UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

**Architectural Views for the Representation of Dynamic
Software Architectures**

João Paulo Biazotto

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências
de Computação e Matemática Computacional (PPG-C²MC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

João Paulo Biazotto

Architectural Views for the Representation of Dynamic Software Architectures

Master dissertation submitted to the Institute of
Mathematics and Computer Science – ICMC-USP,
in partial fulfillment of the requirements for the
degree of Master of Science – Computer Science and
Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Profa. Dra. Elisa Yumi Nakagawa

USP – São Carlos
October 2020

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

B579   Biazotto, João Paulo
            / João Paulo Biazotto; orientadora Elisa Yumi
            Nakagawa. -- São Carlos, 2020.
            87 p.

Dissertação (Mestrado - Programa de Pós-Graduação
em Ciências de Computação e Matemática
Computacional) -- Instituto de Ciências Matemáticas
e de Computação, Universidade de São Paulo, 2020.

1. Software Architecture. 2. Dynamic Software
Architecture. 3. Architectural Views. 4.
Architectural Representation. I. Nakagawa, Elisa
Yumi, orient. II. Título.

João Paulo Biazotto

Visões Arquiteturais para a representação de Arquiteturas de Software Dinâmicas

Dissertação apresentada ao Instituto de Ciências
Matemáticas e de Computação – ICMC-USP,
como parte dos requisitos para obtenção do título
de Mestra em Ciências – Ciências de Computação e
Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e
Matemática Computacional

Orientador: Profa. Dra. Elisa Yumi Nakagawa

USP – São Carlos
Outubro de 2020

To God. To my family. To University of São Paulo.

Acknowledgements

I would like to thank God for granting me life and health to go through the hard way until the conclusion of this master's degree. I also thank the Immaculate Conception of Aparecida, who, with all her love, prays for us in difficult times, and certainly has prayed for me during these years many and many times.

I thank my parents, João Batista Biazotto and Maria Margarida dos Santos Biazotto, for the support and dedication to me since the first years of education, always being by my side for encouraging me to keep looking for more and more knowledge. They have never gone out of their way to provide me with the best education possible, and I have been taking the rewards of their work until today. I am grateful to my sister, Luzia Fernanda Biazotto, who has always been my example of an academic and encouraged me to follow the path of teaching. To my girlfriend, Carla Eduarda Parma Duarte, who had all the patience in the world with me during those years. I am fully convinced the journey would have been much more difficult without her support.

I also thank my friends from São Carlos, who have always helped me and made the walk lighter and fun. In particular, I would like to thank João Choma Neto, Diego Moreira, Lucas Beraldinelli Nardini, Henrique Araujo Silva, and Luiz Antonio Lima Rodrigues, with whom I have shared most of my days over the past two years.

To my dear advisor, Prof. Dr. Elisa Yumi Nakagawa, who trusted my work and helped me with the development of this project; I highly acknowledge her affection and patience.

I also thank Lina Garcés, who greatly helped me elaborate this project, and the entire START Research Group and the other students of the Software Engineering Laboratory for the tips given.

Finally, I thank the Coordination for higher Education Staff Development (CAPES) and the National Council for Scientific and Technological Development (CNPq) for the financial support provided to this project.

*To one who has faith, no
explanation is necessary. To
one without faith, no
explanation is possible.*

St. Thomas Aquinas

Abstract

BLAZOTTO, J.P. **Architectural Views for the Representation of Dynamic Software Architectures**. 2020. 86 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

Presently, several critical application domains have adopted large, complex, and software-intensive systems, whose architectural design demand special attention. Such systems, occasionally display dynamic architectures and specific structures are required for supporting changes at run-time. Moreover, many of them require high levels of reliability due to their critical domains. Towards ensuring reliability, formal languages/notations have been used to represent dynamism in software architectures, which demand efforts and time to be learned and to model architectures; besides, the resulting models are of difficult understanding. Moreover, most of these representation are incomplete, lacking some relevant information. To mitigate this problem, this Master's project proposes a set of architectural views and associate semi-formal techniques that support architects in the design of software architectures that present dynamism. This Master's project proposes a set of architectural views and associated semi-formal techniques for supporting architects in the design of software architectures that show dynamism. From results of a systematic mapping study, a conceptual model was elaborated towards delimiting the main concepts to be covered by the views. A set of five views (general view, reconfiguration view, ripple-effect view, scenario view, and strategy view) was then defined and a proof of concept was conducted with the use of a system for the Healthcare Supportive Home Systems domain. The results show the feasibility and suitability of using the set in combination with the conceptual model and the associated techniques for representing dynamism in software architectures.

Keywords: Software Architecture; dynamic software architecture; architectural view; architectural representation.

Resumo

BLAZOTTO, J.P. **Visões Arquiteturais para a representação de Arquiteturas de Software Dinâmicas**. 2020. 86 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

Atualmente, diversos domínios de aplicação críticos vem adotando sistemas grandes, complexos e com uso intensivo de software que requerem atenção especial ao projeto de suas arquiteturas de software. Eventualmente, esses sistemas apresentam arquiteturas dinâmicas e estruturas específicas são necessárias para apoiar mudanças em tempo de execução. Além disso, muitos desses sistemas requerem alto nível de confiabilidade devido aos seus domínios críticos. Para garantir a confiabilidade desses sistemas, linguagens/notações formais vem sendo usadas para representar o dinamismo em arquiteturas de software. No entanto, essas linguagens/notações exigem esforço e tempo para serem aprendidas e usadas para modelar arquiteturas; além disso, os modelos resultantes são difíceis de compreender. Além disso, muitas vezes esses modelos são incompletos, deixando de apresentar informações importantes. Para mitigar este problema, este projeto de mestrado propõe um conjunto de visões arquiteturais e técnicas semi-formais associadas que auxiliam os arquitetos no projeto de arquiteturas de software que apresentem dinamismo. Para tanto, a partir dos resultados de um mapeamento sistemático, foi elaborado um modelo conceitual para delimitar os principais conceitos que deveriam ser contemplados pelas visões. A seguir, um conjunto de visões (visão geral, visão de reconfiguração, visão de efeito cascata, visão de cenário e visão de estratégia) foi definido e uma prova de conceito foi conduzida usando um sistema para o domínio de *Healthcare Supportive Home Systems*. Como resultado, observamos a viabilidade e adequação do uso do conjunto de visões arquiteturais aliadas ao uso do modelo conceitual e técnicas associadas, para representar o dinamismo em arquiteturas de software.

Palavras-chave: Arquitetura de software; Arquitetura de software dinâmica; visão arquitetural; representação arquitetural.

List of Figures

Figure 1 – Taxonomic Dimensions of Dynamic Software Architectures	7
Figure 2 – SMS Search String	11
Figure 3 – Approaches for the Representation of Dynamic Software Architectures	16
Figure 4 – Conceptual Model for Dynamic Software Architectures	28
Figure 5 – Representation Model	39
Figure 6 – General View Model	40
Figure 7 – Reconfiguration View Model	41
Figure 8 – Ripple-Effect View Model	42
Figure 9 – Scenario View Model	43
Figure 10 – Strategy View Model	45
Figure 11 – Concepts of General View	47
Figure 12 – Concepts of Reconfiguration View	48
Figure 13 – Concepts of Ripple-Effect View	49
Figure 14 – Concepts of Scenario View	50
Figure 15 – Concepts of Strategy View	51
Figure 16 – Diagram of Integration of HomecaARe	59
Figure 17 – Conceptual View of DiaManT@Home	60
Figure 18 – Conceptual model for DiaManT@Home	62
Figure 19 – Proof of Concept - Reconfiguration View	68
Figure 20 – Proof of Concept - Reconfiguration View (Result)	69
Figure 21 – Proof of Concept - Strategy View	70
Figure 22 – SMS Steps	85

List of Tables

Table 2 – Relation among Views	54
Table 3 – DiaManT@Home Adaptivity Scenario	58
Table 4 – Mapping of System’s Specification Concepts	63
Table 5 – Mapping of Dynamic Concepts	65
Table 6 – Mapping of MAPE-K Concepts	66
Table 7 – Mapping of Architectural Concepts	66
Table 8 – Databases used in SMS	84

1	Introduction	1
1.1	Context	1
1.2	Motivation and Problem	2
1.3	Objectives	3
1.4	Organization	4
2	Dynamic Software Architectures	5
2.1	Initial Considerations	5
2.2	Characterization of Dynamic Software Architecture	5
2.3	Problems of Reconfiguring Software Architectures	9
2.4	State-of-the-art of Representation of Dynamic Software Architectures	10
2.5	Representation of Dynamism in Reference Architectures	20
2.6	Final Remarks	22
3	A Conceptual Model for Dynamic Software Architectures	25
3.1	Initial Considerations	25
3.2	Methods	25
3.3	Model	27
3.4	Guidelines and Model Usage	34
3.5	Threats to Validity	35
3.6	Final Remarks	35
4	Architectural Views for the representation of Dynamic Software Architectures	37
4.1	Initial Considerations	37
4.2	General View	40
4.3	Reconfiguration View	40
4.4	Ripple-Effect View	41
4.5	Scenario View	42
4.6	Strategy View	44
4.7	Discussion	45
4.8	Final Remarks	53
5	Illustrative Example of the Use of Views	55
5.1	Initial Considerations	55
5.2	Methods	56
5.3	Characterization of DiaManT@Home	56
5.4	Description of DiaManT@Home	65

5.5	Final Remarks	71
6	Conclusions	73
6.1	Solutions Proposed	74
6.2	Research Contributions	74
6.3	Research Limitations	75
6.4	Future Work	75
	References	77
	APPENDIX A Dynamic software architectures representation: a systematic mapping study protocol	83
A.1	Planning	83
A.2	Conduction	85
A.3	Threats to validity	86

Introduction

1.1 Context

Software architectures have played a significant role in the current society, since they have been considered one of the main artifacts, or the backbone for improvements in the quality of software systems in several application domains, including critical ones, such as telecommunication, transportation, robotics, health, and smart grids (Clements; Bass, 2010). Software architecture can be understood as “the set of structures needed to reason about the system, comprising software elements, relations among them, and properties of both” (BASS, 2013). Such architectures play a fundamental role in determining the quality of software systems, and are constructed towards achieving organization’s business goals (BASS, 2013). They are a bridge between those (often abstract) business goals and the final (concrete) resulting system (GARLAN, 2000).

Over the past two decades, software systems, as well as their architectures, have rapidly evolved and become more complex. Highly available public information systems and safety critical systems are examples of software systems of high level of development complexity (FU *et al.*, 2006). Dynamic software architectures modify their architecture during the running of a system (ALLEN *et al.*, 1998), an action most commonly known as run-time evolution or reconfiguration. A software system that reacts to certain events at run-time by supporting reconfiguration is modeled into a dynamic software architecture (WEYNS *et al.*, 2015). Typical reconfiguration operations for dynamic software architectures include addition, removal or update of components or connectors (WERMELINGER; FIADDEIRO, 1999). Changes in an architecture topology through the addition or removal of connections between components and connectors are also considered a type of reconfiguration. Due to the advantage of the continuous availability of systems promoted by their dynamic architecture, this research subject has become a hot issue in the software engineering field (DING, 2016).

The design of dynamic software architectures must enable stakeholders to understand changes that can occur at run-time and their impact on the system (DING, 2016). Moreover, representations of software architectures are used by parties that create, utilize, and manage modern systems for improving communication and co-operation, so that they work in an integrated and coherent way. Finally, they can reduce rework and be used as receptacles for decisions made by architects, thus facilitating future maintenance (CLEMENTS *et al.*, 2010). Therefore, architectural representations must promote the understanding of work assignments by developers, imposition of architecture in the source code by testers, and scheduling implications by managers (BASS, 2013), among other requirements for different stakeholders.

In dynamic software architectures, models are mostly represented by formal models, techniques, and architecture-description languages (ADL), such as those based on graphs, logic, and pi-Calculus (BRADBURY *et al.*, 2004), since the application domains of insertion of the architectures are sometimes critical.

In this scenario, reliability emerges as one of the main quality attributes of those architectures, and simulations and formal validations enable its more precise and secure achievement (CAVALCANTE *et al.*, 2015). A fully representation of the architecture that covers all relevant information is fundamental for the understanding of the way it should be reconfigured; therefore, representation completeness is another important subject in dynamic software architectures.

1.2 Motivation and Problem

Many of the current software systems require mechanisms that enable changes at run-time and, concomitantly, their architectures must be prepared towards predicting the way such changes will be made and measuring their impact on the whole systems (LI *et al.*, 2012).

The literature reports evidence of the complexity of using formal techniques and languages to represent dynamic software architectures (HAN *et al.*, 2005; KALLEL *et al.*, 2012; SMAALI *et al.*, 2014; OQUENDO, 2016) caused mainly by (KACEM *et al.*, 2006): (i) their difficult learning and use - architects unfamiliar with them must devote considerable efforts and time on their use, and (ii) their uncommon use (e.g., in projects that involve static architecture). Therefore, communication among stakeholders is hampered, since most of them do not know those languages and techniques; besides, formal techniques and languages are rarely understood by those who are not part of the architecture design process.

One of the ways to mitigate the problem of using formal techniques and language to represent dynamic software architecture is to combine semi-formal techniques/languages (such as UML (MILADI *et al.*, 2008)) with formal ones (e.g., graphs (KACEM *et al.*, 2006))

or Petri-nets (HUSSEIN *et al.*, 2017)), thus reducing problems of communication among stakeholders, but not decreasing the efforts and time required. (OK?) Semi-formal representation might be a way for reducing them and maintaining the same levels of reliability (HUSSEIN *et al.*, 2017). The systematic mapping study (SMS) conducted revealed a lack of guidelines that address concerns to be considered in dynamic software architectures.

Among the several unsolved problems related to the representation of dynamism in software architectures are:

- (i) in essence, dynamic software architectures are abstract and complex to understand (HASSAN *et al.*, 2016);
- (ii) many concerns must be considered, e.g., the components that must be stopped or replaced;
- (iii) the use of formal notations ensures the reliability of software architectures, but increases costs; and
- (iv) if no directions on "what" to be represented, architects can forget some relevant concepts, thus decreasing quality.

Considering these problems, specifically for the context of this project, **the main problem addressed in this project is the lack of resources that might help architects in the analysis and representation of dynamic software architectures.**

1.3 Objectives

This project proposes a set of architectural views and associated semi-formal techniques for supporting architects in the design of software architectures whose structures show dynamism. An SMS conducted provided an overview of the state-of-the-art of the representation of dynamic software architectures.

The definition of the set was based on a conceptual model for dynamic software architectures proposed in the project. The model guided the development of the views, and enabled the necessary concepts to be addressed in dynamic software architectures. To evaluate the architectural views and associate semi-formal techniques, we conducted a proof of concept to observe their viability.

The project aims at contributing to the state-of-the-art of software architectures, since the views can be used by practitioners for establishing new (or improving) software architectures for contemporary software systems, such as big data, IoT, smart cities, Systems-of-Systems SoS, and self-* systems. It can also collaborate with ICMC/USP and Fraunhofer Institute of Experimental Software Engineering (Fraunhofer IESE), Germany,

which have contributed to architect industrial software-intensive systems towards meeting quality standards developing joint research projects with industrial partners and other research institutes, thus combining academic know-how with industrial practice.

This Master's project has also contributed to a larger research project entitled "RASoS: Construction of Reference Architecture for Systems-of-Systems", for the building of architectures for Systems-of-Systems (SoS) including in different domains: (i) Flood Monitoring, to create alerts and help stakeholders to prevent from disasters; (ii) robotics systems; (iii) ambient assisted living; and (iv) eHealth, aiming at improving the interoperability of systems present in hospitals, ambulances, among other systems.

1.4 Organization

This chapter introduces the context in which the research is inserted, the problem and motivations, and the main objective. The remainder of the dissertation is organized as follows:

Chapter 2 provides the background information that supports the topics investigated. Basic concepts related to dynamic software architectures are discussed, and the results of an SMS are provided, describing the approaches that can represent dynamic software architectures.

Chapter 3 presents a conceptual model for dynamic software architectures that shows the main concepts to be considered in a dynamic software architecture representation. Each step of the method used for the creation of the model is detailed.

Chapter 4 describes the set of architecture views that represent dynamic software architectures and whose construction considered all concepts contained in the conceptual model.

