Style and Meta-Style : Another way to reuse Software Architecture Evolution

Adel Hassan

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité: Génie logiciel

Par

« Adel HASSAN »

« Style and Meta-Style: Another Way to Reuse Software Architecture Evolution»

Thèse présentée et soutenue à « L2SN », le « 24 Septembre 2018 »
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)
Thèse N° :

Rapporteurs avant soutenance :

M. Kamel Barkaoui, Professeur, Le Cnam-Paris
M. Henri Basson, Professeur Université du Littoral Côte d'Opale

Composition du Jury:

Président
Mme Isabelle BORNE, Professeur, Université Bretagne Sud

M. Kamel BARKAOUI, Professeur, Le Cnam-Paris
M. Henri BASSON, Professeur Université du Littoral Côte d'Opale
M. Djamel SERIAI, Maître de conférences, HDR, Université de Montpellier
M. Christian ATTIOGBE, Professeur, Université de Nantes
Directeur de thèse
M. Mourad OUSSALAH, Professeur, Université de Nantes

Dedication

I dedicate this dissertation to my beloved parents, wife and kids

Adel

Acknowledgments

First of all, I would like to express my sincere gratitude and appreciation to those who over the years have played a role in the realisation of this thesis.

Many thanks go to Professor, Mourad Oussalah, the director of my thesis for his input and guidance during these years. Without his patience, support and encouragement, this thesis would not have been possible. I also want to thank Audrey Queudet for her guidance and her valuable comments that helped me in realising Chapter 5.

I am further grateful to my thesis monitoring committee, Professor Philippe Collet and Professor Kamel Barkaoui for all their guidance and feedback throughout this process.

I would also like to thank Professors. Isabelle Borne, Henri Basson, Djamel Seriai, Kamel Barkaoui and Christian Attiogbé for accepting membership of my Jury Committee during my viva voce examination.

My sincere thank also goes to all the wonderful staff of LS2N (LINA) for their hospitality and their technical and administrative help.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Context

Nowadays, information systems are becoming more and more complex and more widely distributed, with most being software-intensive systems. They often include large-scale heterogeneous distributed software components, embedded systems, telecommunications, wireless ad hoc systems etc. As markets and technology are continually changing, these systems also need to change in order to fulfill the new requirements of the market and technology. Whenever a system needs to be changed, the ideal starting point for the evolution team is to understand the system (design) and then to attempt to find a suitable set of modifications. In this context, planning the large-scale evolution is a highly challenging activity that requires an understanding of the overall system structure, consideration of pervious design decisions, and principled trade-offs among candidate evolution scenarios and selecting the optimal scenario (Barnes, Pandey, and Garlan 2013).

One of the main difficulties of software evolution lies in the fact that all artifacts produced and used during the entire software lifecycle are subject to changes (Buckley et al. 2005). Since software systems change fairly frequently, and all software artifacts are subject to change, it is essential that their architectures must be restructured. However, software architecture is the backbone of the software system (Taylor, Medvidovic, and Dashofy 2009) and plays a key role in guiding software development and evolution. Thus, architectural changes are inevitable for a software system in order to guide the evolution process, or prevent architectural drift and erosion, and maintain the system's consistency (to avoid inconsistency between the architecture and source code over time).

In this vein, the role of software architecture in the software evolution process can be considered from two points of view: as an artifact for evolution, which guides the planning and conducting of the evolution process, and as an artifact of the evolution, because it must be evolved itself in order to be consistent with system change (Cuesta et al. 2013).

In practice, the architecture of a software system is the first design artifact which captures those early architectural decisions which had a significant impact on the quality of the system. Whenever this system needs to change, it is important, if not essential, to understand these decisions as early as possible in the evolution process. However, architecture evolution is about making new design decisions, or removing obsolete ones, in order to embrace the new requirements. It concerns the changes applied to the architecture, including their components, connectors or configuration.

In fact, software architecture evolution is a very complex process to plan and thus the person or team who plays this role should have a blend of architectural knowledge from different fields including: business architecture, enterprise architecture, data architecture, application architecture and infrastructure or technical architecture.

Therefore, the architects are in needs of tools, methods and techniques that help them to prevent the evaporation of the architectural evolution knowledge (Oussalah et al. 2006). This is especially true in regard to the sharing and reuse of this knowledge by those architects who do not have such experience.

This thesis focuses on both software architecture evolution knowledge modeling and on reusing. It aims to explore the best techniques for capturing the architectural evolution knowledge and practices, allowing these to be demonstrated in fashions that can fit the different type of stakeholder involved in this process. This solution must take into account the existing evolution styles and standards and, thus, they can be reused and exploited.

The structure of this chapter is as follows: Section 1.1 introduces the context of this work, Section 1.2 elaborates on the questions that this work endeavours to address, Section 1.3 presents the research contribution, and Section 1.4 summarises the structure of the thesis.

1.2 Challenges

Reuse has formulated the optimal goal of several methodologies proposed and used in software product engineering. It varies from fine products, such as objects or methods, to very large and complex products, such as components or architecture styles. For software process engineering, especially here in the software architecture evolution process, the reused parts can be operations, activities, steps, a solution or even a set of possible solutions (to evolve the current architecture towards the intended target architecture).

Evolution style is one of software engineering approaches, the aim of which is to enhance reuse in the software architecture evolution knowledge. The main idea behind the evolution styles is to capture and share the best architectural evolution practices and knowledge relevant to a particular domain. To this end, they model the possible evolution scenarios that represent different ways of evolving this domain. These modeled scenarios can be grouped together into an evolution style that the architect can compare and analyse in order to plan and reason about software evolution. Several evolution styles have been introduced, each of which defines their own method and tool to depict the process using a relevant modelling technique (notions, syntax and semantic) in accordance with a certain point of view (Hassan and Oussalah 2016). Notwithstanding all these considerable achievements, some important issues related to standardisation and reuse among evolution styles themselves are still absent and need more exploration.

To address this limitation, this thesis considers the common existing approaches to both process and product modeling (MOF, UML, SPEM and Essence) and evolution styles (Hassan and Oussalah 2016), and provides a meta-style evolution (meta-meta-model) and methodology for software architecture evolution modeling. To this end, some issues have to be addressed:

— We need to define a meta-style evolution which can specify all necessary (meta-) elements (the meta-modeling language) that can instantiate the elements of the evolution styles.

— It is necessary to validate the ability of the meta-style evolution in terms of conforming to the different evolution styles and managing the mapping among these styles. Moreover, it is necessary to explore the extent to which this meta-style is equivalent to common standard models.
— We need to assess the extent to which this meta-style can be extended to embrace new aspects of the evolution process (e.g. dynamism) or a new mechanism of modeling this process (a new stakeholder perspective).
— The proposed meta-style and methodology should be implemented in order to explore the feasibility of the approach.

This thesis endeavours to explore all these concerns and to provide a framework that can support reuse in software architecture evolution.

1.3 Contribution

The main contributions of this thesis are:

Proposition of a meta-meta-model for software architecture evolution: We introduce The MES (meta-style evolution) component-oriented framework. It is a reflective methodology whereby the notion of the component reflects on the modeling of the process that evolves it. Thus, the component is the main entity of the MES model, as in the object-oriented modeling in which everything is subclass of the abstract class "ModelElement". Mapping procedures have been conducted with some standard models and evolution styles, the aim of which is to allow transformation between different evolution styles themselves as well as to the discipline of object modeling.

Extension of Meta-style evolution: We have extended the MES with some concepts that allow for modeling dynamic evolution in software architecture. We explore some issues that accompany dynamic evolution, such as: safe stopping, state transfer, change management and dynamic scheduling. This is done in order to annotate MES with the information required to fit the analysis and model the dynamic evolution of software architecture.

A methodology for multi-view/abstraction evolution style: We present a methodology in which the architect (modeler) of an evolution style can decompose an evolution style into different views and abstraction-levels. Several views of the process can be produced which cover different irrelevant perspectives separately and fit a wide range of stakeholders.

Implementation of a prototype tool supporting the proposed meta-style and methodology: In order to validate the applicability and feasibility of our approach, an initial tool prototype was developed, based on the conception of MES. To demonstrate this tool, a set of proposed views has been developed and integrated, in according to the relation notion introduced in MES.

1.4 Thesis Structure

The remainder of this document is organised as follows.

Chapter 2. Software architecture and software evolution: In this chapter, we present the background and the relevant realms of this work: Software Architecture (SA), Software Evolution (SE) and Architecture and the Software Architecture Evolution (SAE). We begin with a historical overview of the important definitions of software architecture and the terminology used in this field. In the second part of this chapter, we discuss the concept of software evolution, its motivations (the laws of software evolution) and the dimensions of software evolution. The third part handles the relationship between software evolution and software architecture. From this overview the importance of software architecture in guiding the planning and restructuring of the software system become clear.

Chapter 3. Evolution styles: This chapter presents the state of the art related to our work software architecture evolution modelling "evolution styles". In the first part, we position the evolution styles' realm, including some meta-modelling approaches in software process engineering. The second part presents a review of the on evolution styles and explores these approaches form different perspectives: conceptual, theoretical and practical. The objective of this chapter is to analyse the important research approaches in software architecture evolution, especially in sharing and reusing architectural knowledge and practices in the modeling of this process.

Chapter 4. Evolution Meta-Style: This chapter presents our first important contribution: Meta-style evolution which is a meta-meta model for software architecture evolution modeling. The first part is described the meta-modeling methodology, along with the benefits of this methodology and an example of The Meta-Object Facility MOF. The second part exploits this methodology in the domain of evolution style and, thus, Meta style and its main concepts are defined. The last part of this chapter puts MES in the mapping process as a means of evaluation.

Chapter 5. Dynamic Meta-Style Evolution: This chapter discusses the issues of dynamic evolution and looks at how these issues are handled at the architecture level. The objective is to extract necessary concepts and information that need to be annotated with MES in order to fill the requirement of modeling the dynamic evolution of software architecture.

Chapter 6. Multiple views/abstractions with Meta-style evolution: This chapter introduces the notion of multi-view and multi-abstraction modeling into the domain of evolution style. The first part presents a summary of the literature associated with this notion of view/viewpoint in the discipline of computer science. The second part represents our contribution in which we integrate these notions into the Meta-style evolution.

Chapter 7.Prototype: The chapter presents an initial prototype tool, implementing a multi-view evolution style editor based on the meta-style evolution specification. To implement the prototype, the ADOxx Meta-modeling platform has been selected as a development toolkit. Four different types of views are developed to visualise architectural evolution knowledge: path view, component-connector view, deployment view and decision view. The last part of this chapter presents a scenario of realisation an evolution style model on the initial prototype tool.

Chapter 8.Conclusions: This chapter summarises the contribution of the thesis and the limitations of the proposed approach, and outlines future work that would formulate possible directions for future research.

1.5 Publications

The following publications present some of the results represented in this thesis:

International conference

[1]. Adel Hassan and Mourad Oussalah. "Evolution Styles: Multi-View/Multi-Level Model for Software Architecture Evolution." *7th ACM International Conference on Software and Computer Applications*. 2018.

[2]. Adel Hassan, Audrey Queudet, and Mourad Oussalah, "Evolution style: framework for modeling dynamic evolution of real-time software architecture." *10TH EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE (ECSA 2016)*. 2016.

[3]. Adel Hassan, Mourad Oussalah, "Meta-evolution style for software architecture evolution" 42th *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, Berlin, Heidelberg, SOFSEM 2016.

International journal

[1]. Adel Hassan, Mourad Oussalah, "Evolution Styles: Multi-View/Multi-Level Model for Software Architecture Evolution," *Journal of Software* vol. 13, no. 3, pp. 146-154, 2018.

Chapter 2. Software Architecture and Software Evolution

This chapter relates the work in this thesis to background realms and is divided into a number of sections, including software architecture, software evolution and software architecture evolution, as well as software process models. In each section, there is also an explanation of how the thesis is related to each paradigm

2.1 Software Architecture

Software architecture serves as a blueprint for a software system's construction and evolution (Garlan 2000), (Clements, Garlan, Bass, et al. 2002). It is the backbone of the software system (Taylor, Medvidovic, and Dashofy 2009). In the last years, several definitions of software architecture have been published, although it was first defined by Perry and Wolf as a 3-tuple consisting of *Elements*, *Form*, and *Rationale* in 1992.

> *Elements capture the system's building blocks, which can be of three types: processing elements, data elements and connecting elements.*
> *Form captures how the (architectural) elements are organized in the architecture, by means of weighted properties and relationships. That is, the form captures how the elements are composed (i.e. the architecture configuration), the characteristics of their interactions, and their relationship with their operating environment.*
> *Rationale captures the motivation for the choice of an architectural style, the choice of elements, and the form. That is, the system designer's intent, assumptions, choices, external constraints, selected design patterns, and other information that is not easily observable from the architecture.*
>
> *(Perry and Wolf 1992)*

In 1993, Shaw and Garlan argued that software architecture represented a problem of designing and specifying the overall system structure that was beyond the algorithms and data structures of the computation. These structural issues include: gross organisation and global control structure; protocols for communication, synchronisation and data access; the assignment of functionality to design elements; physical distribution; the composition of design elements; scaling and performance; and selection among design alternatives. Based on this the authors define the term "architectural style" as denoting a family of architectures that share a common vocabulary (of components and connectors) and meet a set of constraints for that style.

> *An architecture of a specific system is a collection of computational components or simply components together with a description of the interactions between these components the connectors.*
>
> *(Garlan and Shaw 1993)*

These definitions are some of the most widely extended and accepted definitions of software architecture. Over the years, other researchers have refined and extended these early classical definitions and several definitions have been presented based on them. One of these extended definitions was proposed by Garlan and Perry for their guest editorial in the April 1995 IEEE Transactions on Software Engineering, devoted to software architecture:

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*
>
> *(Garlan and Perry 1995)*

Another extended definition was proposed by Kruchten in 1995:

> *Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well -chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability.*
>
> *Software architecture deals with abstraction, with decomposition and composition, with style and esthetics.*
>
> *(Kruchten 1995a)*

Another definition that considers the needs of stakeholders (users, acquirers, analysts, developers, testers, maintainers, inter-operators, and others) as a concept necessary for complete software architecture is that of Gacek *et al.*:

> *A software-system architecture comprises:*
>
> - *a collection of software and system components, connections and constraints;*
> - *a collection of system stakeholders' need statements;*
> - *a rationale which demonstrates that the components, connections, and constraints define a system, that if implemented, would satisfy the collection of system stakeholders' need statements*
>
> *(Gacek et al. 1995)*

Another definition introduced by Shaw and Garlan in 1996, implicitly includes the elements defined by Perry and Wolf:

> *Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.*
>
> *(Shaw and Garlan 1996)*

Bass, Clements and Kazman, in 1998 proposed a definition of software architecture whereby they insisted on the external visible properties of the elements defining their expected behaviour:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
>
> *(Bass, Clements, and Kazman)*

Another definition was provided by the ANSI/IEEE Standard 1471-2000. It presents a conceptual framework and embodies a theory and practice of architectural descriptions (AD). It includes the use of multiple views, reusable specifications for models within views, and the relationship of architecture description to system context. This conceptual framework is introduced via a class diagram (meta-model) with a set of content requirements for the architecture description and its context (views, viewpoints, stakeholders and concerns).

> *Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*
>
> *(Maier, Emery, and Hilliard 2001)*

One of the more interesting extended forms of Perry and Wolf is provided by Taylor, Medvidovic and Dashofy, who embrace the meaning of *What*, *How* and *Why* questions:

> ***Elements*** *help to answer the What questions about the architecture: What are the elements of a system? What is their primary purpose and the services they provide?*
>
> ***Form*** *helps to answer the How questions about the architecture: How is the architecture organized? How are the elements composed to accomplish the system's key task? How are the elements distributed?*
> ***Rationale*** *helps to answer the Why questions about the architecture: Why are particular elements used? Why are they combined in a particular way? Why is the system distributed in a given manner?*
>
> *(Taylor, Medvidovic, and Dashofy 2009)*

Finally, there is one more definition by Taylor, Medvidovic and Dashofy in their book: Software Architecture: Foundation, Theory and Practice. They state that the system architecture represents the set of principal design decisions which depend on the system goals defined by the stakeholders (as requirements drive architecture). The definition places the software architecture as a central artifact in the software life-cycle which is specified in the early stages of software development and which constitutes the process model:

> *A software system's architecture is the set of principal design decisions about the system.*
>
> *Design decisions encompass every aspect of the system under development, including: system structure, functional behavior, interaction, nonfunctional properties and implementation.*
>
> *Principal is a term that implies a degree of importance and topicality that grants a design decision architectural status, that is, that makes it an architectural design decision (i.e. it impacts a system's architecture).*
>
> *(Taylor, Medvidovic, and Dashofy 2009)*

None of these definitions conflict with each other. Instead, they are similar in that they are all concerned with the structure and behaviour of the system. "Structure" describes how the system is built of interconnected elements called components. "Behavior" refers to the visible interaction of these components in order to achieve the overall system's functionality. Both the structure and

behaviour of software architecture are formally described using Architecture Description Languages (ADLs).

2.2 Architecture modeling concepts

Generally, the overall ADLs defined to date according to Taylor and Medvidovic (Medvidovic and Taylor 2000), focus on the basic component, the connector, the interface and the configuration. All of these treat components as first-class citizens, but in some languages neither the connector nor the architectural configuration are considered first-class citizens. In order to facilitate a better comprehension of the work of this thesis, we are presenting a precise definition of the most relevant concepts, some of which are represented in Figure 1.

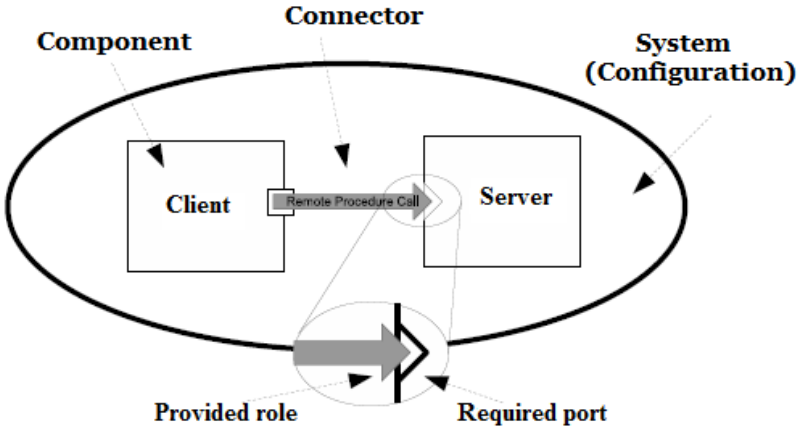

Figure 1: A group of architectural elements can be depicted using an ADL

Component

The component represents the basic concept of software architecture and of the first-class citizens that Architecture Description Languages (ADLs) share par excellence. Szyperski is one of the leading authors on component software:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only, a software component can be deployed independently and is subject to composition by third parties.*
>
> *(Szyperski 2002)*

A software component is a decoupled architectural entity which provides functionality and/or a data source. It can be deployed independently or within a software system. A server, database or mathematical functions are examples of components. A component should contain three constituents: interfaces, an implementation and a descriptive specification (Taylor, Medvidovic, and Dashofy 2009).

The interface is the port of the component that defines the points of interaction with its environment *(Szyperski 2002)*. Furthermore, a component may have multiple interfaces (ports), separated into two main categories: provided and required interfaces. "Implementations" refer to the concrete, executable code of components that are defined by the developer and hidden behind the interfaces. "Specification" defines the component's semantics as a set of functional and non-functional properties (external definition), based on which the component can be reused (selected).

Connector

Connectors identify the interaction between the components and correspond to lines in box-line descriptions. The connector may play one role or many different roles, in a system, each of which is identified by the service that the connector provides. These services belong to one of four categories: communication, coordination, conversion and facilitation (Mehta, Medvidovic, and Phadke 2000). It is possible to have a multi-category connector in order to serve different interaction requirements. These four categories have been further classified into eight types, depending on the way in which they realise interaction services: procedure call, event, data access, linkage, stream, arbitrator, adaptor and distributor. Perry has provided a high-level classification of the roles that software connectors play in architecture (Perry 1997). Mary Shaw was the first to introduce explicit connectors that separate the concerns of computation, handled by components, from inter-component communications, handled by connectors.

> *Connectors are the locus of relations among components. They mediate interactions but are no "things" to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction, such as ordering, performance, etc*
>
> *(Shaw 1993)*

Shaw presents the connector as an architectural element that defines the coordination process among components. The separation between the interaction and the computation is promoted so as to obtain more reusable and modularised components (the same component can be used in a variety of environments, each of which uses different communication primitives) and to improve the level of abstraction of the software architecture descriptions. Furthermore, it provides an architectural view of the system instead of the object-oriented view of compositional approaches. Shaw also defends the idea of considering connectors as first-class citizens of software architecture in terms of their description languages (Shaw 1993).

Allen and Garlan (Allen and Garlan 1997) formalise the semantics of connectors, whereby the specification of the connector types is based on the idea of characterising the protocols of interaction (interaction patterns) between architectural components. In general, connectors represent mechanisms for transferring control and data around a system (Lau, Elizondo, and Wang

2005). Each of these mechanisms has specific characteristics and properties. Examples of connectors are: procedure call, sending events, pipes and data stream. A procedure call can be local or remote. In the case of remote, various kinds of remote procedure calls (or remote method invocation) middlewares can be used to implement such a call (Java RMI, DCOM, CORBA, SOAP, etc.).

Software architecture represents the structure and behaviour of software systems. It captures the behaviour of a software system from the behaviour of individual components and how they interact with each other. Thus, interaction is the central focus of software architecture (Kiwelekar 2013).

Several ADLs have introduced the concept of connectors as first-class entities, such as ACME (Garlan, Monroe, and Wile 2000), Wright (Allen 1997), Archware (Oquendo et al. 2004), π-ADL (Oquendo 2004) and COSA (Smeda, Oussalah, and Khammaci 2004), whereas for example, Darwin language uses implicit connections. The connections among components are specified in terms of direct bindings of requires and provides interfaces.

Configuration

An architectural configuration (a system's instance or topology) is a graph which shows how components and connectors are composed in a specific way in order to accomplish the system‘s objectives. “The basic elements of architectural description are components, connectors and configurations “(Garlan, Allen, and Ockerbloom 1994).

> *An architectural configuration is a set of specific associations between the components and connectors of software system architecture.*
>
> *(Taylor, Medvidovic, and Dashofy 2009)*

The graph is obtained via associating the ports of the components with the roles of the appropriate connectors, in order to formalise the system (or subsystem). The associations among components and connectors are sometimes called attachments (in the same composite and binding among different levels).

