code to create an hierarchical structure of AS elements

```
modeltype kdm "strict" uses kdm('http://www.eclipse.org/MoDisco/kdm/kdm');
modeltype code "strict" uses code('http://www.eclipse.org/MoDisco/kdm/code');
modeltype core "strict" uses core('http://www.eclipse.org/MoDisco/kdm/core');
modeltype action "strict" uses action('http://www.eclipse.org/MoDisco/kdm/action');
modeltype source "strict" uses source('http://www.eclipse.org/MoDisco/kdm/source');
modeltype structure "strict" uses structure('http://www.eclipse.org/MoDisco/kdm/structure
  ↪ ');

transformation addImplement(inout sourceModel:kdm);

configuration property abstraction: String;
configuration property abstractionType: String;
configuration property codeElement: String;
configuration property elementType: String;
configuration property elementPath: String;

main() {

     --code Elements
     if (abstractionType.equalsIgnoreCase("component"))
     {
        sourceModel.objectsOfType(Component)-> map addImplementationComponent();
     }
     elif (abstractionType.equalsIgnoreCase("subsystem")){

        sourceModel.objectsOfType(Subsystem)-> map addImplementationSubsystem();
     }
}

mapping Component::addImplementationComponent():Component when {self.name = abstraction}{
    init{

        if (elementType.equalsIgnoreCase("package")) {

            var pk : Package := getPackage(codeElement);
            var path: String := getCompletePath(pk);
            path := path + codeElement;



            if (path.equalsIgnoreCase(elementPath))
            {
                result := self->forOne(r){
                    r.implementation +=getPackage(codeElement);
                }
            }
        }
        elif (elementType.equalsIgnoreCase("class")){

            var class_:ClassUnit := getClass(codeElement);
            var path:String := getCompletePath(class_);
            path := path + codeElement;

            if (path.equalsIgnoreCase(elementPath))
            {
                result := self->forOne(r){
                    r.implementation +=getClass(codeElement);
                }
            }
        }
        elif (elementType.equalsIgnoreCase("interface")){

            var class_:InterfaceUnit := getInterface(codeElement);
            var path:String := getCompletePath(class_);
            path := path + codeElement;

            if (path.equalsIgnoreCase(elementPath))
            {
                result := self->forOne(r){
                    r.implementation +=getInterface(codeElement);
                }
            }
        }
        elif (elementType.equalsIgnoreCase("method")){

            var method:MethodUnit := getMethod(codeElement);
            var path:String := getCompletePath(method);
            path := path + codeElement;

            if (path.equalsIgnoreCase(elementPath))
            {
                result := self->forOne(r){
                    r.implementation +=getMethod(codeElement);
                }
            }
        }
        elif (elementType.equalsIgnoreCase("variable") or elementType.equalsIgnoreCase("field")){

            var varOrField:StorableUnit := getVariable(codeElement);
            var path:String := getCompletePath(varOrField);
            path := path + codeElement;

            if (path.equalsIgnoreCase(elementPath))
            {
                result := self->forOne(r){
                    r.implementation +=getVariable(codeElement);
                }
            }
        }
    }
}

mapping Subsystem::addImplementationSubsystem():Subsystem when {self.name = abstraction}{
    init{

        if (elementType.equalsIgnoreCase("package")) {
            result := self->forOne(r){
                r.implementation +=getPackage(codeElement);
            }
        }
        elif (elementType.equalsIgnoreCase("class")){

            result := self->forOne(r){
                r.implementation +=getClass(codeElement);
            }
        }
        elif (elementType.equalsIgnoreCase("interface")){

            result := self->forOne(r){
                r.implementation +=getInterface(codeElement);
            }
        }
        elif (elementType.equalsIgnoreCase("method")){

            result := self->forOne(r){
                r.implementation +=getMethod(codeElement);
            }
        }
        elif (elementType.equalsIgnoreCase("variable") or elementType.equalsIgnoreCase("field")){

            result := self->forOne(r){
                r.implementation +=getVariable(codeElement);
            }
        }
    }
}
```

```

query getPackage(name:String):Package {

    var packageName : String := name;
    var pkg:Package:= null;

    sourceModel.objectsOfType(Package) -> forEach(r){

            if (packageName.equalsIgnoreCase(r.name))
            {
                pkg := r;
                break;
            };
    };
    return pkg;
}

query getClass(name:String):ClassUnit {

    var className : String := name;
    var class_:ClassUnit:= null;

    sourceModel.objectsOfType(ClassUnit) -> forEach(r){

            if (className.equalsIgnoreCase(r.name))
            {
                class_ := r;
                break;
            };
    };
    return class_;
}

query getInterface(name:String):InterfaceUnit {

    var interfaceName : String := name;
    var class_:InterfaceUnit:= null;

    sourceModel.objectsOfType(InterfaceUnit) -> forEach(r){

            if (interfaceName.equalsIgnoreCase(r.name))
            {
                class_ := r;
                break;
            };
    };
    return class_;
}

query getMethod(name:String):MethodUnit {

    var methodName : String := name;
    var method:MethodUnit:= null;

    sourceModel.objectsOfType(MethodUnit) -> forEach(r){

            if (methodName.equalsIgnoreCase(r.name))
            {
                method := r;
                break;
            };
    };
    return method;
}


query getVariable(name:String):StorableUnit {

    var variableName : String := name;
    var variable:StorableUnit:= null;

    sourceModel.objectsOfType(StorableUnit) -> forEach(r){

            if (variableName.equalsIgnoreCase(r.name))
            {
                variable := r;
                break;
            };
    };
    return variable;
}

query getCompletePath(elementInput:Element): String{

    var name_:String := "";
    var element:Element := elementInput;

    while (not element.oclIsTypeOf(CodeModel)){

        element := element.container().oclAsType(Element);
        if (element.oclIsTypeOf(MethodUnit)){

            name_:= element.oclAsType(MethodUnit).name + "." + name_;

        }
        elif (element.oclIsTypeOf(ClassUnit)){

            name_:= element.oclAsType(ClassUnit).name + "." + name_;
        }
        elif (element.oclIsTypeOf(InterfaceUnit)){

            name_:= element.oclAsType(InterfaceUnit).name + "." + name_;
        }
        elif (element.oclIsTypeOf(Package)){

            name_:= element.oclAsType(Package).name + "." + name_;
        }
        elif (element.oclAsType(StorableUnit))
        {
            name_:= element.oclAsType(StorableUnit).name + "." + name_;
        }
    };
    return name_;
}
```

Listing A.3 – QVT-o file to add implementations in the KDM CA instance

```
modeltype kdm "strict" uses kdm('http://www.eclipse.org/MoDisco/kdm/kdm');
modeltype code "strict" uses code('http://www.eclipse.org/MoDisco/kdm/code');
modeltype core "strict" uses core('http://www.eclipse.org/MoDisco/kdm/core');
modeltype action "strict" uses action('http://www.eclipse.org/MoDisco/kdm/action');
modeltype source "strict" uses source('http://www.eclipse.org/MoDisco/kdm/source');
modeltype structure "strict" uses structure('http://www.eclipse.org/MoDisco/kdm/structure');

transformation addAggregated(inout sourceModel:kdm);

configuration property relationship: Set(KDMRelationship);
configuration property abstractionType: String;
configuration property abstractionFrom: String;
configuration property abstractionTo: String;

main() {

    if (abstractionType.equalsIgnoreCase("component"))
    {
        sourceModel.objectsOfType(Component)-> map aggregatedComponent();
     }
    elif (abstractionType.equalsIgnoreCase("subsystem")){
        sourceModel.objectsOfType(Subsystem)-> map aggregatedSubsystem();
    }
}

mapping Component::aggregatedComponent():Component when {self.name = abstractionFrom }{

    init{

        var found:Boolean = false;
        result := self->forEach(a){

            a.aggregated -> forEach(b){

                if (b._from.name.equalsIgnoreCase(abstractionFrom) and b.to.name.equalsIgnoreCase(abstractionTo)){
                    b.relation += relationship;
                    b.density := b.relation->size();
                    found := true;
                };
            };

            if (found = false){

                a.aggregated +=object AggregatedRelationship{
                    _from:=getKDMEntity(abstractionFrom);
                    to:=getKDMEntity(abstractionTo);
                    relation += relationship;
                    density := relation->size();
                };
            }
        }
    }
}

mapping Subsystem::aggregatedSubsystem():Subsystem when {self.name = abstractionFrom }{

    init{

        var found:Boolean = false;
        result := self->forEach(a){

            a.aggregated -> forEach(b){

                if (b._from.name.equalsIgnoreCase(abstractionFrom) and b.to.name.equalsIgnoreCase(abstractionTo)){
                    b.relation += relationship;
                    b.density := b.relation->size();
                    found := true;
                };
            };

            if (found = false){

                a.aggregated +=object AggregatedRelationship{
                    _from:=getKDMEntity(abstractionFrom);
                    to:=getKDMEntity(abstractionTo);
                    relation += relationship;
                    density := relation->size();
                };
            }
        }
    }
}

query getKDMEntity(name:String):KDMEntity {

    var abstractionName : String := name;
    var abstraction:KDMEntity:= null;

    sourceModel.objectsOfType(Component) -> forEach(r){

            if (abstractionName.equalsIgnoreCase(r.name))
            {
                abstraction := r;
                break;
            };
    };

    sourceModel.objectsOfType(Subsystem) -> forEach(r){

            if (abstractionName.equalsIgnoreCase(r.name))
            {
                abstraction := r;
                break;
            };
    };

    return abstraction;
}
```