Chapter 5 addresses the evaluation of the architecture views, which consists of a proof of concept that considered a dynamic software architecture inserted in an eHealth domain. The improvements perceived with the use of the views were analyzed.

Finally, Chapter 6 concludes the project, revisiting the contributions made, summarizing limitations, and suggesting future research.

Dynamic Software Architectures

2.1 Initial Considerations

In the last two decades, software systems have rapidly evolved and become more complex, as well as their architectures. Public information systems with high availability and mission, and safety critical systems are examples of software with a high level of complexity in their development (FU *et al.*, 2006). Furthermore, modern software systems must be continuously available, which blurs the traditional separation between run-time and development time (WEYNS *et al.*, 2015).

Software architectures have been widely used to facilitate the development software systems. Through them, it is possible to analyze software systems under different stakeholders perspectives, allowing to evaluate impacts of taken decisions by architects (BASS, 2013). However, traditional (or static) software architectures are not able to capture all characteristics that complex and dynamic software systems, such as SoS have (ALLEN *et al.*, 1998). Thus, mechanisms to measure architecture dynamism are required.

Dynamic software architectures are those that modify their structure during the system's execution (ALLEN *et al.*, 1998). Due to the advantage of continuous availability, the research on dynamic software architectures has become a hot issue in software engineering field (XU *et al.*, 2010).

2.2 Characterization of Dynamic Software Architecture

Software design methodologies are not already adapted to the development and design demand of new software systems, which are becoming more complex and large-scale. Hence, the selection of the specific algorithms and structures are not more important than overall system structure design and specifications. Furthermore, software architecture is a crucial

success factor for system design and development (ZHENHUA; YUANLI, 2004). Thus, software architecture area has also investigated ways to support these new requirements, specially dynamism, arising the term dynamic software architecture, which represents structures capable of reconfiguring at run-time (TORKAR, 2002). This constitutes a research field for the study of dynamic software architectures. Reconfiguration of architectures by maintaining their conformity is the main objective of these software architectures (LOULOU *et al.*, 2004).

Interaction behavior can also be captured and specified in dynamic software architecture, as well as in the architecture structure by designing guidelines for the architecture level. Therefore, dynamic evolution can be implemented in a dynamic software architecture during the system execution Li *et al.* (2012). These architectures are usually implemented with two types of change: change in the structural elements (e.g., components or connectors) or change in the topology and interaction behavior (LI *et al.*, 2012). In this aspect, three levels of dynamism can be considered in the architectures according to Cîmpan *et al.* (2005): (i) The first-level or *interactive dynamism* requires the dynamic communication of data in a fixed structure; (ii) The second level or *structural dynamism* allows the modification of this structure, usually expressed as the creation and removal of component instances and links; and (iii) The third level allows the modification of the infrastructure of the structures, i.e., the dynamic (re)definition of new component types, called *architectural dynamism*.

2.2.1 Taxonomic Dimensions of Reconfiguration Process

Relevant characterizations of dynamic reconfiguration on software architectures were provided by Wermelinger and Fiadeiro (1999) and Bradbury (2004). In this sense, Cavalcante (2016) summarizes a taxonomy, presented in Figure 1, extracted from the work of that author.

Source (or initiation). Dynamic reconfiguration of software architectures can be either programmed or ad-hoc. In programmed (a.k.a. foreseen or proactive) reconfigurations, changes are known at design time and applied at run-time under a given condition or event (ENDLER; WEI, 1992). In this sense, during the design of the software architecture, the architect can predict and measure the changes, defining the constraints and requirements to perform change as well as the operations that must be done. On the other hand, ad-hoc (a.k.a. unforeseen or reactive) reconfiguration, the changes that will be performed at run-time are not known at design time, hence, the changes can not be evaluated at design time (CAVALCANTE, 2016). In most cases, the changes come from outside of system (from environment, users, among others) and they are applied through an interface. It is noteworthy that most of the works about dynamic software architectures addresses programmed reconfiguration (BRADBURY *et al.*, 2004).

Figure 1 – Taxonomic Dimensions of Dynamic Software Architectures

Source: Cavalcante (2016).

Autonomically reconfigurations can be applied in architectures with programmed reconfigurations in response to certain conditions. Apply changes/updates without necessarily foreseeing them in advance can be an advantage that ad-hoc reconfigurations can allow. Anyway, both programmed and ad-hoc reconfigurations have their advantages and drawbacks, making them complementary approaches that shall be supported by architectural approaches (CAVALCANTE, 2016). However, ad-hoc reconfigurations must be somewhat constrained, carefully applied to avoid architectural erosion.

Operations. Different nomenclatures are adopted in the literature to define reconfiguration operations. However, four operations can be highlighted (ALLEN *et al.*, 1998; ENDLER; WEI, 1992; CAVALCANTE, 2016) to be applied on the architectural elements: (i) creation of architectural elements; (ii) removal of architectural elements; (iii) attachment of architectural elements; and (iv) detachment of architectural elements.

Management. According to Cavalcante (2016), the management of the reconfiguration process can be either centralized with a special entity or distributed across architectural elements. The two typers are:

- In **exogenous dynamism** a central entity (e.g., a configuration manager) controls all architectural elements, applying reconfiguration actions on the architecture; and
- **Endogenous dynamism**, in it turn, decentralizes the reconfiguration process, i.e., each element are able to reconfigure itself.

In exogenous dynamism, the centralization of the reconfiguration process can reduce the performance of the architecture at run-time due to bottleneck at the implementation level, what characterizes its main disadvantage. Moreover, different architectural

elements can require a specific set of operations which must be described and performed independently from each other (Costa-Soria *et al.*, 2009). On the other hand, maintainability and reuse can be decreased in the endogenous approach due to reconfiguration actions may be tangled with components's behavior/functionality. Furthermore, the components has to hace knowledge about other components, what could compromises basic features of a component or a connector (CAVALCANTE, 2016). Nevertheless, trade-off analyses between these dynamism approaches, considering each specific scenario and architecture, is necessary to establish some sort of according about their usage.

2.2.2 Types of Reconfiguration

There are different forms to perform run-time changes in architectural level. Bradbury (2004) summarized 11 types of possible reconfiguration as described below:

- **Programmed dynamism:** the system itself triggers the reconfiguration, and possible changes are previously known;
- **Ad-hoc dynamism:** the change is made often during a maintenance task and they are not known at design-time;
- **Constructible dynamism:** Constructible dynamism is a kind of ad-hoc change, because it is initiated by an external event. Three types of tools are used to represent this architecture: one of them shows the initial state of the architecture, other represents the modifications and, finally, a middleware supports the architectural change;
- **Adaptive dynamism:** it consists in a type of a programmed change. This type of dynamism has a predefined set of configurations and events that can reconfigure the architecture. When one of these events happens, the system selects the run-time changes for that event;
- **Intelligent dynamism:** intelligent dynamism is also a kind of programmed change, similar to adaptive dynamism, but without restricting of a predefined set of configurations. In intelligent dynamism, possible configurations are dynamically constructed and the architecture assesses the quality of the configurations and implements an appropriate choice;
- **Constrained run-time dynamism:** in this type of dynamism, a change can occur only if a constraint is satisfied;
- **Unconstrained run-time dynamism:** No constraints are placed on the program state or in the topology prior to running the system;

- **Transient connectors:** this type of change is made by a boolean condition in the connectors, defining its role and when it should be applied. Addition or removal of connectors is based on a condition;
- **Self-organising architectures:** this type of architecture allows components connect or remove from the architecture, without a centralized management;
- **Self-repairing systems:** it is similar to self-organising architectures, but the difference between them is that self-repairing systems do not need distributed management. A self-repairing system has a control-loop of three primary activities: monitoring, interpretation, and reconfiguration. In a first moment, run-time behavior is monitored to determine the need for change. Second, behavior is interpreted and the change is identified. Finally, the reconfiguration is conducted; and
- **Self-adaptive software:** Systems that can adapt to their environments by enacting run-time changes. These systems monitor observable behavior that can be external or internal to the systems and, in response, they enact change at architectural level.

2.3 Problems of Reconfiguring Software Architectures

To efficiently handle the dynamic evolution process of software architectures some issues should be addressed by any evolution approach. These issues are aware of the mechanism and evolution model that is used to perform the dynamic architecture evolution ([OREIZY, 1998](#)).

The first issue is when stop the component to perform the reconfiguration. Evolving an artifact at run-time without considering its thread may disrupt or suspend its service for an arbitrarily long time, which can lead real-time tasks to miss some deadlines ([HASSAN *et al.*, 2016](#)). Hence, one of the main issues that must be considered when handling dynamic evolution is to leave systems in a consistent state after a change is performed. A crucial key to guarantee the system's consistency state is detect when it is safe to actually evolve the artifacts. [Hassan *et al.* \(2016\)](#) affirms that are two states for the elements in an architectures. The passive elements are those that do not have any execution, but typically provide services for other elements. On the contrary, the active elements are those that have active execution. This means that the execution in the element would need to be suspended, thus, potentially resulting in deadline misses for the execution of the element being under evolution. Therefore, the evolution operation should respect the some constraints of the active element that is subject to change ([HASSAN *et al.*, 2016](#)).

According to [Oreizy \(1998\)](#), the consistency between the representation and the implementation of dynamic software architectures is also an issue. Run-time changes to

the system are described in terms of its architectural model, what can origin problems like: a modeling language may assume that run-time changes are instantaneous, when in fact the implementation does not support instantaneous changes. Whether or not particular differences between the model and implementation are significant depends on the type of analysis being performed. In any case, the correspondence between the two must be ensured. In the same perspective, components that may have internal information, or connectors that may have buffers full of messages (OREIZY *et al.*, 2008) characterize another crucial issue of system consistency and should be considered when addressing a dynamic evolution is that of handling stateful elements. According to Hassan *et al.* (2016), in the case of replacing elements, information integrity requires that the state of the old element must be transferred, or possibly transformed (in the case the data structure is different), to the new element. Practically, it is difficult to develop a generic abstract that can fit the internal data structure for all the system elements, to store the state of any element during its evolution (HASSAN *et al.*, 2016).

How evolution events can be detected is another relevant issue in dynamic evolution as well as how the changes are driven, and how the suitable reactions can be effected (HASSAN *et al.*, 2016). More accurately, it relates to how the evolution process can be managed. Generally, the software system can be reactive (changes are driven externally), or proactive (drives changes to itself) (BUCKLEY *et al.*, 2005). In dynamic evolution, management can be represented as an evoluter (Role) who is responsible for dynamically performing the evolution. Thus, either the system is instrumented with change management, or with an interface to allow an external agent to dynamically introduce the changes. This can be formulated as a controlling system that monitors a controlled system to detect and analyze an evolution event when it occurs on the controlled system or its environment to select or synthesize the appropriate action or scenario of evolution.

2.4 State-of-the-art of Representation of Dynamic Software Architectures

Designing dynamic software architectures clearly must allow all stakeholders understand changes that can occur at run-time, and their impact on the system (DING, 2016). Hence, the necessity for clear and complete representations is increased due to its abstraction level. It is necessary to identify changes and reasons of changes and, if the architecture was not documented, the system could be not implemented in a right way, what compromises its quality and do not achieve its goals (ALLEN *et al.*, 1998). To identify and understand initiatives to help the representation of dynamic software architectures, we performed a Systematic Mapping Study (SMS). An SMS is a research technique proposed by the Evidence-Based Software Engineering (EBSE) that allows to summarize the evi-

dences¹ about a given subject or technology (PETERSEN *et al.*, 2015; KITCHENHAM B. A.; CHARTERS, 2007). This technique uses a research protocol defined in the beginning of the research process that ensures impartiality in the investigation. These guidelines are divided in three steps: (i) planning: in which the research objectives and the systematic mapping protocol are defined; (ii) conduction: primary studies are identified, selected, and evaluated according to the inclusion and exclusion criteria previously established in the protocol; and (iii) synthesis: information extracted from the studies are analyzed. The following research questions (RQs) were established:

- **RQ1:** Which approaches have been used to represent dynamic software architectures?
- **RQ2:** What the level of formalism of these approaches are?
- **RQ3:** How can the representation approaches solve dynamic software architectures issues?

For the accomplishment of this work, five databases were selected (Scopus, IEEE Xplore, ACM Digital Library, Web of Science, and ScienceDirect), which we consider to be the most relevant for the secondary study. The search string used to perform the research are shown in Figure 2. The research returned 7595 studies. Parsif.al² was used for the organization of studies. After the removal of duplicate studies, 5076 studies remained. After reading titles, abstracts and keywords, 50 studies, were selected for the complete reading and will be used as a basis for the development of this work.

```
( "software architecture" OR "reference architecture" )
AND ( "dynamic" OR "dynamism" OR "reconfigura*"
OR "adaptive" ) AND ( "model*" OR "design" OR
"notation" OR "represent*" OR "language" )
```

Figure 2 – SMS Search String

¹ Evidence is any result obtained in a scientific work and published in a publication vehicle – such like periodicals, technical reports, and other – that be relevant to the research topic.

² <http://parsif.al>

Table 1 – Selected Studies

Reference	ID	Title	Year
S01		Describing software architecture styles using graph grammars	(??)
S02		Specifying and analyzing dynamic software architectures	(ALLEN <i>et al.</i> , 1998)
S03		A simple description language for dynamic architectures	(WERMELINGER, 1998)
S04		An architectural application framework for evolving distributed systems	(??)
S05		Algebraic software architecture reconfiguration	(WERMELINGER; FIADDEIRO, 1999)
S06		Specification and refinement of dynamic software architectures	(??)
S07		Specifying and Verifying Reconfigurable Software Architectures	(??)
S08		A graph transformation approach to software architecture reconfiguration	(WERMELINGER; FIADDEIRO, 2002)
S09		Two Graph-Based Techniques for Software Architecture Reconfiguration	(HIRSCH; MONTANARI, 2002)
S10		A Scalable Approach to the Design of SW Architectures with Dynamically Created/Destroyed Components	(??)
S11		A Graph Based Architectural (Re)configuration Language	(??)
S12		Style-Based Refinement of Dynamic Software Architectures	(??)
S13		Automated adaptations to dynamic software architectures by using autonomous agents	(??)
S14		Towards a unified graph-based framework for dynamic component-based architectures description in Z	(LOULOU <i>et al.</i> , 2004)
S15		Novel Architecture Description Language Based on High-Level Petri Nets	(ZHENHUA; YUANLI, 2004)
S16		Structure analysis for dynamic software architecture	(HAN <i>et al.</i> , 2005)
S17		Using UML2.0 and GG for describing the dynamic of software architectures	(KACEM <i>et al.</i> , 2005)

S18	Handling Dynamic Behaviour in Software Architectures	2005	(CÎMPAN <i>et al.</i> , 2005)
S19	Modeling Dynamic Architectures Using Nets-Within-Nets	2005	(??)
S20	An “abstract process” approach to algebraic dynamic architecture description	2005	(??)
S21	Modeling Dynamic Software Architecture Based on Pi-Net	2006	(FU <i>et al.</i> , 2006)
S22	Temporal superimposition of aspects for dynamic software architecture	2006	(??)
S23	Describing dynamic software architectures using an extended UML model	2006	(KACEM <i>et al.</i> , 2006)
S24	Modelling Dynamic Software Architectures using Typed Graph Grammars	2008	(BRUNI <i>et al.</i> , 2008)
S25	Dynamic Software Architectures: Formally Modelling Structure and Behaviour with Pi-ADL	2008	(F. Oquendo, 2008)
S26	Web Service Interface Extending and description based on pi-calculus	2008	(??)
S27	Graph-Based Design and Analysis of Dynamic Software Architectures	2008	(??)
S28	A UML rule-based approach for describing and checking dynamic software architectures	2008	(MILADI <i>et al.</i> , 2008)
S29	Towards a Formal Model for Reconfigurable Software Architectures by Bigraphs	2008	(??)
S30	A Graph Transformation based Approach for Runtime Constrained Evolution of Service-Oriented Architectures	2009	(ZHAO <i>et al.</i> , 2009)
S31	Dynamic Software Architecture Development: Towards an Automated Process	2009	(BEEK <i>et al.</i> , 2009)
S32	A Method for Modeling Aspect-Oriented Dynamic Software Architecture	2010	(??)
S33	Description and Verification of Dynamic Software Architectures for Distributed Systems	2010	(XU <i>et al.</i> , 2010)
S34	Dynamic Software Architectures: Formal Specification and Verification with CSP	2012	(LI <i>et al.</i> , 2012)
S35	Specifying Dynamic Software Architectures with Dynamic Description Logic	2012	(ZHAO; LI, 2012)

S36	Modeling and enforcing invariants of dynamic software architectures	2012	(KALLEL et al., 2012)
S37	Modeling and verifying composite dynamic evolution of software architectures using hypergraph grammars	2013	(??)
S38	BGG: A Graph Grammar Approach for Software Architecture Verification and Reconfiguration	2013	(??)
S39	A Categorical Approach for Modeling and Verifying Dynamic Software Architecture	2013	(LING, 2013)
S40	On Specifying Reconfigurable Component-Based Systems Using Strategies	2014	(DJOUDI et al., 2014)
S41	Architecture Reconfiguration in Pervasive Environment: an AADL-based Approach	2014	(CHEN et al., 2014)
S42	K-Maude Definition of Dynamic Software Architecture	2014	(SMAALI et al., 2014)
S43	An Architecture Description Language for Dynamic Service-Oriented Product Lines	2015	(??)
S44	Approach for integrated modular avionics reconfiguration modelling and reliability analysis based on AADL	2016	(ZHANG et al., 2016)
S45	Formally describing the software architecture of Systems-of-Systems with SosADL	2016	(OQUENDO, 2016)
S46	An Approach for Modeling and Analyzing Dynamic Software Architectures	2016	(DING, 2016)
S47	Safe adaptation of vehicle software systems	2017	(HUSSEIN et al., 2017)
S48	Ensuring Consistent Dynamic Adaptation: an Approach from Design to Runtime	2017	(HUYNH et al., 2016)
S49	Reconfigurable Service-Based Architecture Based on Variability Description	2017	(ADJOYAN; SERIAI, 2017)
S50	Formal Specification and Verification of Dynamic Parametrized Architectures	2017	(CIMATTI et al., 2018)

To obtain the data, we extracted, from primary studies, information about the applied/proposed approaches, such as: name, theoretical basis (i.e., in which formalism/technique the language is based), limitations, and examples of use. With this data, we classified the approaches into five categories (as shown in [Figure 3](#)), considering its theoretical basis.