Oussalah et al. in different works, COSABuilder (Smeda et al. 2008) and C3 meta-model (Amirat and Oussalah 2009), defines configuration as a first-class citizen like the component and the connector. He argues that, since a component and connector are perceived and handled from the outside as primitive elements, their inside may be composite with a configuration which constrains the wiring of all the internal elements. This configuration should be also handled as a first-class entity. It represents a graph of components and connectors and describes how they are fastened to each other. Configuration can be hierarchical where each component and connector represents a sub-configuration that has internal architectures. So, they do not treat the configuration as an instance of a system but as an architecture “Type” which can be also instantiated.

System

A system is a collection of components and connectors instantiated in a configuration. This definition, unlike others definitions (e.g. Oussalah) which consider the configuration as another kind of architectural element, defines the configuration only as an instance of a system. Therefore we can state that the concept of system here differs from the concept of configuration in that a system is a building block that can be reused in several software systems, whereas a configuration defines a specific instance of a system which cannot be reused.

It is often represented as a hierarchical network (i.e. a composition may be composed of other subsystems) of components linked together by connectors in accordance with a certain set of rules, which facilitates the understandability and the specification of the architectural description.

Several ADLs have introduced the concept of the system as a configuration of components and connectors, such as UniCon, Wright, Acme and Archware.

Interface

An interface is the "gate" to the outside world, made up of components, connectors and configurations. It is also known as a port for components and configurations and as a role for connectors. The interface (port) of a component provides the information (services) that allow use of the component without knowledge of its implementation. In addition to the services provided by component, the interfaces also specify the services required by the component. Require interfaces declare the operations that allow the component to interact accurately with its environment. Thus, the component can have multiple ports which either provide or require services to/from its environment.

A connector consists of a set of roles. Each role serves as a connection point that defines the behaviour of one participant in the interaction. The role is also either a provided role or the required role. A provided role serves as an entry point to a component interaction and is intended to be connected to a required interface of a component (or to a required role of another connector).

Interactions between interfaces are defined by attachments and bindings.

- *Attachment: represents the connection of a port of a component and a role of a connector, if they are compatible with each other (required / provided).*
- *Bindings establish the mappings between the internal and external interfaces of a system (Garlan, 2001).The connections between the ports (or roles) of a system, or of a composite component and the ports (roles) of one of its architectural elements. Thus the binding is different from attachments because binding represent a hierarchical composition mechanism, used between different granularities, while attachments are a flat composition mechanism, used among the same granularity.*

Other concepts

Even though, the previous concepts are adequate for describing a software architecture, there are other relevant concepts which have been added by most ADLs, or which are common to software architecture that remain to be defined.

First there are the concepts of **Architecture Style** which are widely used in order to represent families of software architecture descriptions that belong to software systems which have something in common: resource types, configuration patterns and constraints (Garlan 2002). Examples of styles are: event-based, layered, blackboard, pipe-and-filter, client-server, peer-to-peer etc.

In addition to the aforementioned high-level elements, most architecture also associates properties with their internal elements. For example, for an architecture in which its components are associated with periodic tasks, the properties could define the periodicity, the priority, and the worst case execution time of each component. Properties may also be non-functional, such as the relative ease of evolution, the reusability of components, the efficiency and the dynamic extensibility, and these are often referred to as quality attributes (Bass, Clements, and Kazman 1998).

A **Type** is another concept that simulates the paradigm of object and class. A type can be instantiated several times in the same architecture, or can be reused in other architectures. A system with explicit types (such as pipes, filters, clients, servers, parsers, databases etc) facilitates understanding and allows analysing of the architectures (Garlan, Allen, and Ockerbloom 1994). Thus, it is possible to reuse components, connectors and configurations by component types, connector types, and configuration types. In this regard, an architectural style can be seen as a particular type of configuration.

The last concept that we aim to mention is that of the **Constraints** which express the restrictions on the architectural elements. They are often applied to an architectural design in order to induce the architectural properties desired of the system. For example, the uniform pipe-and-filter style obtains by constraining the components to a single interface type. Thus, any single "port" of a filter can be connected to one pipe at most. The constraints can be specified either in a separate constraint language (such as Architecture Constraint Language ASL and Object Constraint Language OCL) or directly by ADLs such as Aesop, SADL and Wright.

2.3 Architecture Description Languages (ADLs)

According to the standard (ISO/IEC/IEEE 42010, 2011), an Architecture Description Language (ADL) is:

"Any form of expression for use in architecture descriptions".

Box and line have for a long time, been the only means for informally describing software architecture. Occasionally, it is useful to have a formal description that can assist in understanding

the alternatives and trade-offs of the design decisions. Nevertheless, the continual increase in the size and complexity of software systems have drawn a lot of interest in develop more rigorous ways for describing software architecture. Thus, architecture description languages (ADLs) overcome the informality of most box-and-line descriptions of software architecture, and provide notations and tools for accurately representing the *structures and behaviours* of the *architecture* and for supporting the *reasoning (architecture knowledge)* about them (Clements 1996). In this vein, a prominent thread of research on formal ADLs has been conducted over the past two-and-a-half decades (Garlan 2014), with the definitions of dozens of different of ADLs. Accordingly, a number of ADLs have been developed by the academic community for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages. Some of the prominent ADLs proposed include Darwin (Magee and Kramer 1996), Rapide (Luckham and Vera 1995), UniCon (Shaw et al. 1995), Wright (Allen 1997), Acme (Garlan, Monroe, and Wile 2000), Koala (Van Ommering et al. 2000), xADL (Dashofy, Van der Hoek, and Taylor 2001) , ArchWare (Oquendo et al. 2004), UML 2 (Ivers et al. 2004), AADL (Feiler, Gluch, and Hudak 2006) and COSA (Smeda, Oussalah, and Khammaci 2004). Each of these ADLs comes with certain distinctive capabilities (Medvidovic and Taylor 2000), such as multi-view, dynamism and analysis mechanisms. Some of these ADLs are domain-specific, for example: AADL, SysML[1] and EAST-ADL (Cuenot et al. 2010) which have been presented to deal with real-time and embedded computer systems; DiaSpec (Cassou et al. 2009) used for control-loop applications; Koala used for product-line architectures; π-ADL (Oquendo 2004) used for dynamic and mobile software architectures, and ArchiMate, used for enterprise architecture.

2.4 Architecture Knowledge

Whenever a software system needs to be modified, it must be understood in the first step, before non-destructive modifications can be proposed or enacted. Measurements have shown that system comprehension consumes more than half of the time and effort of the evolution (Gacek et al. 1995), (Zelkowitz, Shaw, and Gannon 1979).

As mentioned before, Perry and Wolf in (1992) provided one of the most insightful definitions of software architecture, defining it as a 3-tuble of element, form, and rationale. Even though, the concept of rationale is missing in some definitions such as that of Garlan and Shaw (1993), it reflects the essence of the software architecture. It aims to answer the Why questions about architecture design decisions in accordance with the What, How and Why of Taylor, Medvidovic and Dashofy's definition of software architecture (Taylor, Medvidovic, and Dashofy 2009). Capturing the architecture decisions can help to get into the architects' reasoning process, reduce knowledge vaporisation and avoid architectural drift.

[1] https://www.omg.org/spec/SysML/1.5.

Jansen and Bosch emphasised this view, and this was shown in their definition of software architecture as a composition of a set of architectural design decisions (Jansen and Bosch 2005). In accordance with this point of view, a different software architecture paradigm emerged, which represents the software architecture from the point of view of its stakeholders (concerns, design decisions, alternatives, rationale) (van Heesch et al. 2014), rather than from the physical view (structure and behaviour).

Thus, different research into capturing, modeling sharing, and (re)using architecture knowledge have been conducted, whereby several architecture knowledge managements methods, techniques, and tools have been proposed in order to support the decision-centric architecture approaches (Capilla et al. 2016). In this thesis, we are more interested in capturing design decisions throughout the architecture life-cycle and thus enabling the management of its advanced evolution processes.

2.5 Meta-modeling in software architecture

Meta-modeling is a modeling technique that aims to manage the complexity of a system, enhancing the understandability and promoting standardisation and interoperability. It has been widely used to address real problems in programming languages, databases, engineering models or distributed systems. However, it often defines two or more levels using the definition of a metamodel (Oussalah 2014). Meta-knowledge is knowledge about knowledge and meta-model is modeling of model. Based on this view, meta-architecture is the act of architecting (modeling) applied to the architecture. This mechanism can be applied several times yielding multiple levels of modeling. One prevailing framework that implements the four modeling levels was proposed by the Object Management Group (OMG). Oussalah et al. (Smeda, Oussalah, and Khammaci 2005) argued that the meta-modeling technique could be applied to the field of software architecture (i.e. it could be implemented in the component- based modeling like on the object-oriented approach). Based in this view, they used this technique to define Meta Architecture Description Language (MADL). Accordingly, an architecture hierarchy with four modeling levels was defined, starting from a meta-meta-architecture and going up to an application level. Moreover, COSA Builder, as ADL tool based on this language, was developed. Figure 2 illustrates the MADAL construction which applied the meta-knowledge to the field of architecture. It identifies four levels of modeling: the meta-meta-architecture level, the meta-architecture level, the architecture level and the application level.

The meta-meta-architecture level (A3): this level provides the minimum components for modeling architecture, which represent the basic concepts for an ADL. The meta-meta-architecture is an instance of itself (self-defined).

The meta-architecture level (A2): this level provides the basic modeling components for an architecture description language (ADL) – component, connector, architecture, ports, roles, etc. by which a system architecture can be defined. Meta-architecture conforms to the meta-meta-

architectures (its meta). Each element in the A2 level is an instance of an element (its meta-element) in the A3 level.

The architecture level (A1): at this level, several types of components, connectors and architectures can be described. Each architecture model should comply with its meta-architectures (ADL), whereby each element of A1 is associated with an element of A2.

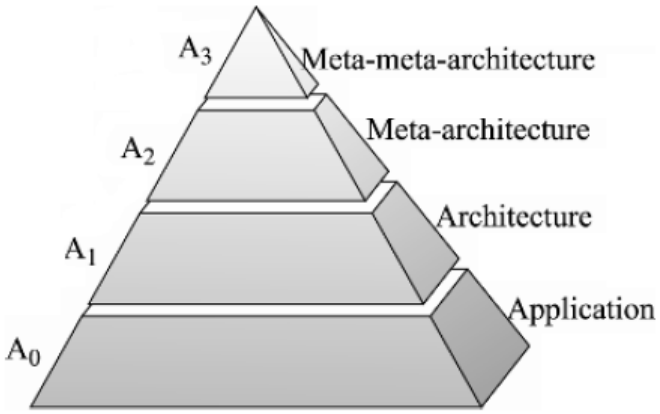

Figure 2: Four levels of modeling in software architectures

The application level (A0): This level refers to the place where the executive units are located. An application is considered as a set of instances of architecture-element types. Applications conform to the architecture whereby each element of A0 is associated with an element of A1.

Some advantages of introducing the multi-level (meta-modeling) in the domain of architecture have been summarised as follows:

- The meta-architecture can play a vital role in standardisation among their instances as a meta-architecture can semantically conform to different architectures.
- The meta-architecture can support the reuse in the definition of new architecture (instantiates new architecture).
- The meta-architecture can provide the foundation for comparison and mapping among different architectures.
- The meta-architecture facilitates and supports the exchange of architectural elements between the ADLs.

This view of software architecture will be seen as important when the problem of the evolution of software architecture is discussed in the next chapter, for example in distinguishing the granularity of changes, i.e. the changes impacting architecture type (level) and/or architecture instance.

Figure 3 represents an example of these modeling levels applied to client-server architecture.

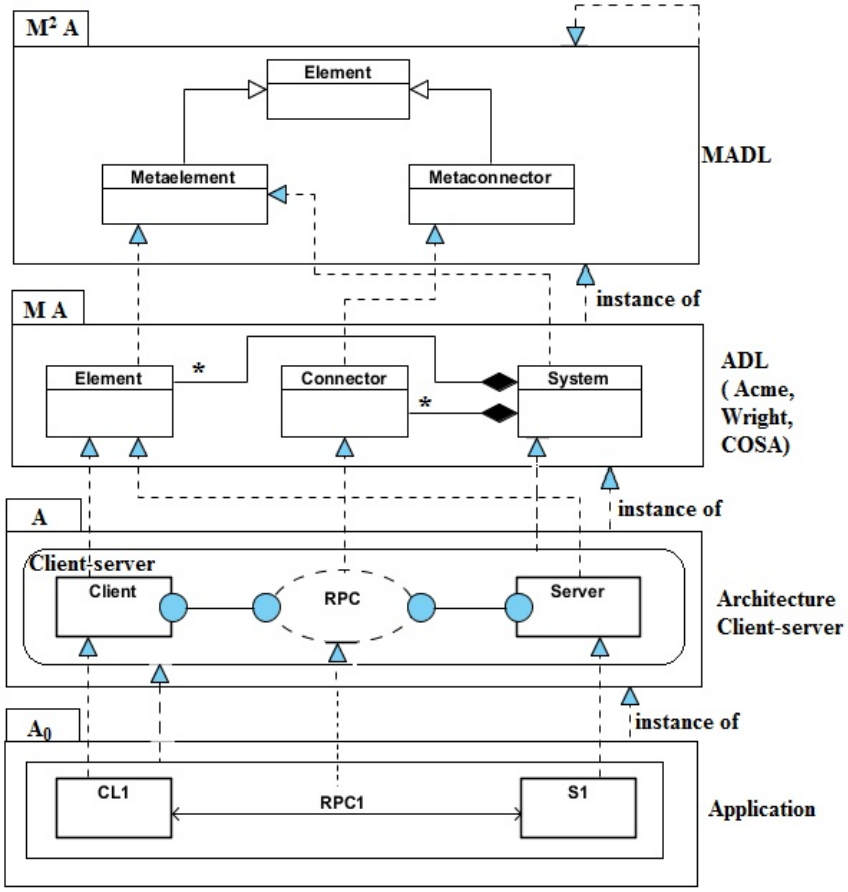

Figure 3: The implementation of the four modeling levels in software architecture

2.6 Software evolution

Change is inevitable for any software system in order to achieve its business goals during its lifetime. The necessity and the characteristics of software change are defined as Lehman's laws of software evolution (Lehman 1996). During the last 40 years, many empirical studies have been conducted to address the validity of these laws (of software evolution). Herraiz et al. (Herraiz et al. 2013) summarised these studies and found that all of them have validated the first law ''*Law of Continuing Change: An E-type system that is used must be continually adapted else it becomes progressively less satisfactory.*" It can be stated that systems must be evolved continuously in order to be useful, - i.e. evolution is a recurrent process that accompanies a system from its initial creation to its eventual retirement or abandonment. However, several studies have suggested that software evolution consumes the largest share of the software life-cycle budget (50% to 90%) (Moad 1990) and (Erlikh 2000). This confirms the need for techniques, methods and tools that guide the different participants in the software evolution process. Different approaches have been suggested

to facilitate and support such software evolution (Mens, Serebrenik, and Cleve 2014). Lehman provides the following definition of evolution:

> *"Software evolution is the collection of all programming activities intended to generate a new version from an older and operational version."*
>
> *(Lehman 2000)*

2.7 Laws of software evolution

The laws of software evolution were formulated by Lehman et al. and the first version was based on the observations of the IBM OS/360 operating system (Lehman 1974). Initially, Lehman defined three basic principles for the evolution of software systems. He uses the term "E-type software" to denote programs that address a problem or an activity of the real world. Accordingly, changes in the real world will necessitate changes in the software which serves this environment. The laws of software evolution, then, have seen several refinements over the years, with the most recent version of the laws published in 1996 (Lehman 1996) and summarised as follows:

- Law of Continuing Change:
  An E-type system that is used must be continually adapted, else it becomes progressively less satisfactory in use.
- Law of Increasing Complexity:
  As an E-type system evolves, its complexity increases unless work is done to maintain or reduce the complexity.
- Law of Self-Regulation:

  Global E-type system evolution processes are self-regulating.

- Law of Conservation of Organisational Stability:

  Average global activity rate in an E-type process tends to remain constant over periods or segments of system evolution.

- Law of Conservation of Familiarity:
  The average growth rate of E-type systems tends to remain constant or to decline.
- Law of Continuing Growth:

  The functional capability of an E-type system must be continually increased to maintain user satisfaction over its lifetime.

- Law of Declining Quality:
  Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
- Law of Feedback System:
  E-type software processes are multilevel, multi-loop, multi-agent feedback systems.

The laws concerning continuing change, increasing complexity, and continuing growth are of particular interest for this thesis. However, the ever-changing requirements of technology and the business environment necessitate integrating new features or improving the quality of the system for it to remain useful. Changes cannot be made over-night, so the evolution team must develop a plan to manage and conduct this process. In fact, the ideal start for the evolution team is to understand the system and then to find a suitable set of system modifications. Then, the optimal scenario, which best fits certain properties, will be selected. Ultimately, implementation of the selected scenario will lead to the new target system through a series of phased releases. All these have motivated this work, i.e. when evolving a system, it is important to explore the potential selections of evolution, to understand the rationale of these scenarios and to make trade-offs among them. Our thesis aims to investigate this trade at the architectural level. This is based on the fact that, architecture evolution is an essential complement to software evolution because it permits planning and system restructuring at a high level of modeling where business goals, quality requirements and alternative scenarios of evolution can be explored (Barnes, Garlan, and Schmerl 2014).

2.8 Dimensions of Software Evolution

Software evolution is a complex process, due to the fact that all artifacts produced and used during the software development life-cycle are subject to changes, starting from requirements, through analysis and design documents to the executable code. All these have inspired several studies about the dimensions and taxonomies of software change (Lientz and Swanson 1980), (Chapin et al. 2001), (Buckley et al. 2005), all of which attempt to answer the Why, How, What, When and Where of software evolution.

Lientz and Swanson (Lientz and Swanson 1980) proposed a mutually exclusive and exhaustive software maintenance typology, falling into three basic categories, namely perfective, adaptive and corrective. This definition was further extended by the ISO/IEC Standard for Software Maintenance (ISO 1999) to include the category of preventive maintenance activities. This typology was further refined by Chapin et al. (Chapin et al. 2001) into an evidence-based classification to include 12 different types of software evolution and software maintenance.

A few years later, a more complex taxonomy of evolution was proposed by Buckley et al. (Buckley et al. 2005), which focuses more on the characteristics of software change mechanisms and the factors that influence these mechanisms. Several dimensions were refined into four logical themes: temporal properties (When is the change made?); object of change (Where is a change made?); system properties (What is being changed?); and change support (How is the change accomplished?), as shown in Figure 4.

However, the dimensions of evolution are classified as characterising the mechanism or as influencing factors. An influencing factor, for example, a system's 'availability' can be affected by the change mechanism applied to that system. If a system is required to be highly available, then a

run-time change mechanism should be applied. The characteristics of software change are those factors of software change which can be used to describe the nature of the changes (e.g. "time of change").

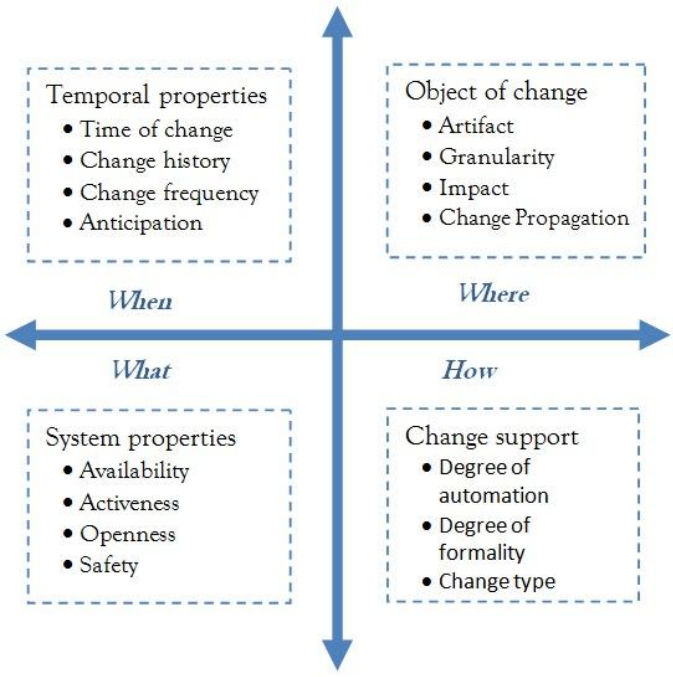

Figure 4: Themes and dimensions of software change.

In order to facilitate a better comprehension of the work of this thesis, the theme "object of change" is presented in more detail here. This theme describes the location(s) in the system where changes are made, divided into four factors:

- ***Artifact***: refers to any produced and/or used artifact during the entire software development life-cycle which may be subject to change, ranging from requirements through architecture and design, to source code, documentation, configurations and test suites.
- ***Granularity***: refers to the scale of the artifacts to be changed and can range from very coarse (system architecture), through medium (modules/components), to a very fine degree of granularity (class, methods).
- ***Impact***: The impact of a change, ranging from local to system-wide change, which can span different artifacts with different abstraction levels (from source code to architecture).
- ***Propagation***: refers to the process of ensuring that a change with global impact is propagated to all related entities in the software system. Propagation can be performed in two directions; horizontal or vertical (top-down or bottom-up).

Each of these dimensions can be affected by the others. For example, if we take system architecture as an artifact of change, the evolution mechanism of this artifact can be affected by other factors from the same or from another, theme. The impact of change at the architecture can vary from the small scale of a component to the entire architecture (system/configuration). Moreover, the impact

of change can remain within the same abstraction level of the architecture (vertical impact), or span across different levels (horizontal impact).

Propagation at the architecture artifact also can be performed in two directions, whereby vertical propagation affects the same abstraction level and horizontal propagation is done across different architecture levels (top-down or bottom-up), from an architecture type to its instance or vice versa. The time-of-change dimension at the architecture evolution, can widely affect the mechanism of the changes, arranging them from static, through load-time, to dynamic (at runtime). Dynamic evolution refers to modifying the architecture and enacting those modifications in the system without halting the system. This is a critical requirement for many software systems, such as air traffic control, telecommunications and financial systems (Hassan, Queudet, and Oussalah 2016).

2.9 Evolution as part of the development process

The term "software evolution" refers to a phase of the software life cycle, which lasts from the initial creation of the software product until its eventual retirement or abandonment (close down phase). Several factors, such as frequency, effort and cost of changes have promoted evolution to be an important stage in the life-cycle of a software product. Based on this view, several software development methods have been introduced and gained widespread acceptance in the research community. Each of these methods has the evolution as a crucial stage of the process model. Some examples are: Evolutionary Development (Gilb 1981), the Spiral Model (Boehm 1988), Staged Model (Rajlich and Bennett 2000), and Agile Software Development (Cockburn 2001). In these models, the stage of software evolution can involve the adaptation of new requirements or the performance of maintenance activities.