Listing A.4 – QVT-o file to add the relationships in the KDM CA instanca

A.3 Architecture Conformance Checking Code

```
public void checkConstraint(IFile currentArchitecturePath, IFile constraintPath) throws
    ↪ SQLException  {
    DataConstraint dataConstraint = new DataConstraint(workspacePath + projectName);
    try {
      dataConstraint.deleteTables();
    } catch (Exception e3) {
      // TODO Auto-generated catch block
      e3.printStackTrace();
    }

    OCLstdlib.install();
    CompleteOCLStandaloneSetup.doSetup();
    OCLstdlibStandaloneSetup.doSetup();
    EssentialOCLStandaloneSetup.doSetup();

    KdmPackage.eINSTANCE.eClass();
    Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(Resource.Factory.
    ↪ Registry.DEFAULT_EXTENSION,new XMIResourceFactoryImpl());
    ResourceSet resourceSet = new ResourceSetImpl();
    Resource resource = resourceSet.getResource(URI.createURI(currentArchitecturePath.
    ↪ getFullPath().toString()), true);
    OCL ocl = OCL.newInstance(resource.getResourceSet());

    // get an OCL text file via some hypothetical API
    Resource asResource = ocl.parse(URI.createFileURI(constraintPath.getRawLocation().
    ↪ toOSString()));

    Map<String, ExpressionInOCL> constraintMap = new HashMap<String, ExpressionInOCL>();
    for (TreeIterator<EObject> tit = asResource.getAllContents(); tit.hasNext(); )
    {
      EObject next = tit.next();
      if (next instanceof Constraint)
      {
        Constraint constraint = (Constraint)next;
        ExpressionInOCL expressionInOCL = null;
        try {expressionInOCL = ocl.getSpecification(constraint);} catch (ParserException
    ↪ e) {e.printStackTrace();}
        if (expressionInOCL != null)
        {
          String name = constraint.getName();
          if (name != null)
            constraintMap.put(name, expressionInOCL);
        }
      }
    }

    StructureModel structureModel = (StructureModel) resource.getContents().get(0).
    ↪ eContents().get(4);
    for (String key : constraintMap.keySet() ) {
      Boolean check = false;
      ExpressionInOCL expressionInOCL = constraintMap.get(key);
      String type = key.split(Pattern.quote("_"))[0];
      String abstraction = key.split(Pattern.quote("_"))[1] + "_" +key.split(Pattern.
    ↪ quote("_"))[2];

      try {
        check = (Boolean) ocl.evaluate(structureModel, expressionInOCL);

        if (type.equals("exist")) {

          try {
            int row = dataConstraint.insertExistence(projectName.replaceAll("\\/", ""),
    ↪ abstraction, (check.booleanValue() ? 1 : 0 ));
            dataConstraint.insertExistenceRules(projectName.replaceAll("\\/", ""), key,
    ↪ expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),
    ↪ row);

          } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }
        }
        else {
          if (type.equals("composite")) {

            try {
              int row =dataConstraint.insertComposite(projectName.replaceAll("\\/", ""),
    ↪ abstraction, (check.booleanValue() ? 1 : 0 ));
              dataConstraint.insertCompositeRules(projectName.replaceAll("\\/", ""), key,
    ↪  expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),
    ↪ row);
            } catch (Exception e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
            }
          }
          else {

            if (type.equals("access")) {
              String abstraction2 = key.split(Pattern.quote("_"))[3] + "_" + key.split(
    ↪ Pattern.quote("_"))[4];
              try {
                int row =dataConstraint.insertAccess(projectName.replaceAll("\\/", ""),
    ↪ abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
                dataConstraint.insertAccessRules(projectName.replaceAll("\\/", ""), key,
    ↪ expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),
    ↪ row);
              } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
              }
            }else
            {
              if (type.equals("not")) {
                String abstraction2 = key.split(Pattern.quote("_"))[4] + "_" + key.split(
    ↪ Pattern.quote("_"))[5];
                abstraction =  key.split(Pattern.quote("_"))[2] + "_" + key.split(Pattern
    ↪ .quote("_"))[3];
                try {
                  int row =dataConstraint.insertAccess(projectName.replaceAll("\\/", ""),
    ↪  abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
                  dataConstraint.insertAccessRules(projectName.replaceAll("\\/", ""), key
    ↪ , expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),
    ↪ row);
                } catch (Exception e) {
                  // TODO Auto-generated catch block
```

```java
                e.printStackTrace();
              }
            }
            else {
              if (type.equals("domain")) {

                if (key.split(Pattern.quote("_"))[1].equals("not")) {

                  String abstraction2 = key.split(Pattern.quote("_"))[5] + "_" + key.split(Pattern.quote("_"))[6];
                  abstraction =  key.split(Pattern.quote("_"))[3] + "_" + key.split(Pattern.quote("_"))[4];
                  try {
                    int row =dataConstraint.insertDomain(projectName.replaceAll("\\/", ""), abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
                    dataConstraint.insertDomainRules(projectName.replaceAll("\\/", ""), key, expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),row);
                  } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                  }
                }
                else
                {
                  String abstraction2 = key.split(Pattern.quote("_"))[4] + "_" + key.split(Pattern.quote("_"))[5];
                  abstraction =  key.split(Pattern.quote("_"))[2] + "_" + key.split(Pattern.quote("_"))[3];
                  try {
                    int row =dataConstraint.insertDomain(projectName.replaceAll("\\/", ""), abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
                    dataConstraint.insertDomainRules(projectName.replaceAll("\\/", ""), key, expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),row);
                  } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                  }
                }
              }
            }
          }
        }
      }
    }
    catch (InvalidValueException e)
    {
      if (type.equals("exist"))
        try {
          int row =dataConstraint.insertExistence(projectName.replaceAll("\\/", ""), abstraction, 0);
          dataConstraint.insertExistenceRules(projectName.replaceAll("\\/", ""), key, expressionInOCL.getBody().replaceAll("\'","") , 0,row);
        } catch (Exception e1) {
          // TODO Auto-generated catch block
          e1.printStackTrace();
        }
      else {
        if (type.equals("composite")) {
```

```
      try {
        int row =dataConstraint.insertComposite(projectName.replaceAll("\\/", ""),
↪ abstraction, 0);
        dataConstraint.insertCompositeRules(projectName.replaceAll("\\/", ""), key,
↪  expressionInOCL.getBody().replaceAll("\'","") ,0,row);
      } catch (Exception e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
      }
    }
    else {
      if (type.equals("access")) {
        String abstraction2 = key.split(Pattern.quote("_"))[3]  + "_" +  key.split(
↪ Pattern.quote("_"))[4];
        try {
          int row =dataConstraint.insertAccess(projectName.replaceAll("\\/", ""),
↪ abstraction, abstraction2, 0);
          dataConstraint.insertAccessRules(projectName.replaceAll("\\/", ""), key,
↪ expressionInOCL.getBody().replaceAll("\'","") ,0,row);
        } catch (Exception e2) {
          // TODO Auto-generated catch block
          e2.printStackTrace();
        }
      }
      else
      {
        if (type.equals("not")) {
          String abstraction2 = key.split(Pattern.quote("_"))[4] + "_" + key.split(
↪ Pattern.quote("_"))[5];
          abstraction =  key.split(Pattern.quote("_"))[2] + "_" + key.split(Pattern
↪ .quote("_"))[3];
          try {
            int row =dataConstraint.insertAccess(projectName.replaceAll("\\/", ""),
↪  abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
            dataConstraint.insertAccessRules(projectName.replaceAll("\\/", ""), key
↪ , expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),
↪ row);
          } catch (Exception e3) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }
        }
        else {
          if (type.equals("domain")) {

            if (key.split(Pattern.quote("_"))[1].equals("not")) {

              String abstraction2 = key.split(Pattern.quote("_"))[4] + "_" + key.
↪ split(Pattern.quote("_"))[5];
              abstraction =  key.split(Pattern.quote("_"))[2] + "_" + key.split(
↪ Pattern.quote("_"))[3];
              try {
                int row =dataConstraint.insertDomain(projectName.replaceAll("\\/",
↪ ""), abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
                dataConstraint.insertDomainRules(projectName.replaceAll("\\/", ""),
↪  key, expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 :
↪ 0 ),row);
              } catch (Exception e4) {
                // TODO Auto-generated catch block
                e.printStackTrace();
          }

         }
         else
         {
          String abstraction2 = key.split(Pattern.quote("_"))[5] + "_" + key.split(Pattern.quote("_"))[6];
          abstraction =  key.split(Pattern.quote("_"))[3] + "_" + key.split(Pattern.quote("_"))[4];
          try {
           int row =dataConstraint.insertDomain(projectName.replaceAll("\\/", ""), abstraction, abstraction2, (check.booleanValue() ? 1 : 0 ));
           dataConstraint.insertDomainRules(projectName.replaceAll("\\/", ""), key, expressionInOCL.getBody().replaceAll("\'","") , (check.booleanValue() ? 1 : 0 ),row);
          } catch (Exception e5) {
           // TODO Auto-generated catch block
           e.printStackTrace();
          }
         }
        }
       }
      }
     }
    }
   }
  }

  //Update constraints according to the existence of abstractions
  try {
   dataConstraint.checkRealConstraints();
  } catch (Exception e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
```