2.4.1 Approaches to Represent Dynamic Software Architectures

In this section we summarize the results of our SMS to answer the follow research question: **”RQ1 - Which are the approaches to represent dynamic software architecture?”**. Thus, each approach category for the representation of dynamic structures are described. We have identified five categories of approaches: graphs, modeling languages, process algebra, programming languages, and multi-formalism, which are explained in this section, and summarized in [Figure 3](#).

Several works have proposed approaches that allow the representation of decisions in dynamic architectures to enhance their understood and evaluation. [Figure 3](#) presents approaches identified in this work and are organized as follows: the vertical axis shows the year in which studies were published; the horizontal axis presents categories of each one of approaches; finally, bubbles represent the number of studies framed in a particular category in a given year.

The use of *graphs* has being proposed since the beginning of the studies on representation of dynamic architectures. This approach can be used through circles (vertices) and lines (edges) which allows a better understanding of the models. In addition, graphs have mathematical formalism which reduce ambiguities, allowing the automatic validation of these models ([ZHAO et al., 2009](#)). During the use of these approaches, each node represents an architecture state, and each edge represents rules for the states change, that is, the rules of reconfiguration ([HIRSCH; MONTANARI, 2002](#)). Through this language it is possible to represent all three types of reconfiguration, since the rules that define each one of the state changes present no type of restriction.

Another approach that was used is *petri-nets*, another type of graphs. Petri-nets is a formal method with graphical and mathematical notations are noted for their advantages in specifying and analyzing concurrent systems and are promising for the representation of dynamic architectures ([DING, 2016](#)). Petri-nets can describe the communication between processes (in this case components), modeling their behavior and consequently the dynamism. In this model, transitions between states that components can assume are presented as mathematical formulas, which guarantees semantic coherence to the models. During analysis of studies that present this model of formalism, no restriction was identified, since rules defined b formulas are variable, and can represent all types of dynamism.

Figure 3 – Approaches for the Representation of Dynamic Software Architectures

Modeling languages have been widely used to represent software architectures, including dynamic architectures. Architectural Description Languages (ADLs) are languages proposed specifically to represent software architectures, which differentiate them from other languages that are applied for this purpose but not proposed for it (such as graphs and UML). Overall, 10 studies used ADLs, among which the ADLs applied are: Dynamic Wright ([ALLEN et al., 1998](#)), SosADL ([OQUENDO, 2016](#)) and AADL ([CHEN et al., 2014](#); [ZHANG et al., 2016](#)). One of the main advantages in the use of ADL's is the possibility of using tools for automatic validation and verification, assessing requirements

such as performance and availability. In a similar way to the use of graphs, no restrictions on the type of dynamism were identified, however, it should be considered that most of these ADL's were proposed for the context of dynamic arrays, which caused the authors to have these concerns during the development of them.

UML was also used to represent dynamic architectures. In the study, the author extends the UML2.0 meta-model and offered a UML profile to model the evolution of software architecture (MILADI *et al.*, 2008). The proposal presents three aspects: the static first that defines the structures of the architecture, the second encompasses the dynamic aspect, which is related to the process of addition or removal of components, finally, the third uses the activities diagram for the coordination of the reconfiguration. The authors considered expanding the client (tool) that is used in UML modeling so that it is possible also the automatic validation of these models. Thus, the tool translates the diagrams into language Z and so it is possible to perform the simulation of the proposed architecture. Through the use of these models it is possible to represent any of the three types of dynamism, considering the three steps defined by the author.

Process-algebra are formal languages that present the use of process-algebra-based formalism. This language is widely used for representation of concurrent programs and allows the communication of objects through channels, being possible to model aspects of synchronization and interaction between them. Approaches based on pi-calculus are Turing-complete, i.e., any type of computation can be expressed through them. These approaches are able to represent both structural and behavioral aspects, which encompass dynamic characteristics. In addition, languages such as pi-ADL (OQUENDO, 2004) support automatic checking through the use of simulators and tools for having algebraic laws derived from process algebra. In the same way as the others, the use of this language does not present any restriction as to the type of dynamism, since the rules are variable and can be defined in the necessary way.

Logic-based approaches, in its turn, are a kind of process-algebra. In these languages, "parameters representing the infinite-state space of possible configurations, and first-order formulas represent the sets of initial configurations and reconfiguration transitions." (CIMATTI *et al.*, 2018). Because they are languages based on mathematical formalism, its semantics is well defined, which entails the absence of ambiguities. In addition, the number of components and configurations are unlimited, which is an advantage of using this type of language. In systems with self-managed dynamism, for example, the number of components that can exist in the architecture is undefined, therefore, that language presents good performance to represent this type of system. For the other types of dynamism, no type of constraint has been identified.

Programming languages also support the representation of dynamic architectures. TOM language was applied in (DJOUDI *et al.*, 2014) because it is a formalism that

allows both representation of structures and dynamism (add or removal of components). In addition, because it is a programming language has a well defined syntax, which leads to a reduction of errors. Another important point of the use of this type of language is the possibility of automatic verification through tools. It is even possible that the reengineering process is done, so, from the representation it would be possible to generate graphic representations of the structures, which may help in understanding the problems. About supported types of dynamism, the example presented in (DJOUADI *et al.*, 2014) is not explicit if there is any limitation.

Other programming language used was *Chemical Abstract Machine (CHAM)*. CHAM "is a general-purpose, simple, intuitive and operational programming model that has been used for the specification and analysis of the computational behaviour of software architectures" (WERMELINGER, 1998). This formalism is inspired by the chemical reactions that occur between elements, so components are defined as "molecules" that make up the architecture and "reactions" are the interactions between the components. Architectural style and reconfiguration are specified by two different CHAMs, the creation CHAM and the evolution CHAM (WERMELINGER, 1998). The work that presents the use of this formalism presents an example of a client server system, which presents a programmed dynamism, so it is not explicit if this can be applied to any of the dynamism types.

Applying more than one language to the representation of dynamic architectures had proven to be a good solution for the balance between formalism and graphic representation. From the researched studies, seven of them present *multiformalism* by combining languages like UML and EAST-ADL (HUSSEIN *et al.*, 2017), making it possible to visually model the architecture components (what is considered simpler) in addition to enable the automatic validation of models through formal languages. In addition, there is evidence of the possibility of using such approaches in the industry, for example, the approach is applied to the automotive domain as part of the evaluation. Through these models it is possible to represent any of the dynamism types, since the use of ADL's do not restrict the rules of reconfiguration.

2.4.2 Level of Formalism of the Approaches to Represent Dynamic Software Architectures

This section aims to answer our second question: "**RQ2 - Which the level of formalism of these approaches are?**". In this section we will analyze how the formalism level impacts representation activity, this is, what the use of formal (or not formal) languages allows to architects.

Most of the approaches presented in this secondary study (49 studies) use formal languages (such as Dynamic Wright, pi-ADL, ACME, Fractal, among other) to represent

dynamic software architectures. We observed that the highly number of formal approaches for representing dynamic architectures are due the following factors: (i) dynamic software architectures had their inspiration in distributed systems architectures (MAGEE *et al.*, 1995) and these architectures were represented by formal models process-algebras, so it is natural that most authors tend to continue using these approaches, adapting it to the context is being applied. Dynamic Wright (ALLEN *et al.*, 1998), for example, was first proposed to lead with dynamic structures in concurrent and distributed systems and are used in dynamic software architectures; and (ii) the use of formal languages makes possible the automatic validation of the described architectures, and the main validated points are related to availability and reliability (ALLEN *et al.*, 1998; HIRSCH; MONTANARI, 2002; OQUENDO, 2016; CIMATTI *et al.*, 2018; WERMELINGER; FIADEIRO, 2002);

Another important aspect related to formal representations is the possibility of architecture simulation, a very important factor while several architectural decisions can only be evaluated at run-time (F. Oquendo, 2008). Allen *et al.* (1998) give an example of reconfiguration that we will consider to explain simulation advantages: consider a client-server system that receives requests for e-mail firing, where the router must be reconfigured in a way that balances the load between two servers, that is, if the demand exceeds the capacity of one server, another server must be inserted in the architecture, and, similarly, when the demand decreases, the server must be removed from the architecture. In this situation, simulating the reconfiguration process allows architects to evaluate the ripple-effect of the change as well as other consistency features that are essential for a dynamic system. Semi-formal models (such as UML) are unable to simulate such behaviors. However, learning these languages takes time and cost, which depending on the project is not feasible. However, among formal languages, graphs present an interesting alternative to this problem. Because they can be represented as a set of vertices and edges (graphical representation), these languages can be automatically validated and simulated, they can still be interpreted as a set of states of the architecture, facilitating the understanding of the stakeholders.

The use of semi-formal models becomes another interesting alternative to the problem of the complexity of formal languages. Semi-formal languages is most simple to understand and are widely used in systems project. In dynamic software architectures its use are lower due to the necessity of automatic simulation on many domains (ZHANG *et al.*, 2016). However, Miladi *et al.* (2008) presented an extension of the language that allows it to be translated into formal language (in this case, Z language), which makes possible the simulation of these models. The main advantage of using UML is the possibility of visual representation, which facilitates understanding and reduces the learning curve by the stakeholders.

2.4.3 Problems on Reconfiguring Dynamic Software Architectures handled by Representation Approaches

Section 2.3 presented a set of concerns that should be considered during the design of dynamic software architectures, characterizing some difficulties that can be found by the architects in this activity. In this sense, it is important to relate how existing approaches can support the design of architectures regarding these issues and if they can represent them in a coherent way. This section presents results of answering the **RQ3 - How can the representation approaches solve dynamic software architectures issues?**

To ensure the consistency, some studies (S37, S43, S48) proposes the concept of transaction is applied. Transactions consist of pre-established processes in which components can not participate in a reconfiguration if they are in use. For this, a reconfigurator must be used to manage the running transactions and which components are involved [Huynh et al. \(2016\)](#). To ensure the consistency of these systems, representation languages (Wright, for example) must provide mechanisms that allow the reconfigurator to identify components that depend on each other, i.e., if a component is in use, components on which it depends can not be reconfigured in that moment. Another way to solve the consistency problems of architectures is the application of variability concepts and product line of software. According to [Huynh et al. \(2016\)](#), the points that give rise to variability in the systems can be used to represent points that would generate dynamic aspects in the architecture.

One of the main problems with respect to dynamic software architectures is its high-level of abstraction. Understanding (at design time or before implementation) how software architecture can evolve at run-time still one of the main gaps on this topic. Trying to solve that, some approaches (S17, S28, S32, S36, S48) apply more than one language to give an graphical representation to facilitate design of that structures. For example, [Hussein et al. \(2017\)](#) combined EAST-ADL, a formal ADL, with UML. With this combination, the authors improved the stakeholders understanding, allowing better representation of the decisions made.

2.5 Representation of Dynamism in Reference Architectures

During the conduction of our SMS, we do not identify approaches concerned on represent dynamic aspects in reference architectures. Even the first works on dynamic architectures was published more than 20 years ago, the use of its concepts in reference architectures, according to our SMS, is not explored, what can decrease the quality of concrete systems which are based on it. However, some contributions on software architecture serve as basis

for the description of dynamism in the reference architecture level. We identified three works : [Beek et al. \(2009\)](#), [Smaali et al. \(2014\)](#), and [Hassan et al. \(2016\)](#).

[Beek et al. \(2009\)](#) proposed an automated process to develop dynamic software architectures. This process is divided in three main phases: *structural design and analysis*, *behavioural design and analysis* and *automatic code generation*. In the first phase, the architecture style is defined and the components and connectors are instantiated. Moreover, in this phase the reconfiguration rules are defined. In this phase, authors proposed the use of typed graph grammars (TGG) due to: (i) the usual way of representing architectures is achieved using formal basis and a graphical representation; and (ii) a natural way to describe styles, configurations and reconfigurations. To model rules and structures, the authors used *Alloy*, once it has a simple notation and can represent the formalism of TGG.

In phase two (behavioural design and analysis) the authors seek for validate models built in phase one. In this sense, they used a specific model checking approach called UMC. For each DSA configuration in the set of architectural configurations generated during the phase one, we associate to each component (i.e. hyperedge) a communicating UML state machine describing its behaviour. Thus, in this phase, rules and structures are validated. That last phase is the code generation and uses ARCHJAVA, extending JAVA with component classes (describing objects that are part of the architecture), connections (enabling components' communication) and ports (connection endpoints). In this sense, after the three phases, dynamic software architectures is automated generated.

In a second work, [Smaali et al. \(2014\)](#) proposed the Dynamic Software architecture Meta-Model (DySAM), in this model the concept of interface is considered the core of the architectural description, thus, the DySAM model considers it as first class entity while leaving components and connectors as grey boxes. This will offer more flexibility and a loose coupling of architectural elements. Considering the interfaces, DySAM also needs to define the possible configurations of the architecture, making a dynamic software architecture meta-model. To represent the configurations, the DySAM is integrated with K-Maude, an approach based on rewrite logic that allows formal specification of architectures and, consequently, its automated validation and simulation. In this approaches, rules are defined as a set of change Actions, resulting Events, Conditions and state's Transitions. MDA development techniques and formal K-method was combined to provide the DySAM model intuitive modeling notation, supporting a graphical view, and rigorous syntax and semantics.

Finally, [Hassan et al. \(2016\)](#) proposed a Meta-Evolution style (MES). MES consists in defining meta-concepts for describing a software architecture evolution aiming to capture the main characteristics of the set of activities that are performed to evolve software architecture. Different of the approach proposed by ([BEEK et al., 2009](#)), in this

approach components are the core of the evolution. The authors define five elements that compose the MES:

- **Role:** the stakeholders, tools and techniques that participate in performing the architecture evolution operation;
- **Operation:** The operation are associated with roles and architecture element through the interaction element and is one of the units of process that consumes or produces an architecture element;
- **Architecture Element:** the inputs and the outputs of an architecture evolution operation;
- **Interaction:** an entity that governs the relationships between these elements and their behavior; and
- **Interface:** a place in which elements interact and which determines rules that should be followed by elements to intercommunicate compatibly.