Evolution planning is often the first phase of software evolution and often starts with analysing the change impact and checking the completeness and consistency of the intended design. This step requires a deep understanding of the system (design decisions) and of the changes (suitable modifications) in order to satisfy the evolution requirements without breaking the existing functionality (loss of design quality). Based on these perspectives, one of the first attempts to move towards a more evolutionary model was proposed by Yau et al. (Yau, Collofello, and MacGregor 1978) with the so-called *change mini-cycle* which consists of the five main phases shown in Figure 5.

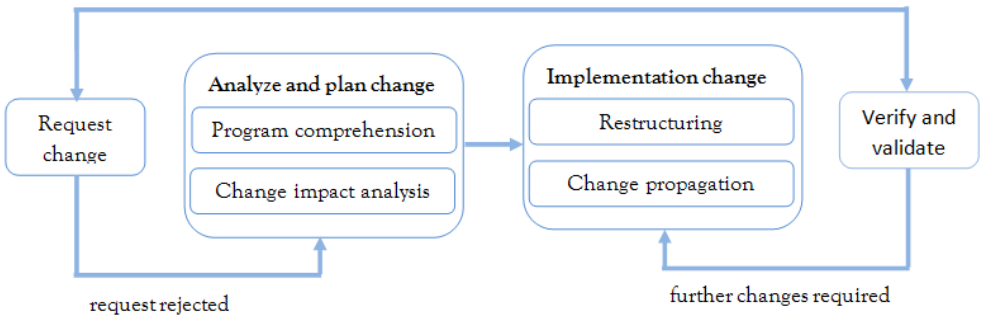

Figure 5: The staged process model for evolution (adapted from (Yau, Collofello, and MacGregor 1978))

2.10 Evolution and software architecture

Information systems are more and more complex and widely distributed and most are software-intensive systems. They often include large-scale heterogeneous distributed software components, embedded systems, telecommunications, wireless ad hoc systems and COTS. As the market is continual changing, these systems also have to change in order to embrace new business opportunities, customers' requirements and technological infrastructure. In this context, the designing, building, and evolving of such heterogeneous software systems is an important issue of research on which several studies have been conducted. One discipline proposed "*architecture-centric development methods*" which claim that architecting is the most appropriate point of view from which to address this complexity as it enables stakeholders to work at a higher abstraction level. Software architectures shift the focus of developers from implementation to coarser-grained architectural elements and their overall interconnection structure (Medvidovic and Taylor 2000). As the software architecture captures early design decisions (architectural design rationales) that have a significant impact on the quality of the intended system, it is important if not essential to understand those decisions as early as possible in the evolution process. This step can ensure the consistency between design and evolution decisions and avoid architectural drift and erosion. On this point of view, we can cite, in particular, IEEE Standard 1471–2000 which provides an interesting definition of architecture, which mentions evolution: *"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" (Maier, Emery, and Hilliard 2001).*
Since the system architecture is one of the artifacts that may be impacted by changes, so the methods of evolving the architecture is part of the system-evolution problem. A better understanding of architecture changes is the prerequisite to planning the evolution process and enacting the intended changes. Moreover, it is recognised that, architecture plays a beneficial role in the evolution process. This is due to the fact that: 1) architecture can elucidate the reason behind the design decisions which guided the building of the system; 2) it can expose the dimensions along which a system is expected to evolve (Garlan and Perry 1994); 3) it can permit planning and system restructuring at a high level of modeling, where business requirements and quality goals can be ensured and alternative solutions can be explored; 4) it can provide foundations for the evolution process (Medvidovic 1998). Thus, software evolution begins to shift its underlying basis from source-code to software-architecture models.

2.11 Software architecture evolution

Planning large-scale evolution is a high-challenging activity that requires an understanding of overall system structure, consideration of design rationale and principled trade-offs among various issues of concern (Barnes, Garlan, and Schmerl 2014). A design decision is a key concept in software evolution (Jansen and Bosch 2005). There are different strategies to deal with evolution. The classical approach in reengineering is to evolve the software at their source-code level. In fact,

the source code cannot provide a complete and synthetic view of software that will clearly reflect its design decisions. On the contrary, the architecture presents a high-level view of a system, which includes the software elements and their relations, whereby design decisions can be an explicit part of the architecture model (Jansen and Bosch 2005). Architecture-design decisions are often documented using templates, a decision model or annotations. These documents often capture the rationale behind an architecture relevant decision, the selection of a certain solution, a set of potential alternatives and the reasoning about a trade-off between alternatives.

However, architecture evolution is about making new design decisions or removing obsolete ones in order to satisfy new requirements. It concerns the changes applied to architecture, including their components, connectors or their configuration.

Architecture evolution has been an active area of research, to which a number of software architecture researchers have turned their attention. This attention is also presented in some interesting semantics literature reviews and comparative studies. Breivold et al. (Breivold, Crnkovic, and Larsson 2012) and Jamshidi et al. (Jamshidi et al. 2013) have studied the available architecture-evolution research and classified it in two main categories: 1) research concerns about designing and assessing the evolvability of the software architecture; 2) research concerns about planning and enacting software-architecture evolution. Our approach focuses on the latter category, on providing architects with assistance when software-architecture evolution is carried out.

2.12 Conclusion

This chapter has presented a brief overview of software architecture and the software-architecture evolution. The most important concepts have been introduced in order to ensure a good understanding of the research context of the thesis. We have explained how software architecture has gradually come to the fore and now plays a vital role in software development and evolution. We have also devoted sections to software evolution, explained the rationale and the laws of the evolution and looked at what role software architecture can play in this process.
The evolution of software is a complex process that consists of multiple tasks in which multiple groups of stakeholders are involved. It has been detected and highlighted as one of the most complex aspects of the software life-cycle. Architecture can play a vital role in guiding the planning and restricting the system in a high level far away from the complexity of low-level details.

The reusing of software-architecture evolution knowledge is the motivation for this thesis, i.e. we need to investigate means of capturing and sharing best architectural-evolution knowledge and practice in a particular domain.

Chapter 3. Evolution Styles

In this chapter we give an overview of the current state of art of the evolution styles with respect to different realms that are relevant to software-architecture evolution modeling. We then discuss the related work and position our proposal.

3.1 Process modeling

Knowledge is essential in everyday work. Everyone knows how to carry out his work and this knowledge can be reused later in similar tasks by adopting this knowledge to fit new situations. However, sometimes this knowledge needs to be shared with others (e.g. throughout an organisation) and this necessitates knowledge acquisition and modeling. A process model contains (aims to capture) the information relevant for performing a certain piece of work.

Process-modeling techniques, in terms of formalisation can be classified in two categories. Descriptive models utilise semi-formal graphical modeling techniques. They take an analytical perspective, aiming to provide a communication environment, the better to understand (use) and improve the process. These models often adhere to the semantic and syntactic rules of the modeling method, but not in a strict manner such as those are applied in execution models. On the other hand, formal modeling techniques are founded on rigorous mathematical paradigms (process algebra, Petri nets, Z notations). They are often used to describe the dynamic aspect of the process for automation purposes (Garcia, Vizcaino, and Ebert 2011), or other usages including process analysis and simulation (Recker et al. 2009).

Within the scope of this work, what we mean by the software architecture evolution process model is the description of this process (descriptive model). Process-modeling techniques are usually concerned with (visual) mapping and workflow to facilitate understanding, analysis and positive changes to the real process.

3.1.1 Domain Specific Modeling

Martin Fowler (Fowler 2010) defines a Domain-Specific Language (DSL) as “a computer programming language of limited expressiveness focused on a particular domain”, such as spreadsheet, HTML, SQL, or CSS. One of the main benefits of DSLs is that, since they lack the complexity of general-purpose languages, domain experts can easily comprehend and learn these languages and can create or enhance the code for their application.
But the term has gained more popularity with the advent of Domain-Specific Modeling DSM. Several modeling languages (García-Borgoñon et al. 2014) have been devised for different purposes that have brought many benefits such as managing complexity, or preserving and reusing expert knowledge. Model-Driven Engineering (MDE) is strongly focused and relies on models and it has

been claimed that many potential benefits can be gained in terms of understandability, productivity, portability, reusability and interoperability (Hutchinson et al. 2011).

The diversity of modeling languages is due to the different perspectives of modeling, which often require different concepts with different properties (Kent 2002), as well as the particularity of some domains. For example, we can take the discipline of software modeling and the most known generic-purpose modeling languages, Unified Modeling Languages (UML), which come with different diagram categories (structural diagrams, functional diagrams, behavioural diagrams and interaction diagrams) for modeling software systems. Nonetheless, many approaches have extended UML, using profiles to add specific concepts in order to appropriate a domain-specific system. For example, even though the UML has activity and state-chart diagrams to model the system behaviour, there are several UML extensions for modeling real-time and embedded systems, such as the UML profile for Architectural Analysis and Design Language (AADL) and the UML profile for the Modeling and Analysis of Real-Time and Embedded systems (MARTE)[2], where the schedulability analysis is critical (Evensen and Weiss 2010).
In the same vein, process modeling aims to capture the main characteristics of the set of activities performed in the production and/or maintenance of a software system in order to define a suitable process-modeling language that can express the process in an understandable model. Various modeling languages have been created for this purpose, such as Multi-View Process (MVP-L), Software Process Engineering Meta-model (SPEM) (Münch et al. 2012) and Essence – Kernel and Language for Software Engineering Methods (OMG Submitters 2012). SPEM is a process-engineering meta-model as well as a conceptual framework. It provides standard concepts for modeling software-development processes and it is also specified as an MOF2.0 based metamodel and as a UML2.0 superstructure-based profile (OMG 2008).

3.1.2 SPEM 2.0

The Software and Systems Process Engineering Meta-Model (SPEM2.0)[3] is one of the most used meta-models for developing several software process models. It provides a means of designing the development process with the basic conceptions for modeling method contents and processes. This meta-model (SPEM 2.0) defines a clear separation between the method content's elements and its process element (as in Figure 6). This means that it separates the content of the methodology (e.g. tasks, work products and roles), from its instantiation within a particular process (e.g. activities, role use and work product use). There are several tools available to implement the SPEM 2.0 models; the Eclipse Process Framework (EPF) composer is an Eclipse open-source project that aims to provide an extensible framework and exemplary tools based on SPEM 2.0 concepts. for defining and managing software-development processes (Haumer 2007). Rational Method Composer (RMC)[1] is a commercial version of EPF from IBM (Haumer 2005) and IRIS Process Author, from Osellus[2], is also a commercial-process-authoring and -tailoring tool which complies with SPEM

[2] https://www.omg.org/spec/MARTE/1.1/

[3] https://www.omg.org/spec/SPEM/2.0/

methodology. On the other hand, SPEM has also attracted the process-engineering community as a subject of extension and several studies have been conducted to extend or annotate SPEM with concepts and semantics related to specific domains in order to fit these domains. Maciel et al. propose an integrated approach to modeling the Model-Driven Architecture (MDA) process, based on SPEM 2.0 concepts (Maciel et al. 2009). FlexSPMF is an extension of SPEM 2.0 which defines a framework for modeling controlled flexibility in software processes (Martinho, Varajão, and Domingos 2009). A. Koudri proposes MODAL (Koudri and Champeau 2010) as an SPEM 2.0 extension for specifying model-based engineering processes. Aoussat (Aoussat, Oussalah, and Nacer 2011) proposes an SPEM extension that defines explicit software process connectors in order to facilitate software-process reuse based on software architecture. T. Martínez-Ruiz proposes vSPEM as an SPEM 2.0 extension which aims at supporting the variability implied in a Software Process Line (Martínez-Ruiz, García, and Piattini 2008).

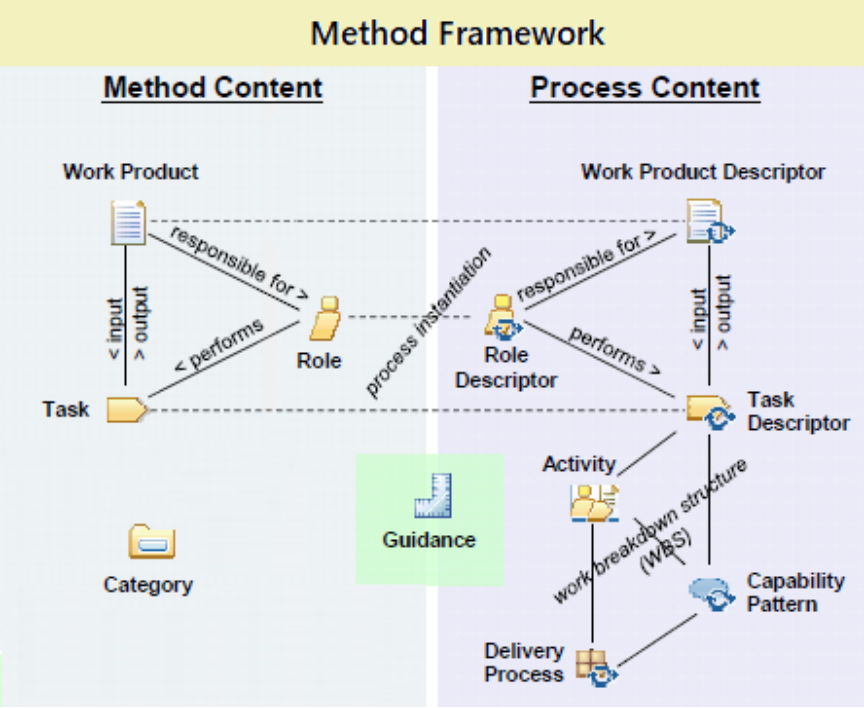

Figure 6: Overview of key concepts in SPEM 2.0

Each of these extensions aims at meeting a domain-specific modeling need and thus they extend SPEM with concepts and semantics that fit the properties of their domains. In this context, software architecture evolution is a particular domain in software engineering, which has its own properties and concerns. Thus, modeling software architecture evolution also requires specific concepts and notations in order to cover the particularity of this discipline in an efficient manner. A domain-specific modeling language for software-architecture evolution can be developed, which may provide significant help in capturing and acquiring architectural-evolution knowledge and practices. For example, an SPEM extension can also be used to embrace the concepts of the architecture-evolution process in order to express this process in a better way.

3.1.3 Essence 1.0

The REuse and Migration of legacy applications to Interoperable Cloud Services (REMICS) (Ilieva et al. 2010) project aims to develop agile methodologies that specifically address model-driven modernisation to service clouds Essence defines a kernel and a language for software engineering method specification. The REMICS project has participated in the Software Engineering Method and Theory (SEMAT) initiative (Kajko-Mattsson et al. 2012) which has led to a new specification named "Essence – Kernel and Language for Software Engineering Methods" (OMG Submitters 2012). In June 2014 the Essence was officially adopted as an official OMG standard. The Essence 1.1 specification was published in December 2015 (http://www.omg.org/spec/Essence/1.1). It aims to provide better support for the definition of agile practices and method enactment, compared with the SPEM specification. The Essence specification introduces a method architecture that distinguishes between ***kernels***, ***practices*** and ***methods***. A ***kernel*** provides a stripped-down, light-weight model of the essential aspects of software engineering. A ***practice*** is a description of how to handle a specific aspect of a software engineering endeavor. A ***Method*** is the composition of a Kernel and a set of practices to fulfill a specific purpose (describe of how an endeavor is performed).
Figure 7 shows the Essence architecture in which the language represents the domain-specific language for defining methods, practices and the essential elements of the kernel.

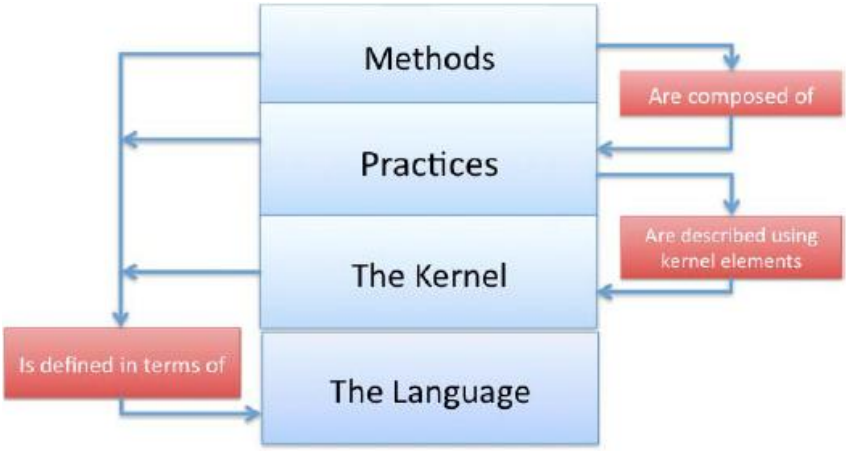

Figure 7: Essence Methods architecture (taken from REMICS)

Figure 8 illustrates the key language concepts defined in the Essence specification. A *kernel* is expressed using the concepts *Alpha*, *Activity Space* and *Competency*. *Alphas* represent things to work with and have *Alpha States* to track their progress. *Alphas* can be seen as placeholders for *Work Products*. *Activity Spaces* represent things to do and can be seen as placeholders for *Activities*. *Competencies* represent skill sets. A *Practice* adds specific details by defining *Work Products* and *Activities*, but can also introduce new *Alphas*, *Activity Spaces* and *Competencies*. The language also includes *Resource* so as model different resource types (Ilieva et al. 2010).

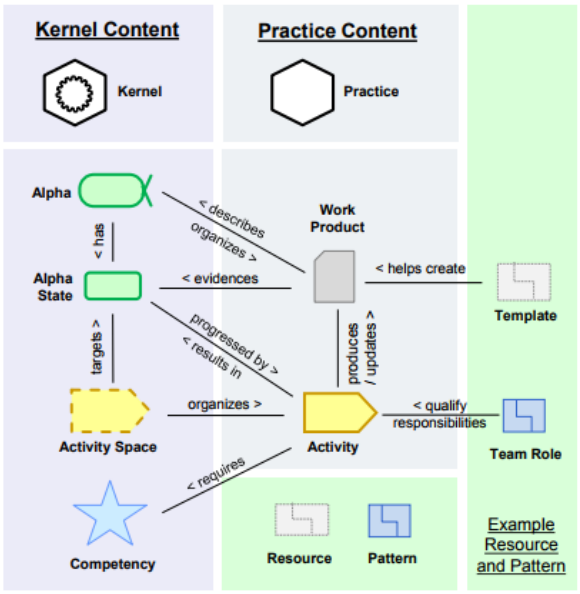

Figure 8: Overview of key language concepts in Essence [REMICS]

3.2 Software architecture evolution modeling

The software architect plays a pivotal role in software development and evolution. Documenting software architecture is extremely important in order to facilitate understanding thus we can analyse it, build systems from it and evolve it. The documentation can be considered from two points of views: firstly, to describe the process of creating an architectural artifact of a software system (descriptive process); secondly, to describe the documents, diagrams, views and documented architectural-design decisions that together describe software systems' architecture (architecture framework) (Clements, Garlan, Little, et al. 2002).

Architecting can be viewed as a decision process (Bosch 2004). Thus architecting evolution modeling aims to capture, document and disseminate the architectural evolution activities, decisions and knowledge that have reshaped the system architecture. In this context, this is more than documenting the architecture in an ADL. It requires the architectural evolution operations and their decisions to be recorded, so that the reasons underpinning the decisions can be better understood. Whenever the architecture is evolved, this architectural evolution knowledge can be captured and acquired as a process model which encapsulates the evolution alternatives and the knowledge that supports the trade-offs between these alternatives. (Novice) architects, (especially those who lack experience in this domain) can benefit from this modeling knowledge. It makes it possible for them to understand the potential evolution scenarios, and to analyse and compare these scenarios, in order to select or synthesise the suitable solution.

A software-architecture evolution model should provide an architectural view of the process of the architecture evolution. Instead of merely providing a descriptive model for the architecture-evolution process, it aims to integrate this model with different views that can support the reasoning process.

3.3 Evolution style

Modeling (visualisation) aims to reduce the cognitive overload in our understanding (Cassou et al. 2009). In this vein, the process model became a critical factor in organisations in order to reduce the overall processes' complexity (Dumas et al. 2013). As a result, several modeling methodologies have been proposed in different areas of software engineering. Likewise, the modeling and the visualisation of software architecture evolution have also attracted some academic attention. In this vein, the term "evolution style" has been introduced by Oussalah (Oussalah, Sadou, and Tamzalit 2006) as a methodology for modeling software-architecture evolution, the aim of which is to capture the main characteristics of a set of activities performed in evolving domain-specific software architecture. The set of practices used in evolving this domain can be grouped together as a library of evolution styles, whereby analysis and comparison of these alternatives can assist the architect in planning and conducting this process [MES]. A number of other evolution styles have been introduced (Le Goaer et al. 2008), (Garlan et al. 2009), (Cuesta et al. 2013), each of which defines an individual method and tool to depict the process using a relevant modeling technique (notations, syntax and semantics) in accordance with a certain point of view (Hassan and Oussalah 2016).

In the following sections we will present some of the evolution styles approaches from different perspectives, looking at their methodologies and solutions with formalism and tool support to model, analyse, and implement evolution reuse in software architectures. These perspectives will be classified in three categories; conceptual (method), theoretical (formalism) and practical (tool).

3.3.1 Garlan et al Evolution Style

Garlan et al. (Garlan et al. 2009) propose an architecture evolution style that provides a completed vision of how to achieve reuse in a particular domain of architecture evolution.

- ***Conceptual perspective:***

This approach defines an evolution style for modeling the architecture-evolution process in order to support architects in planning and reasoning about, domain-specific architecture evolution. The main idea of evolution style is to capture and share the best practices and knowledge relevant to a particular domain of architecture evolution. To this end, they define an evolution style in order to model the potential scenarios of software-architecture evolution formally as a set of finite evolution paths, each of which defines a sequence of architectural states. The transitions between successive states are effected by a set of evolution operators. The paths share common properties and satisfy a common set of constraints. The evolution style defines the constraints that each path must obey. Ultimately, they support the architect in selecting the optimal evolution path from among the set

of all possible evolution paths since an evaluation function is defined to assess the overall utility of each evolution path.

Therefore, the evolution style defines the vocabulary of concepts necessary to model such an evolution process. These vocabularies are: the set of ***operators*** available to define the evolution ***transitions***; a set of ***evolution states*** that represents the stages from the initial architecture to the target architecture, which may be by several paths; a set of evolution-path ***constraints*** that define which paths are permissible in the evolution style; and a set of ***evaluation functions*** that can be used to evaluate the overall qualities of each path. Thus, it is possible to make comparisons between these paths in order to facilitate the selection of the optimal path which satisfies business and management goals.

- ***Theoretical perspective:***

This allows architecture evolution to be represented as a sequence of constrained transitional architectural states, beginning with the initial architecture style (the current system architecture) and ending with a target architecture (final state), along with evolution operators that characterise the transitions between these states. A way of evolving from the initial to the final architecture is an evolution path and it is possible that there may be several these. Constraints are imposed on evolution paths for permitting them to be in the evolution style.