Listing A.5 – Code to check architectural drifts in the CA

A.4 Codes to Transform from UML Package Model to the Structurizr DSL

```
public void createPlantComponentDiagram(Resource r, String path, String projectName, String mappingPath, String title) {

    Workspace workspace = new Workspace("Component Diagram", title);
    Model model = workspace.getModel();
    ViewSet views = workspace.getViews();
    SoftwareSystem adaptiveSystem = model.addSoftwareSystem(Location.Internal, "Adaptive System Architecture", "Allows customers to view information about their bank accounts, and make payments.");

    List<Package> memory1 = new ArrayList<Package>();
    List<String> roots = new ArrayList<String>();

    for (int z=0; z<r.getContents().get(0).eContents().size(); z++ )
    {
      if (r.getContents().get(0).eContents().get(z) instanceof Package)
      {
        Package package1 = (Package)r.getContents().get(0).eContents().get(z);
        memory1.add(package1);
        roots.add(package1.getName());
      }
    }
    dependenciesList =  LinkedListMultimap.create();
    packagedList = LinkedListMultimap.create();

    while (!memory1.isEmpty())
    {
      Package node = memory1.remove(0);
      EList<Package> children = node.getNestedPackages();
      ECollections.reverse(children);
      if (!children.isEmpty()){

        for (int i =0; i< children.size(); i++){
          packagedList.put(node.getName(), children.get(i).getName());
          memory1.add(0, children.get(i));
        }
      }

      EList<Dependency> deps = node.getClientDependencies();
      if (!deps.isEmpty()){
        for (int i=0 ; i< deps.size(); i++) {

          Dependency dependency = deps.get(i);
          EList<NamedElement> suppliers = dependency.getSuppliers();
          for (int j=0; j< suppliers.size(); j++)
          {
            NamedElement element = dependency.getSuppliers().get(j);
            dependenciesList.put(node.getName(), element.getName());
          }
        }
      }
    }

    this.loadMappings(mappingPath);

    List<DeploymentNode> architecture = new ArrayList<DeploymentNode>();
    for (String key : packagedList.keySet()) {
      if (roots.contains(key))
      {
        architecture.add(model.addDeploymentNode(key, key, key, mappingMap.get(key)));
        roots.remove(key);
      }
    }

    while (!architecture.isEmpty())
    {
      DeploymentNode n1 = architecture.remove(0);
      List<String> children = packagedList.get(n1.getName());
      if (!children.isEmpty()){
        for (int i =0; i< children.size(); i++)
          architecture.add(0, n1.addDeploymentNode(children.get(i), children.get(i), mappingMap.get(children.get(i))));
      }
      else
      {
        n1.addTags(END_NODE);
        Container container = adaptiveSystem.addContainer("c"+n1.getName(), "c"+n1.getName(), "");
        n1.add(container);
      }
    }


    //Root nodes without children
    for (int i = 0; i< roots.size(); i++)
    {
      DeploymentNode node= model.addDeploymentNode(roots.get(i), roots.get(i), roots.get(i), mappingMap.get(roots.get(i)));
      Container container = adaptiveSystem.addContainer("NULL_"+i, "NULL", "NULL");
      node.add(container);
    }

    List<Relationship> lRelationships = new ArrayList<Relationship>();

    for (Element element : model.getElements())
    {
      if (element instanceof DeploymentNode || element instanceof Container )
      {
        List<String> dependencies = dependenciesList.get(element.getName());
        for (String dependency : dependencies)
        {
          Optional<Element> oElement = model.getElements().stream().filter(e -> (e instanceof DeploymentNode || e instanceof Container) && (e.getName().equals(dependency))).findFirst();;
          Element e = oElement.get();
          if (e instanceof DeploymentNode && element instanceof DeploymentNode)
          {
            DeploymentNode d1 = (DeploymentNode) element;
            DeploymentNode d2 = (DeploymentNode) e;
            Relationship relationship = d1.uses(d2, "must-use","");
            lRelationships.add(relationship);
          }
        }
      }
    }

    DeploymentView developmentView = views.createDeploymentView(adaptiveSystem, "LiveDeployment", title);
    developmentView.setEnvironment("");
    for (DeploymentNode node : model.getDeploymentNodes())
      developmentView.add(node);

    for (Relationship relation : lRelationships)
      developmentView.add(relation);

    StringWriter stringWriter = new StringWriter();
    PlantUMLWriter plantUMLWriter = new PlantUMLWriter();
    plantUMLWriter.write(workspace, stringWriter);

    String diagram = stringWriter.toString();
    diagram = diagram.replaceAll("(?m)rectangle.*", "");
    diagram = diagram.replaceAll("(?m)@enduml.*", "");

    String style = "skinparam node {\n" +
        "\n" +
        " backgroundColor<<Reference Input>> #3498db\n" +
        " backgroundColor<<Alternative>> #3498db\n" +
        " backgroundColor<<Measured Output>> #3498db\n" +
        " backgroundColor<<Executor>> #3498db\n" +
        " backgroundColor<<Sensor>> #3498db\n" +
        " backgroundColor<<Monitor>> #3498db\n" +
        " backgroundColor<<Analyzer>> #3498db\n" +
        " backgroundColor<<Planner>> #3498db\n" +
        " backgroundColor<<Effector>> #3498db\n" +
        " FontColor<<Reference Input>> white\n" +
        " FontColor<<Alternative>> white\n" +
        " FontColor<<Measured Output>> white\n" +
        " FontColor<<Executor>> white\n" +
        " FontColor<<Sensor>> white\n" +
        " FontColor<<Monitor>> white\n" +
        " FontColor<<Analyzer>> white\n" +
        " FontColor<<Planner>> white\n" +
        " FontColor<<Effector>> white\n" +
        "}\n" +
        "@enduml";

    diagram = diagram + style;

    try {
      Files.write(Paths.get(path + "ComponentDiagram.txt"), diagram.getBytes(), StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING );
    } catch (IOException e) {

      e.printStackTrace();
    }
}
```

Listing A.6 – Code to create the Structurizr DSL from UML Package model

```
@startuml(id=LiveDeployment)
scale max 2000x2000
title Adaptive System Architecture
```