As stated, we do not identify approaches concerned in represent dynamism in reference architectures. Furthermore, approaches concerned in the representation of reference architectures do not explicit how they address dynamism with that views and languages. In the same perspective, the approaches presented in this section are efficient in their context but they have some issues that could constraint their use in reference architecture's context. First, reference architectures has a higher level of abstraction and (some of them) do not consider the concrete component, but a generalization of types of components, for example, what have to be considered during its representation ([ANGELOV *et al.*, 2009](#)). Moreover, the approaches are not concerned about what dynamic aspects need to be represented and for what stakeholder, for example. Do not consider those characteristics can compromises stakeholder's understanding, what can result in mistakes during system's development, causing rework and elevated costs. Thus, it is important to establish an approach that could fulfill these gaps and could be applied since reference architectures, improving its quality.

2.6 Final Remarks

In this chapter, we characterized dynamic software architectures, highlighting their main characteristics, the main issues, and types. Moreover, we provide an overview about the approaches that have been used to represent dynamic software architectures. We identified five categories distributed in 50 studies over the last 20 years. Most of the identified studies

use formal representation, which makes possible the automatic verification of the architectures, but it makes difficult to understand the models. Most of these case studies present simple systems that have programmatic dynamism and are didactic examples. Most case studies are not concerned with evaluating the specific characteristics of the approaches, but rather their application to the representation of dynamism in architecture.

Hence, based on the panorama provided by this chapter, we can highlight the following issues: (i) lack of graphical approaches that allow better analysis of architectural decisions; (ii) lack of practical examples of the use of these approaches; (iii) formal languages with a large learning curve; and (iv) lack of approaches that address dynamism in reference architectures. Considering these gaps, Chapters 3 and 4 will detail our research contributions.

A Conceptual Model for Dynamic Software Architectures

3.1 Initial Considerations

The design of dynamic software architectures enables stakeholders to understand changes that may occur at run-time and their impact on the systems (DING, 2016). Changes and reasons for them must be identified; if the architecture is not documented, the system may not be implemented in the right way, thus compromising both its quality and goals (ALLEN *et al.*, 1998).

This chapter defines a conceptual model for dynamic software architectures which considers 14 concepts to be considered for design and representation of a dynamic software architecture. It also enables a concern-based design that considers the concerns individually so that each dynamism necessity can be precisely solved. Furthermore, it can help architects improve their architectural representations.

3.2 Methods

The development of the model followed four steps, namely: (i) **literature survey on dynamic architectures**, (ii) **selection of relevant characteristics**, (iii) **synthesis and definition of concepts**, and (iv) **definition of relationships**. For better understanding and organization, each of the steps will be described below.

Three types of search were conducted for the **literature survey on dynamic architectures**: (i) **ad-hoc**, (ii) **systematic mapping study**, and (iii) **snowballing**. The search engines were diversified for the achievement of largest number of studies and avoidance of lack of relevant literature. In what follows is the way each search was conducted.

- **(i) Ad-hoc search:** mechanisms such as Google Scholar and Scopus were used; terms related to dynamic architectures were searched and the results were analyzed according to titles and abstracts;
- **(ii) Systematic Mapping:** the elaboration of this model is part of a larger project that has been conducted at the ICMC/USP by the author. The systematic mapping followed the guidelines of [KITCHENHAM B. A.; CHARTERS \(2007\)](#) and [Petersen et al. \(2015\)](#) on the ADLs for the representation of dynamic architectures. Among the data extracted from the selected studies are those related to characterizations of dynamic software architectures and those also used for the elaboration of this model. More results of this study are provided in Chapter 2; and
- **(iii) Snowballing:** after the reading of selected studies from previous searches, a backward snowballing was conducted, according to the guidelines provided by [Wohlin \(2014\)](#), for selecting more studies that would provide relevant information on the characteristics of dynamic architectures to be used in the development of model.

A form was developed for the extraction of data on characteristics of dynamic architectures. Initially, the focus was given on the reading of studies that aimed to explain the concepts of dynamic architectures towards the composition of the background necessary for the interpretation of subjective characteristics that might appear in other studies, especially those to be read from systematic mapping. Therefore, for achieving **(ii) selection of relevant characteristics**, the extraction form was divided into two namely (i) General Information, and (ii) Characterization Information:

- **(i) General Information:** general information on the studies (authors, year, vehicle, among others) was extracted towards a more efficient consultation to the study; and
- **(ii) Characterization Information:** for each study read, possible characteristics and their definitions were extracted. Regarding such characteristics, the authors tried to understand (i) the way architectures were reconfigured, (ii) why they must be reconfigured, and (iii) the way the reconfiguration process was managed. Since several characteristics were subjective, the extraction depended on interpretation, which was performed and later discussed by the authors.

After extraction, a set of heterogeneous characteristics that often diverged in nomenclature and converged in meaning was obtained, analyzed by the authors, and synthesized, thus achieving **(iii) synthesis and definition of concepts**, and concrete and unique concepts in the model. The concepts were again interpreted to be related in

terms of their semantics. Towards supporting this step and the definition of the model, three architectures (KALLEL *et al.*, 2012; PORTOCARRERO *et al.*, 2016; HUSSEIN *et al.*, 2017) with dynamic characteristics were analyzed, after which the concepts were defined. The nomenclature of the concepts involved their definitions and was used very intuitively for facilitating their use.

Finally, towards **(iv) definition of relationships**, the model relationships were established according to the characteristics extracted from the selected studies. As in the concepts, words were chosen for an intuitive definition of relationship semantics, therefore, they were action verbs for facilitating understanding. Each relationship indicates the way a concept leads to others and complements each other for the characterization of dynamic architectures.

This section presented the methods followed for the development of the conceptual model of dynamic architectures, and the four steps and what they included were described. The following section presents the complete model, explaining each of its terms and relationships.

3.3 Model

The construction of the model (Figure 4) was based on several concepts extracted from the literature and the analysis of software architectures, as described in Chapter 2.

The model is divided into two parts, namely concepts (represented by rectangles) and relations (represented by the lines connecting the rectangles). The former present the characteristics of dynamic architectures, usually with nouns, whereas the latter indicate the way each concept should be considered in terms of dependencies. The model was built for orienting the dynamic characteristics of the architecture towards a specific concern, therefore, each concept establishes relationships that build a workflow from the definition of the concern to be covered by a dynamic solution. For a better organization and greater details of the model's artifacts, the definition of concepts and relationships are described into topics addressed in the next sections.

Figure 4 – Conceptual Model for Dynamic Software Architectures

Source: Elaborated by the author.

3.3.1 System's Specification Concepts

This section describes the concepts related to the characterization of the system, its requirements, and its domain, especially about concerns related to reconfiguration. In this sense, the related concepts are described below:

- **Concern:** architecturally significant requirements normally related to quality attributes to be covered by the system. In dynamic architectures, the strategy for achieving quality attributes is the reconfiguration in time of the system components (LOULOU *et al.*, 2004; BRUNI *et al.*, 2008). As an example, let us consider a temperature management system must receive information from a sensor that works via both Bluetooth and WIFI. By applying the concepts of dynamic architecture and towards meeting the availability attribute, the software system (depending on the delay time between one data delivery and the other) can identify failures in one of the connections and switch the protocols so that the other connection is used and keeps the system available;
- **Application Domain:** the application domain of a software system refers to the characteristics related to the functionalities and quality attributes inherent to the environment where the system will be applied (health systems, management systems, among others) (NAKAGAWA *et al.*, 2014). The domain characteristics must be considered, since they can assist in the definition of the requirements, hence, the reconfiguration policies; and
- **Software systems:** is any system that requirements demands for reconfiguration mechanisms.

In the same perspective, the relationship among the concepts describes their dependencies on and uses with other concepts. The relationships of this section are described below:

- *Software System* **Has** *Application Domain*: A Software System has an Application Domain that defines, among other things, the environment in which it is inserted and some concerns related to it (NAKAGAWA *et al.*, 2014);
- *Application Domain* **Frames** *Concern*: An Application Domain helps the definition of some concerns over a software system. Therefore, its characteristics are fundamental for the establishment of both architecture requirements and reconfiguration mechanisms (NAKAGAWA *et al.*, 2014);

3.3.2 Dynamic Concepts

Dynamic Concepts are related to concepts specifically present in dynamic software architectures to be considered in the representation of the models.

- **Reconfiguration Policy:** consists in the definition of needs for the software architecture adaptation. It is established through the system's quality requirements and mapped under reconfiguration conditions (or triggers) (OQUENDO, 2016; DING, 2016). An example of a reconfiguration policy is related to system availability. The system must be reconfigured according to the number of connections on the server and must be rebalanced according to the number of connections. Reconfiguration policies are also related to trade-offs among quality attributes (WERMELINGER; FLADEIRO, 2002; BRADBURY *et al.*, 2004). For example, security can be compromised (by disabling authentication, for instance) in systems that prioritize performance;
- **Reconfiguration Strategy:** strategies are related to the way a software system must reconfigure given a situation (DJOUDI *et al.*, 2014). It is a process to be adopted for the reconfiguration of an architecture. According to the review conducted, the strategies can be classified into two categories:
  - *Parametric reconfiguration* - the system monitors the execution conditions and, once a trigger has been identified, it changes previously defined parameters, thus allowing adaptation (PORTOCARRERO *et al.*, 2016); and
  - *Element-based reconfiguration* - the software system loads (or destroys) elements and connections, creating redundancies and even applying other algorithms for processing its adaptation (KACEM *et al.*, 2006; F. Oquendo, 2008). For example, the system can instantiate a new instance and a new connection for users on a server that supports 200 connections per system image.
- **Trade-Offs:** In dynamic architectures, trade-offs are common as in other software architectures (BASS, 2013; KAZMAN *et al.*, 1998), i.e., architectural decisions that meet certain quality requirements can compromise others. Therefore, the architect must prioritize which are more important regarding others (OREIZY, 1998; MADNI; SIEVERS, 2014). In dynamic architectures, reconfiguration policies, as well as strategies, must be planned towards reducing the impacts of changes in other parts of the architecture;
- **Reconfiguration Triggers:** Conditions or triggers are stimuli, situations, or events that trigger an adaptation at run-time (OQUENDO, 2016; LING, 2013; ZHANG *et al.*, 2016) and enable the Reconfiguration Manager to identify a reconfiguration to

be proceeded, requesting the addition/removal of components, for example (BRADBURY *et al.*, 2004). In general, such conditions refer to system failures, changes in requirements, changes in the environment, among other events (DING, 2016; ADJOYAN; SERIAI, 2017; CIMATTI *et al.*, 2018). Triggers can be classified into two main categories:

- *Internal* - internal conditions refer to situations or failures in a system (e.g., failures in execution, parameters, encryption or security) (MADNI; SIEVERS, 2014; ZHANG *et al.*, 2016); and
  - *External* - this type is more common and related to the system's execution environment. An external trigger is an event that occurs with no interaction of the system, i.e., it is independent of it. Examples include wifi connection failures, lack of battery in sensors, overload of requests on the server, among others (ZHAO; LI, 2012).
- **Autonomy Level:** This concept defines the autonomy level of the components and is related to those that have access to Reconfiguration Policy. (tirei, ok?). Such an autonomy has 2 levels, namely:
    - *Endogenous* - each component "knows" what to do when a trigger is identified (OQUENDO, 2004); and
    - *Exogenous* - a centralized manager knows each reconfiguration policy and performs one of them when a trigger is fired (OQUENDO, 2004; CAVALCANTE, 2016).

In the same perspective, the relationships among the concepts are described in what follows.

- **Reconfiguration Policy Defines Reconfiguration Triggers:** according to the reconfiguration needs, events that start a process are defined (ALLEN *et al.*, 1998; OREIZY, 1998; CIMATTI *et al.*, 2018);
- **Reconfiguration Policy Defines Reconfiguration Strategy:** processes to be followed for the reconfiguration of a system, as well as the elements involved are defined;
- **Reconfiguration Policy Composes Knowledge Base:** knowledge bases are defined and augmented (IBM, 2003), so that better strategies can be chosen by the manager;
- **Reconfiguration Policy Consider Trade-Offs:** trade-offs must be taken into account among requirements for a correct definition of strategies that prioritize concerns (ALLEN *et al.*, 1998; MADNI; SIEVERS, 2014);

Reconfiguration Policy Defines Autonomy Level: Based on reconfiguration policies, it is necessary to define how autonomous is the system ie, how much the system will have mechanisms to reconfigure without previous knowledge of the triggers (CAVALCANTE, 2016; OQUENDO, 2004);

- *Reconfiguration Strategy Composes Knowledge Base:* Just as reconfiguration policies make up the knowledge base, strategies provide data on the success and failure of strategies so that the choice of strategy is better and better (IBM, 2003);
- *Autonomy Level Impacts Reconfiguration Strategy:* According to the desired autonomy level, reconfiguration strategies used must follow the system's autonomy (CAVALCANTE, 2016; OQUENDO, 2004).

3.3.3 MAPE-K Concepts

As addressed elsewhere, MAPE-K (IBM, 2003) is a reconfiguration management model widely used in Self-Adaptive Systems (LEMOS *et al.*, 2013). Since this type of system has dynamic architectures, it was used here as a basis for the definition of concepts related to the management of reconfigurations, described in what follows:

- **Reconfiguration Management:** among the several ways of managing reconfiguration at run-time, two types can be elucidated taking into account which components can perform this management:
  - *Centralized:* a central entity manages all other components, assessing the need for adaptation, and planning and executing it (F. Oquendo, 2008; OQUENDO, 2016); and
  - *Decentralized:* components manage adaptations. MAPE-K model, which manages changes according to a body of knowledge, is commonly applied (CAVALCANTE, 2016; GARCÉS, 2018; OQUENDO, 2016; LEMOS *et al.*, 2013).
- **Knowledge Base:** it can be used by the Reconfiguration Manager to support the choose of the best Reconfiguration Strategy.

The relationships of concepts in Mape-k Section are described in what follows.

- *Reconfiguration Management Performs Reconfiguration Strategy:* when it identifies a trigger, a strategy is chosen and applied (OQUENDO, 2004; CAVALCANTE, 2016);
- *Reconfiguration Management Observes Reconfiguration Triggers:* it checks if any event (trigger) has triggered reconfigurations, selects a strategy, and performs the

necessary reconfiguration (ALLEN *et al.*, 1998; CAVALCANTE, 2016; MADNI; SIEVERS, 2014);

- *Reconfiguration Management Uses Knowledge Base*: is uses the knowledge base to choose the best reconfiguration strategy, i.e., according to previous successes and failures of the strategies (IBM, 2003);
- *Reconfiguration Management Ensures Reconfiguration Policy*: the policy is then followed and strategies are applied to it (MILADI *et al.*, 2008; OREIZY *et al.*, 2008)
- *Reconfiguration Management Implements Autonomy Level* in other words, it defines a set of components towards the implementation of the autonomy level desired for the system (KACEM *et al.*, 2006);

3.3.4 Architectural Representation Concepts

The Architectural Representation Concepts are based on ISO 42010 (ISO/IEC/IEEE, 2011) and related to elements to be represented in dynamic software architectures. An architectural configuration represents a set of software and hardware elements (described in what follows) that can be modified during a reconfiguration process:

- **Architectural Element**: any element that comprises a software architecture (ISO/IEC/IEEE, 2011), e.g., a hardware or s software one. In the proposed model, the architectural elements are primarily responsible for reconfiguring the architecture, since they are (un)connected, created, destroyed or rearranged during the architecture adaptation process (OQUENDO, 2016; ALLEN *et al.*, 1998);
- **Link**: connections are elements of specific architectural elements that enable communication between two or more elements (BASS, 2013; KAZMAN *et al.*, 2013) and provide the necessary interfaces for their exchange of information. They are also part of the reconfiguration process, since they can be created and destroyed, depending on the context (ALLEN *et al.*, 1998; CAVALCANTE, 2016). Connection failures between components can trigger changes in run-time;
- **Model-Kind**: according to ISO 42010/2011 (ISO/IEC/IEEE, 2011), Model-Kind is any model that represents architectural elements. Model types encompass diagrams, languages, architectural description and even informal models that represent software architectures; and
- **Architectural Configuration**: refers to the set of architectural elements that comprises the architecture at a certain time in execution (ALLEN *et al.*, 1998; BRADBURY *et al.*, 2004), and represents the components and connections used as ar-

chitecture. A change in architectural configuration is used as a form of dynamic reconfiguration (DING, 2016).