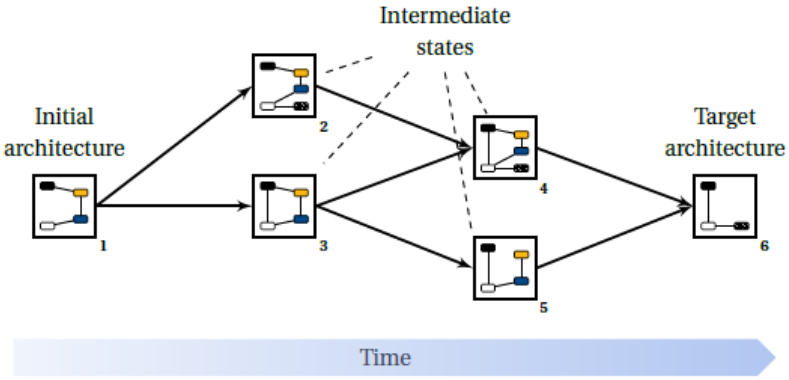

Figure 9: Depiction of an evolution style

- ***The practical perspective:***

This approach presumes that the software architecture is well documented and uses the notion of architecture style to represent the family of architecture and thereby the first and final state are well-represented in formal modeling language. They illustrate the architecture evolution as graphical paths from the first architecture to the target and each path represent a possible scenario of evolution whereby a path can comprise several intermediate states. Two prototype tools have been developed to implement the conception (Figure 9) of the evolution styles. The Ævol workbench (Figure 10) was developed as an Acme Studio plug-in (Garlan and Schmerl 2009). The framework supports the evolution analysis and planning by allowing architects to define an evolution graph and associate its nodes with system architectures that are represented in Acme Studio. With each architectural-evolution transition a set of properties is associated for calculate

the overall utility of the path. Paths comparison can result in reason for the selection of an evolution path. The second implementation was an extension of Magic-Draw (a commercial UML tool). This prototype provides some features and supports multiple architectural views of the evolution state, whereby a component diagram and deployment diagram can be linked with nodes to represent these two views (Garlan et al. 2009).

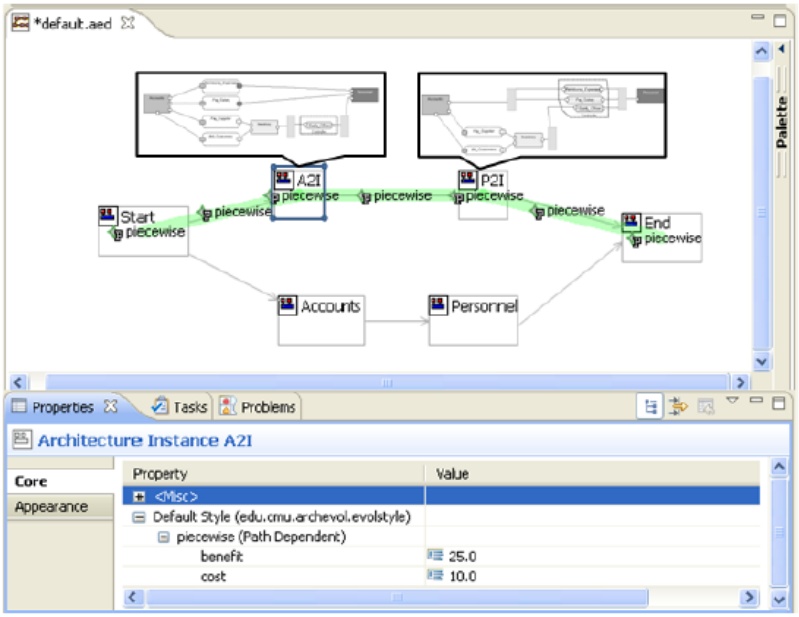

Figure 10: Screenshot of the Ævol workbench

3.3.2 Cuesta et al. Evolution Style

Cuesta et al. (Cuesta et al. 2013) proposed an evolution style called Architectural Knowledge-driven Evolution Styles (AKdES) which considers architectural knowledge as a valuable asset in the evolution process.

- ***Conceptual perspective:***

This approach describes software architecture evolution activities through the concept of evolution styles. Contrary to previous approach, it combines the novel concept of evolution style with the basis provided by Architecture Knowledge (AK) to simplify the reuse; by capturing the information and decisions related to architectural design (evolution). Therefore, topological information and architecture knowledge are both constrained and trigger evolution in software architectures.

- ***Theoretical perspective:***

This approach is centered on (Perry and Wolf's definition) of the software architecture as a model composed of elements (components, connectors), form (constraints) and rationale (motivation). The authors consider the rationale as an active actor in the evolution, one which drives its application and represents the architecture knowledge. Therefore, they emphasise that architecture knowledge is an equally critical driver in the decision-making process in architecture evolution and thus they make the modeling of architectural decisions the centre piece of their approach.

They presume that the connection between the evolution process and the software architecture knowledge is provided by evolution decisions. Each evolution step is preformed because an evolution decision is taken as result of an evolution condition being verified. Therefore the architect, while working with a concrete system, identifies the significant steps (Architectural Design Decisions [ADD]) as a generic situation. Then he gathers sequences of such steps, which denote patterns. Finally, a set of related patterns defines the evolution styles.

- ***The practical perspective:***

The evolution style is conceived as being applied as part of a semi-automatic process: each step in every pattern corresponds to an architectural design decision (ADD) that is stored in the AK. These patterns (evolution styles) are documented in the form of an AK decision tree that can support the automatic implementation of evolution style. The system must detect an evolution decision if the condition is a logical expression and then a sequence of evolutionary steps is followed in the original order in which ADD is applied. Hence, the first step is performed. If the evolution condition has been satisfied, the process can stop; otherwise, the second evolution decision (ED) is applied, the previous ADD is inhibited and a new ADD is taken. This continues until the termination condition is met. ATRIUM (Software Architecture Driven by Requirements) (Montero and Navarro 2009) has been used to validate this approach. ATRIUM provides explicit support for AK and allows the manipulation of its models in an easy way by providing automatic/semi-automatic support for traceability (as it is an MDD process designed to support the development process from requirements to architecture).

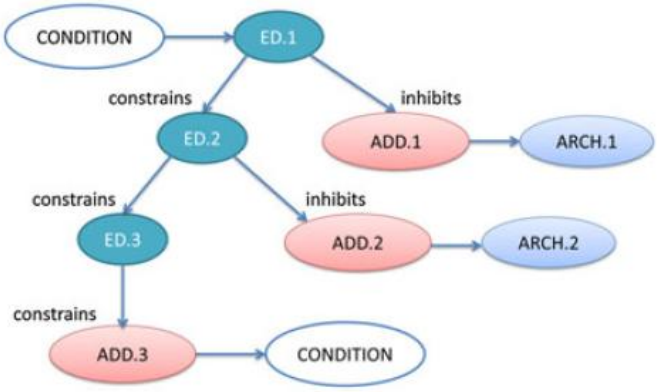

Figure 11: Presentation of evolutions pattern

3.3.3 Oussalah et al. Evolution Style

The concept of evolution styles was first introduced by this research team, with the aim being to capitalise and transfer architectural-evolution knowledge about a particular domain.

- ***Conceptual perspective:***

The main idea of this approach (Le Goaer et al. 2008) is to model software-architecture evolution tasks in order to provide a reusable expertise in domain-specific evolution. To this end, they

provide a classification scheme to describe evolution-style libraries with the ability to acquire, update and retrieve evolution knowledge matching the domain-specific evolution. They also provide assistance techniques for software architects that support and exploit the evolution-styles concept.

This approach considers an architectural evolution as consisting of modifications, in terms of addition, deletion, and updating as first-class entities. In the context of component-based architectures, the component, the connector, the interface, and the configuration were defined as first-class elements of design, which could provide the main concepts for describing a domain-specific architecture. The set of Architectural Elements (AE), evolution operations (modifications) and constraints (which defined how AE could be modified) all of which provide a domain-specific design vocabulary whereby a particular type of configuration represents an architectural evolution style.

- ***Theoretical perspective:***

Authors define a metamodel represented in an object-oriented context (Figure 12). They propose to reify their conceptual evolution style by defining a specific formalism consisting of two complementary parts: the header and the competence. The header denotes an evolution task while the competence denotes a problem-solving method used to achieve the task.

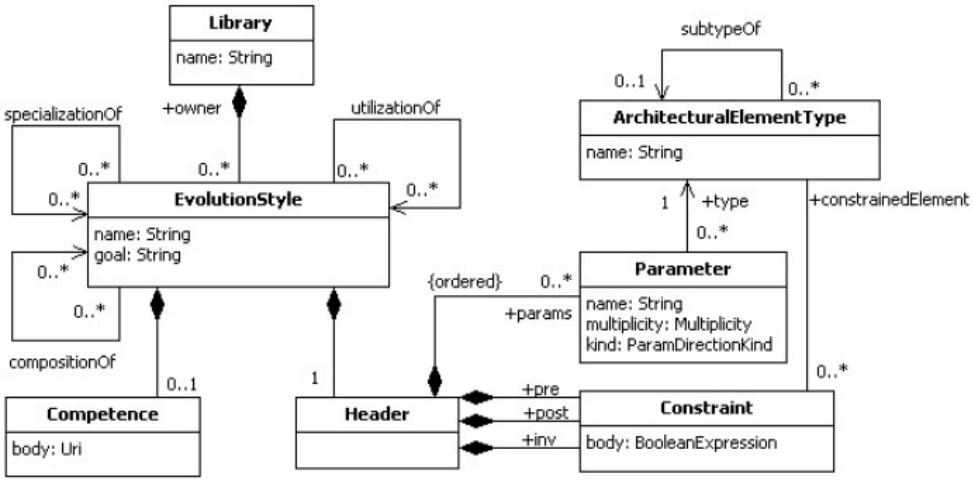

Figure 12: Meta-model of the concept of evolution style

Syntactically, the header is responsible for publishing the interface (the set of parameters defined by the types of design elements provided by a particular architecture style) and the behaviour of the evolution (the three essential constraints for the style are: a precondition, a postcondition, and an invariant), while the competence ensures that the header traces a particular strategy. The inter-relation between evolution styles is defined in three categories: specialization relation, composition relation and utilization relation. Finally, the Evolution Shelf provides a repository which serves as an infrastructure to support the classification, storage and retrieval of evolution styles.

- ***The practical perspective:*** No tool has been developed to implement this approach.

3.4 Conclusions

After the analysis and comparison of the different approaches of evolution styles, it is possible to conclude that, notwithstanding all their solid contributions, there are some important concepts related to standardisation, reusability and interchange among these styles which are still absent, and which need more exploration.

A unified method can also bring other benefits, such as the provision of a better communications environment among the evolution team, ensuring that different categories are on the same terms and allowing multiple views of the evolution process to be used within different information levels (Hassan and Oussalah 2016).

Chapter 4. Evolution Meta-Styles

4.1 Introduction (Model-Based Engineering)

Modeling is a fundamental concept in software engineering. A model is an abstraction of reality that has been used for several issues, such as analysis, problem-solving, decision support, training, testing, research and education. From the software-engineering perspective, a model has been used to form a conceptual expression the two major parts: software product and software process.

With the increase in the size and complexity of software systems, controlling and understanding every detail became mostly impossible using a general-purpose programming language (textual documents). This has grabbed more attention on models as an important enabler in the analysis, designing, developing and testing of a software-intensive system.
Therefore, Model-Based Engineering (MBE) was introduced with the aim of supporting developers during the entire development process (analysis, design, validation, testing, documentation and evolution) with several CASE tools being developed to accommodate this trend. Here, models aim at helping developers by providing an insight into the system under development (analysis and design). They endeavour to enhance communications between stakeholders and to assist them to r manage the complexity better. MBE takes the software-engineering process from the document-centric to the model-centric paradigm.
Model-Driven Engineering (MDE) has been suggested as a software engineering methodology the aim of which is to leverage models to first-entity by shifting the focus of the software-engineering process from coding to modeling (Kent 2002). Thus, modeling can be considered as a high-level programming language that enables the developers to manage the complexity of the system (Levendovszky et al. 2002). To this end, transformation techniques have been introduced in order to transfer models between different modeling (abstraction) levels and meta-models have been suggested to specify the modeling languages. Model-Driven Development (MDD) is the software-engineering process that relies on the principles of MDE. Several approaches (Jeusfeld, Jarke, and Mylopoulos 2009) have applied the MDD process, such as Model-Driven Architecture (MDA) (Kleppe, Warmer, and Bast 2003), Model Integrated Computing (MIC) and Eclipse Modeling Framework (Küçükkeçeci Çetinkaya 2013).

The metamodels are often utilised to define the concepts that the modeling languages provide as well as how composing them to the models. Based on the metamodels, the modelers build editors and code generators to support the use of the modeling language. These tools provide users with graphical notations by which they can build models that conform to the metamodel.

4.2 Model, Modeling Language and the Meta-model

A model is an abstract description of an original, the aim of which in software engineering is to reduce the complexity of software development by abstracting away the details of the implementation level.

From the software engineering perspective, a model can be used for several purposes. First, it can be used only for documenting software system, thus making it easier to understand the decisions behind the design. Second, a model can be used for requirements-gathering and validation by checking the model (system prototype) against the system requirements. In addition, a model can be used for testing by stimulating the behaviour of the system. Moreover, a model can be used to generate the implementation code of the system in order to improve the development productivity.

Whatever the purpose of the model was, the modelers need a modeling language by which they can define the model. Ideally, models are built by using adequate modeling languages (Bézivin and Heckel 2006). A language usually consists of syntax and semantics. The syntax specifies the vocabulary of concepts that the modeling language provides as well as showing how these may be combined to create models. Often, language engineers build a model of the syntax of the modeling language — a so-called metamodel — which is in the center of the definition of a modeling language. This metamodel represents the abstract syntax, based on which the concrete syntax and semantics of the modeling language (and mapping) are (implicit) defined (Clark, Sammut, and Willans 2015). Since the abstract syntax conveys little information about the meaning of the language concepts, additional information is needed in order to capture the semantics of the modeling language in better way. Defining the semantics for a language is important in order to be clear about what the language represents and means (Clark, Sammut, and Willans 2015).

Therefore, it can be stated that metamodels are the primary asset by which modeling language artifacts can be expressed and they are therefore the essence of DSMLs and of many of their development tools (e.g. EMF, MS SDL, Sirius). As the metamodel (the abstract syntax) of the language is independent from the concrete syntax and the semantics, more than one concrete syntax and also the semantics can be associated with the metamodel, i.e. a metamodel can instantiate different models each of which is defined by a different modeling language and thus the metamodel can specify or represent the abstract syntax for more than one modeling language.

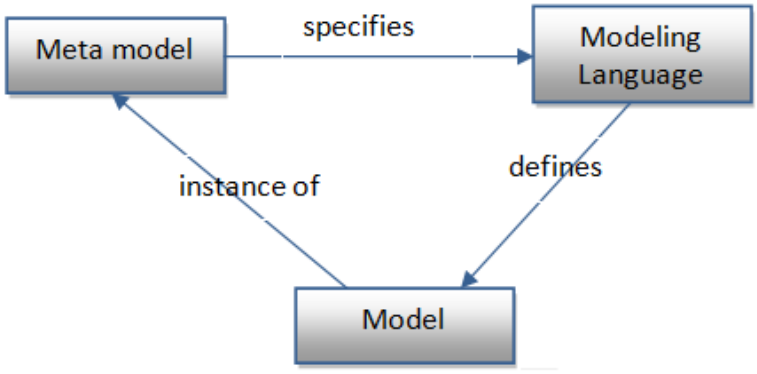

Figure 13: Modeling language

The purpose of the metamodel is to represent the modeling language in an abstract form in order to provide a proper way of developing models in conformance with that language. Thus, in order

to develop a model for a problem in a particular domain, a metamodel is first defined and this specifies the main concepts that can cover a larger set of problems in that domain. Then, a modeling language based on this metamodel is used to express the model. Here, the model is said to be an instance of the metamodel, whereby each element of the model is an instance of some element (its meta-element) in the metamodel. Meanwhile the specification relations express the link between the metamodel and the modeling. Figure 13 illustrates this relation between model, metamodel and modeling language.

4.3 Metamodeling

Metamodeling is abstracting the syntax of the modeling language, the process of precisely specifying a modeling language in the form of metamodel, or the process of creating a metamodel. A metamodel specifies the modeling language which the modelers use to define models. This metamodel is defined by, or written in, a metamodeling language and is therefore similar to a modeling language that can also be specified by a metametamodel. (i.e. this metamodel is defined by a metamodeling language, which can also be specified by a meta-metamodel). Figure 14 illustrates the metamodeling process (technique) and the relations between models and modeling languages at different levels. This allows different metamodels to be described by a single meta-metamodel.

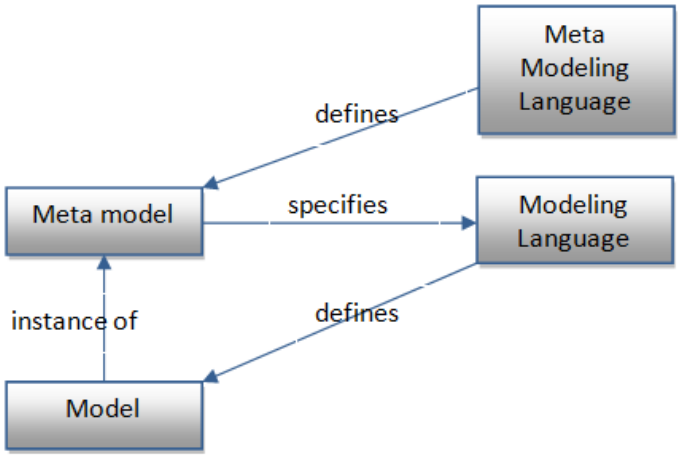

Figure 14: Metamodeling

In this context, the metamodeling emerges as an ascending technique, the benefit of which is to describe several languages in a unified way, which means that the languages can be uniformly managed and manipulated, thus tackling the problem of language diversity.

Several meta-metamodels have been proposed, all of which aim to control the variety of modeling languages and manage the translations between them. Among the popular metamodels are: Meta-Object Facility (MOF[4]), Eclipse Ecore metametamodel[5], MetaGME (Emerson, Sztipanovits, and

[4] https://www.omg.org/spec/MOF/2.5/

Bapty 2004), and MADL (Smeda, Oussalah, and Khammaci 2005). Moreover, several meta-modeling languages [tools/meta-modeling environment] that have been used to defer the metamodels to languages or editors such as Generic Modeling Environment (GME), AToM3, MetaEdit+, Microsoft SDK DSL, Sirius, and ADOxx.
These metamodeling environments significantly reduce the effort need to build modeling languages' tools (editors) based on metamodels. They provide the means to edit, parse, and interpret models in accordance with the modeling language.

4.3.1 Meta-Object Facility MOF

Due to the rapid growth of the domain-specific modeling paradigm (method, tools), different modeling languages have been developed, which necessitate a common means that can embrace this variety and manage the transition between them.
To this end, in order to permit the transformation between these models (or their instances objects), additional specification must be provide to describe precisely the structure of these objects – this is the aim of Meta-Object Facility (MOF).
MOF is standardised by the OMG (Object Management Group) as a language to define the metamodel of the modeling language. Thus, MOF specification has become the de-facto metamodeling language in OMG for many modeling languages (UML, SPEM, CWM, Java EJB, EAI etc) where models can be created, integrated and then transformed into different formats (for example, the set of rules used to transform an MOF model into a UML model has been derived from the OMG UML profile for MOF specification).

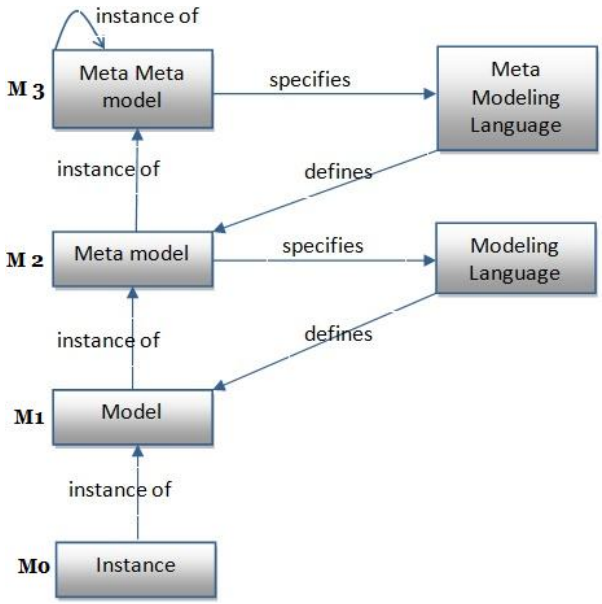

Figure 15: Four-layer Metamodeling Architecture

[5] http://www.eclipse.org/modeling/emf/

The metamodeling in MOF is architecturing in a four-level hierarchy, as Figure 15 shows. The M3 level consists of the MOF meta-metamodel which is self-defining to avoid an unnecessary stack in the number of meta levels. It is used to define the M2-level languages. The M2 level consists of metamodels which are instances of the MOF model. This level provides metamodel specifications, defining the syntax and the semantics of M1-level elements. The M2 level includes languages such as the UML, CWM, SPEM, Java and XML. The M0 level consists of model instances. These represent models with a user data which might be instantiated class objects, instantiated database tables or Java/XML code. Table 1 shows the four modeling levels introduced by OMG with an example of UML language.

Table 1: Four-layer metamodeling architecture

Modeling Level	Descriptions	Model	Concept example
M3 metametamodel	An instance of itself. Specifies the language for defining metamodels	MOF	In MOF ➔MetaClass and MetaAttribute
M2 metamodel	An instance of the meta-metamodel. Specifies the language for defining models.	UML, CWM, SPEM	In UML➔Class and Attribute
M1 model	An instance of the metamodel. Specifies the model.	Model	Customer and Name
M0 original	An application instance of the model. Specifies the data value.	System	Cus0132424 Name = "John"

The metamodeling process has also been used in the domain of software architecture, as mentioned in Section 2.5. Oussalah et al. (Smeda, Oussalah, and Khammaci 2005) have applied the metamodeling to the component-based model, as OMG has done to the object-oriented approach . They define Meta Architecture Description Language (MADL) as meta-modeling language for the ADLs.

We argue - in line with other engineering disciplines - that the meta-modeling technique can bring many benefits to the domain of evolution styles (Hassan and Oussalah 2016).