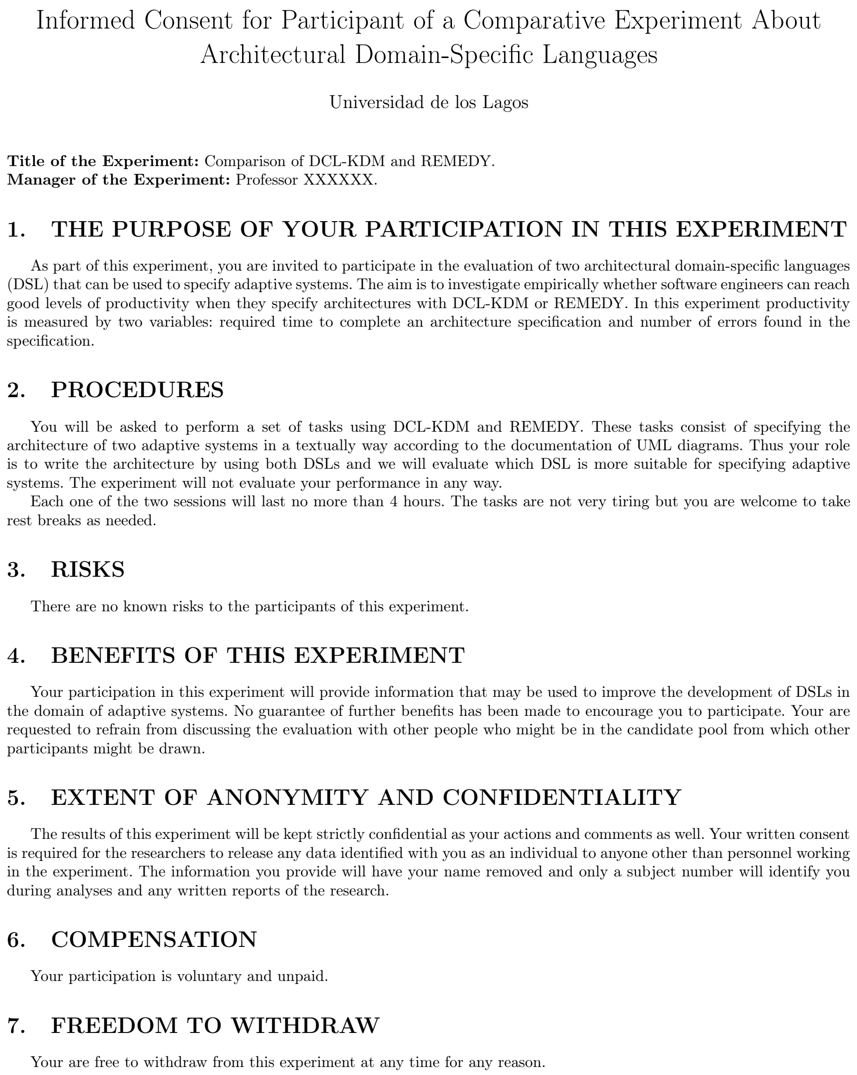
```
caption This is the planned architecture of project: EnvironmentGuardRobotPlannedArchitecture

skinparam {
  shadowing false
  arrowColor #707070
  actorBorderColor #707070
  componentBorderColor #707070

  noteBackgroundColor #ffffff
  noteBorderColor #707070
}
node "adaptationManager" <<Managing Subsystem>> as 2 {
  node "loopManager_1" <<Loop Manager>> as 4 {
    node "slaveLoop" <<Loop>> as 6 {
      node "slaveMonitor" <<Monitor>> as 9 {

      }
      node "slaveExecutor" <<Executor>> as 10 {

      }
      node "slaveAnalyzer" <<Analyzer>> as 11 {

      }
      node "knowledge" <<Knowledge>> as 8 {
        node "strategy_1" <<Alternative>> as 18 {

        }
        node "strategy_2" <<Alternative>> as 19 {

        }
        node "rotationReference" <<Reference Input>> as 21 {

        }
        node "proximityReference" <<Reference Input>> as 20 {

        }
      }
      node "slavePlanner" <<Planner>> as 7 {

      }
    }
    node "masterLoop" <<Loop>> as 5 {
      node "parameterMonitor" <<Monitor>> as 34 {

      }
      node "parameterExecutor" <<Executor>> as 35 {

      }
      node "masterAnalyzer" <<Analyzer>> as 33 {

      }
      node "masterPlanner" <<Planner>> as 32 {

      }
    }
  }
}
node "environmentGuardRobot" <<Managed Subsystem>> as 3 {
  node "wheels" <<Effector>> as 47 {

  }
  node "angularSpeed" <<Measured Output>> as 45 {

  }
  node "distance" <<Measured Output>> as 49 {

  }
  node "proximity" <<Sensor>> as 44 {

  }
  node "tachometer" <<Sensor>> as 48 {

  }
  node "speed" <<Effector>> as 46 {

  }
}
33 .[#707070].> 32 : must-use
33 .[#707070].> 11 : must-use
32 .[#707070].> 35 : must-use
32 .[#707070].> 7 : must-use
35 .[#707070].> 10 : must-use
34 .[#707070].> 33 : must-use
34 .[#707070].> 9 : must-use
44 .[#707070].> 49 : must-use
11 .[#707070].> 8 : must-use
11 .[#707070].> 33 : must-use
11 .[#707070].> 20 : must-use
11 .[#707070].> 21 : must-use
11 .[#707070].> 7 : must-use
10 .[#707070].> 8 : must-use
10 .[#707070].> 35 : must-use
10 .[#707070].> 46 : must-use
10 .[#707070].> 47 : must-use
9 .[#707070].> 8 : must-use
9 .[#707070].> 34 : must-use
9 .[#707070].> 44 : must-use
9 .[#707070].> 11 : must-use
9 .[#707070].> 48 : must-use
7 .[#707070].> 8 : must-use
7 .[#707070].> 32 : must-use
7 .[#707070].> 10 : must-use
7 .[#707070].> 18 : must-use
7 .[#707070].> 19 : must-use
48 .[#707070].> 45 : must-use

skinparam node {

  backgroundColor<<Reference Input>> #3498db
  backgroundColor<<Alternative>> #3498db
  backgroundColor<<Measured Output>> #3498db
  backgroundColor<<Executor>> #3498db
  backgroundColor<<Sensor>> #3498db
  backgroundColor<<Monitor>> #3498db
  backgroundColor<<Analyzer>> #3498db
  backgroundColor<<Planner>> #3498db
  backgroundColor<<Effector>> #3498db
  FontColor<<Reference Input>> white
  FontColor<<Alternative>> white
  FontColor<<Measured Output>> white
  FontColor<<Executor>> white
  FontColor<<Sensor>> white
  FontColor<<Monitor>> white
  FontColor<<Analyzer>> white
  FontColor<<Planner>> white
  FontColor<<Effector>> white
}
@enduml
```

Listing A.7 – DSL of PlantUML for the Environment Guard Robot

Appendix B

CONTROLLED EXPERIMENT ARTIFACTS

B.1 Consent Form



Informed Consent for Participant of a Comparative Experiment About Architectural Domain-Specific Languages

Universidad de los Lagos

Title of the Experiment: Comparison of DCL-KDM and REMEDY.
Manager of the Experiment: Professor XXXXXX.

1. THE PURPOSE OF YOUR PARTICIPATION IN THIS EXPERIMENT

As part of this experiment, you are invited to participate in the evaluation of two architectural domain-specific languages (DSL) that can be used to specify adaptive systems. The aim is to investigate empirically whether software engineers can reach good levels of productivity when they specify architectures with DCL-KDM or REMEDY. In this experiment productivity is measured by two variables: required time to complete an architecture specification and number of errors found in the specification.

2. PROCEDURES

You will be asked to perform a set of tasks using DCL-KDM and REMEDY. These tasks consist of specifying the architecture of two adaptive systems in a textually way according to the documentation of UML diagrams. Thus your role is to write the architecture by using both DSLs and we will evaluate which DSL is more suitable for specifying adaptive systems. The experiment will not evaluate your performance in any way.

Each one of the two sessions will last no more than 4 hours. The tasks are not very tiring but you are welcome to take rest breaks as needed.

3. RISKS

There are no known risks to the participants of this experiment.

4. BENEFITS OF THIS EXPERIMENT

Your participation in this experiment will provide information that may be used to improve the development of DSLs in the domain of adaptive systems. No guarantee of further benefits has been made to encourage you to participate. Your are requested to refrain from discussing the evaluation with other people who might be in the candidate pool from which other participants might be drawn.

5. EXTENT OF ANONYMITY AND CONFIDENTIALITY

The results of this experiment will be kept strictly confidential as your actions and comments as well. Your written consent is required for the researchers to release any data identified with you as an individual to anyone other than personnel working in the experiment. The information you provide will have your name removed and only a subject number will identify you during analyses and any written reports of the research.