The only relation in this section is described bellow:

- *Architectural Element Is represented by Model-Kind*: Architectural Elements are represented by Model Kinds (notations, languages, among others) (ISO/IEC/IEEE, 2011);

3.3.5 Relation among Sections

Relationships among sections express the way they are related to each other. Below are the descriptions of the three that were listed:

- *Concern Has Trade-Offs*: dynamism in software architectures is related to a specific concern, which imposes trade-offs in the architecture if other concerns are considered (ALLEN *et al.*, 1998; KACEM *et al.*, 2006);
- *Concern Defines Reconfiguration Policy*: concerns define the requirements to be covered by the architecture. When concerns cannot be addressed by static mechanisms, a reconfiguration policy that supports such requirements is defined (OREIZY, 1998; OQUENDO, 2004; OREIZY *et al.*, 2008);
- *Reconfiguration Strategy Modifies Architectural Configuration*: in other words, when a reconfiguration is performed, the strategy changes the elements and links of the architecture so that this architecture assumes a new configuration towards meeting the concerns (ALLEN *et al.*, 1998; WERMELINGER; FIADDEIRO, 1999; DING, 2016; CIMATTI *et al.*, 2018);
- *Software System Has Architectural Configuration*: a set of hardware and software elements meets the architectural concerns (ALLEN *et al.*, 1998; OREIZY, 1998; ISO/IEC/IEEE, 2011; BASS, 2013);

3.4 Guidelines and Model Usage

The proposed model consists of a set of concepts related to dynamic software architectures, as well as the relationships among such concepts. It enables analyses of the architecture's characteristics and guides (OK?) the architect on decisions to be made during the design of software architectures that demand dynamism. During its elaboration, we assumed the architect had conducted a previous analysis of the domain and concluded dynamic solutions should be applied. Therefore, it can be employed in the following two different scenarios:

- **Initial design of software architectures:** the model is applied during the definition of architectural decisions on dynamism. Architecturally significant requirements are analyzed by the architect and each necessary concept is defined so that the dynamic solution can be reached; and
- **Architectural recovery and update:** the architect can use the model to understand and document solutions applied to an existing architecture (under use or not). The proposed documentation must be analyzed for the identification of the components defined and of whether all concepts have been covered. In this scenario, the model supports improvements in the documentation of the architecture and its solutions, since it presents the concepts to be covered and enables the identification of missing ones.

In both cases, the model instance is related to a quality attribute that needs dynamic solutions to be achieved. As an example, Chapter 5, addressed its for characterizing DiaManT@Home (VICENTE *et al.*, 2017), a software architecture for the health domain. The decisions made and the logic used for that representation will be shown, to help architects to better understand the use of the model.

3.5 Threats to Validity

For the construction of this model, a customized scientific method was adapted. The authors aimed at a method that would reduce possible bias displayed in the process and ensure accurate and valid information extracted. Due to the subjective level of the information, some threats to validity must be considered. Below are the descriptions of the main threats and the actions taken by the authors for mitigating them

- **Subjectivity of Information:** much of the information extracted from studies was not explicit, therefore, the authors needed to interpret some data, especially regarding the characteristics of dynamic architectures; and
- **Lack of relevant works:** some work may not have been recovered and might contain information relevant to the construction of the model. Towards mitigating this threat, three different types of search were conducted for recovering as many studies with quality information as possible;

3.6 Final Remarks

This chapter presented a conceptual model for dynamic software architectures that consists of a set of concepts and relationships directly related to the components that form

dynamic solutions. It enables improvements in analyses of architectural requirements that demand dynamic solutions and efficiency in software projects, since it guides architects on decisions to be made.

This conceptual model can be understood as a starting point for the construction of dynamic structures - however, they must be represented coherently. Representation techniques must provide a low learning curve and be simple, efficient, and consistent with the agility found in software development environments. In this sense, the next chapter presents a set of architectural views and associated techniques for the representation of dynamic architectures.

Architectural Views for the representation of Dynamic Software Architectures

---

4.1 Initial Considerations

Architectural views have been widely used for representing software architectures by software architects in both academia and industry ([BASS, 2013](#)). This approach consists in dividing a software system into different models so that it is presented according to different concerns. Different approaches show predefined sets of views, of which many are consolidated among researchers in the area([CLEMENTS *et al.*, 2010](#)), and others are innovative and aim to represent new perspectives of a software system. [Tekinerdogan and Sozer \(2017\)](#) suggest the use of such views for representing architectural decisions, which, in dynamic software architectures, is not as widespread as in static software architectures.

The use of two views, namely structural and behavioral, was observed in a systematic mapping conducted in this project. As suggested by the name, such views display characteristics related to the components of architecture (layers, source codes, sensors, APIs, among others) and behavioral ones (communication protocols, information exchanged, interfaces, among others). However, such information is suppressed in those models, since more information would be necessary for a complete representation (e.g., trigger for reconfiguration, strategy for reconfiguration, among others).

Towards mitigating this lack of information and increasing the reliability of the architecture, such views are, in most cases, represented by formal notations, as in the systematic mapping. However, their use is complex and requires efforts from stakeholders for both communicating and understanding the documentation of the software system.

In such a scenario, an alternative is the development of a set of views, which, based on the Conceptual Model, can represent all concepts identified as relevant to dy-

dynamic architectures. Moreover, the views are semi-formal, which reduces efforts for the construction and understanding of decisions. Therefore, with their use, descriptions are expected to be more complete while maintaining reliability.

ISO 42010 ([ISO/IEC/IEEE, 2011](#)) was used as a basis for our views, and a meta-model, shown in [Figure 5](#) and which lists the necessary concepts for their construction and use, was built.

According to [Figure 5](#), each view is composed of 4 concepts, namely Dynamic Viewpoint, Dynamic View, Dynamic Architectural Model, and Dynamic Model Kind that must be considered and related. This model was built through the instantiation of the ISO 42010 concepts ([ISO/IEC/IEEE, 2011](#)) related and adapted to the context of dynamic architectures. Therefore, they inspired the descriptions presented in what follows - each concept is explained and related to both view and Conceptual Model:

- **Dynamic Viewpoint:** the viewpoint that governs the view and establishes what must be represented. The conceptual model constitutes the Viewpoint, since it presents the information to be included in the view;
- **Dynamic View:** the view itself, which is governed by a Viewpoint and represented by a model;
- **Dynamic Architectural Model:** the representation of the view by a model-kind; and and
- **Dynamic Model Kind:** the notation that can represent the view, such as UML Module Diagram or Deployment Diagram. Variations in such notations in other languages (e.g., SysML) can also be applied, as long as the function that presents parts of the system is maintained.

Figure 5 – Representation Model

4.2 General View

General View is a Module View (CLEMENTS *et al.*, 2010) that represents the system with all modules that compose it (considering the dynamic concern to be represented) and the way they are related, and that provides a brief view of the dependencies among them. In general, it holistically shows the system as a whole. Figure 6 shows a model of the concepts covered by this view.

Figure 6 – General View Model

This view must be designed, since it represents the "Architectural Configuration" section of the Conceptual Model, showing the main components that can be reconfigured and the initial configuration of the architecture. It also enables evaluations of semantic problems that may have occurred during both analysis of the domain and definition of relationships among the modules, and represents dynamic components at run-time. Although it does not assess the dynamism of the architecture, it provides a blueprint of the architecture as a whole, as well as its static view (CLEMENTS *et al.*, 2010).

4.3 Reconfiguration View

Reconfiguration View is a variation of General View that presents a system before and after reconfiguration. Since a system is represented according to a reconfiguration scenario, which, in turn, is dependent on a requirement, this view must show only the components that are part of the reconfiguration in question.

It must present the components of the system and evidences the way the reconfiguration will change the constitution, connection, and organization of these components. Therefore, notations of easy understanding must be selected, since the view aims at presenting the way a reconfiguration changes the components and organization of the architecture. Notations such as UML Component Diagram, Deployment, and the like can

be used. Variations in diagrams in other languages are also interesting (e.g., SysML or other semi-formal languages for representing software architectures). Figure 7 illustrates the conceptual model of the view.

Figure 7 – Reconfiguration View Model

Reconfiguration View represents dynamism in architecture, because it can be used as a validation of the reconfiguration requirement. Since the reconfiguration process aims to support a quality requirement, measurements of the changes enable assessments of the impact of the reconfiguration on the whole architecture.

4.4 Ripple-Effect View

The third view in the set is the Ripple-Effect View. As the name suggests, it aims to represent the effects the reconfiguration can trigger on a whole architecture, i.e., the way components that do not participate in the reconfiguration are affected. It includes components presented in General View and the way they are related to the reconfiguration represented, and shows the components of the Reconfiguration View as a black box, since its main objective is to present those that are not part of the reconfiguration, but are under some type of modification.

Similarly to the other view, Ripple-Effect should be designed with semi-formal structural notations, such as UML's Component and Module Diagram, since this type of representation enables relations and connections of components for the identification of their dependencies. Variations (e.g., diagrams in other notations, such as SysML) are also supported, and textual descriptions regarding redundancy of the models can be applied. Tables and texts can also represent this view.

ISO 42010 (ISO/IEC/IEEE, 2011) was used for the structuring of the concepts, since the conceptual model of this view is the instantiation of the model present in the standard (see Figure 8).

Figure 8 – Ripple-Effect View Model

As addressed in Section 4.2, the concepts of ISO 42010 were applied for the construction of the views, and the concepts shown in Figure 8 are based on the standard. This view is an important source for the assessment of coupling and dependence on the components of the architecture - even those that do not participate in certain reconfigurations. In addition, possible architectural trade-offs can be identified, since a certain quality attribute must be prioritized to the detriment of others, depending on their impact on the architecture.

4.5 Scenario View

Scenario View, the fourth view of the set of views that represent dynamic architecture, is based on the scenario model proposed by Bass (2013). It describes important concepts for the analysis of architecture and the way it must behave in some situations. In dynamic models, this type of analysis is fundamental, since the entire construction of the architecture must be based on a certain reconfiguration requirement (ALLEN *et al.*, 1998; GARLAN *et al.*, 1995), derived from the whole scenario.

A model based on ISO 42010/2011, as shown in Figure 9, was designed for the construction of the view, and identified the concepts to be covered by the view.

The reconfiguration scenario organizes and joins the other views so that they can be related to their function in the representation of architecture. It is composed of the

Figure 9 – Scenario View Model

same elements defined by Bass (2013), which can be mapped in 5W1h (What, Where, When, Who, How, Why), and shows those that participate in the reconfiguration, as well as their roles. In what follows is the description of the 5W.

- Source of the stimulus (Why): the reason why an architecture should be reconfigured. According to the SMS, architectures must be reconfigured when static solutions are not able to achieve the quality attributes required (BRADBURY, 2004; CÎMPAN *et al.*, 2005; CIMATTI *et al.*, 2018; ADJOYAN; SERIAI, 2017). This element is textual and should express the quality attributes that require those dynamic solutions. The reconfiguration needs must be clear, since all analysis and representation (in both model and set of views) must be based on a reconfiguration scenario defined by a quality attribute;
- Stimulus (When): defines the moment at which an architecture must be reconfigured, i.e., it represents the reconfiguration triggers, which must be explicitly identified and related to quality attributes with the use of measurable data (e.g., reconfiguring the system when the number of connections has reached 200) for the avoidance of ambiguities (GARLAN *et al.*, 1995));
- Environment (Where): corresponds to the Reconfiguration View and shows the location in the system that requires a change. General View can also be related to this element, since it displays the architecture structure in its normal operation. This section describes the way a system should operate in a normal situation and after reconfiguration (as shown in Reconfiguration View), and is also related to "Architectural Configuration", depicted in the model in Figure 4;

- Response (How/Who): a process to be adopted for the reconfiguration of a system, indicating the way it should occur. Regarding the conceptual model, this section should describe the Reconfiguration Strategies to be adopted for the achievement of each quality attribute. Two views must be considered: Strategy View and Ripple Effect View, and the events that should occur and their sequence must be described. Moreover, the impact of such changes on the remainder of the system must be measured; therefore, the key relationships among the components (as shown in General View) must be observed. Another important point to be considered refers to "WHO" will control and proceed to reconfiguration, since reconfiguration management is also part of the system's response. [Figure 4](#) shows the reconfiguration management is contained in the "Reconfiguration Management" element, which, should follow MAPE-K guidelines ([WEYNS et al., 2015](#)), for a better process management; and
- Response Measure (What): consists of an evaluation of the reconfiguration regarding the meeting of the requirement. Another important point in this section is the Ripple-Effect view, since it displays the impact of reconfiguration. The Trade-Offs considered for the construction of the reconfiguration model must also be analyzed, since different reconfigurations in different attributes can impact attributes that require no dynamic solutions. Therefore, the trade-offs considered, those prioritized, and the way others were mitigated must be recorded.

In summary, the scenario view organizes the other views, and provides architects with an overview of the problem of reconfiguration of the system. The use of textual elements, i.e., texts, tables, items, among others, is recommended for this view representation. This view should be built first, since elements to be used in the others can be identified.

4.6 Strategy View

The strategy view consists of a view that represents the behavior of the system during reconfiguration. It shows which operations must happen when a Reconfiguration Trigger is identified. [Figure 10](#) displays an instance of the ISO 42010 model with the main elements of the view.

Apart from the components addressed in Section 4.2, the figure shows the model to be represented by this view and the Reconfiguration Process. The views must be represented by behavioral notations, especially in semi-formal languages (eg, UML, BPMN, among others), since such notations are simpler to be learned and used, improving and facilitating communication among stakeholders.

Figure 10 – Strategy View Model

The components to be involved in the reconfiguration, as well as the way messages will pass them must be defined. The way the manager sends messages to the components and the components' level of autonomy are represented in this view.

4.7 Discussion

The use of architectural views is widespread in the scientific community of software architecture, and one of the main techniques for the presentation of parts of a system, their relationships, features and organization (both logical and physical) (BASS, 2013; CLEMENTS *et al.*, 2010). However, architectural views have not yet been explored in the context of dynamic architectures. Although evidence has shown their use (GARLAN *et al.*, 1995; OQUENDO, 2016; CAVALCANTE *et al.*, 2015), the way they are proposed and used ignores some characteristics inherent to the dynamic domain, thus compromising the quality of the architecture. Such views tend to use generic views (structural and behavioral), and the way important information is handled in the context of a dynamic solution is not clear.

Combined with the Conceptual Model, our views present the main concepts and information to be identified, analyzed, and represented regarding dynamic architectures. As important as identifying such characteristics, relating them semantically in the representation results in an easy understanding of the solution, especially by stakeholders. Therefore, the views proposed in this chapter aim to cover all concepts presented in the model, separating them semantically and generating self-contained models for a better analysis of both features and quality requirements.

The following figures provide a list of the concepts present in each view for showing such relationships.

Figure 11 – Concepts of General View

Figure 12 – Concepts of Reconfiguration View

Figure 13 – Concepts of Ripple-Effect View

Figure 14 – Concepts of Scenario View

Figure 15 – Concepts of Strategy View

[Figure 11](#) displays the concepts related to General View, which shows those related to the initial architecture configuration, listing all components (software and hardware) that are part of the description. Therefore, the elements of the architecture are represented without relating concepts and dynamic elements.

[Figure 12](#), depicts the concepts related to the impacts of reconfiguration represented on the items that are part of the Reconfiguration View. The reconfigured elements are shown and highlight the initial and final states of the architecture (Architectural Configuration), hence, the elements of the architecture (Software Elements, Hardware Elements, and Architectural Configuration) and those that are part of the reconfiguration process (Reconfiguration Manager, Reconfiguration Triggers, among others) and that impact the reconfigured elements.

The [Figure 13](#) shows the concepts related to the Ripple-Effect View, i.e., concepts that address elements that suffer the effects of reconfiguration despite not being part of it. As it is possible to measure, in this view again we are going to present the architectural configuration, built to relate the elements to a reconfiguration scenario in which they do not participate.

Scenario View shows concepts that encompass the entire environment that requires a dynamic solution. It lists the concepts and elements to be reconfigured, the reason for reconfiguration, and the strategy adopted. [Figure 14](#) depicts the concepts of the Conceptual Model that are represented by this view.

Finally, [Figure 15](#) displays the concepts related to Strategy View, i.e., "Dynamic Concepts" (according to [Figure 4](#)), that is, they represent the processes and elements present in the reconfiguration.