4.4 Metamodeling in software-architecture evolution

Several evolution-styles approaches have been developed, each of which identifies the modeling elements to express the architecture evolution and define the mechanism for modeling this process. A variety of instantiations of a process model (evolution style), which represents a set of possible ways to restructure the system to form the intended architecture, can provide a reusable knowledge for architects, assisting them in planning and carrying out the evolution process in this system's domain. Thus, each approach is defined in a way that reflects the perspectives of the modeler with his own conceptual vocabulary (syntactic notations) and a mechanism/tactic, composed of a set of interactions between these vocabularies, to model software-architecture evolution. In the modeling methodology we can consider the evolution style as a modeling

language (meta-model) that specifies the modeling notations to define a particular model of the software-architecture evolution process. Therefore, we have to find a way to approach these concepts (of evolution styles) and metamodeling is one of the best techniques to adopt, providing many benefits in different modeling engineering fields (Ter Hofstede and Verhoef 1997). In this context, our approach is compatible with the four-layer architecture promoted by the Object Management Group (OMG). Ultimately, its aim is to specify a metamodeling language for software architecture evolution. Figure 16 shows a proposed metamodeling architecture of evolution style.

The meta-models provide the abstract syntax of a modeling language, which means that all available modeling concepts are defined, as well as the permitted ways to combine them in order to create models. Therefore, the semantics of the basic concepts of our approach will be on an ontological basis. "The ontology explicitly expresses the semantics of the modeling concepts whose syntax is defined by the metamodel." (Brand et al. 2011).

Ontologies can support translation between different languages and representations; moreover, modules (concepts) can be imported or exported among different evolution styles (White, Schmidt, and Mulligan 2007). In the same vein, a meta-evolution style will use ontologies as a means to support the mapping among evolution styles (for finding equivalences between evolution styles' elements), as well as with the relevant modeling languages.

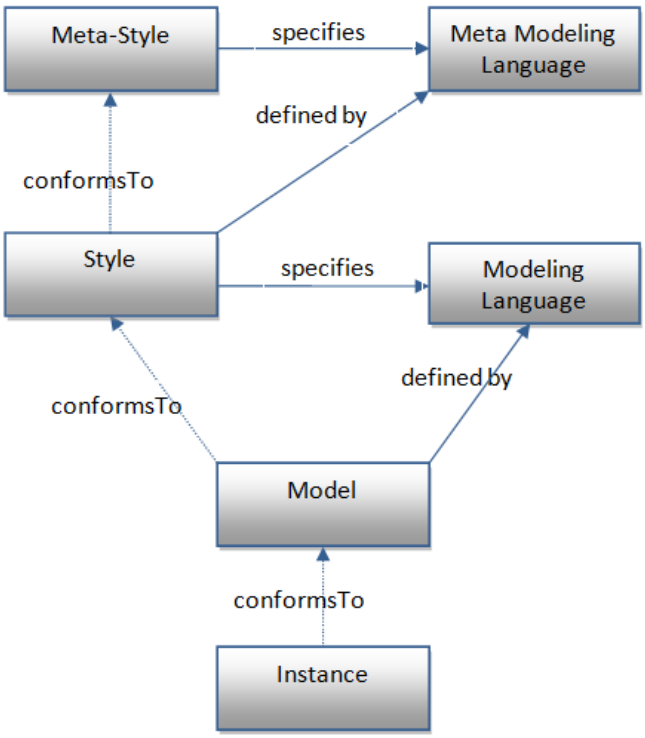

Figure 16: Four Layer Meta-Style Architecture

4.4.1 The basic concepts of architecture evolution

A meta-level should consist of the necessary (meta-) concepts that can instantiate elements of the level below. Each concept in the level below should have only one meta-concept, while meta-concepts can have more than one instance in the level below.
To this end, this section endeavours to capture the main concepts that can provide the abstract syntax for the evolution styles.
Software-architecture evolution is a process whereby system architecture is modified through a series of steps, leading finally to the target architecture, along with the process roles which participate in this process. Each step may be comprised of a set of operations used to transfer the architecture towards the final state. In accordance with this generic view and with definitions of the evolution-styles' approaches [in Section 3.3], a basic concept for software-architecture evolution is proposed (as Figure 17 shows), which consists of the core elements of this process, by which evolution-style elements can be instantiated. This concept formulates the foundations of a meta-style.

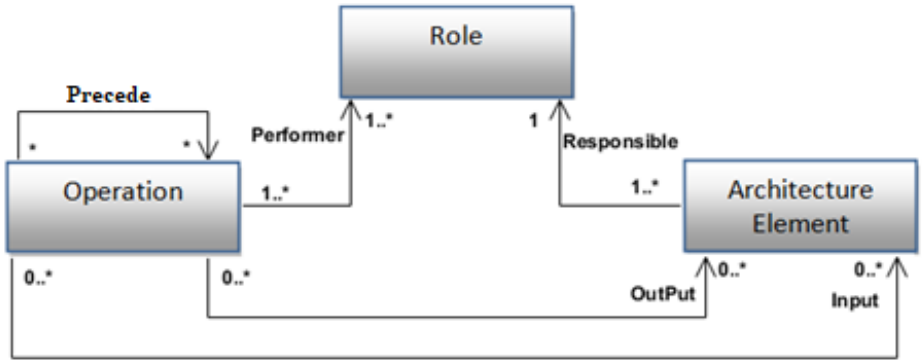

Figure 17: The basic concepts of evolution styles

The evolution meta-style will be based on this concept and will be sufficiently abstracted to specify the metamodeling language for software-architecture evolution. A critical problem in today's practice of software-architecture evolution style is that it is missing a high modeling level that could express all the style elements. Therefore, our contribution assumes that it introduces an evolution style at a higher layer (meta-style) that can offer a common language to facilitate the representation, reuse, and sharing of the architecture-evolution knowledge and practices. The construction of the basic concept, as illustrated in Figure 17, is composed of the three core elements and their interrelation as follows. 1) Operation: to define the activities. 2) Role: to define the stakeholders, tools and techniques. 3) Architecture Element: to define the inputs and outputs architecture of the operation.

Software-architecture evolution is a specific phase of software development that provides a framework to support software evolution and its workflow. The software engineering discipline studies the processes of both product engineering and process engineering (Rombach and Verlage

1995). Our approach exploits and emulates the SPEM methodology (Caplat and Sourrouille 2005) in defining a metamodel for evolution style.

4.4.2 Evolution Meta-Style MES

Software-architecture evolution process modeling aims to capture the main characteristics of the set of activities performed to evolve software architecture. For better process modeling which expands the reuse of experience and knowledge, a variety of styles have been created. To reason out and unify the modeling concepts that formulate these styles, which represents different perspectives of modelers, a metamodeling language (meta-style) comprising the necessary vocabularies is needed.
Meta-modeling, as we mentioned, consists of ascending techniques used to define a higher level, where a minimum number of concepts are, in order to define and reason about models in the level below. Meta-style is an intended layer which has the essential concepts to specify an architecture-evolution metamodeling language. To remain compatible with the four modeling levels of the OMG, each evolution model is an instance of the model in the level above (its meta), including the meta-style which is an instance of itself. The meta-style introduces the essential elements and their interrelationships that represent the concepts required for modeling the architecture-evolution process. These concepts are: *Operations, Roles, Architecture Elements, Interfaces, and Interactions*. These essential concepts are comprised in the meta-evolution style called MES, illustrated in Figure 18. MES is a component-oriented concept for modeling the software-architecture evolution process. Indeed, it is a reflective concept, whereby the concept of the component is reflected on the modeling of the process that evolves it. This is in the sense that everything is a component, such as class in object-oriented modeling where everything is a subclass of the abstract class "model Element". Thus, the component is the basic evolution entity of the MES model.

Role: This represents a set of responsibilities, i.e. it describes the required skills, tools and techniques required to perform a specific architecture-evolution operation.

Operation: This is one of the units of the process that produces visible changes in the architecture state. The operation is associated with roles and architecture elements through the interaction elements.

Architecture element: This refers to the inputs and outputs of an architecture-evolution operation. An architecture element produced by an operation can be used later as raw material for the same-or another-operation in order to produce other architecture elements.

Interaction: This represents an entity that governs the relationships between these elements and their behaviours. Generally, it determines what roles participate in an operation and what kind of inputs and outputs there should be.

Interface: This refers to a place in which elements interact and which determines rules that should be followed by elements in order to intercommunicate compatibly.

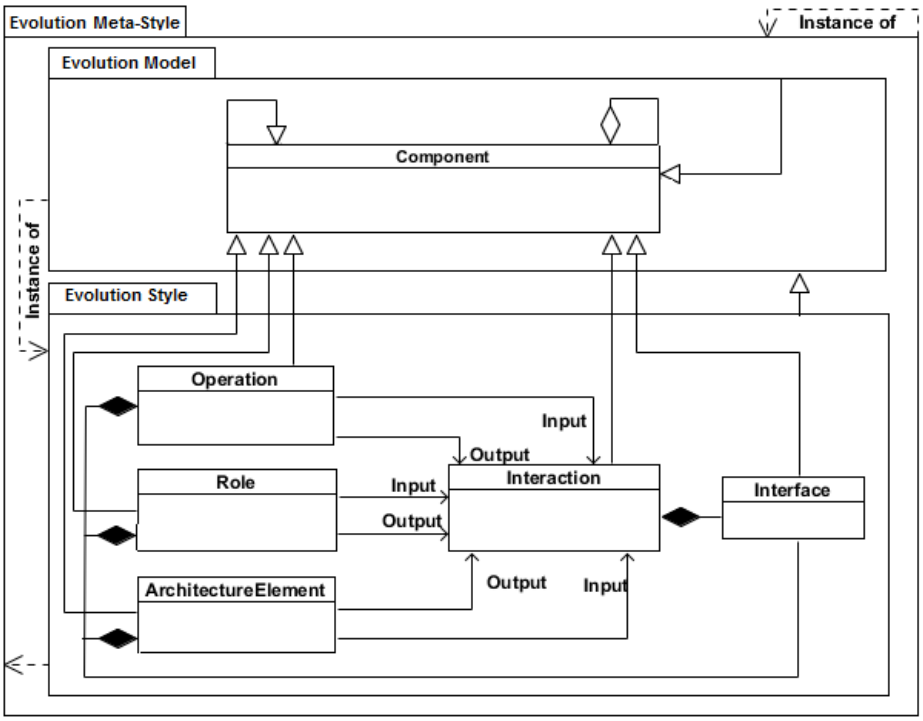

Figure 18: Evolution Meta-Style MES

Normally, an architecture evolution can be implemented in different ways. In this context, the way to the target architecture from the current architecture might take different trajectories, depending on both the initial and target styles and on stakeholders' needs, constraints, and perspectives. Evolution planning is extremely difficult, considering the multiple choices and the different ways of evolving the system. Therefore, different models (instantiations) of the implementation could be established; a meta-evolution style MES aims to provide the core elements for defining any meta-model (style) and for managing the comparison between them.

For example, Figure 19 shows a proposed for a metamodel for an evolution style based on a general conception of re-architecting the workflow (as well as Garlan's style). In general, a metamodel provides a specification of the concepts, the relationships that exist between these concepts and rules for better formation, which declare how these concepts can legally be combined.

This metamodel expresses the abstract syntax of an evolution style, comprising a set of architecture states and transitions. Transitions link successive states whereby a sequence of transitions between the internal and the final state formulate (an evolution path) one possible way of system evolution.

Each architectural transition may be conducted using one or more evolution operations. An architecture-evolution operation may be a single operation, such as adding an architecture element operation or a composed operation such as replacing an architecture element.

The abstract class of ArchitecturState has three instance types: InitialState, IntermediateState and FinalState. The evolution style must have at least two states, the current architecture (the initial

state) and the target architecture (the final state). Thus we can see the relations between the Evolution style class and the InitialState and FinalState classes.

The rule that governs the relations between successive states has been applied to the ArchitecturState class which states that each ArchitectureState has one or more following - as well pervious – states and this permits the development of multiple paths. The InitialState must not have a pervious state and the FinalState should have not a following state. These two exceptions to this rule can be applied at the implementation level (e.g., using Object Constraint Language [OCL]).

However, the model is not created in vain; it should be associated with an original. Each concept or rule that the model consists of should have (stimulate) its counterpart in the original and this is called semantic mapping.

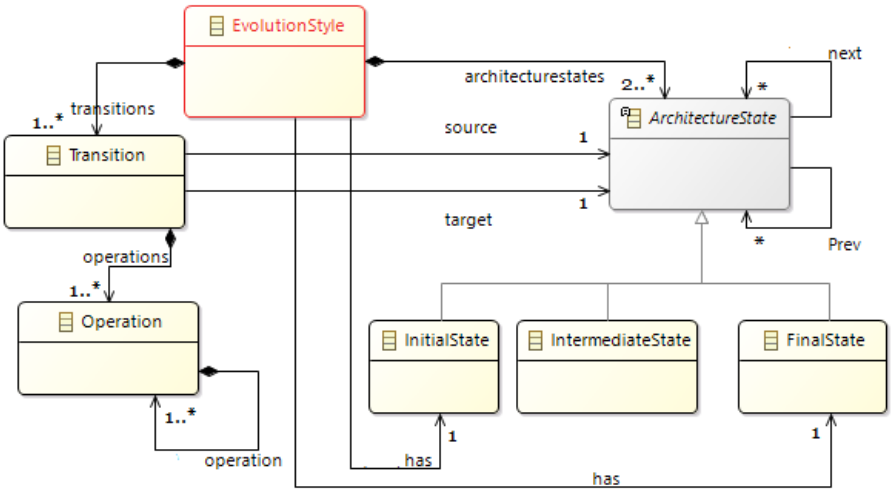

Figure 19: Metamodel for an evolution style

Languages expressing models do not exist in isolation. They need to be translated to each other among different abstraction levels (Vertical) for code generation, or - in the same abstraction level they need to be translated from one language to another, for example from UML to XML.

Often, this translation is specially done for each pair of languages in a directional or bidirectional way and a model transformation is usually defined using a transformation language (model to model or model to text) such as ATL, QVT, ETL, OAW, Xpand, or Acceleo.

Mapping is the first step of the validation and translation process. Figure 20 shows the mapping between the proposal metamodel and the evolution meta-style MES to investigate whether MES has the necessary meta-concepts that can instantiate the concepts of the proposed evolution style.

Table 2: Conformation of MES with a proposed style

Proposed style	MES
Transition, Operation	Operation
Architect	Role

ArchitectureState	Architecture Element
Association, constraints	Interaction
Attributes	Interface

Table 2 represents the corresponding elements in MES and the proposed style, with the first column referring to the elements of the proposed style and the second column including those counterparts that are the most suitable meta-concepts (super-element) in MES of each conceptual element in the proposed style.

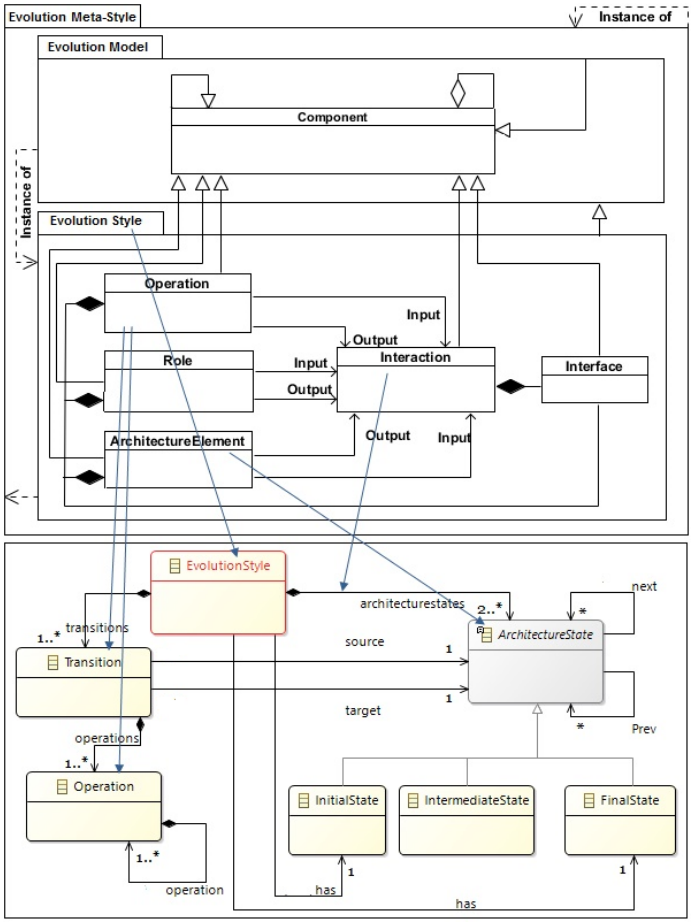

Figure 20: Mapping between evolution style and MES

4.5 Meta-style based Transformations

The introduction of DSM has yielded a number of different, yet related, modeling languages that attempt to address very similar problems (domains). This phenomenon necessitates the need for the transformation of models into other models, otherwise every existing model must be developed and understood separately and manually converted into other forms.

However, when automatic model transformations are used, the mapping between the metamodel concepts has to be developed only once. It can then be used for all their instances. Defining the mapping between two conceptually different models requires a common basis that describes both the source and target domains. In this context, meta-modeling techniques can offer a common language (for the models in the level below) to facilitate the reuse and representation of models specified in one domain-specific modeling language in another domain-specific modeling language. Metamodel-based mappings permit descriptions of transformations between models created using different concepts from possibly overlapping domains (Levendovszky et al. 2002). In the same vein, an evolution meta-style can facilitate the evolution-style mapping and unify the transformation rules for these evolution styles and other solution domains.

A meta-modeling technique as previously mentioned is a powerful technique which enables us to reason about the mapping between models. One of the important reasons for defining MES is to manage the transformations between evolution styles. Therefore, our approach aims to provide solid principles for evolution-style transformations with a twofold benefit: on the one hand, it permits mapping between different evolution styles; on the other hand, it permits mapping between these styles and other meta-models (e.g. those defined by MOF such as SPEM and UML) to improve the process efficiently and to exploit the best practices and knowledge (frameworks and tools) in the domain of object modeling.

4.5.1 Vertical Mapping

In the context of Model-Driven Engineering, models are the primary artifacts and model transformations are one of the most important operations applied to models.
The basic assumption in using modeling levels is the possibility of serialising model at different levels and utilising the reflection of elements at a higher level to determine their counterparts at the succeeding level. We can relate or compare models that are on the "same" modeling level with horizontal mapping, while vertical mappings relate or compare models at different modeling levels. For example, the mapping between the Platform-Independent Model (PIM) and the Platform-Specific Model (PSM) in the MDA paradigm represents vertical mapping. Figure 21 illustrates the concepts of the V and H mapping.

The vertical mapping between MES and evolution styles provides two benefits: first, it can be used to evaluate the ability of MES to define an evolution style; second, it can facilitate the matching of corresponding elements between a pair of evolution styles, where counterparts should have the same meta-element in MES. Moreover, it can also be exploited in mapping an evolution style with MOF-compatible models.

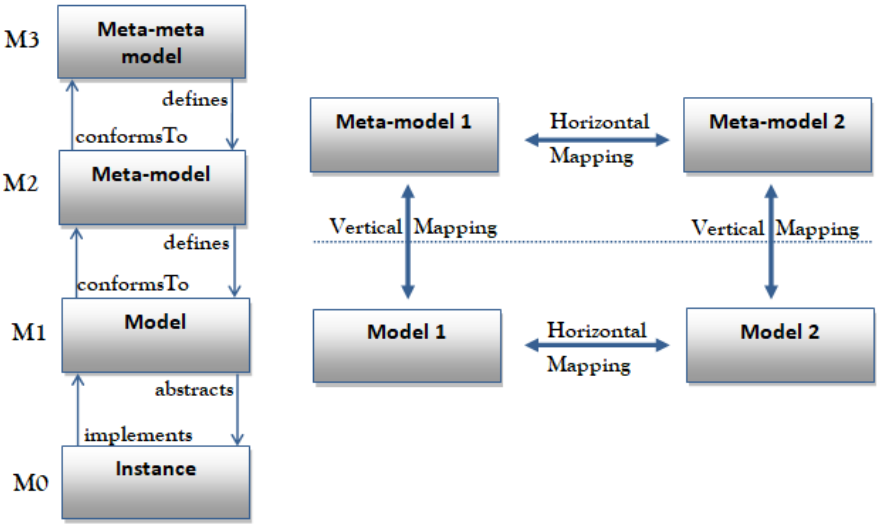

Figure 21: The conception of models mapping

4.5.1.1 Mapping between MES and Garlan et al. evolution style

One of the vital reasons for defining MES is to set up the rules to govern the transfer of evolution styles; i.e., to investigate the efficiency of MES in defining and transferring evolution styles models. A vertical mapping between MES and an evolution style (Garlan et al.) is reported in this section in order to conform MES with this style.

Garlan et al. define the evolution styles as follows:-

> "*An evolution style describes a family of domain-specific architecture evolution paths that share common properties and satisfy a common set of constraints*".

An evolution style, then, provides the vocabulary of concepts necessary to define and analyse such an evolution model: the set of operators that is available to define the evolution transitions (which represent the evolution operations that can be carried out in the domain at hand), a set of evolution path constraints that define which paths are permissible in the evolution style (which capture elements such as ordering constraints or invariants that must hold for all nodes), and a set of evaluation functions that can be used to compare different evolution paths with respect to quality metrics (which are used to facilitate selection of an optimal path). (Barnes, Garlan, and Schmerl 2014)

According to this definition (conception of evolution styles), a meta-model has been derived which is shown in Figure 22. This metamodel will be used to facilitate the (semantic) mapping between MES and Garlan et al. evolution styles in order to check MES conformance with an evolution style instance.

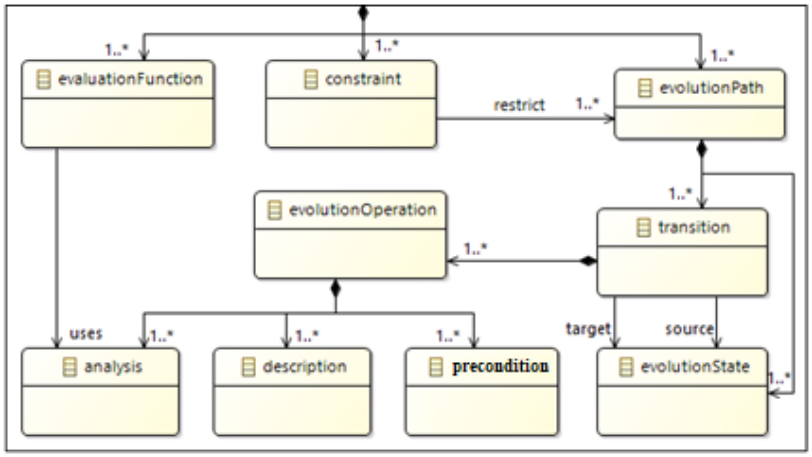

Figure 22: Metamodel for Garlan style

Model validation and verification activities are key features for checking the conformance of the built model to the original (in order to verify whether the models behave as expected). This can usually be provided as mapping from the abstract syntax (metamodel) to the semantic domain. This process can be also conducted to check the equivalence (conformance) between different modeling languages in order to verify whether models can be translated from/to each other.