6. COMPENSATION

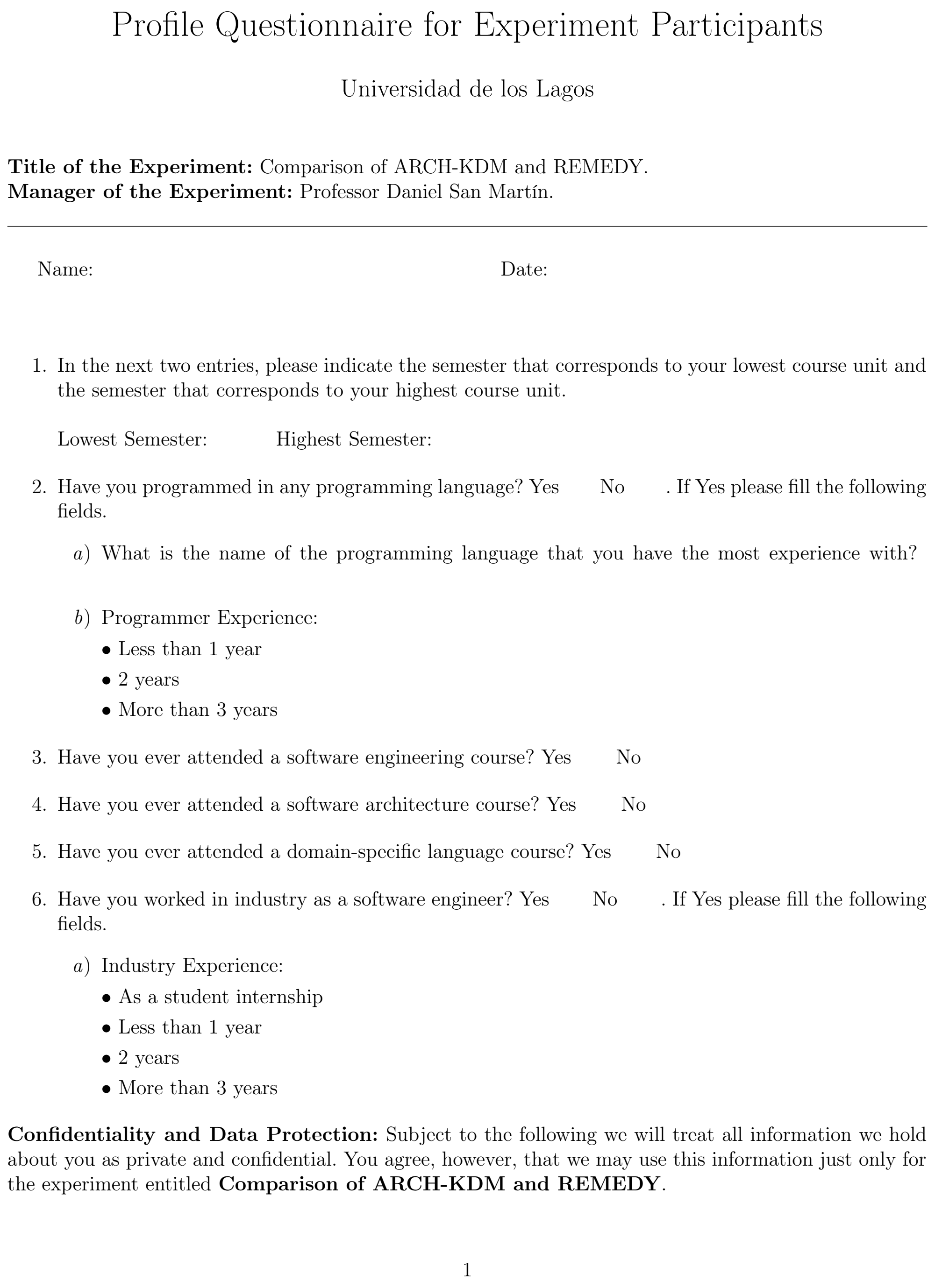
Your participation is voluntary and unpaid.

7. FREEDOM TO WITHDRAW

Your are free to withdraw from this experiment at any time for any reason.

Figure B.1 – Consent form in controlled experiment

B.2 Subject Profile Form



Profile Questionnaire for Experiment Participants

Universidad de los Lagos

Title of the Experiment: Comparison of ARCH-KDM and REMEDY.
Manager of the Experiment: Professor Daniel San Martín.

Name: Date:

1. In the next two entries, please indicate the semester that corresponds to your lowest course unit and the semester that corresponds to your highest course unit.

   Lowest Semester: Highest Semester:

2. Have you programmed in any programming language? Yes No . If Yes please fill the following fields.

   a) What is the name of the programming language that you have the most experience with?

   b) Programmer Experience:
   - Less than 1 year
   - 2 years
   - More than 3 years

3. Have you ever attended a software engineering course? Yes No

4. Have you ever attended a software architecture course? Yes No

5. Have you ever attended a domain-specific language course? Yes No

6. Have you worked in industry as a software engineer? Yes No . If Yes please fill the following fields.

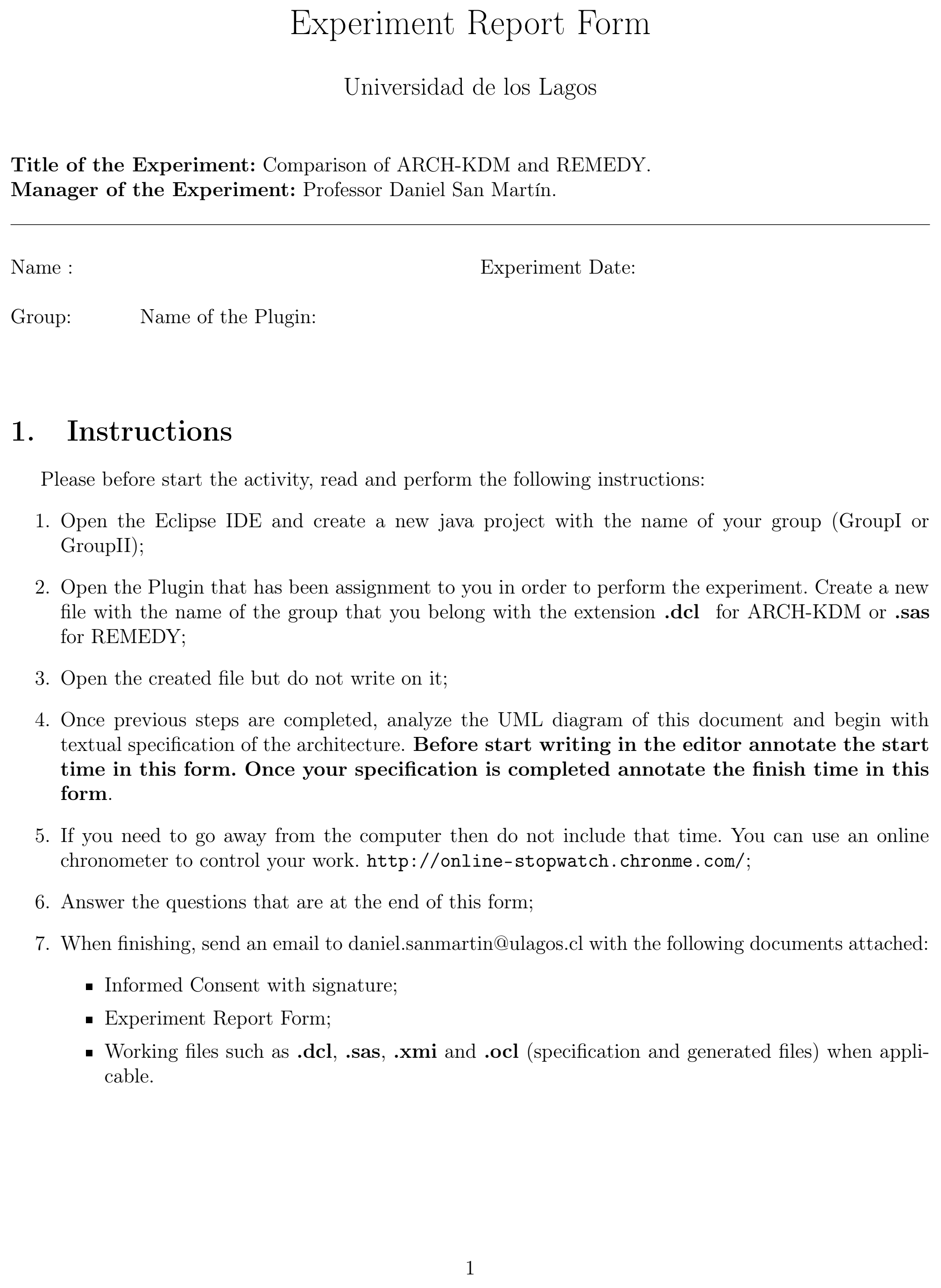
   a) Industry Experience:
   - As a student internship
   - Less than 1 year
   - 2 years
   - More than 3 years

Confidentiality and Data Protection: Subject to the following we will treat all information we hold about you as private and confidential. You agree, however, that we may use this information just only for the experiment entitled **Comparison of ARCH-KDM and REMEDY**.

1

Figure B.2 – Subject profile form

B.3 Experiment Form



Experiment Report Form

Universidad de los Lagos

Title of the Experiment: Comparison of ARCH-KDM and REMEDY.
Manager of the Experiment: Professor Daniel San Martín.