The relationship among the views must be highlighted, since such dependencies result from the views representing the same architectural elements at different times of the execution with different objectives. [Table 2](#) shows the data related to each view, evidencing their main differences, towards the understanding of such relationships.

The analysis of the information provided in [Table 2](#) revealed the set of views proposes the representation of the master section of the architecture in several different ways with different objectives. This approach is interesting, since, as discussed in [Chapter 2](#), most notations that represent dynamic architectures are formal. However, its use is costly regarding both computational resources and architect's effort. Towards reducing such efforts, we have built a set of views that approaches the reliability of formal representation, but with a reduced learning curve, so that all necessary elements are represented in a less complex way.

4.8 Final Remarks

This chapter presented a set of architectural views for the representation of dynamic software architectures. The set was elaborated in a way all concepts of the Conceptual Model are addressed and all information necessary for the design of such a software architecture type is present.

Due to abstract characteristics and critical domains where dynamic architectures are usually inserted, formal models are generally applied for representation. However, due to the complexity of such notations, the proposed set of views aims to reduce efforts during the architectural design, while maintaining the highest possible level of reliability. Therefore, semi-formal notations are suggested for the use of views.

Redundancies are observed in the represented components, i.e., they are represented in different places with different objectives, which enables the visualization of the architecture at different times (project, execution, re-configuration scenario), providing stakeholders with a broader view of the solution and mitigating the lack of dynamism in the notations.

Towards measurements of the feasibility of our set of views, [Chapter 5](#) addresses their application for representing an existing software architecture. Improvements in their use regarding level of understanding in each representation (with and without the views) are also analyzed.

Table 2 – Relation among Views

Name	Objective	Type	Notations	Notes
General View	Shows all elements which will be part of the dynamic software architecture.	Structural View	Class Diagram Module Diagram Component Diagram	Shows elements in a static way, i.e., at the system's starting.
Reconfiguration View	Presents initial and final state of the architectural configuration, considering the reconfiguration scenario.	Structural View	Class Diagram Module Diagram Component Diagram	It differs from General View due to represented elements i.e., General View shows all architectural elements and Reconfiguration View only those present in reconfiguration scenario.
Ripple Effect View	Presents the impacts of reconfiguration in elements which are not present in the process.	Structural View	Class Diagram Module Diagram Component Diagram Textual Description	It differs from Reconfiguration View due to the elements which are presented, i.e., in Ripple Effect View, only elements those are not on scenario reconfiguration are represented.
Scenario View	Presents the reconfiguration scenario i.e., presents reconfiguration requirements, triggers, quality attributes and trade-offs related to reconfiguration.	Supportive View	Scenario Template (Bass et.al, 2013)	This view aims to join and relate the Conceptual Model and the views, to analyses and take decisions about the project. Hence, other views are "inside" this one.
Strategy View	Shows the reconfiguration process and how the manager reconfigures the architecture	Behavior View	Sequence Diagram BPMN	This view considers the elements of the other views i.e., shows how messages are exchange among components during reconfiguration process.

Source: Elaborated by the author.

Illustrative Example of the Use of Views

5.1 Initial Considerations

This chapter provides results of a proof of concept of the conceptual model and the set of views proposed in Chapters 3 and 4. The set enables the representation of dynamic architectures in a certain reconfiguration scenario based on some quality attributes that cannot be achieved by a static solution.

A dynamic architecture, called DiaManT@Home (VICENTE *et al.*, 2017), was used in the proof of concept for the healthcare supportive home systems domain, which integrates several sensors that measure data on patient health (e.g., temperature, blood pressure, blood oxygenation, among others), managing and sending them to the responsible health center. As it is a set of sensors, failures may occur, or new sensors may be required. The system cannot stop the insertion of sensors, i.e., the insertion of a new sensor consists in a dynamic solution necessary for the architecture.

Therefore, both conceptual model and set of views represented a reconfiguration scenario extracted from the architecture for measurements of the applicability of the views considering the ability to perceive the requirements and the understanding of the solution. The aim was also to evaluate whether the views and the conceptual model could fully represent a dynamic software architecture and whether any improvement from their application would be perceived.

Ideally, the views should be evaluated by specialists towards measurements of more attributes, such as learning curve and completeness. However, since such assessments are complex and require an unavailable amount of time, others have been planned as future work, as described in Chapter 6.

5.2 Methods

Five steps (described in what follows) were elaborated in a reconfiguration scenario that used the set of views proposed in Chapter 4 for the development of the proof of concept.

1. **Review of Dynamic Software Architectures:** Initially, a review of the software architectures found in the systematic mapping was conducted, and revealed DiaManT@Home would best suit our context, considering the elements described, the scenarios identified, and the views used;
2. **Review of DiaManT@Home:** After the selection of DiaManT@Home as the architecture of our proof of concept, a new review of the elements that composed the architecture was conducted, thus identifying the architectural decisions that would characterize a reconfiguration scenario;
3. **Identification of dynamic concepts:** After the selection of the reconfiguration scenario, the aim was the identification of the dynamic concepts covered by the architecture, considering the conceptual model, which was then instantiated for the proof of concept through analyses of the existing concepts and missing ones;
4. **Design of the Views:** As addressed in Chapter 4, the views are based on the conceptual model, which, after instantiation, enabled the identification of the elements to be modeled and the view in which they should be placed. The views were then built with the use of UML diagrams and textual descriptions; and
5. **Comparing the Solutions:** since the main objective of this proof of concept is to identify the applicability of views in the dynamic architecture design, the two solutions were compared regarding the advantages and disadvantages of the use of the views proposed.

5.3 Characterization of DiaManT@Home

DiaManT@Home is a software architecture designed for the Healthcare Supportive Home Systems (HSH) domain, i.e., a type of system that monitors patients in their homes (GARCÉS; NAKAGAWA, 2017). It is composed of a set of constituents (sensors, systems, and services) that collect, process, and make available data on several measurements of patients' vital signs.

Two documents were consulted to analyze the characteristics of the architecture: in Vicente *et al.* (2017) the functional and non-functional requirements of the system are defined. The architecture was instantiated according to the guidelines defined by HomecARe (GARCÉS, 2018) reference architecture. Some HomecARe models were reused

in DiaManT@Home, and were necessary for representing the use of the conceptual model proposed. Despite the relationship between the architectures, this example considers only the requirements of DiaManT@Home, and the other models support the explanation of the characteristics through the model.

After the analyses of the documents, requirement "Adaptivity, Integration" was selected, since this quality attribute is related to the dynamic characteristics of the architecture and its ability to reconfigure itself at run-time, according to some events. This fact is consistent with the main definitions of dynamic software architectures, presented in Chapter 2.

To present which elements and architectural decisions were made to achieve this quality attribute, [Garcés \(2018\)](#) defines a scenario, according to the guidelines defined by ([BASS, 2013](#)). The scenario consists of seven elements: attribute, environment, stimulus, response, architectural decisions, reasoning, and architectural diagrams, as shown in [Table 3](#).

Two other models were also considered for the model instantiation. The Diagram of Integration of HomecaARe, shown in [Figure 16](#), was cited in the scenario as the one that presented the architectural decisions made for the integration of new services. Additionally, the Conceptual View, displayed in [Figure 17](#), showed the components at auto level and through layers. Such a type of abstraction is highly relevant, since a holistic view of the architecture can be obtained, thus enabling the understanding of the way each component is distributed in the layers and definition of the dynamic characteristics to be instantiated in the model.

Table 3 – DiaManT@Home Adaptivity Scenario

Element	Description
Attribute(s)	Adaptivity, Integration
Environment	Normal execution of DiaManT@Home.
Stimulus	Physician updates the patient care plan, requiring incorporating extra services to monitor new conditions of the patient.
Response	One of the following: • (i) New services discovered and integrated, and reconfigurations of the HSB done; • (ii) No service discovered; • (iii) no service integrated.
Architectural decision(s)	• The discovery service is responsible for discovering available services to be integrated into the HSH system. • The discovery service performs service verifications regarding its identities, message, format, vocabulary type, services interfaces, capabilities, contracts, and qualities. • After correct verification, the service is integrated, otherwise, no integration is done. • A broker is allocated, by the controller, to offer interoperable communication between the new service and the HSB.
Reasoning	• Benefits: Automatic discovering and integration of new services, reducing manual configuration of brokers and interfaces. • Liabilities: (i) If none service offering desired capabilities is found, functionalities required to support new patient's conditions can not be offered; (ii) Only services using message formats and vocabularies stored in repositories in the interoperability standards layer can be integrated, otherwise, the new service must offer transformation rules to be executed by the broker
Architectural Diagram	Diagram of Integration of HomecARe

Source: Adapted from [Garcés \(2018\)](#).

Figure 16 – Diagram of Integration of HomecaARe

Source: [Garcés \(2018\)](#).

Figure 17 – Conceptual View of DiaManT@Home

Source: Garcés (2018).

After analyses of the three sources, the model proposed was instantiated, as shown in [Figure 18](#). Concepts were identified in DiaManT@Home and related to those established in the model, according to their semantics, which regarded both functionalities of the components and their organization and distribution in the architecture layers. An interpretative analysis was then conducted.

Figure 18 – Conceptual model for DiaManT@Home

Source: Elaborated by the author.

The description of the mapping between the concepts of architecture and the model is provided in tables, which show the concept of the model and its correspondent in DiaManT@Home, as well as the rationale followed for the relation. Table 4, shows the concepts related to the system's specifications and, as explained in Chapter 3, that deal with general information on the domain of the system, since such data can provide important support for architectural decisions. (tirei, ok?)

Table 4 – Mapping of System's Specification Concepts

Concept	Concept On DiaManT@Home	Rationale
Concern	Adaptivity	All decisions regarding dynamic architectures need to be made based on a concern. In this case, the Adaptivity concern cannot be achieved without a dynamic solution, therefore, it is the starting point of the model instantiation.
Application Domain	Healthcare	DiaManT@Home is an architecture designed for a system that manages patient data in their homes, so it belongs to the health domain.
Software Sytem	DiaManT@Home	Although DiaManT@Home is a software architecture, it designs a system, so the architecture under analysis itself consists of a system.

Source: Elaborated by the author.

The data from Table 4 were extracted by analyzing the information extracted from the scenario described in Table 3. Such a scenario was built towards supporting a quality attribute (Adaptivity), which characterizes the concern of the context. Additionally, the system and the application domain were identified from the analysis of the documents available in (GARCÉS, 2018) and (VICENTE *et al.*, 2017), and that show the application domain as being Healthcare Supportive Home Systems (HSH). The documents also present a software architecture; therefore, the software system in question was considered an instance of DiaManT@Home.

The second group of concepts mapped between the model and the architecture was "Dynamic Concepts", according to Table 5. The scenario used for this example, shown in Table 3, was also sufficient for the identification of dynamic concepts, since it presents "Why", "When" and "How" an architecture reconfiguration occurs. This information promoted a complete understanding of the most relevant information for decision-making.

No concept was identified to be mapped in Trade-Offs and Autonomy Level, since such information was not explicit in the sources. Trade-offs are exchanges necessary for the achievement of dynamic capabilities. According to the way the system was structured,

we can infer Adaptivity has a direct trade-off with performance, since the system must manage the entire reconfiguration.

The approach adopted in the level of autonomy is not clear; however, [Vicente et al. \(2017\)](#) used MAPE-K, an architectural standard widely employed for dynamic solutions, as a solution for Adaptivity. According to [Müller et al. \(2009\)](#), MAPE-K extends the system's autonomy, according to its five elements, which enable decision-making regarding adaptations. Therefore, we can infer the system under study has a high level of autonomy.

MAPE-K is a widespread standard for the implementation of dynamic solutions in architectures, and consists of four components, namely **M**onitor, **A**nalyze, **P**lanning, **E**xecuting, and **K**nowledge. In this scenario, the pattern displays several characteristics that enable the implementation of dynamic features in the architecture, which are covered by the conceptual model. Furthermore, MAPE-K was used to manage the adaptation of the architecture, as show in [Table 6](#), which presents the mapping between the concepts of the model and the analyzed architecture.

From [Table 6](#), the architectural elements (extracted from the analysis of [Figures 16](#) and [17](#)) involved in the reconfiguration are defined. The models present the components of the architecture structure and, together with the description of those models (available in ([GARCÉS, 2018](#))) enable measurements of their function in the architecture.

When analyzing [Figure 16](#), we seek to identify which components were part of MAPE-K. It was noted that the architects represented architectural elements following this standard, including the nomenclature that indicates the components of MAPE-K, what is explicit in [Table 6](#). Furthermore, in [Figure 17](#) shows a layer for the quality of service, where services that manage reconfigurations are described. The layer also displays other elements that comprise the reconfiguration manager.

The last set of concepts analyzed was "Architectural Concepts", shown in [Table 7](#). Such concepts are important for the understanding of the architectural elements used for solving certain situations that demand dynamism. Data were extracted from this set by the models, towards the identification of where the reconfiguration policies would be applied and which architectural elements was included in it.

When new services needed to be instantiated, the architectural configuration changed was the set of Constituent System Services, as depicted in [Figure 17](#). This part of the architecture is directly modified when new services are integrated, thus characterizing the configuration of the architecture. Constituents are the sensors and services that provide data to the system, and can be understood as the architectural elements that comprise the service layer and that must be somehow represented. In this example, the Model Kind chosen was SysML.

Table 5 – Mapping of Dynamic Concepts

Concept	Concept On DiaManT@Home	Rationale
Reconfiguration Policy	New patient care plan	According to the Adaptivity scenario, new services need to be added to the system when a new care plan is inserted. Therefore, we can consider this new plan as the Reconfiguration Policy, which defines why the system needs to be reconfigured.
Reconfiguration Triggers	Integration of new service	When a new service is needed, the system needs to take a series of steps to discover and integrate this new service. Thus, the need for a new service starts the reconfiguration, so it can be considered a trigger.
Reconfiguration Strategies	Integrate new service	The scenario describes what needs to happen when a new service needs to be integrated into the system. Even at a high level, the integration of a new service (Architectural decision(s), in the scenario), defines this step-by-step to perform this. Other steps could be defined, such as the redirection of information, other services that would need to be stopped, among others. However, for this example, the walkthrough presented is sufficient.
Trade-Offs	-	No trade-off was considered in information about the scenario. This type of change would probably impact the performance of the system, for example, since more processing would be needed to manage the integration of the new service, while the others are running. However, it is not possible to state this precisely, since it was not described in the sources used for the elaboration of the example.
Autonomy Level	-	The level of autonomy of the system has also not been explicitly defined. However, it is possible to measure that it is “Self- *” level, since it implements the MAPE-K standard, and this pattern has a database to provide a certain level of smart to the system.

Source: Elaborated by the author.

5.4 Description of DiaManT@Home

This section describes DiaManT@Home with the use of the set of views, displayed in the next sections.

Table 6 – Mapping of MAPE-K Concepts

Concept	Concept On DiaManT@Home	Rationale
Reconfiguration Management	Quality Service Layer	It was explained, in the description of the architecture, that MAPE-K was implemented to provide the possibility of architectural adaptation. Therefore, the Quality Service Layer is responsible for defining how adaptations should be made, in addition managing and triggering reconfigurations.
Planner	plannerService	Implements the MAPE-K Planner function.
Execute	qualityPlanExecuting	Implements the MAPE-K Execute function
Analyze	analyserService	Implements the MAPE-K Monitor function
Monitor	qualityMonitoring	Implements the MAPE-K Analyze function
Knowledge Base	qualityPlanRepository	The quality plans, that is, the reconfiguration needs, are stored in a repository, so it is understood as a database that can be consulted to improve the solutions.

Source: Elaborated by the author.

Table 7 – Mapping of Architectural Concepts

Concept	Concept On DiaManT@Home	Rationale
Architectural Configuration	Services Layer	The Services Layer is the architectural configuration, that presents the set of available services in the system. Therefore, when a new service is inserted, this layer is updated, and its configuration changes.
Architectural Element	Constituent Service	The elements that make up the Services Layer are the Constituent Services, which are the services available to the system.
Software Element	Service	The software elements are the available services.
Hardware Element	Sensor	The sensors that integrate the system constitute the hardware elements of the architecture.
Model Kind	SysML	To represent the elements that make up the architecture, the SysML modeling language was used, which characterizes Model Kind.

Source: Elaborated by the author.