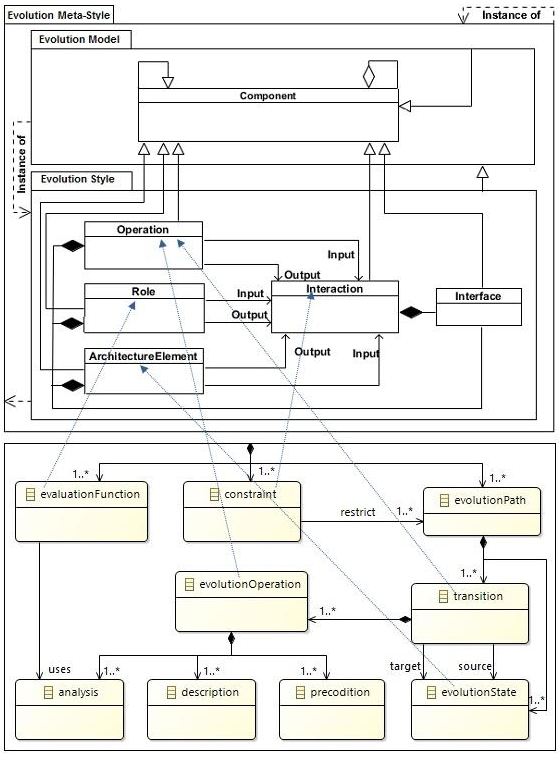

Figure 23: Mapping Garlan style to MES

Languages can have equal syntaxes with radically different semantics, just as they can have different syntaxes with equivalent mappings to a certain semantic domain. Mapping is one of the methods

used for model conformance checking and in order to find the counterparts or equivalent elements in source and target languages. It can be based on an ontological domain, structural semantics or on the behavioural semantics.

Ontology mapping aims to finds the equivalent elements which serve the same goal. The notion of completeness at the level of language is related to the notion of ontological expressiveness, whereby every concept in a domain is covered by at least one modeling element of the language. Therefore, if two languages are ontologically expressed in a certain domain, this means that a model in one language can easily be expressed by the other language.

Ontology-based mapping has been widely used as a solution for managing semantic interoperability (Arnarsdóttir et al. 2006). Thus, we use ontology as a base for the checking the semantics conformance between the MES and evolution style approaches. Figure 23 show a mapping among MES and Garlan et al. evolution styles.

Table 3 represents the corresponding elements in MES and Garlan styles, indicating the most suitable meta-concept (super-element) in MES for each conceptual element in the Garlan style.

Table 3: Equivalent elements between MES and Garlan et al. evolution style

MES	Garlan et al. Style
Operation	Operator, Transition
Role	Architect, Evaluation Function
Architecture Element	Evolution State
Interaction	Constraint
Interface	Does not have explicit element

4.5.1.2 Mapping between MES and Cuesta et al. evolution styles

Cuesta et al. have defined their own evolution styles, as mentioned earlier (Section 3.3.2). Figure 24 shows a metamodel that abstracts their styles. MES can present an ontological mapping of their evolution styles as follows: the evolution pattern and evolution step can be defined by the operation element in MES, the evolution decision, evolution condition and AK can be defined by the role element in MES and the architectural element can be defined by the architecture element in MES. Figure 25 illustrates this mapping.

Table 4 represents the corresponding elements between MES and Cuesta style. The first column contains the elements of MES. The second column comprises their counterparts in Cuesta et al.'s evolution style.

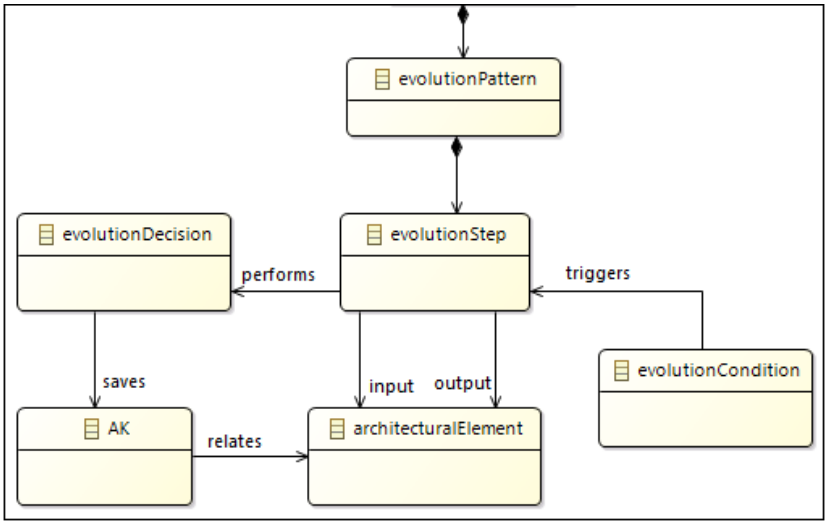

Figure 24: Metamodel for Cuesta et al. evolution styles

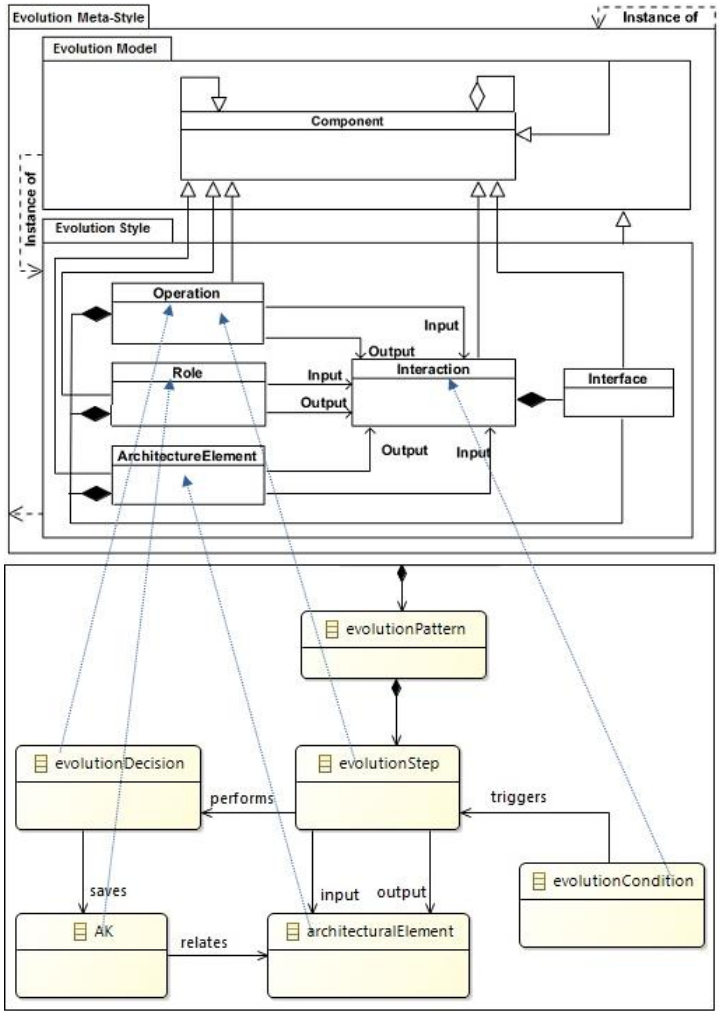

Figure 25: Mapping Cuesta et al. evolution style to MES

Table 4: Equivalent elements between MES and Cuesta et al. evolution styles

MES	Cuesta et al. Evolution Styles
Operation	Evolution Step
Role	Architect, Decision, Architecture Knowledge
Architecture Element	Architectural Element
Interaction	Condition
Interface	Does not have explicit element

4.5.1.3 Mapping between MES and Oussalah et al. evolution styles

The SAEM (Style-based Architectural Evolution Model) was developed by Oussalah et al. (Le Goaer et al. 2008). It is previously mentioned in Section 3.3.3. They defined a metamodel for their evolution style as shown in Figure 26. An ontological conformance between the MES meta-style and SAEM evolution style can be done in this way: the competence and header can be associated with operation, constraint can be defined as a role and the architectural element type can be defined by the architecture element in MES. Figure 27 illustrates this mapping.

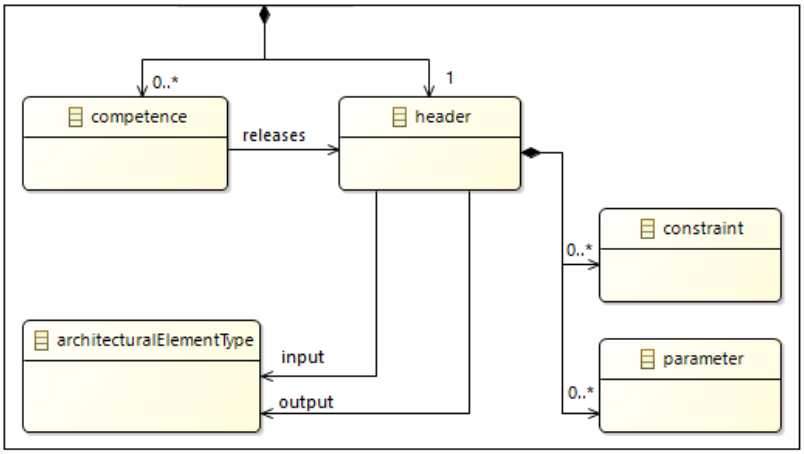

Figure 26: SAEM metamodel

Table 5 represents the corresponding elements from both the MES and SAEM metamodels, whereby the first column comprises the elements of the SAEM metamodel and the second column includes their counterparts in MES.

Table 5: Equivalent elements between MES and SAEM

Oussalah et al. Evolution Styles	MES
Header, Competence	Operation
Architect	Role
Architectural Element Type	Architecture Element

Constraint	Interaction
Parameter	Interface

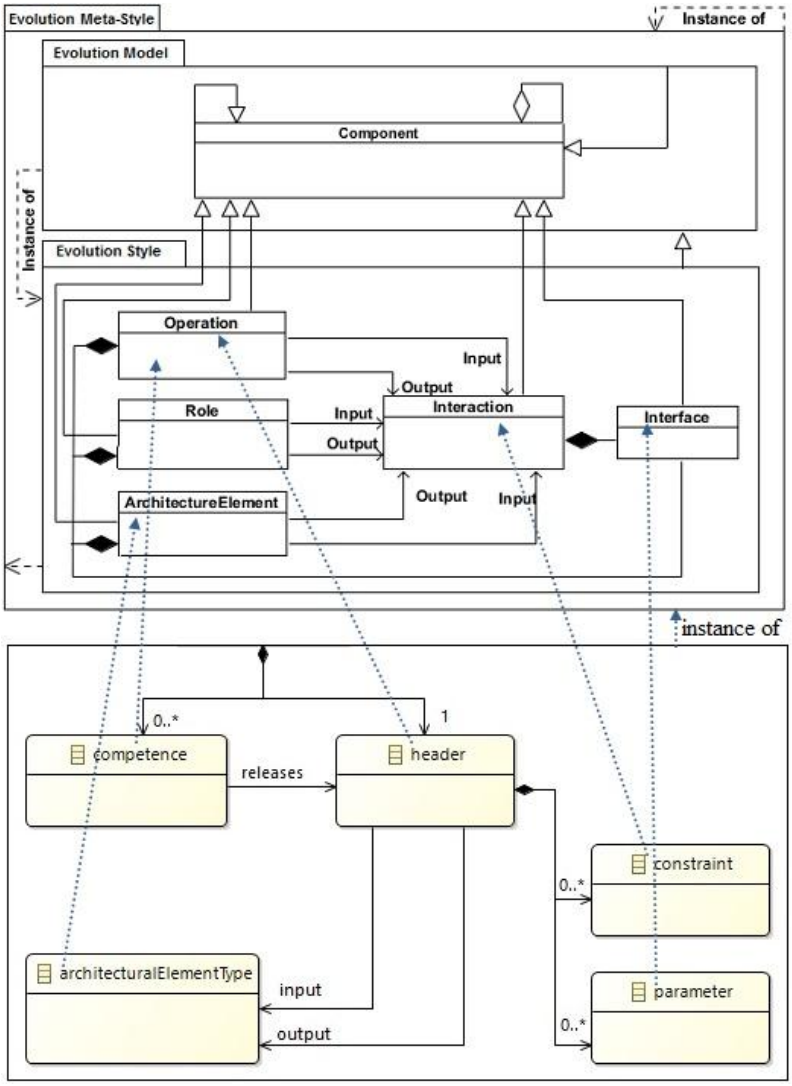

Figure 27: Mapping Oussalah et al. evolution style to MES

4.5.2 Horizontal Mapping

A vertical model transformation results in a change in the level of abstraction. The level of abstraction can be increased by a refinement transformation (Top-Down transformation) or decreased by an abstraction transformation (Bottom-Up transformation). Meanwhile, a horizontal-model transformation leaves the level of abstraction unchanged and changes the representation of the model (Model-to-Model [M2M]).

Horizontal mapping can relate or integrate models at the same modeling level which might be defined by the same meta-model or by different meta-models; this mapping could be at any level. For example in the Model-Driven Architecture, a vertical transformation can be done when moving from a Platform-Independent Model (PIM) to a Platform Specific Model (SIP) or even to code. We use a horizontal model transformation when a migrating model to other modeling languages at the same level. In this section, we are first going to consider horizontal mapping at the level of a metametamodel between meta-style evolution (MES) and MOF.

MOF is the foundation of OMG's industry-standard environment; it is a meta-language for many modeling languages (UML, SPEM, CWM, Java EJB, EAI, etc) and supports translation from one to another (for example ,the set of rules used to transform an MOF model into a UML model has been derived from the OMG UML pro file for MOF specification[6]).
In general, mapping is a complex task. To be precise, mapping is a particular directional or bidirectional translation solution defined for two specific models, so it cannot be reused for other models. Indeed, meta-modeling provides many opportunities for facilitating this process, whereas in the upper level there are always fewer conceptual elements to be translated.
Therefore, we enact the set of rules that are used to transform an MES compatible model to an MOF-compatible model. For example, an evolution style can be transferred to an EPF (Eclipse Process Framework) (Haumer 2007). In the same vein, MES provides two benefits: first, it unifies the rules transferred from the domain of software architecture evolution modeling to OMG's modeling environment; and second it achieves the mapping and transfer of knowledge between the evolution styles.

4.5.2.1 Mapping MES concepts to MOF

As mentioned above, MOF is standard throughout the industry widely supported and accepted as the cornerstone of an open information interchange model. Indeed, this standardised method is bringing consistency, interoperability and compatibility to applications created in collaborative environments.
Following this standard by defining rules translated from MES to MOF will help architects to model the evolution styles using MOF compatible technologies.
Actually, MOF is a metametamodel, and is also used to define a metamodel for software process and software architecture. However, MOF has some limitations regarding the architectural description languages (Oussalah 2014) as well in the modeling of software architecture evolution (evolution styles) and we have defined MES to address the following shortcomings:

- the absence of meta-entities representing meta-style evolution;
- the absence of meta-entities representing evolution style;
- the absence of a meta-entity representing evolution operation;

[6] https://www.omg.org/spec/MOF/2.5/

_ the absence of a meta-entity representing architecture;

_ the absence of a meta-entity representing role;

_ the absence of a meta-entity representing interaction;

_ the lack of an explicit relationship between style and its meta-style

In summary, we can consider MES as a UML profile which defines the necessary stereotypes (tagged values, and constraints) for adapting the UML meta-model in order to fulfill the modeling requirements of software-architecture evolution

It can be stated that mapping rules between MDA/MOF and software Architecture Evolution/MES have more benefits at this level than at those levels below.

_ At this level there are fewer elements to be translated;

_ These mapping rules can be reused by every style in the domain of software;

_ Architecture evolution that is compatible with MES;

_ The translations from MOF to its metamodels (e.g. UML, SPEM) already exist and are well defined.

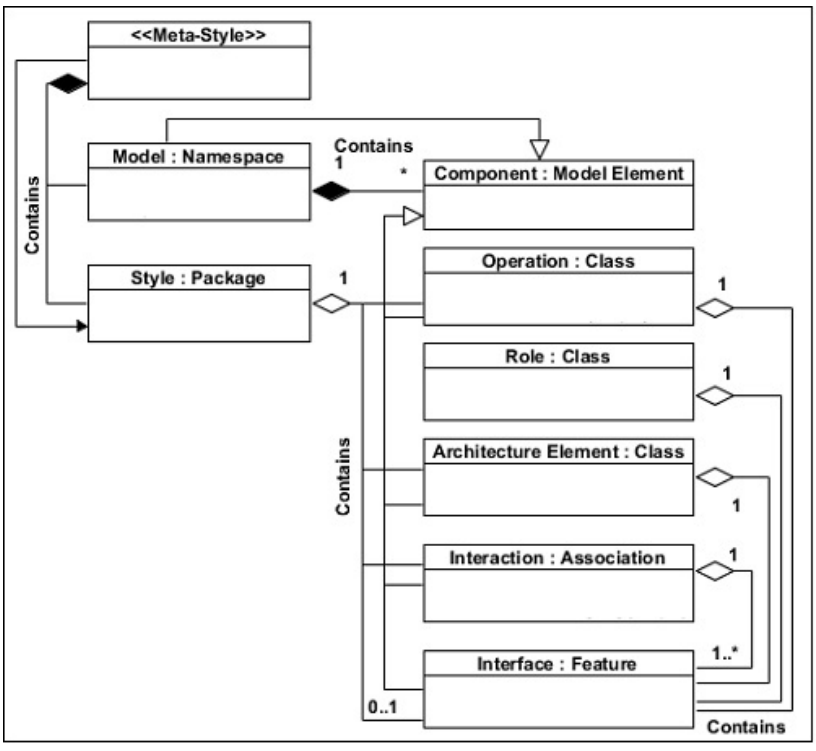

Figure 28: Mapping MES with MOF

Table 6 represents the corresponding elements in MES and MOF, which are useful for translating a model of evolution style to a model-driven development MDD area to be implemented by one of its modeling environments such as EPF.

Table 6: MES mapping with MOF

MES	MOF
Meta-style	Stereotype Package
Style	Package
Model	Namespace
Component	Model Element
Operation	Class
Role	Class
Architecture Element	Class
Interaction	Association
Interface	Feature

4.5.2.2 Mapping an evolution style with SPEM

SPEM (Software & Systems Process Engineering Metamodel) is a standard metamodel based on MOF and developed by Object Management Group (OMG). SPEM is a process engineering metamodel as well as a conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing interchanging, and enacting development methods and processes[7]. The conceptual model of SPEM is represented by the three core elements of process and their interplay, whereby Roles are responsible for and execute Activities that consume and produce Work Products.

SPEM is considered one of the most influential process-modeling notations in software engineering and has industrial support through the Eclipse and Rational Unified Process (RUP) community. However, several open-source and proprietary tools have been developed based on SPEM, for example, Eclipse Process Framework Composer (EPF), IBM Rational Method Composer (RMC) and Osellus IRIS Process Author (IPA) (Suryadevara 2010).

Our aim is to bring the evolution style concept closer to the MOF-compatible modeling environments, in order to exploit the previous achievements that have been adopted by most software engineering communities and supported by their methods and tools.

To this end, we are going to carry out mapping between the Garlan evolution style and SPEM, which is a process engineering metamodel and the nearest to our basic concept of evolution style, even though it lacks an explicit concept of the software architecture evolution process based on the architecture perspective.

[7] https://www.omg.org/spec/SPEM/2.0/

According to the transformation strategy in this paper, the first step must be independently defined for each evolution style, while the other steps (2, 3) are defined only once and can then be used by any evolution style compatible with MES. The second step has already been defined, which is the mapping between MES and MOF. The third step is the instantiation of the MOF by SPEM; this operation is specified by OMG, which describes the instantiation of the MOF by SPEM. Furthermore, SPEM is defined as a UML profile and each element (stereotype) of an SPEM is defined by a corresponding UML element (super-class or meta-class). The instantiation of the MOF by UML is done automatically using the specifications of OMG. Table 7 shows the possible SPEM instances for each element of MOF that we used in the mapping with MES.

Table 7: MOF instantiation

MOF	SPEM 2.0
Class	Activity, Work Product , Role , Guidance
Association	Work Product Relationship, Work Sequence, Process Responsibility Assignment
Feature	Process Parameter
Package	Package

The class is the fundamental element of MOF, and the meta-element (super class) for activity, work product, role, and guidance elements in SPEM. In the mapping between MES and MOF despite the presence of different elements in MES, the class is also the equivalent element that fits the three core elements in MES, which are operation, role and architecture element. Consequently, in Tables 6 and 7, we notice that the translation of the three elements to the SPEM through MOF gives the same counterparts.

Operation, Role and Architecture Element ⇒ Class ⇒ Activity, Work Product, Role or Guidance

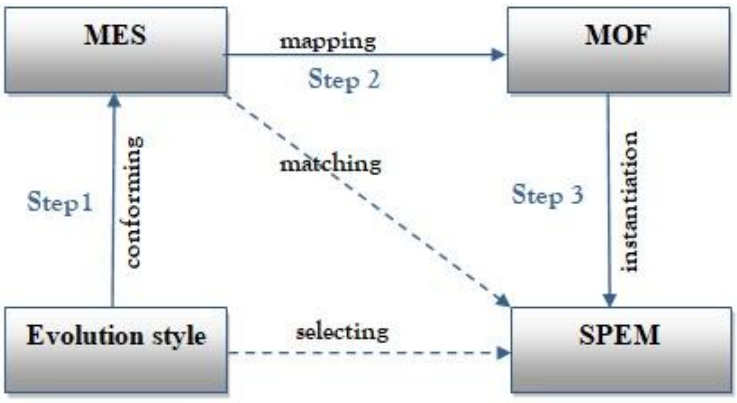

Figure 29: Mapping MES with MOF

We have mapped between MES and MOF those elements belonging to the same modeling level (metametamodel), to minimise the elements and to apprehend the matching. Therefore, to make the transformation more specific we could match the MES and SPEM, considering both the mapping of the super-classes (meta-element) and the ontology of the elements in the two domains (Figure 29 depicts this process).
Thus, Table 8 illustrates the matching between their elements that could provide a more accurate translation.

Table 8: Matching MES with SPEM

MES	SPEM
Style	Package
Model	Process Model
Operation	Activity
Role	Role, Guidance
Architecture Element	Work Product
Interaction	Responsibility Assignment , Work Product Relationship
Interface	Parameter

Finally, we reach the last step in this mapping with the selection of the final concepts. At the end of the matching of MES elements with SPEM elements, we reach a reduced number of choices for each conceptual element in the evolution style. This step is specific for each evolution style, while all evolution styles that are compatible with MES reuse the three previous steps in this mapping strategy. To this end, the end user can define and apply some criteria that will assist and guide him in his choices (the last step), to transfer his style to a target environment according to these rules.