Name : Experiment Date:

Group: Name of the Plugin:

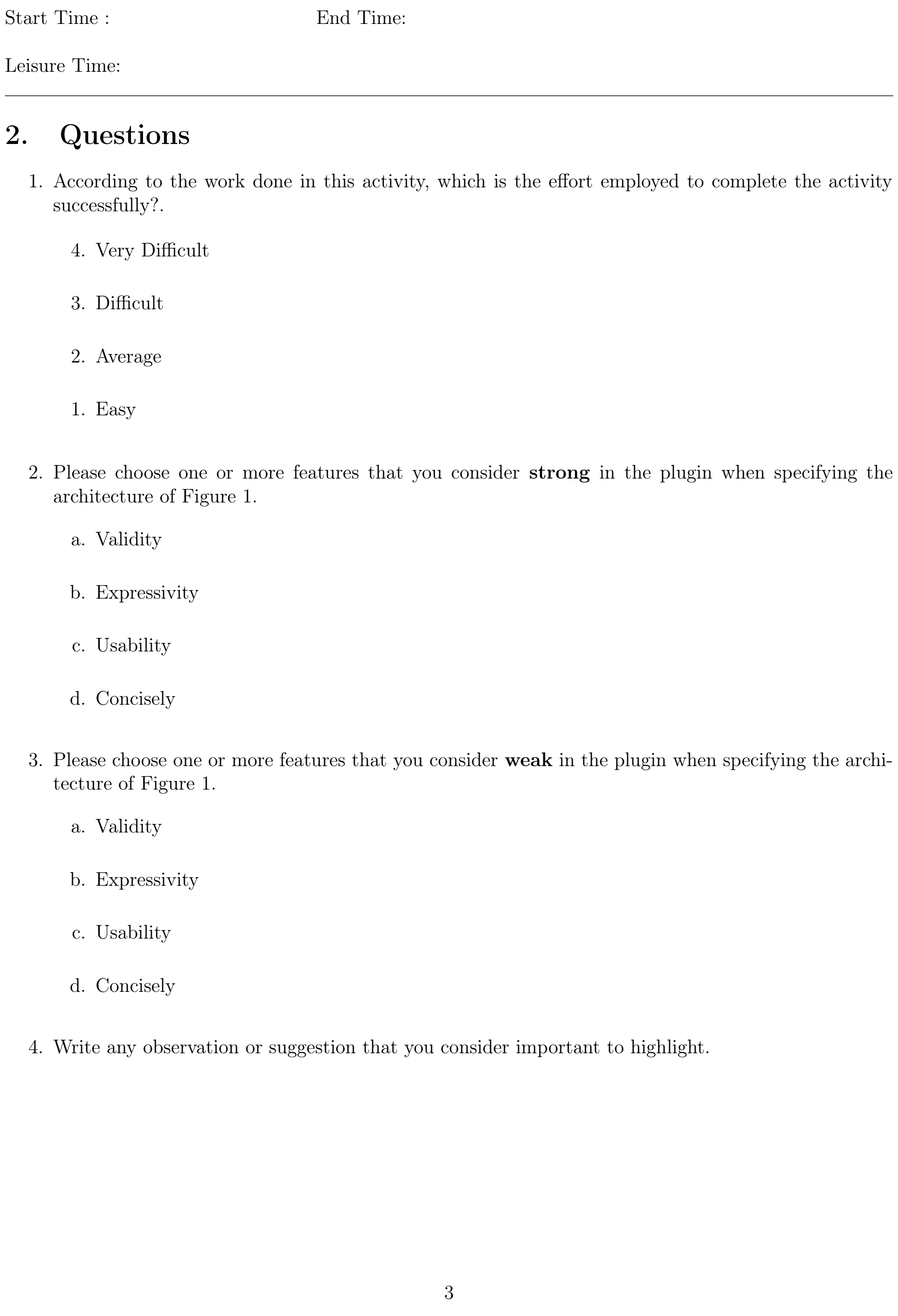
1. Instructions

Please before start the activity, read and perform the following instructions:

1. Open the Eclipse IDE and create a new java project with the name of your group (GroupI or GroupII);
2. Open the Plugin that has been assignment to you in order to perform the experiment. Create a new file with the name of the group that you belong with the extension **.dcl** for ARCH-KDM or **.sas** for REMEDY;
3. Open the created file but do not write on it;
4. Once previous steps are completed, analyze the UML diagram of this document and begin with textual specification of the architecture. **Before start writing in the editor annotate the start time in this form. Once your specification is completed annotate the finish time in this form**.
5. If you need to go away from the computer then do not include that time. You can use an online chronometer to control your work. `http://online-stopwatch.chronme.com/`;
6. Answer the questions that are at the end of this form;
7. When finishing, send an email to daniel.sanmartin@ulagos.cl with the following documents attached:
   - Informed Consent with signature;
   - Experiment Report Form;
   - Working files such as **.dcl**, **.sas**, **.xmi** and **.ocl** (specification and generated files) when applicable.

1

Figure B.3 – Experiment form (Page 1)



Start Time : End Time:

Leisure Time:

2. Questions

1. According to the work done in this activity, which is the effort employed to complete the activity successfully?.

   4. Very Difficult

   3. Difficult

   2. Average

   1. Easy

2. Please choose one or more features that you consider **strong** in the plugin when specifying the architecture of Figure 1.

   a. Validity

   b. Expressivity

   c. Usability

   d. Concisely

3. Please choose one or more features that you consider **weak** in the plugin when specifying the architecture of Figure 1.

   a. Validity

   b. Expressivity

   c. Usability

   d. Concisely

4. Write any observation or suggestion that you consider important to highlight.

3

Figure B.4 – Experiment form (Page 3)