5.4.1 Scenario View

The first view built was Scenario View, which enables evaluations of the reconfiguration elements, such as Reconfiguration Triggers and Quality Attributes involved. Each of the elements that comprise Scenario Views is represented below:

- **Source of stimulus (Why):** In this example, the source of stimulus for reconfiguration is a new treatment policy for the patient, presented as one of the requirements that demand architectural reconfigurations (VICENTE *et al.*, 2017). Architecturally, the quality attributes involved in the reconfiguration are Adaptability and Reliability, since such changes must be performed with no interruption of the system, towards ensuring a constant acquisition of patient data. Since new policies must be inserted, the architecture needs to change in run-time, so a dynamic solution is needed;
- **Stimulus (When):** the request for a new service triggers a system reconfiguration. Therefore, we can infer that there are two ways for reconfiguration:
  - *Reactive*: the system is not reconfigured immediately after a new care plan has been inserted. Since no service provides the new data requested by the system, a new service must be integrated; and
  - *Proactive*: When a new care plan is inserted, the new data necessary are automatically mapped, and the reconfiguration is performed.

This example considers the Reactive reconfiguration of the system, i.e., the Reconfiguration Trigger will be the request for new data by the system;

- **Environment (Where):** The environment encompasses the software elements that are part of the reconfiguration and is represented by the Reconfiguration View, displayed in Figures 19 and 20;
- **Response (How/Who):** This section includes the Strategy View and Ripple-Effect View; and
- **Response Measure (What):** In addition to the Ripple-Effect View, the trade-off between reliability and performance was identified. Since more processes must be conducted for the insertion of a new service at run-time, the system's performance may be compromised. Consequently, the necessary dynamic solution requires a reduction in such performance (even if only during reconfiguration).

5.4.2 General View

General View presents the system modules as a whole, i.e., the way they are related and organized (in layers, for example). As the system presents a General View, we chose not

to rewrite it, since it meets the requirements that we identified as important for this view. Figure 17 illustrates the General View of the Architecture.

5.4.3 Reconfiguration View

Reconfiguration View presents the elements of the system that are part of the reconfiguration, i.e., before and after the reconfiguration of the system. Figure 19, displays the architecture prior to reconfiguration, consequently, HSB(Home Service Bus) is monitored by collectorService, which detects if data required are not available. Moreover, plannerService and analyserService use data from GovernanceLayer to improve the insertion of new services.

Figure 19 – Proof of Concept - Reconfiguration View

Source: Elaborated by the author.

In Figure 20, after executing the strategy (Figure 21), it is noted that in addition to the *serviceD* being available, the reconfiguration data is saved in *GovernanceLayer*, so that they can be reused in the next necessary reconfiguration.

5.4.4 Strategy View

Strategy View presents the steps to be taken by the system for the integration of a new service, thus categorizing a reconfiguration of the architecture. Figure 21 illustrates the way the software elements interrelate to achieve the reconfiguration. The architectural decision for the solution was the implementation of MAPE-K, therefore, the Reconfiguration Manager uses its four elements.

Figure 20 – Proof of Concept - Reconfiguration View (Result)

Source: Elaborated by the author.

Figure 21 – Proof of Concept - Strategy View

Source: Elaborated by the author.

Additionally, Figure 16 shows the behavior of Health Service Bus in function of the integration of new services, regarding the mediators involved.

5.4.5 Ripple-Effect View

In Ripple-Effect View we have to identify the possible impacts of the architectural reconfiguration. In our proof of concept, we consider the insertion of a new service, thus, we may have some impacts related to data integration. Besides that, during insertion, processes can be created, making *Health Service Bus* slower due to this integration.

Figure 17 depicts the parts of the architecture that are related to the elements to be reconfigured. A variation of the model can be understood as a possible visualization of the Ripple-effect. In this view, both Services layer and Quality of Service layer should be represented, since they are related, and may be affected by the integration of a new service.

5.5 Final Remarks

This chapter presented a proof of concept regarding the use of the architectural views, the conceptual model, and associated techniques for the representation of a dynamic architecture. A software architecture for healthcare domain, called DiaManT@Home, was considered for the proof.

Although the representation of the architecture converged to what the literature points (and which is proposed in the set of views), some points could be improved with the use of views and the conceptual model, such as (i) use of trade-offs with the reconfiguration processes, (ii) use of the reconfiguration view for the understanding of the elements to be present before and after reconfiguration, and (iii) improvement in the representation of the reconfiguration data flow by Strategy view.

The proof of concepts enables measurements of improvements in the representation of architecture, especially concerning the reconfiguration process, and its reasons and effects. However, the views showed redundancy, since they present the same elements more than once, even if at different instants. Such an increased number of generated models may increase the documentation size, although decreasing complexity.

Conclusions

Today's software systems are increasingly demanding mechanisms for reliability and availability since many of them are inserted in critical domains. Such reliability and availability impose on architects the need to model architectures that are flexible and subject to change even at run-time. In this scenario, the concept of dynamism has been studied by the software architecture community for over 20 years. The representation of dynamic architectures characterizes a relevant and well-explored topic; however, despite different solutions, the representation of these architectures is still quite costly, especially due to the use of formal notations. Furthermore, the completeness of the representation have not being explored in literature i.e., there are no approaches to help architects in the analysis and representation approaches, then, some relevant information could be missed.

Formal notations are applied to guarantee the reliability of the architectures; however, they are very complex to be learned and applied. Considering this, this Master's work proposed the use of a set of architecture views that allows representing the main concepts related to dynamic software architectures using semi-formal notations, which usually reduce the learning curve and, consequently, the time required for the modeling of architectures. For the elaboration of architectural views, a conceptual model of dynamic architectures was proposed. In this model, the main concepts of these architectures were included so that the set of views can represent them.

The use of architectural views is one of the main practices used for the representation of software architectures, since they allow the separation of interests and, thus, they facilitate the analysis of architectural decisions considering certain perspectives of the system. In the context of dynamic architectures, the use of a set of views allows, in addition to this analysis, to increase the reliability of the solutions, allowing the architect to analyze the architecture in different states, ensuring the consistency of one model in relation to the others. In the same hand, the application of this set of views improves the completeness of the model, covering all relevant concepts showed in conceptual model.

In the proof of concept, that used the conceptual model and the architecture views, we found that it is possible to identify important information about the architecture (for instance, the quality attribute related to reconfiguration need, the possible trade-offs, among others) and provides a completeness representation of the architecture.

6.1 Solutions Proposed

We highlighted four problems considering dynamic software architectures and, specially, their representations. In this scenario, we propose some solution that were able to solve those problems. Thus, each problem and the related solution are related as follows:

- **(i) dynamic software architectures are, in their essence, abstract and complex to understand (HASSAN *et al.*, 2016):** to solve this problem, we proposed a set of views that can represent different aspects of the architecture, allowing an complete view of the components and the reconfiguration process;
- **(ii) many concerns have to be considered in their representation e.g., what components must be stopped or replaced:** to cover this problem, a conceptual model was proposed to guide the analysis and representation of these many concerns;
- **(iii) using formal notations ensures the reliability of software architectures, but increases the costs:** to handle with complexity of formal notations we proposed to represent our set of views using semi-formal notations. In addition, the five proposed views covers all concepts defined in conceptual model, hence, this completeness improves the reliability of the representation; and
- **(iv) without any directions concerning about "what" must to be represented, architects can forget some relevant concepts, what could decrease architecture's quality:** this problem was solved with the conceptual model, which relate all necessary concepts which must be considered in dynamic software architectures' design. In addition, the views cover all these concepts, what maintains the quality of the architecture.

6.2 Research Contributions

The main research contributions of this Master's project are:

- **Characterization of the state of the art on Dynamic Software Architectures Representation:** by conducting the SMS presented in Chapter 2, we pro-

vided a view of the state of the art about dynamic software architecture representation, showing the main notations and problems related to this topic. Particularly, the results showed how formal languages improve the reliability, but also increase the costs of the software projects. Moreover, it was also showed that semi-formal languages are still unexplored to the representation of dynamic software architectures;

- **Establishment of a Conceptual Model for Dynamic Software Architectures:** The conceptual model shows the main concepts related to dynamic software architectures, evidencing "*what*" must to be considered and represented in dynamic software architecture's design; and
- **Establishment of a set of Architectural Views for Dynamic Software Architecture:** Using the conceptual model as basis, the set of views proposed allows architects to represent dynamic software architectures using semi-formal notations. By using these notations, costs and time spent in projects may be reduced while quality could be kept.

6.3 Research Limitations

The main limitations of this Master's project are:

- **Implementation Details:** We introduce several concepts of dynamic architectures. However, we do not explore how these structures can be implemented and applied in different scenarios;
- **Experimental Studies:** Despite the proof of concept explores the feasibility of using the proposed architecture views, it is necessary to better explore their use, especially with the conduction of experiments and surveys with specialists; and
- **Types of Reconfiguration Support:** In Chapter 2, we present the main types of reconfiguration that are found in the literature. However, neither the conceptual model nor the architecture views show how these types can be supported.

6.4 Future Work

During the process defining the conceptual model and the set of architecture views, possible future works were identified, as follows:

- Conduct an assessment of the views with experts to obtain their opinion on the usability and completeness of the models;

- Develop and consolidate processes, metrics, and guidelines for the application of the model and proposed views;
- Identify, analyze, and catalog reconfiguration strategies;
- Analyze the different degrees of abstraction that should be considered in Reference Architectures so that models and views can be applied in this domain;
- Identify how different types of reconfiguration are supported by the views;
- Identify the demand for Industry 4.0 regarding the need for dynamic solutions;
- Propose and develop semi-formal representation methods that can be simulated and validated formally (Translation Engines); and
- Identify how Crowdsourcing can support the identification of strategies and architectural patterns for dynamic software architectures.

References

- ADJOYAN, S.; SERIAI, A.-D. Reconfigurable service-based architecture based on variability description. In: **Proceedings of the Symposium on Applied Computing**. 2017. p. 1154–1161. Citations on pages [31](#) and [43](#).
- ALLEN, R.; DOUENCE, R.; GARLAN, D. Specifying and analyzing dynamic software architectures. In: **Fundamental Approaches to Software Engineering**. 1998. p. 21–37. Citations on pages [1](#), [5](#), [7](#), [10](#), [16](#), [19](#), [25](#), [31](#), [33](#), [34](#), [42](#), and [84](#).
- ANGELOV, S.; GREFEN, P.; GREEFHORST, D. A classification of software reference architectures: Analyzing their success and effectiveness. In: **European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on Software Architecture**. 2009. p. 141–150. Citation on page [22](#).
- BASS, L. **Software architecture in practice**. 3rd. ed. : Addison-Wesley, 2013. Citations on pages [1](#), [2](#), [5](#), [30](#), [33](#), [34](#), [37](#), [42](#), [43](#), [45](#), and [57](#).
- BEEK, M. H. T.; BUCCHIARONE, A.; GNESI, S. Dynamic software architecture development: Towards an automated process. In: **Euromicro Conference on Software Engineering and Advanced Applications**. 2009. p. 105–108. Citation on page [21](#).
- BRADBURY, J. **Organizing definitions and formalisms for dynamic software architectures**. 2004. 1–39 p. Citations on pages [6](#), [8](#), and [43](#).
- BRADBURY, J. S.; CORDY, J. R.; DINGEL, J.; WERMELINGER, M. A survey of self-management in dynamic software architecture specifications. In: **Proceedings of Workshop on Self-Managed Systems**. 2004. p. 28–33. Citations on pages [2](#), [6](#), [30](#), [31](#), and [33](#).
- BRUNI, R.; BUCCHIARONE, A.; GNESI, S.; MELGRATTI, H. Modelling dynamic software architectures using typed graph grammars. **Electronic Notes in Theoretical Computer Science**, p. 39 – 53, 2008. Citations on pages [29](#) and [84](#).
- BUCKLEY, J.; MENS, T.; ZENGER, M.; RASHID, A.; KNIESEL, G. Towards a taxonomy of software change: Research articles. **Journal Of Software Maintenance and Evolution: Research and Practice**, p. 309–332, 2005. Citation on page [10](#).
- CAVALCANTE, E.; BATISTA, T.; OQUENDO, F. Supporting dynamic software architectures: From architectural description to implementation. In: **Working IEEE/IFIP Conference on Software Architecture**. 2015. p. 31–40. Citations on pages [2](#) and [45](#).
- CAVALCANTE, E. R. de S. **A Formally Founded Framework for Dynamic Software Architectures**. Phd Thesis (PhD Thesis) — Federal University of Rio Grande do Norte (UFRN), Brazil and University of Southern Brittany, France, 2016. Citations on pages [6](#), [7](#), [8](#), [31](#), [32](#), and [33](#).

CHEN, L.; HUANG, L.; LI, C.; RUAN, M. Architecture reconfiguration in pervasive environment: An aadl-based approach. In: **International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing**. 2014. p. 23–30. Citation on page 16.

CIMATTI, A.; STOJIC, I.; TONETTA, S. Formal specification and verification of dynamic parametrized architectures. In: HAVELUND, K.; PELESKA, J.; ROSCOE, B.; VINK, E. de (Ed.). **International Symposium on Formal Methods**. 2018. p. 625–644. Citations on pages 17, 19, 31, 34, and 43.

CÎMPAN, S.; LEYMONERIE, F.; OQUENDO, F. Handling dynamic behaviour in software architectures. In: **Proceedings of the 2Nd European Conference on Software Architecture**. 2005. p. 77–93. Citations on pages 6 and 43.

Clements, P.; Bass, L. The business goals viewpoint. **IEEE Software**, p. 38–45, 2010. Citation on page 1.

CLEMENTS, P.; GARLAND, D.; BACHMANN, F.; IVERS, J.; STAFFORD, J.; BASS, L.; MERSON, P. **Documenting Software Architectures: Views and Beyond**. 2nd. ed. : Addison-Wesley Professional, 2010. Citations on pages 2, 37, 40, and 45.

Costa-Soria, C.; Pérez, J.; Carrión, J. Handling the dynamic reconfiguration of software architectures using aspects. In: **2009 13th European Conference on Software Maintenance and Reengineering**. 2009. p. 263–266. Citation on page 8.

DING, J. An approach for modeling and analyzing dynamic software architectures. In: **International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery**. 2016. p. 2086–2092. Citations on pages 1, 2, 10, 15, 25, 30, 31, and 34.

DJOUDI, B.; BOUANAKA, C.; ZEGHIB, N. On specifying reconfigurable component-based systems using strategies. In: **International Conference on Computational Science and Its Applications**. 2014. p. 656–670. Citations on pages 17, 18, and 30.

ENDLER, M.; WEI, J. Programming generic dynamic reconfigurations for distributed applications. In: **International Workshop on Configurable Distributed Systems**. 1992. p. 68–79. Citations on pages 6 and 7.

F. Oquendo. Dynamic software architectures: Formally modelling structure and behaviour with pi-adl. In: **International Conference on Software Engineering Advances**. 2008. p. 352–359. Citations on pages 19, 30, 32, and 84.

FU, X.; DA, X.; YU, Z. Modeling dynamic software architecture based on pi-net. In: **Proceedings of 2nd International Conference on Information Communication Technologies**. 2006. p. 2861–2865. Citations on pages 1 and 5.

GARCÉS, L. **A Reference Architecture for Healthcare Supportive Home (HSH) systems**. Phd Thesis (PhD Thesis) — Univesity of São Paulo (USP), Brazil, 2018. Citations on pages 32, 56, 57, 58, 59, 60, 63, and 64.

GARCÉS, L.; NAKAGAWA, E. Y. A process to establish, model and validate missions of systems-of-systems in reference architectures. In: **Proceedings of Symposium on Applied Computing**. 2017. p. 1–8. Citation on page 56.

GARLAN, D. Software architecture: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. 2000. p. 91–101. Citation on page 1.

GARLAN, D.; ALLEN, R.; OCKERBLOOM, J. Architectural mismatch or why it's hard to build systems out of existing parts. In: IEEE. **International Conference on Software Engineering (ICSE)**. Seattle, Washington, USA, 1995. p. 179–179. Citations on pages 42, 43, and 45.

HAN, T.; CHEN, T.; LU, J. Structure analysis for dynamic software architecture. In: **International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing**. 2005. p. 338–348. Citation on page 2.

HASSAN, A.; QUEUDET, A.; OUSSALAH, M. Evolution style: Framework for dynamic evolution of real-time software architecture. In: **European Conference on Software Architecture**. 2016. p. 166–174. Citations on pages 3, 9, 10, 21, and 74.