For instance, the final step could be transforming the Garlan style to SPEM, selecting corresponding concepts for the Garlan style that are the most representative and semantically closest, with consideration of the rules enacted in previous steps. Table 9 illustrates this step.

Table 9: Garlan style, SPEM; corresponding elements

Garlan et al. evolution styles	SPEM metamodel
Operator	Activity
Transition	Activity
Architect	Role
Evaluation Function	Guidance
Constraint	Responsibility, Relationship, Guidance
Evolution State	Work Product

4.6 Mapping scenario

Evolution Meta-Style MES is defined to provide a common environment for specifying evolution styles (modeling languages) and for setting up the rules that manage the transformations between these evolution styles, thus investigating the efficiency of MES in translating the evolution styles model and exploring the extent to which these rules are really useful. Therefore, a case study reported in this section was based upon these translation rules.

4.6.1 Modeling an evolution style by Eclipse Process Framework (EPF) composer

The Eclipse Process Framework (EPF) composer is an SPEM 2.0 implementation that is managed by the Eclipse Foundation. It is an open-source software for authoring method contents and publishing processes, which allows the process engineers to synthesise a process model in order to fit a specific situation (Haumer 2007).

The EPF composer is built on the SPEM metamodel where there is a clear separation between Method Content and Process, as we see in Figure 30. EPFC implements this distinction, thereby providing capabilities for creating method libraries. The method content stores all the process elements (roles, tasks, artifacts and guidance) which are defined regardless of their application in the processes (how they will be used in the processes).

Therefore, it provides an extensible framework for process engineers to select, tailor and assemble their process from the method content.

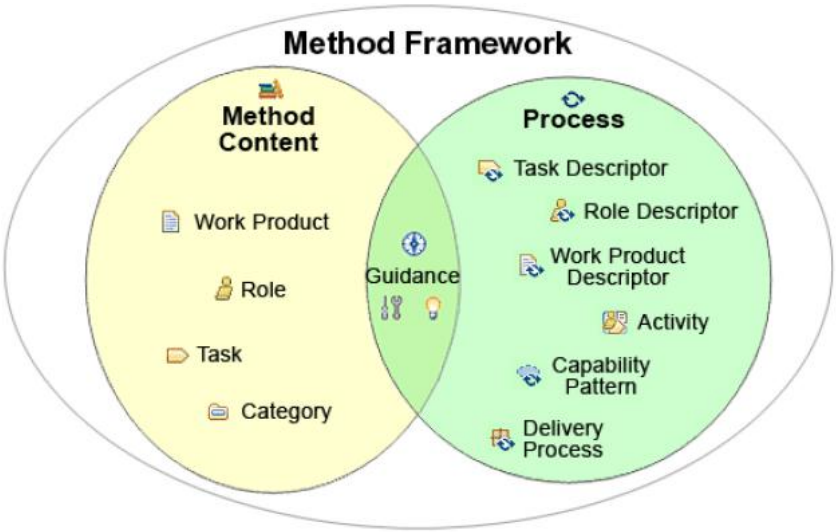

Figure 30: Method Framework of EPF

In this section we aim to investigate the feasibility of the MES and the mapping rules. Therefore, an instance of Garlan evolution style was chosen to be modeled with (translated into) the Eclipse Process Framework (EPF) composer to explore the whole transformation path (rules).

Furthermore, we also aim to compare the difference between these methodologies in supporting the software-architecture evolution process. Moreover, we aim to explore the shortage of SPEM in expressing architecture evolution in order to clarify why we need domain-specific modeling for a software-architecture evolution process.

In the EPF, work products are the elements that clarify the task inputs and outputs. In the same manner, in the Garlan approach; the Evolution style composes of sets of evolution paths that represent the possible scenario of domain-specific evolution. Every path consists of a set of architectural nodes (evolution state), and the evolution operators that make up the transitions on this path. So, the Garlan style is like EPF, it defines the architecture elements (evolution states) as products that clarify the operators' inputs and outputs. Therefore, defining the architecture states (work products) and operators (tasks) help to create a smooth flow in the path.

Garlan et al. (Barnes, Garlan, and Schmerl 2014) presented a case study for their evolution style, which defines an evolution model to evolve a GNU chess engine desktop application (the initial state) to the Amazon EC2 cloud-computing platform (the target state).

We are going to translate the concepts of Garlan style to be modeled by EPF in the method content. Then we will use these elements to model one transition from this evolution style, which is "*Activity: Migrate to cloud*". This transformation will be conducted based on the mapping rules previously enacted in this chapter.

Since EPFC has a clear separation of reusable method content from its application in processes, the process does not directly include core-method elements but creates local references, termed descriptors, that refer to the elements in the method content. For example the activity used task descriptors to refer to a task in method content, which can be reused at different times in the same or in a different activity. Therefore, Table 10 represents the elements of Garlan et al.'s evolution style and their corresponding elements in the EPFC method content. This selection was built on a strategy according to the mapping rules defined in this chapter and the compatibility of EPF with the SPEM metamodel. Figure 31 illustrates the implementation of the style on the EPF composer base on this table.

The model (case study) of the Garlan style represented in their paper is transferred to the EPF composer (Version 1.5.1.6) using these mapping rules as a means of evaluating our approach. We start with the method content to define the elements of the style that are used to model the style (Delivery Process). Figure 31 shows how can we model the style elements in method content (1 in Figure 31) and then included them in the style model (2 in Figure 31).

The elements of the style are defined in the method content under the four main element types in the method content package. These are: Roles, Tasks, Work Products and Guidance, as we can see in the part 1 of Figure 31. Task descriptors are used to instantiate the reusable elements in the content. The task descriptor can be customised further by annotating more details using the prosperities view.

Table 10: Mapping Garlan style to EPF Composer

Garlan et al. evolution style	Eclipse Process Framework
Style	Content Package
Transition	Tasks
Operation	Task
Architect	Role
Evolution State	Work Product
Constraint	Association
Evaluation Function	Tools
Analysis	Guidance(Estimation Consideration)
Transformation Description	Guidance(Guideline)
Precondition	Guidance(Checklist)

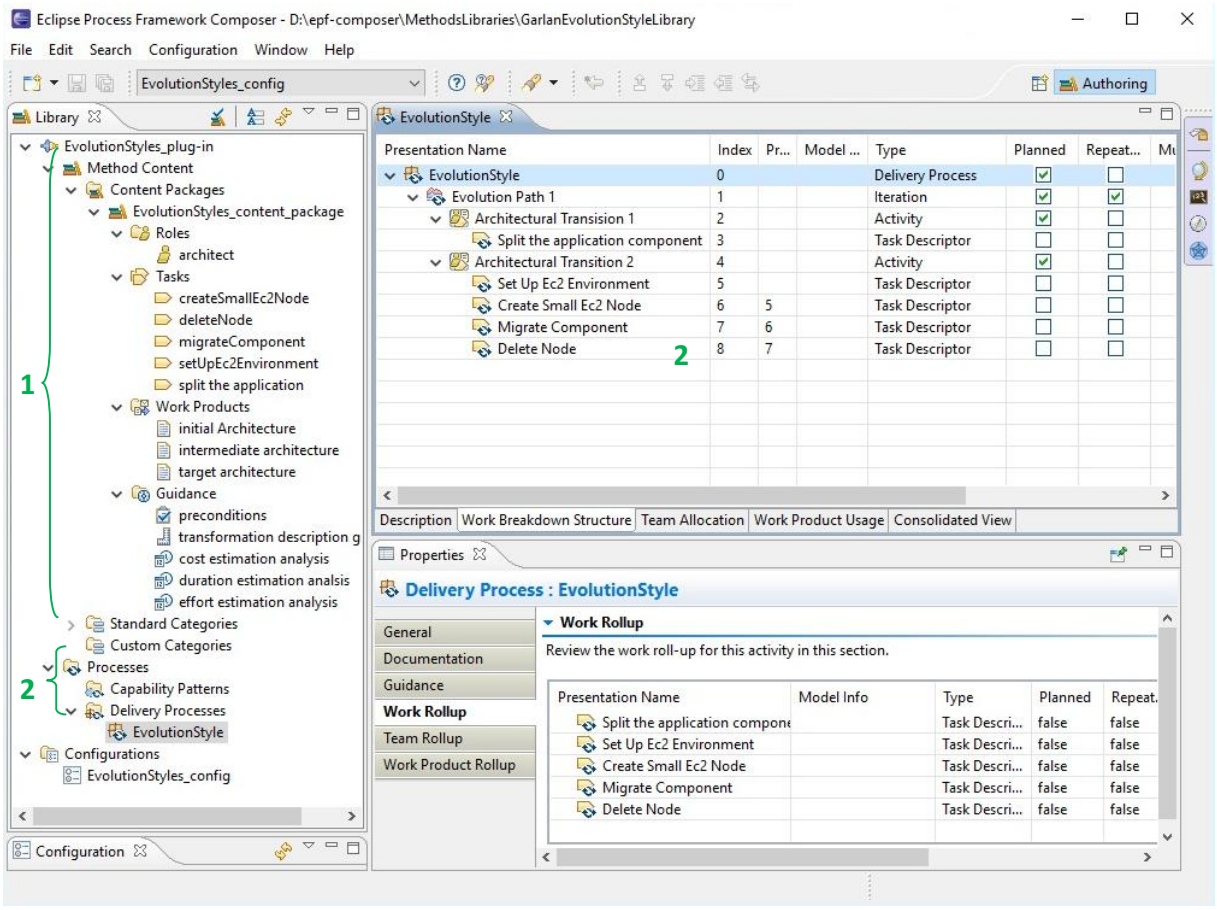

Figure 31: Screenshot of EPF composer

The Evolution operator corresponds to a Task element in the method content of the EPF composer. The Operator in the evolution style (metamodel) is an element representing the evolution activities and is composed of Precondition, Analysis and Transformation Description. Figure 32 shows a preview of the task "Migrates Component", which has a brief description of the task, the purpose, the relationships with relevant elements, the composed steps whereby a complex

task can be broken down (decomposed) into simpler sub-tasks (steps) and more information used to associate the task with the analysis information.

These analyses have been expressed using some Guidance Elements Types. The Checklist element type has been used to represent the preconditions of the evolution operator. The Guidelines element type has been used to provide the description of the evolution operator. The Estimation Consideration element type is used to indicate the analysis information associated with each evolution operator in order to be used to calculate the overall utility of an evolution path.

Each relevant element which is written in blue represents a link to its webpage where the process (evolution style) is published.

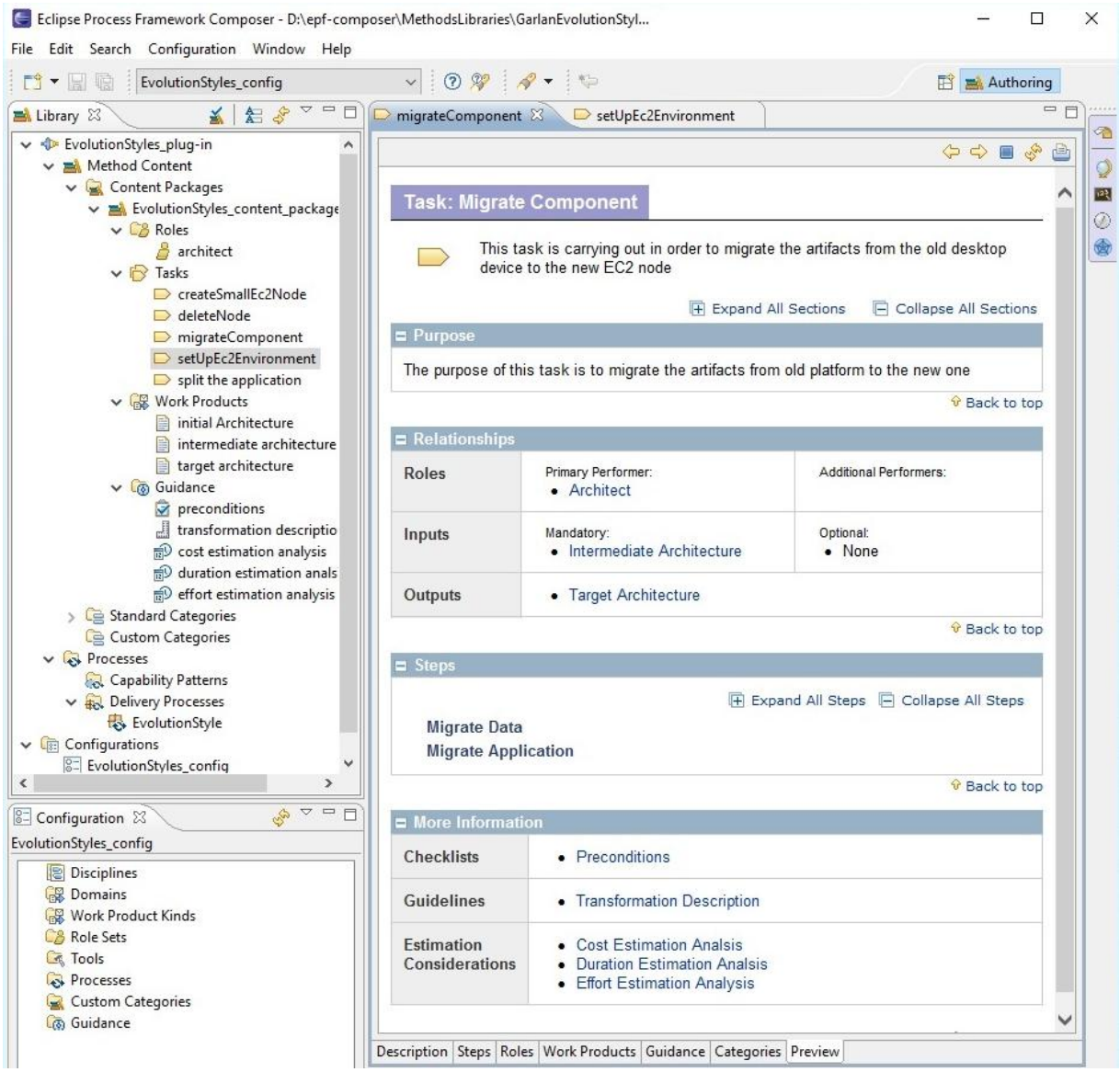

Figure 32: The representation of evolution operator in EPF composer

An evolution state (architecture element) in the Garlan evolution style corresponds to a Work Product Type in the EPF composer Figure 33 show how the first evolution state in the evolution path which represents the architecture of a desktop GNU chess application. In the evolution style content package (on the left side "1" of the Figure 33) we can see that initial architecture is defined as Work Product Type. On the right side we can see a preview of this evolution sate as it will appear in a published process. Other views also exist whereby the details of this work product can be annotated or managed, for example in the Description view where a brief and main description can be added. A component-and-connector view, as a UML component diagram of the initial state, has been added to the main description that illustrates how the EPF composer can support several detail types (text, model/image).

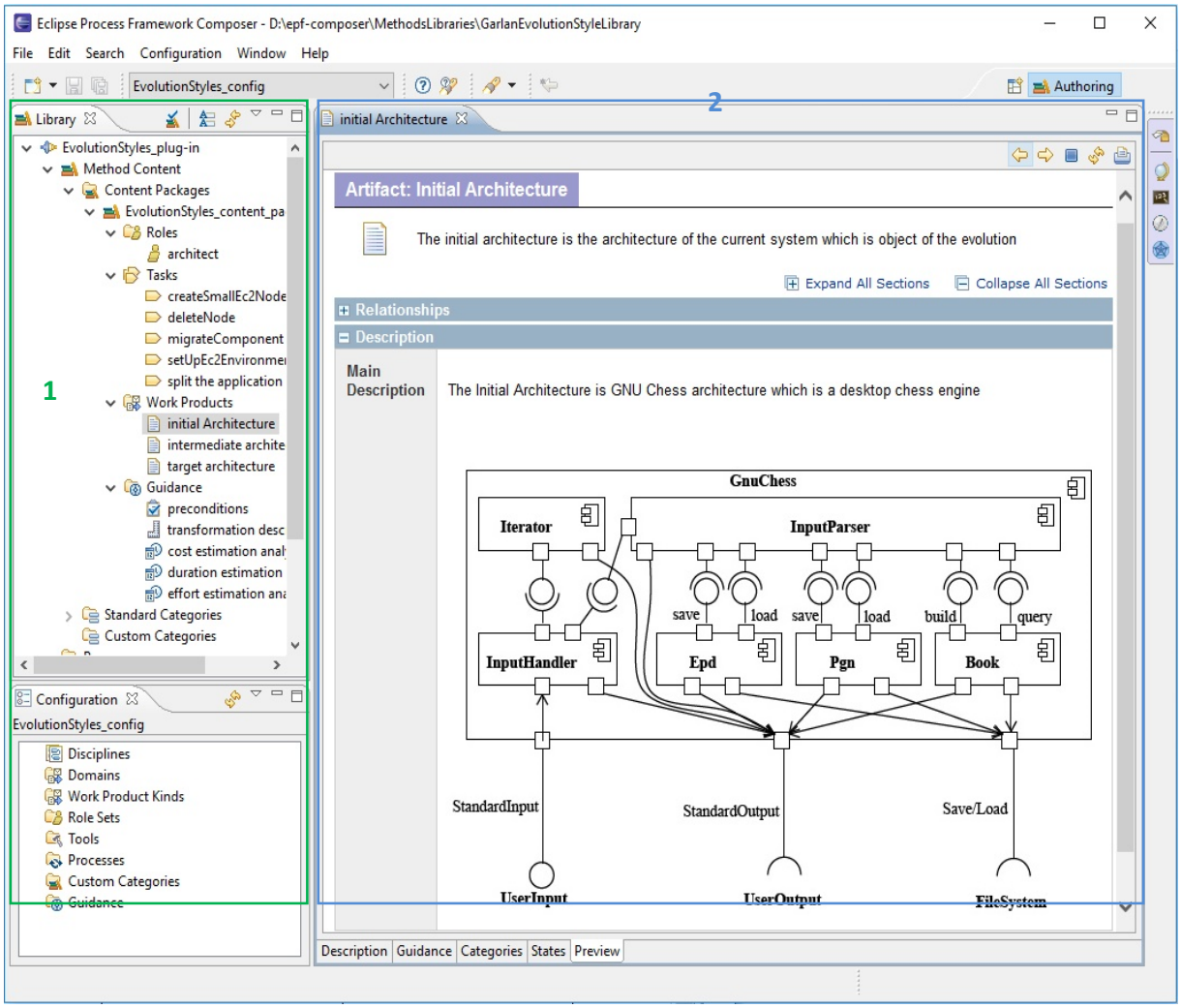

Figure 33: The representation of evolution state in EPF composer

The evolution architect in Garlan style corresponds to Roles Type in the EPF composer as the mapping rules of this work. Thus, we can see in Figure 34, that the architect is defined under Roles Type in the Method Content. The view on the right part of the EPF displays a preview of the

architect, in which we can see a brief description of the architect role, the relationships, modifies, and main description. The "Modifies" part contains the work product elements produced by this Role. The "Relationships" part depicts the relationships between the architect and the other Method Content's elements, i.e. "performs" with tasks and "responsible for" with work products.

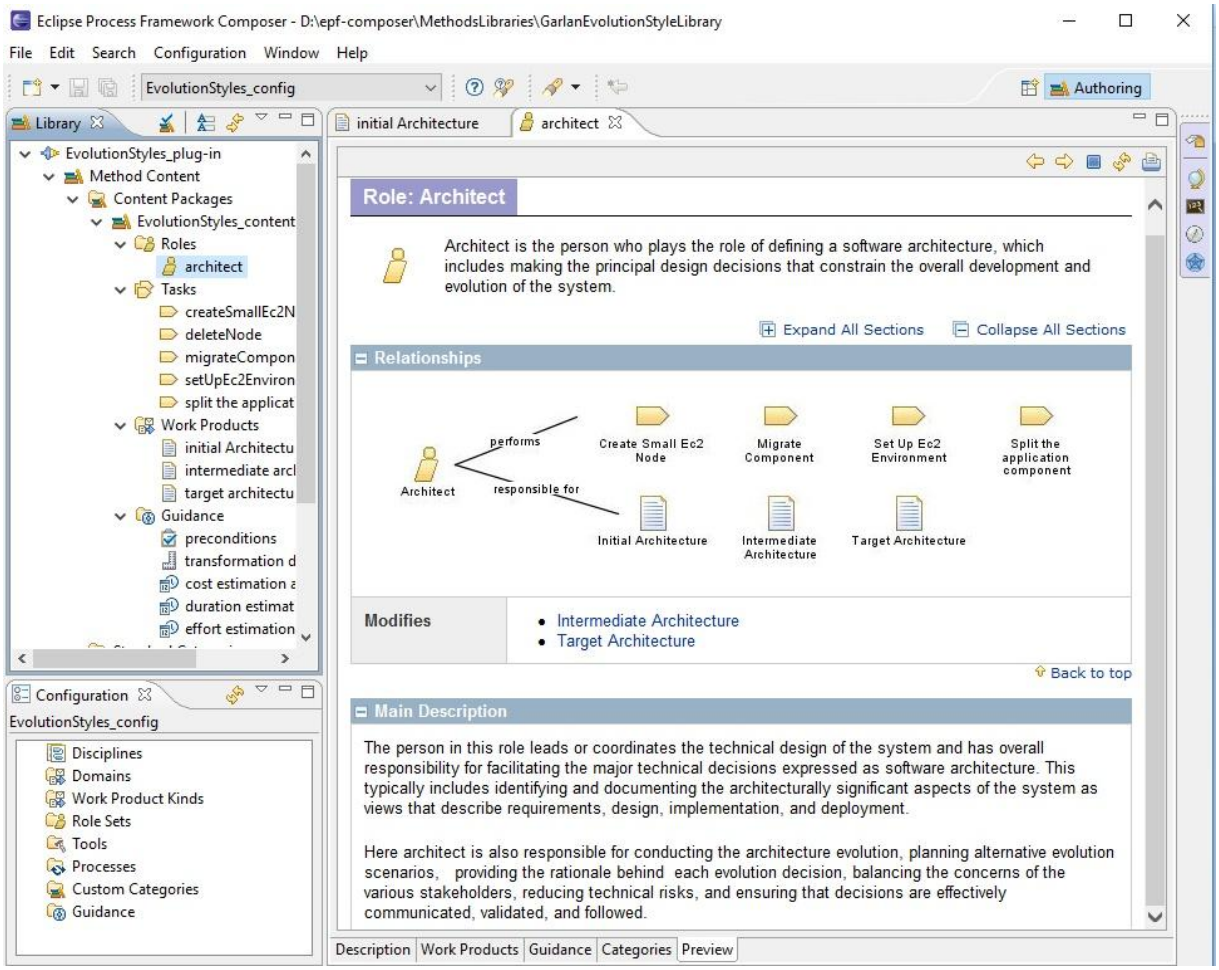

Figure 34: The representation of architect in EPF composer

As soon as the evolution style Content Package is defined, the element types can be used to define a specific evolution style instance under Delivery Processes. Thus, an evolution style that stimulates the above-mentioned case study has been defined as a new Delivery Process in EPF composer, as Figure 31 shows. The evolution style is defined as a Delivery Process which contains one evolution path as Iteration. Evolution Path 1 comprises two architectural Transitions which are defined as an Activity in the Processes. Architectural Transition 1 is composed of one evolution operator (Task) which is a split GNU chess engine component. Each time a task is included in a process, a copy of that task is created in the context of the process as the Task Descriptor.