B.4 Architectural Specifications of Adaptive Systems

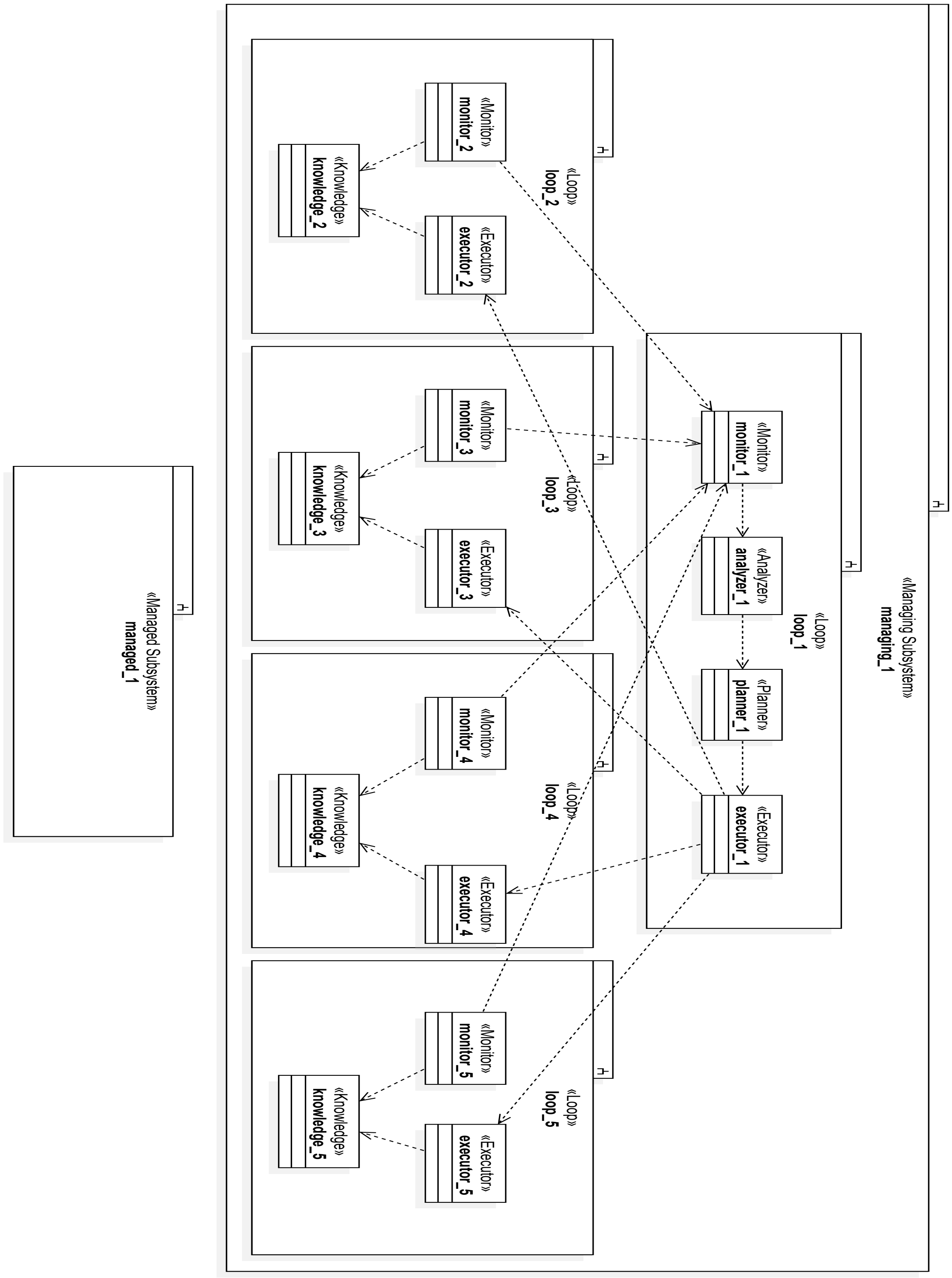


Figure B.5 – Architectural specification used in pilot

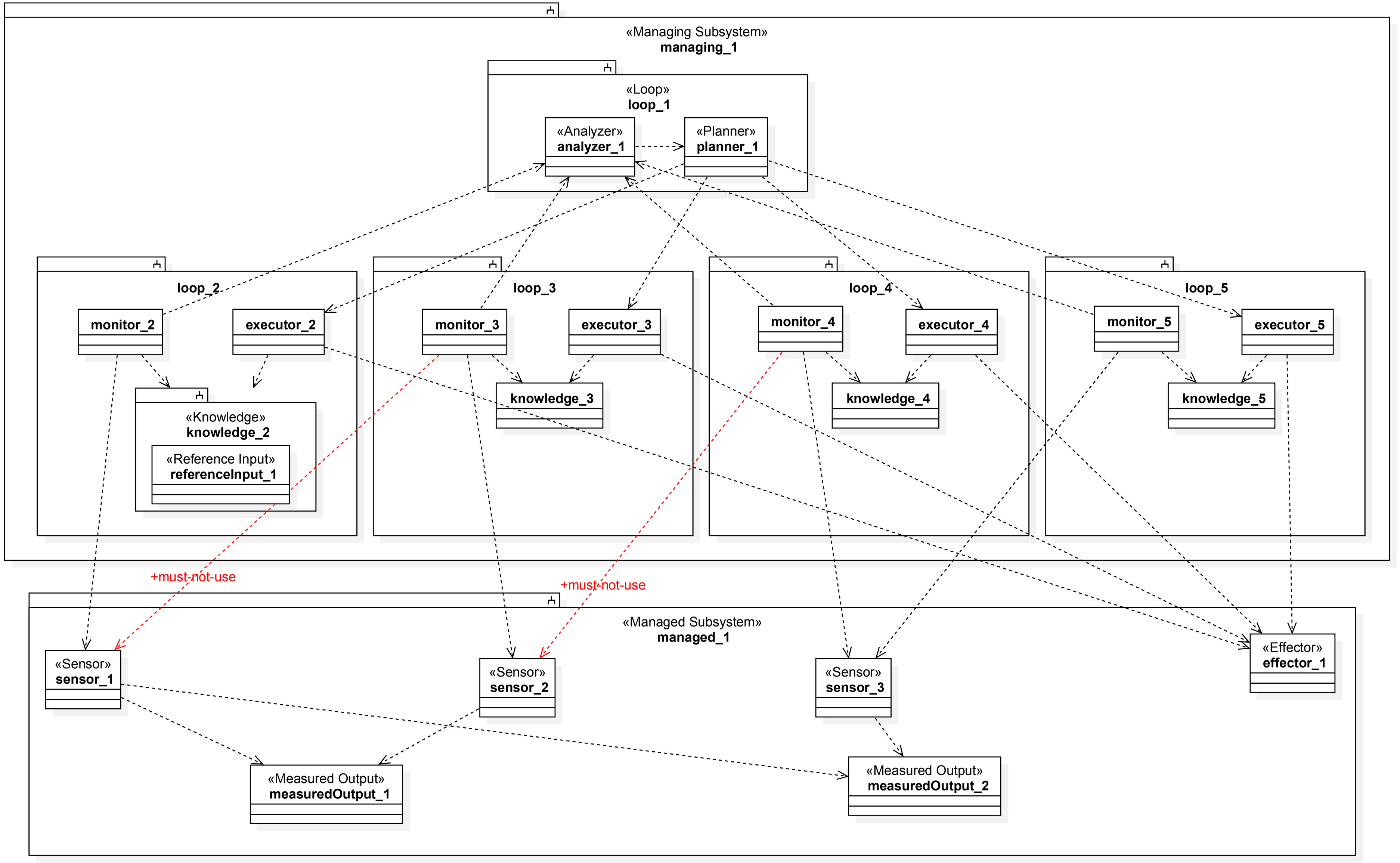


Figure B.6 – Architectural specification used in pilot

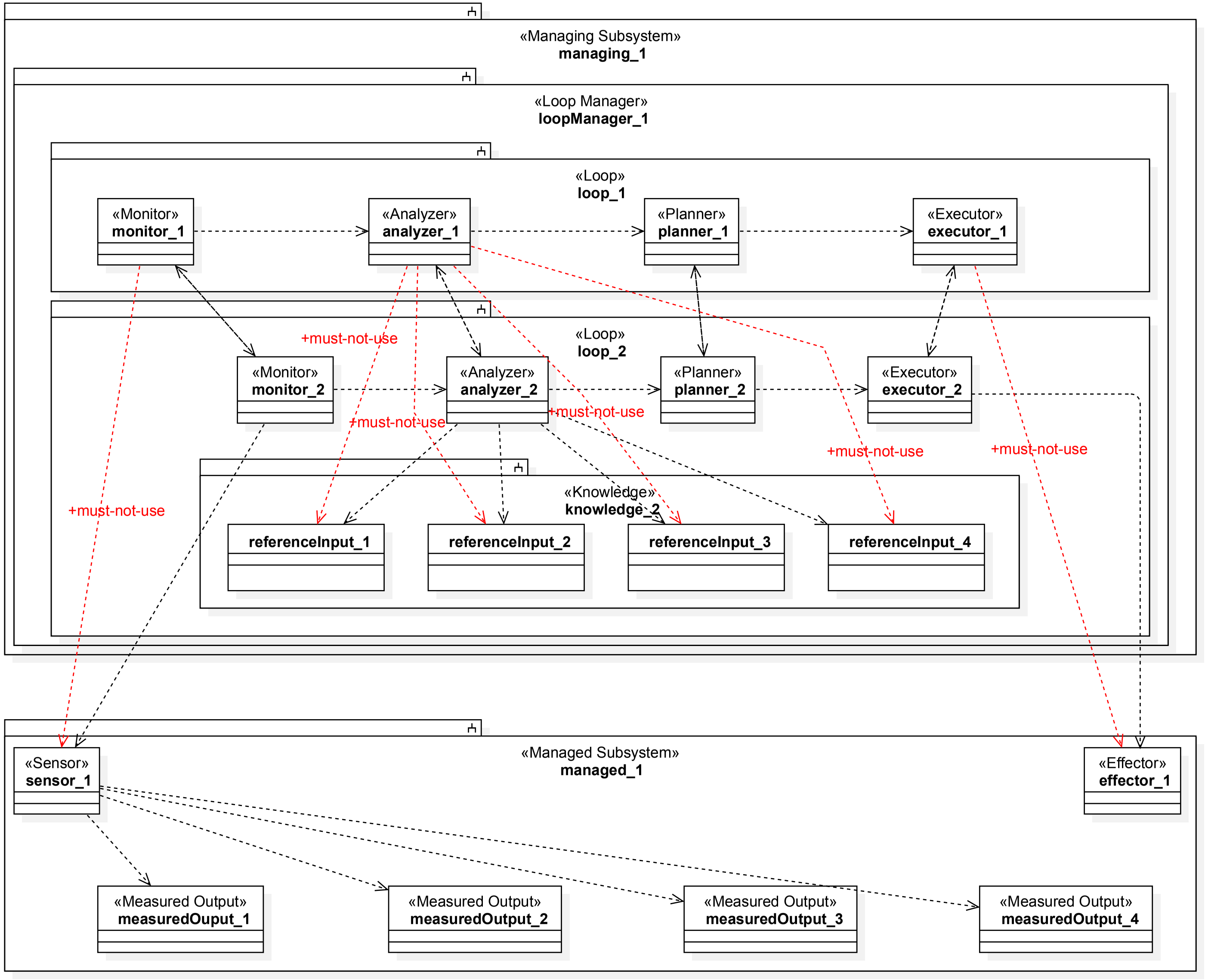


Figure B.7 – Architectural specification used in experiment

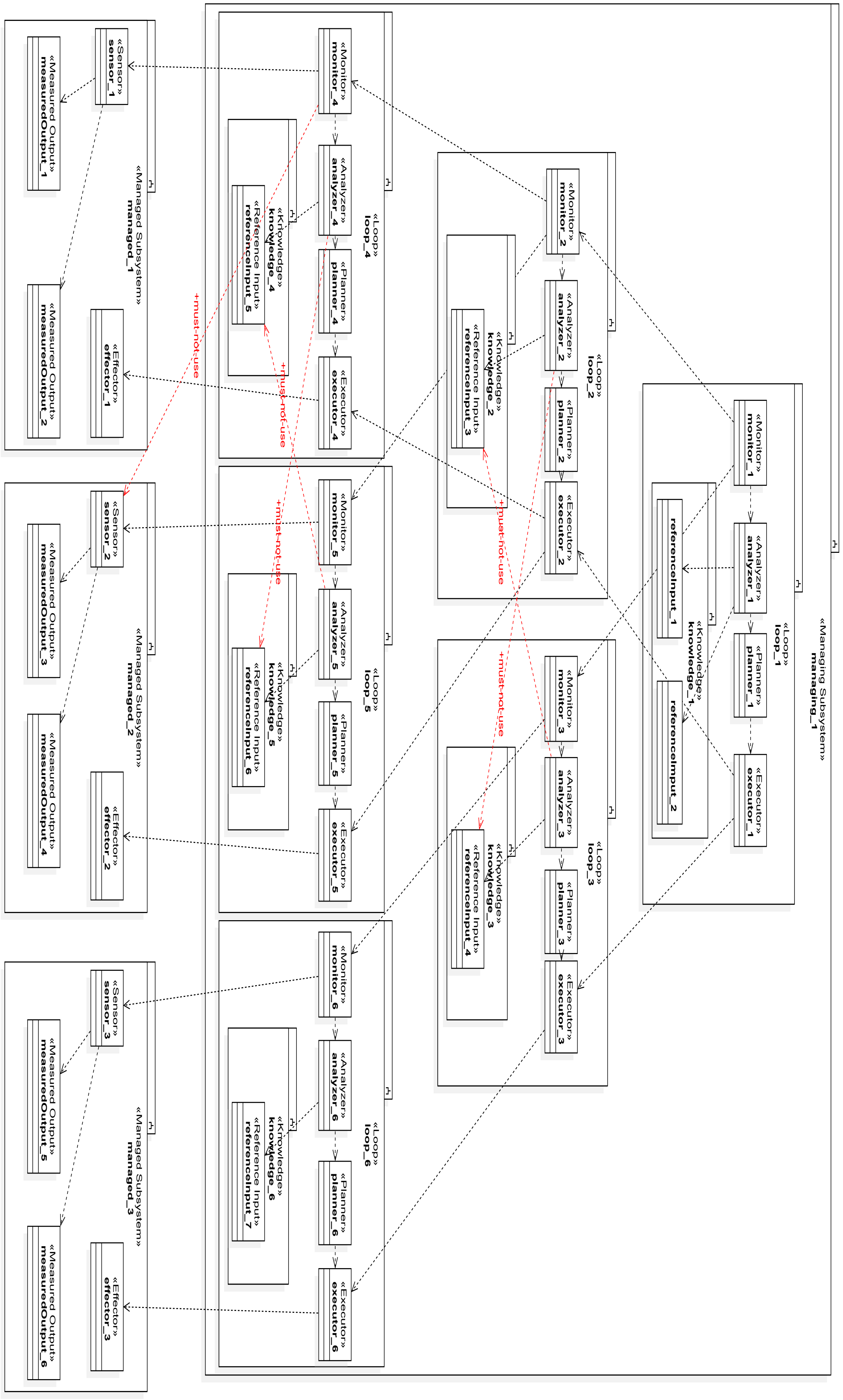


Figure B.8 – Architectural specification used in experiment

ANNEX A

Implementation Requirement of Adaptive Systems

Table A.1 – Requisitos de implementação de um sistema adaptativo

Code	Requirement
S-1	A Sensor element must collect measurements of variables of interest (from now on referred to as sensed data) (e.g., quality attributes specified in the series of standards ISO 25000 like performance of a service, availability of resources, topology information, configuration properties) in the context in which it is located, i.e. its execution context or the context of the domain to which it belongs.
S-2	A Sensor element must temporarily store sensed data. ***Rationale***. The Monitor element's responsiveness relies on the timely availability of the sensed data. This availability can be achieved by supporting temporary storage, which would allow Monitor elements to gather data at any moment. Nonetheless, Sensor elements can use up memory space assigned to the Managed Application, thus, other storage options should be taken into consideration.
S-3	A Sensor element must expose a subset of the sensed data to the set of Monitor elements, whether both Monitor and Sensor elements have been deployed jointly or independently.
S-4	A Sensor element must remove a subset of the sensed data being stored temporarily when instructed by a Monitor element.
S-5	A Sensor element must perform primitive operations (e.g., count repetitions of a measurement in a given time interval) on a subset of the sensed data. ***Rationale***. The ongoing transmission of sensed data from Sensor elements to Monitor elements can overuse network resources, thereby hindering the Managed Application's regular operation. Placing primitive operations in Sensor elements can considerably reduce the amount of data transmitted through the network when Monitor elements do not require the entire collection of sensed data but, instead, calculations over it.

Continua na página seguinte

Table A.1 – continua da página previa

Code	Requirement
M-1	A Monitor element must obtain the sensed data from one or more Sensor elements where it has been captured through the required access modes, i.e. by request (pull) or per occurrence (push).
M-2	A Monitor element must calculate metrics (based on sensed data) related to the variables of interest to characterize the current state of the Managed Application. Said calculation can be made periodically or whenever a new measurement happens, which would produce average or instant calculations, respectively. This calculation can also involve the composition or correlation of metrics calculated by other Monitor elements.
M-3	A Monitor element must make the calculated metrics available, through the Knowledge Manager element, to other Monitor elements so they can compose their own calculations.
M-4	A Monitor element must filter the calculated metrics before being reported to the Analyzer element. The filter must be done through the application of a set of domain-dependent monitoring rules over the calculated metrics.
M-5	A Monitor element must report to the Analyzer element control symptoms, i.e. the metrics (simple or compound) that meet the conditions set by the monitoring rules.
M-6	A Monitor element must allow changing the periodicity in which it calculates its metrics.