HIRSCH, D.; MONTANARI, U. Two graph-based techniques for software architecture reconfiguration. **Electronic Notes in Theoretical Computer Science**, p. 177 – 190, 2002. Citations on pages 15 and 19.

HUSSEIN, M.; NOUACER, R.; RADERMACHER, A. Safe adaptation of vehicle software systems. **Microprocessors and Microsystems**, p. 272 – 286, 2017. Citations on pages 3, 18, 20, and 27.

HUYNH, N.; SEGARRA, M.; BEUGNARD, A. Ensuring consistent dynamic adaptation: An approach from design to runtime. In: **Proceedings of International Conference of Computer Systems and Applications**. 2016. p. 1–8. Citation on page 20.

IBM. **An architectural blueprint for autonomic computing**. 2003. 1–34 p. Citations on pages 31, 32, and 33.

ISO/IEC/IEEE. Iso/iec/ieee-42010 - systems and software engineering–architecture description. **ISO/IEC/IEEE 42010:2011**, p. 1–46, 2011. Citations on pages 33, 34, 38, and 42.

KACEM, M. H.; JMAIEL, M.; KACEM, A. H.; DRIRA, K. Using uml 2.0 and gg for describing the dynamic of software architectures. In: **International Conference on Information Technology and Applications**. 2005. p. 46–51. Citation on page 84.

KACEM, M. H.; KACEM, A. H.; JMAIEL, M.; DRIRA, K. Describing dynamic software architectures using an extended uml model. In: **Proceedings of ACM Symposium on Applied Computing**. 2006. p. 1245–1249. Citations on pages 2, 30, 33, and 34.

KALLEL, S.; KACEM, M. H.; JMAIEL, M. Modeling and enforcing invariants of dynamic software architectures. **Software & Systems Modeling**, p. 127–149, 2012. Citations on pages 2 and 27.

KAZMAN, R.; KLEIN, M.; BARBACCI, M.; LONGSTAFF, T.; LIPSON, H.; CARRIERE, J. The architecture tradeoff analysis method. In: IEEE. **IEEE International Conference on Engineering of Complex Computer Systems**. 1998. p. 68–78. Citation on page 30.

KAZMAN, R.; SCHMID, K.; NIELSEN, C. B.; KLEIN, J. Understanding patterns for system of systems integration. In: **International Conference on System of Systems Engineering**. 2013. p. 141–146. Citation on page [33](#).

KITCHENHAM B. A.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in software engineering**. 2007. 1–39 p. Citations on pages [11](#), [26](#), [83](#), and [84](#).

LEMOS, R. de; GIESE, H.; MÜLLER, H. A.; SHAW, M.; ANDERSSON, J.; LITOIU, M.; SCHMERL, B.; TAMURA, G.; VILLEGAS, N. M.; VOGEL, T.; WEYNS, D.; BARESI, L.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CUKIC, B.; DESMARAI, R.; DUSTDAR, S.; ENGELS, G.; GEIHS, K.; GÖSCHKA, K. M.; GORLA, A.; GRASSI, V.; INVERARDI, P.; KARSAL, G.; KRAMER, J.; LOPES, A.; MAGEE, J.; MALEK, S.; MANKOVSKII, S.; MIRANDOLA, R.; MYLOPOULOS, J.; NIERSTRASZ, O.; PEZZÈ, M.; PREHOFER, C.; SCHÄFER, W.; SCHLICHTING, R.; SMITH, D. B.; SOUSA, J. P.; TAHVILDARI, L.; WONG, K.; WUTTKE, J. Software engineering for self-adaptive systems: A second research roadmap. In: _____. **Software Engineering for Self-Adaptive Systems II**. 2013. p. 1–32. Citation on page [32](#).

LI, C.; HUANG, L.; CHEN, L.; LI, X.; LUO, W. Dynamic software architectures: Formal specification and verification with csp. In: **Proceedings of Asia-Pacific Symposium on Internetware**. 2012. p. 1–8. Citations on pages [2](#) and [6](#).

LING, X. A categorical approach for modeling and verifying dynamic software architecture. In: **IEEE International Conference on Software Security and Reliability Companion**. 2013. p. 168–175. Citation on page [30](#).

LOULOU, I.; KACEM, A. H.; DRIRA, K.; JMAIEL, M. Towards a unified graph-based framework for dynamic component-based architectures description in z. In: **International Conference on Pervasive Services**. 2004. p. 227–234. Citations on pages [6](#) and [29](#).

MADNI, A. M.; SIEVERS, M. System of systems integration: Key considerations and challenges. **Systems Engineering**, p. 330–347, 2014. Citations on pages [30](#), [31](#), and [33](#).

MAGEE, J.; DULAY, N.; EISENBACH, S.; KRAMER, J. Specifying distributed software architectures. In: **European Software Engineering Conference**. 1995. p. 137–153. Citation on page [19](#).

MILADI, M. N.; KACEM, M. H.; BOUKHRIS, A.; JMAIEL, M.; DRIRA, K. A uml rule-based approach for describing and checking dynamic software architectures. In: **2008 IEEE/ACS International Conference on Computer Systems and Applications**. 2008. p. 1107–1114. Citations on pages [2](#), [17](#), [19](#), and [33](#).

MÜLLER, H. A.; KIENLE, H. M.; STEGE, U. Autonomic computing now you see it, now you don't. In: LUCIA, A. D.; FERRUCCI, F. (Ed.). **Software Engineering: International Summer Schools**. 2009. p. 32–54. Citation on page [64](#).

NAKAGAWA, E. Y.; GUESSI, M.; MALDONADO, J. C.; FEITOSA, D.; OQUENDO, F. Consolidating a process for the design, representation, and evaluation of reference architectures. In: **IEEE/IFIP Conference on Software Architecture**. 2014. p. 143–152. Citation on page [29](#).

- OQUENDO, F. π ADL: An Architecture Description Language Based on the Higher-order Typed π -calculus for Specifying Dynamic and Mobile Software Architectures. **Software Engineering Notes**, p. 1–14, 2004. Citations on pages 17, 31, 32, and 34.
- OQUENDO, F. Formally describing the software architecture of systems-of-systems with sosadl. In: **Proceedings of System of Systems Engineering Conference**. 2016. p. 1–6. Citations on pages 2, 16, 19, 30, 32, 33, 45, and 84.
- OREIZY, P. Issues in modeling and analyzing dynamic software architectures. 1998. Citations on pages 9, 30, 31, and 34.
- OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. N. Runtime software adaptation: Framework, approaches, and styles. In: **Proceedings of International Conference on Software Engineering**. 2008. p. 899–910. Citations on pages 10, 33, and 34.
- PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and Software Technology**, p. 1 – 18, 2015. Citations on pages 11, 26, 83, and 86.
- PORTOCARRERO, J.; DELICATO, F.; PIRES, P.; COSTA, B.; LI, W.; SI, W.; ZOMAYA, A. Ramses: A new reference architecture for self-adaptive middleware in wireless sensor networks. **Ad Hoc Networks**, p. 3–27, 2016. Citations on pages 27 and 30.
- SMAALI, S.; CHOUTRI, A.; BELALA, F. K-maude definition of dynamic software architecture. In: **International Conference on Advanced Aspects of Software Engineering**. 2014. p. 92–99. Citations on pages 2 and 21.
- TEKINERDOGAN, B.; SOZER, H. Chapter 4 - an architecture viewpoint for modeling dynamically configurable software systems. In: MISTRIK, I.; ALI, N.; KAZMAN, R.; GRUNDY, J.; SCHMERL, B. (Ed.). **Managing Trade-Offs in Adaptable Software Architectures**. 2017. p. 79 – 97. Citation on page 37.
- TORKAR, R. **Dynamic Software Architectures**. 2002. 1–40 p. Citation on page 6.
- VICENTE, I. Z.; RODRÍGUEZ, L. M. G. 'es; NAKAGAWA, E. Y. **Establishment Of A Software Architecture For Health Care Supportive Home System To Assist Patients With Diabetes Mellitus: Functional And Non-functional Requirements**. 2017. 1–41 p. Citations on pages 35, 55, 56, 63, 64, and 67.
- WERMELINGER, M. A simple description language for dynamic architectures. In: **Proceedings of International Workshop on Software Architecture**. 1998. p. 159–162. Citation on page 18.
- WERMELINGER, M.; FIADEIRO, J. L. Algebraic software architecture reconfiguration. **Software Engineering Notes**, p. 393–409, 1999. Citations on pages 1, 6, and 34.
- WERMELINGER, M.; FIADEIRO, J. 'e L. A graph transformation approach to software architecture reconfiguration. **Science of Computer Programming**, p. 133 – 155, 2002. Citations on pages 19 and 30.
- WEYNS, D.; CAPORUSCIO, M.; VOGEL, B.; KURTI, A. Design for sustainability = runtime adaptation and evolution. In: **Proceedings of European Conference on Software Architecture Workshops**. 2015. p. 1–7. Citations on pages 1, 5, and 44.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. **International Conference on Evaluation and Assessment in Software Engineering**, p. 1–10, 2014. Citation on page [26](#).

XU, H.; ZENG, G.; CHEN, B. Description and verification of dynamic software architectures for distributed systems. **Journal of Software**, p. 721–728, 2010. Citation on page [5](#).

ZHANG, Q.; WANG, S.; LIU, B. Approach for integrated modular avionics reconfiguration modelling and reliability analysis based on aadl. **IET Software**, p. 18–25, 2016. Citations on pages [16](#), [19](#), [30](#), and [31](#).

ZHAO, Y.; MA, D.; LIU, M.; HU, C.; HUANG, Y. A graph transformation based approach for runtime constrained evolution of service-oriented architectures. In: **Euromicro International Conference on Parallel, Distributed and Network-based Processing**. 2009. p. 203–209. Citation on page [15](#).

ZHAO, Z.; LI, W. Specifying dynamic software architectures with dynamic description logic. **Journal of Software**, p. 169–175, 2012. Citation on page [31](#).

ZHENHUA, Y.; YUANLI, C. Novel architecture description language based on high-level petri nets. In: **Proceedings of International Conference on Information and Communication Technologies: From Theory to Applications**. 2004. p. 589–590. Citation on page [6](#).

Dynamic software architectures representation: a systematic mapping study protocol

---

In this appendix, we discuss the planning and execution of the SMS. The research method adopted in this study follows the guidelines proposed by [KITCHENHAM B. A.; CHARTERS \(2007\)](#) and [Petersen *et al.* \(2015\)](#). In short, this process presents three main phases: (i) planning: in this phase, the research objectives and the systematic review protocol are defined. The protocol constitutes a pre-determined plan that describes research questions and how the SMS will be conducted; (ii) conduction: during this phase, primary studies are identified, selected, and evaluated according to the inclusion and exclusion criteria. For each selected study, data are extracted and synthesized; and (iii) reporting: in this phase, a final report is organized and presented. As the results of this SMS were already reported in Chapter 2, this appendix describes the first and second phases in more details.

A.1 Planning

A.1.1 Research Objectives

This SMS aims to present a first overview of the state of the art of how dynamic software architectures have been represented. We are focused on understanding what are the main approaches and languages that are applied in this activity. More over, we intend to analyze how formalism level of these approaches/languages impacts on the representation process. Finally we aim to understand how the approaches can help in solving some issues related to dynamic software architectures.

A.1.2 Research Questions

Aiming to support the research objective, three research questions (RQs) were created:

- RQ1:** Which approaches have been used to represent dynamic software architectures?
- RQ2:** Which the level of formalism of these approaches are?
- RQ3:** How can the representation approaches solve dynamic software architectures issues?

A.1.3 Search Strategy

Aiming to perform the search to answer our RQs, we defined a search string including terms from the research objective, which is the primary purpose of this mapping; and terms from research questions, as suggested by Kitchenham et al (KITCHENHAM B. A.; CHARTERS, 2007). Following this process, we defined the the follow search string:

("software architecture" OR "reference architecture") AND ("dynamic" OR "dynamism" OR "adaptive" OR "reconfigura*") AND ("model*" OR "design" OR "notation" OR "represent*" OR "language").

Our main objective was identify all possible approaches and languages able to represent dynamic software architectures. For this purpose, we select terms related to software architecture, dynamism, and representation.

We performed searches for primary studies using electronic search engines and digital libraries recommended to software engineering research. In particular, we adopted those ones presented in Table 8, chosen for being the most relevant sources in the computer science area (KITCHENHAM B. A.; CHARTERS, 2007). The search string described previously presented was used against metadata (titles, keywords, and abstracts).

Table 8 – Databases used in SMS

Database	URL
ACM Digital Library	http://portal.acm.org
IEEE Digital Library	http://ieeexplore.ieee.org
ISI Web of Science	http://www.isiknowledge.com
Science@Direct	http://www.sciencedirect.com
Scopus	http://www.scopus.com

We had a control group of studies that were previously analyzed and considered essential to this study. Thus, these works were used to validate our search string including the following studies: Allen *et al.* (1998), Kacem *et al.* (2005), F. Oquendo (2008), Bruni *et al.* (2008), Oquendo (2016). This control group was used against our search string to verify if it was properly defined, i.e., whether our string was able to find these studies in the search sources. All these studies were returned using our search string. Hence, we can say that the string is representative to to cover the relevant studies related to dynamic software architectures representation.

A.1.4 Selection Criteria

The selection criteria were used to assess each primary study obtained from the search databases. It allowed us to include relevant studies to answer our RQs, and to exclude non-relevant studies. The inclusion criteria (IC) of our SMS are:

IC1 - The primary study should show an approach to describe dynamic software architectures.

IC2 - The primary study should show an language/notation to describe dynamic software architectures.

As well, the following indicate the exclusion criteria (EC).

EC1 - The full text of the primary study is not available.

EC2 - The primary study is a table of contents, short course description, or summary of a conference/workshop.

EC3 - The primary study is out of scope of our SMS.

EC4 - The primary study is written in a language other than English.

EC5 - The study is a previous version of a more complete one on the same research or of the same authors.

A.2 Conduction

Our SMS was conducted from March to November 2018. During the conduction phase, primary studies were identified, selected, and evaluated by the author using the search strategy previously defined. For each selected study, data were extracted and synthesized. To support the management of the primary studies, we used Parsif.al¹. Figure 22 shows the results for each step described below.

Figure 22 – SMS Steps

¹ Parsifal is a reference management tool that allows storing information on the primary studies (e.g., title, authors, venue, and abstract), as well as the set the inclusion/exclusion criteria applied to select each primary study. Available in: <https://parsif.al/>.

- **Recovery of Studies and Exclusion of Duplicates:** The search string was applied to the selected databases. As a result, we found 7595 studies. From these studies, 2519 were duplicated and, therefore, were removed. In this activity, 5076 primary studies were selected for the next step.
- **First Selection:** We read the title, abstract and keywords of each one of the 5076 primary studies and applied the inclusion and exclusion criteria. When the abstract did not provide enough information, we also read the introduction and the conclusion sections of each study. As a result, we selected a set of 580 potentially relevant studies.
- **Second selection:** We performed the reading of introduction and conclusion of each one of 580 selected studies. Such studies were analyzed again considering the inclusion and exclusion criteria. As a result, 50 primary studies were full reading and considered to answer the research questions.
- **Data Extraction:** To guide the extraction, we created a form using Google Sheets² with the following items: title of study, authors, bibliographic reference, language name, level of formalism and domain application. All these information was extracted of each primary study during the full reading.

A.3 Threats to validity

The main threats to validity of this SMS are described as follows:

Missing of important primary studies: A threat to the validity is that important studies were not found through the systematic search in databases. Furthermore, we followed a predefined protocol proposed by Petersen et al. (PETERSEN *et al.*, 2015) to avoid bias in the study search. Additionally, we searched in five most important digital databases in software engineering area aiming to find all studies available about representation of dynamic software architectures representation.

Study Selection: As the first author selected the studies by reading each one, it is possible that some studies were erroneously assessed. To mitigate this threat, a set of 100 studies were randomly chosen and the title, abstract, and meta data analysis were again analyzed. After that, the results of both analyses were compared and no difference between them was found.

Data Extraction: During data extraction, we created a data sheet to fill and save all answers from each study. However, since not all information was explicitly available in the studies, some data had to be interpreted. To minimize this threat, discussions among the authors were developed.

---

² <https://docs.google.com/spreadsheets>