M-7	A Monitor element must allow to update the set of monitoring rules it applies to perform the filter of metrics. Such update may be triggered by, for example, a structural change of the Managed Application, or a change in the quality scenarios. ***Rationale***. Business and system's operation can make a variable of interest gain or lose relevance, thereby requiring flexibility against such behavior at runtime. Furthermore, providing elements with operations to control their internal behavior help support such flexibility.
A-1	An Analyzer element must evaluate reported control symptoms against reference values previously established (corrective behavior). Reference values must be recovered using the Knowledge Manager element. The evaluation should identify violations that occur with respect to these reference values. A violation indicates an adaptation symptom.
A-2	An Analyzer element must store a record of trends and violations through the Knowledge Manager.

Continua na página seguinte

Table A.1 – continua da página previa

Code	Requirement
A-3	An Analyzer element must reason about the reported control symptoms taking into account the historical records of trends and violations (re-covered using the Knowledge Manager element) to identify observable degradation trends with respect to the reference values (also recovered using the Knowledge Manager element) to avoid future violations (predictive behavior). The evaluation can employ time-series forecasting and queuing models. An observable degradation trend indicates an adaptation symptom.
A-4	An Analyzer must create and send one or more change requests to the Planner element if adaptation symptoms are detected. Such request must include which variable of interest is at risk of being (predictive behavior) or has already been (corrective behavior) violated, the variable's corresponding value, the motive for the request (e.g., violation, risk of violation), and the set of artifacts under the scope of the variable of interest (i.e. affected artifacts).
P-1	A Planner element must reason about the variable of interest, the degree of the violation, and the set of affected artifacts to identify a reachable, optimum resolution. For this reasoning, the Planner element must take into account the quality, quality configuration, and artifact applicability models. This information must be recovered using the Knowledge Manager.
P-2	A Planner element must perform a gap analysis to determine the necessary, high-level actions (e.g., deploy new artifacts, redeploy existing artifacts, replace existing artifacts with alternate ones, remove existing artifacts, update configuration setting) to reach the identified resolution.
P-3	A Planner element must create and send an action plan to the Executor element. Such action plan must include the set of high-level control actions determined with the gap analysis that will modify the Managed Application.
P-4	A Planner element must store a record of optimum resolutions and their corresponding action plans through the Knowledge Manager.
P-5	A Planner element must recover a previous action plan through the Knowledge Manager if the reachable optimum resolution identified matches to one of the action plans stored.
E-1	An Executor element must perform the realization of the action plan given by the Planner element through the scripting of executable commands (e.g., compile, deploy, redeploy) by the corresponding Effector elements.
E-2	An Executor element must use the corresponding Effector element to run commands over the Managed Application.

Continua na página seguinte

Table A.1 – continua da página previa

Code	**Requirement**
EF-1	An Effector element must allow managing a resource or set of re- sources (e.g., manage a middleware to deploy, redeploy, and undeploy components).
K-1	A Knowledge Manager element must perform create, retrieve, update, and delete operations over the repositories where the information of interest to the other elements of the autonomic infrastructure is stored.
K-2	A Knowledge Manager element must provide support operations for the analysis of the information managed by it.