By applying this evolution operator (task) to the initial architecture we can get into the second evolution state (intermediate architecture).

The Architectural Transition 2 is composed of four evolution operators which can be applied to the intermediate architecture in order to obtain the target architecture (final evolution state). These operators are: setUpEc2Environment operator, createSmallEc2Node operator, migrateComponent operator and deleteNode Operator.

Now, as the definition of the delivery process is completed, we can publish the evolution style (process) as a website where we can use the links on the page to navigate through the evolution style. Figure 35 shows a browser showing the published process. On the left side of the site we can see navigation links of the evolution style, where architectural transition 2 is the active link.

On the page of architectural transition 2 we can see a brief description of the transition and, in the workflow part, we can see, at the top, an activity diagram and, at the bottom, an activity detail diagram. In the work breakdown session there is a set of tasks (links) by which this transition can be carried out. The Tasks link on the breakdown session can be used to navigate through these tasks.

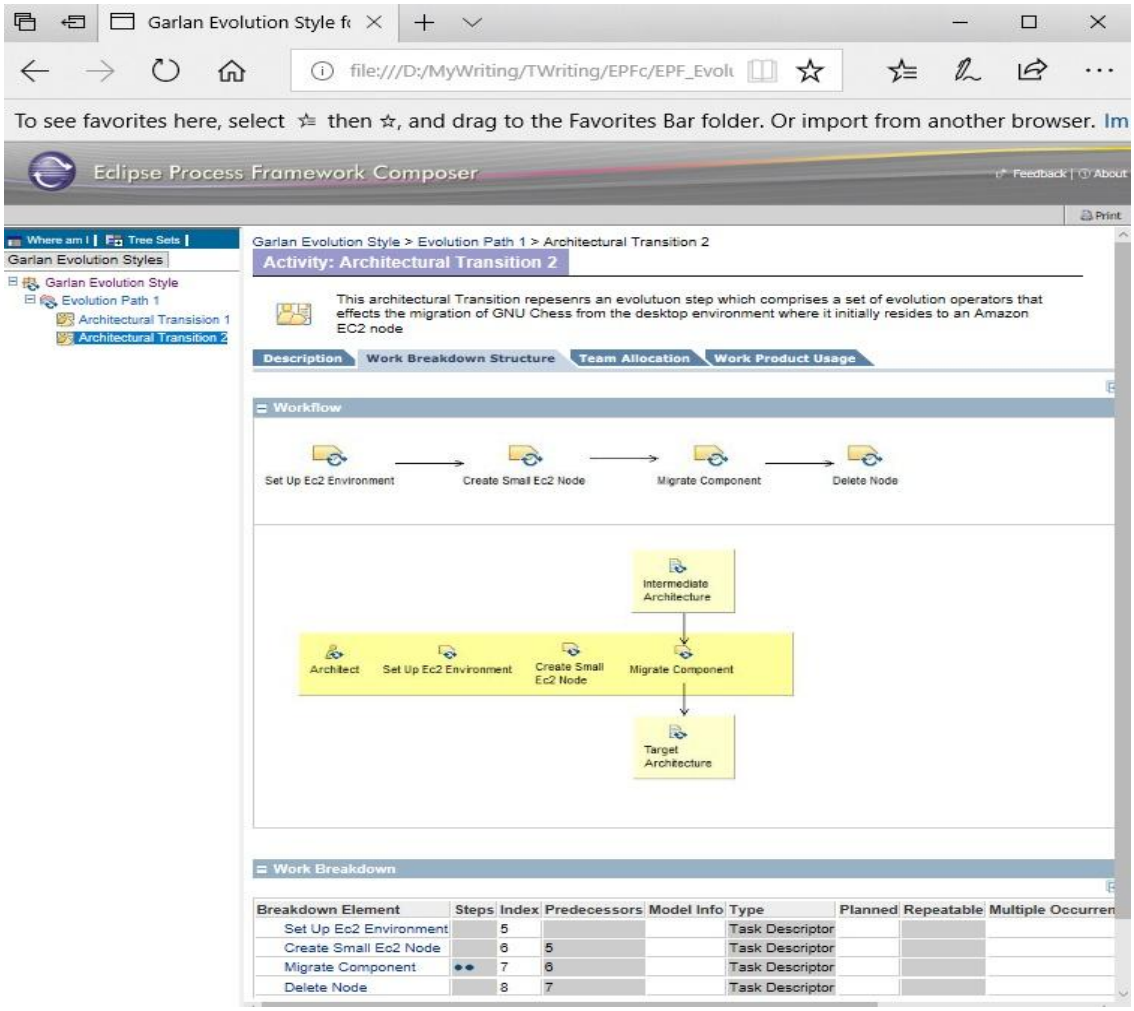

Figure 35: Screenshot of a published evolution style

4.6.2 The Result of the Mapping Scenarios

The case study was conducted in order to answer some research questions. Here we aim to discuss to what extent the case study could answer them and what benefits have been gained from this example.
First, consider whether MES has adequate concepts to instantiate an evolution style's elements and determine their counterparts in another evolution style or modeling environment, in order to facilitate the transformation between them.
The example illustrates how an instance of an evolution style can be transferred to EPF composer and how the most representative elements in the two modeling environments can be identified. The rules were perfectly followed in this example to translate this part of the so it could be published with EPF composer. We can be sure that every element in the example was translated into its corresponding element using these rules and that the transformation was easily conducted with no misunderstanding in the concepts being encountered. As we mentioned in previous sections, meta-level is used to reason about the models at the level below and to determine the corresponding elements in these models. Therefore, the success in translating an instance using the rules based on MES could provide implicit proof of its feasibility and answer the questions of the case study.

EPF is a customizable software process engineering framework, It supports users to author, tailor and publish methods and processes for different software development organizations supporting various kinds of projects and development methodologies (e.g. OpenUp, extreme programming and scrum), as well as, it supports users to create their own process frameworks/models from scratch.

In general, every software process is unique, and the development details differ among different projects. Creating a new process model from scratch can be very expensive and time consuming. Thus, EPF support reuses methodology throughout the process tailoring.

EPF is a tool for deploying software process development and can be used by organisations to evaluate and improve their processes in different ways, i.e. to exploit existing best practices and knowledge, to address the unique needs of individual projects and to enhance the communication among stakeholders. Moreover, it can provide a teaching framework for organisations to deliver their best practices and methodologies in the software-development process.

Even though EPF offers all these benefits, it considers a generic prescriptive framework for the whole software process development phases, including the architecture details. But it does not provide a graphical editor, in which software architecture can be formally represented (e.g. component diagram) or to express the workflow of the potential ways to evolve this artifact.

The evolution style's workbench provides a graphical framework for modeling different paths of software-architecture evolution in a particular domain (in regard to evolving software architecture from a certain architecture style to another style) and also provides the means to compare them.

Support for architects with the detailed information (low abstraction level) about this process and the different ways (paths) of conducting these tasks is missed in the evolution style (exported as HTML documents). In addition the vision of the reuse, which is supported in EPF by the concept of separating the definition of the tasks (methods content) from those used in the process, is also missed.

The main benefits gained from carrying out this case study is summarising and standardising the academic efforts in software-architecture evolution, bringing the academic research closer to the industrial communities, such as OMG and Eclipse, and exploiting their achievements in the related areas.

4.7 Conclusion

This chapter has presented the main contribution of the thesis. In the previous chapter some open challenges that need more research were presented. To this end, to support the standardisation, reusability and interchange in the domain of software-architecture evolution modeling, the metamodeling technique was adopted. Thus, based on the metamodeling, an evolution meta-style (MES) for software-architecture evolution modeling was proposed. Mapping between MES and different evolution style approaches was conducted in order to investigate the ability of MES to express these approaches and to enact the transferring rules between these approaches. Furthermore, mapping with Object-modeling discipline was conducted whereby an example of an evolution style was implemented with Eclipse Process Framework Composer, based on this mapping.

Chapter 5. Dynamic Evolution Meta-Styles

5.1 Introduction

Software systems need to be continuously maintained and evolved in order to cope with ever-changing requirements and environments. As previously mentioned, software evolution is a complex and multifaceted process that requires a great deal of knowledge and skills. This is particularly so when it is required to introduce these changes without halting the system, which is a critical requirement for many software systems. In particular, where the stop for integrating these changes may result in injury, death, serious environmental damage or monetary losses, a mechanism for integrating changes at run-time is needed, which bring more complexity.

Despite the execution model is outside the scope of this work (which focuses on the descriptive process model), the goal of handling dynamic evolution is not to develop an automatic model (e.g. architecture-reflective model for self-adaptive system), but to explore and identify the issues relevant to the dynamic evolution process. Thus, we can elicit the required information which can be annotated with MES in order to model the dynamic evolution of software architecture.

In the same vein, the evolution style approaches should support architects in better understanding and reusing best practices and knowledge when conducting dynamic evolution. Embracing the necessary concepts to model the dynamic architecture changes will help architects in modeling and reusing these activities.

The previous chapter presented the evolution meta-style MES for modeling software-architecture evolution (static evolution style), but we did not delve into the dynamic aspects of this process. Therefore, in this chapter, we endeavour to probe more deeply into the issues and rules that must be considered when handling the dynamic evolution of software architecture in order to extract the needed information, which will be annotated with MES in order to fulfill the requirement of modeling a dynamic evolution at the architectural level.

The behaviour and time aspects (how and when) are among the main issues which accompany the concept of dynamism. In the scope of our work, these issues can influence two modeling concepts in MES; architecture elements and evolution operation. The component (AE) can perform real-time tasks, and/or operation evolution can be performed on the component at runtime (without halting the entire application). Thus this section will handle the issues of the time constraints that affect both the architecture elements and the evolution operation.

5.2 Real-time system

A real-time system is an electronic, or computer, system that is designed to perform only a dedicated application, in many cases as part of a larger system. Real-time systems are those that can provide guaranteed *Worst-Case Response Times* for critical events, as well as acceptable average-case response times for noncritical events. When a real-time system is designed as an embedded component, it is called a real-time embedded system (Fan 2015).

According to Stankovic "*the real-time computing are those in which the correctness of the systems depends not only on the logical result of the computation but also on the time at which the results are produced*" (Stankovic 1988).

Real-time embedded systems have become ubiquitous and have permeated a wide area of applications, such as commercial, industrial, medical, and military applications. Since the embedded technologies are widely dispersed throughout daily life and everyday objects, the demand for engineers with the skill-set for the development and evolution of real-time embedded software has soared in recent years. Developing and evolving software for real-time embedded systems may involve several activities, including requirements specification, timing analysis, architecture design, multi-tasking design and cross-platform testing and debugging. All of these have necessitated the development of techniques, methods, models and tools in order to support the development and evolution process.

Often, the spectrum of real-time systems is classified into three categories based on the timing constraint of their tasks: hard, soft and firm.

Sometimes a system can be deemed soft or hard, depending on where and how it is to be used. For example, a radar system can be either hard or soft real-time. It can be a hard real-time system if it is used in a military surveillance system and it can be a soft real-time system if it used for weather forecasting.

5.2.1 Types of real-time systems

- **Hard real-time systems**

A hard real-time system is a system that cannot tolerate any delay in response. Thus, the correctness of the result of its tasks is related not only to their logical correctness, but also to their temporal correctness (to be completed within a strict deadline). A hard real-time task is one that is constrained to produce its results within certain predefined time boundaries.
Actually, in most cases, temporal correctness is even more important than logical correctness, because a partially functional system may be used as it is and still have its values, whereas a fully functional system is useless if the offered services have no guaranteed service-completion time. Meeting the time constraints is a vital requirement for hard real-time systems. Therefore, time-scheduling analysis is very important in the design and development of these systems (Fan 2015).
An example of a hard real-time system is a nuclear power plant, where the system must periodically monitor the plant conditions. These conditions are based on the values of some properties (sensors) and these include temperature, pressure and water and radiation level. Based the values (event) the monitoring system triggers the corrective actions necessary in order to maintain the safe operation of the power plant.

- **Soft real-time systems**

Tasks in soft real-time also have time bounds associated with them. Unlike hard and firm real time tasks, nothing catastrophic will occur if deadlines are missed. But the utility of a result may be degraded after its deadline, thereby degrading the quality of service in the system. The soft timing constraints are typically expressed either in terms of the average response or standard deviation.
An example of a soft real-time task is a web email service. Normally, after the send button is clicked, the recipient will receive a copy of the email. The email will (should) appear in the receiver's in-box within a couple of minutes on average. However, when it takes several hours to receive an email, we still do not consider the system to have failed, we merely say that the performance of the system has been degraded.

- **Firm real-time systems**

Systems that lie between the hard and soft real-time systems are often called firm real-time systems, whereby each task has to be scheduled within a certain time bound in which result must be produce. However, unlike the hard real-time system, even when a task exceeds its deadline, this does not mean that the system has failed. It just merely makes the utility of the result zero.
An example of a firm real-time system is a video-conferencing system whereby the video and audio are combined and converted into packets delivered to the receiver over a network in the form of real-time video images and an audio stream. The receiving end simultaneously decodes and plays the digital voice stream being received. The temporal correctness of the receiver is shown when it decomposes and plays the received frames at a predetermined constant rate. However, some of these frames may get delayed or lost during the transmission. If a certain frame is late in arriving at the receiver, at the time when one of its predecessors is being played, that frame is out of use and will not be played. Its utility becomes zero and it will be directly discarded.

5.2.2 Types of real-time tasks

Real-time systems are computing systems that must meet their temporal specification in response to an external environment. Recurring tasks are usually driven by clock interruptions (to monitor, or to control) or by (respond to) external events. Thus, the way that real-time tasks recur over a period of time can be classified into three categories: periodic, sporadic and aperiodic. The majority of real time systems consist the two main categories of tasks, periodic and aperiodic (to monitor and to respond). Periodic tasks have regular arrival times and hard deadlines. Aperiodic tasks have irregular arrival times and either soft or hard deadlines. Therefore, it is important for a real-time system to meet the regular deadlines of periodic tasks and the response-time requirements of aperiodic events.

An example of a periodic task is the monitoring task in a nuclear power plant, for example checking the operation conditions at regular intervals. An operator request of online systems, meanwhile, is an example of an aperiodic task.

5.2.3 Architecture Description Languages for real-time system

Real-time is a domain-specific software system which has timing requirements in addition to functional requirements. These additional requirements bring or demand special properties (methodology) in all aspects of analysing, designing and developing such systems. Furthermore, they demand specific (real-time) operating systems which must provide specific functions with respect to tasks (scheduling, dispatching and intercommunication and synchronisation) to ensure that tasks can satisfy the time constraints (deadlines).

Therefore, real-time ADLs are designed to satisfy the required properties for modeling the specific characteristics of those systems and their execution platforms (execution time, memory, processors, buses, devices etc). To this end, a number of domain-specific languages for (architecting) modeling real-time embedded systems have been developed. Among the best known of these are: EAST-ADL (Debruyne, Simonot-Lion, and Trinquet 2005), AADL (Feiler, Gluch, and Hudak 2006) and MARTE (Faugere et al. 2007).

- ***EAST-ADL*** (*Embedded Architectures and Software Technologies-Architecture Description Language*) is an architecture description language originally designed for the domain of automotive embedded systems (Blom et al. 2016). It has been defined in the scope of European research initiatives since 2001 (European ITEA EAST-EEA project) and is being refined to support the modeling of fully electric vehicles. It is aligned with industrial practices and standards AUTOSAR and ISO 26262. EAST-ADL can be used to describe high-level abstract functionalities and both the computer hardware and the software architecture of an automotive electronic system in a standardised form. The language provides a wide range of modeling entities (such as functions, requirements, variability, software components, and hardware components) and consists of four abstraction levels which can describe the system from different views. The **Vehicle Level** is the highest level, the aim of which is to describe the system at high level of abstraction and this contains a model of the electronic vehicle features. The **Analysis Level** comprises the functional design architecture which defines the abstract functionalities based on the requirements and features. The **Design Level** comprises the functional design architecture and the hardware design architecture. It defines the concrete functionalities of software, the topology of the hardware components, the allocation of software functions to hardware components and the middleware. The **Implementation Level** uses the AUTOSAR concepts to realise the model through mapping EAST–ADL functions to AUTOSAR Runnables. The EAST-ADL language is implemented as a UML2 profile whereby the engineer can model the system according to the EAST-ADL semantics, using off-the-shelf UML2 tools.
- ***AADL*** (Architecture Analysis and Design Language) is a modeling language initially designed for avionics (Feiler, Gluch, and Hudak 2006). It is an SEA (Society of Automotive Engineers) international standard, dedicated to model software, hardware and system architecture of complex real-time embedded systems. Thus, it provides the definition of software components (data, thread, thread group, subprogram and process), the execution platform components (processors, memory, buses, devices, virtual processor, and virtual bus) and the hybrid components (system). AADL

supports system designers in early prediction and analysis of the composed components such as system schedulability, reliability, sizing analysis and safety analysis. *ADDL* version 1 was published in 2004 and aligned with the OMG UML standard to provide UML2 Profile of the *AADL* that adapts the semantics of AADL to UML notation, thereby providing a strong semantic structure for specifying real-time systems in UML notation. This UML Profile for AADL later became standardised and defined as a part of the Modeling and Analysis of Real-Time Embedded Systems (MARTE) meta-model. Therefore, there is no longer an effort to continue the development of the UML Profile for AADL outside of incorporating it into MARTE. The version 2.2 of the AADL standard was published in January 2017 (AS5506C)[8]. Carnegie Mellon University has developed an open-source AADL Tool Environment (OSATE)[9], based on EMF Eclipse Modeling Framework, whereas the current version of OSATE is 2.3.1 which was **released on** January 3, 2018 and is based on Eclipse Oxygen.1A.

- ***MARTE*** (Modeling and Analysis of Real-Time and Embedded)[10] is a UML profile extension for modeling real-time and embedded systems, which was standardised by OMG in 2007. It provides fundamental concepts simply to model and analyse concerns such as time, performance and schedulability issues, which facilitate the modeling of both the software and hardware platform components. MARTE does not bring any new diagrams for the UML as part of its metamodel. It includes guidelines for constructing models at various abstraction levels. In general, the MARTE package is divided into two sub-divisions: the MARTE design model and the MARTE analysis model. Since the design model is independent of the analysis model, this allows user to perform several kinds of analysis on the system without needing to modify its design model and this may bring additional benefits to the evolution process. The MARTE profile has some sub-profiles which contain partial entities of the MARTE profile. For example, the High-Level Application Modeling (HLAM) profile provides concepts for constructing diagrams at a higher-level of abstraction, while addressing basic real-time concepts such as schedulability.

5.3 Dynamic software evolution

Dynamic evolution of the software system, where components can be added, removed, or replaced while preserving the availability of the application is becoming increasingly important for many organisations. Adapting the software application at runtime has tremendous potential for increasing an application's capacity to adjust to changing execution environments, to become more resilient and to embrace the requirements continuously.

One of the main difficulties of software evolution lies in the fact that all artifacts produced and used during the entire software lifecycle are subject to changes, starting from early requirements over analysis and design, and moving to the source and executable code, in which the updating of the new executable code is the final task in the evolution process.

[8] https://www.sae.org/standards/content/as5506c/.

[9] http://osate.org/osate-releases.html.

[10] http://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf.

To help us to understand the needs of dynamic software evolution, it is worthwhile to spend some time clarifying how software systems are constructed and put into executions. Software systems are often constructed from one or more modules which may be procedures, classes, components, or so on. The source-code files which compose these models are compiled into an object code, generating separated files where a linker can be used to link together these object files to produce a binary file which can be directly executed. For each module, a header is included in order to map symbol names to external code fragments, allowing source-code files to access externally defined functions (libraries). When the compilation of the modules is done, all the object files must be combined. The objective of the linker is to combine the code of the modules and resolve any reference to external fragments.

Dynamic linking can also be used when the application is starting (runtime), with dynamic linking of the external symbols referenced in the source code (which is defined in a shared library) being resolved by the loader at load time (binding the shared libraries dynamically). The modular (component) is not statically bound to the application but dynamically bound when the application is loaded. This permits applications to inherit changes to the shared libraries automatically, without recompiling or rebinding. (e.g. component lifecycle OSGi in Eclipse IDE).

In this respect, to change a software unit (static linking) that is part of a software application, the source code of the unit must first be edited and enhanced with the required modification and then recompiled. The generated object code must be re-linked to the other modules of the application. Since the references between modules have changed and are defined statically, the application must be shut down and re-linked in order to update the references mapped from the old modules to the new modules. This process of integrating changes is known as static evolution or offline evolution, as the system needs to be restarted.

Shutting down the system for evolution results in the loss of the current state of the application and all open transactions (messages). Furthermore, the system availability will be affected during the restarting process. All of this necessitates the need for a new technique which permits the introduction of the required changes at runtime.

5.3.1 Introducing changes at runtime

For many software systems especially those machine safety-critical applications, halting their executions is inadmissible. Such applications may include life-critical systems, telephone switching, financial systems and air-traffic control, among many others. Shutting down and restarting such applications for evolution would likely cause components' threads to miss their deadlines.
To this end, several methodologies for introducing changes at runtime without violating the time constraints have been proposed. On-the-flay program modification (Fabry 1976) was one of the earliest solutions introduced by R.S Fabry in order to handle systems updating at runtime. Since then, several methodologies under different terms have been proposed: *dynamic updating* (Segal and Frieder 1989), *dynamic change* (Kramer and Magee 1990), *on-line change* (Gupta, Jalote, and

Barua 1996), *runtime evolution* (Oreizy 1998), *dynamic evolution* (Malabarba et al. 2000), *live updating* (Vandewoude and Berbers 2005), or *online evolution* (Wang et al. 2006).
All of these terms revolve around the time aspect (on-the-flay, dynamic, on-line, live) of the process (modifications, updating, changes, evolution). Another tread for runtime evolution has been introduced under the perspective of highly dynamic software systems, whereas the Self-Adaptive capabilities were the major issues of this approach and several self-* terms of methodologies have been proposed, such as: Self-Configuration, Self-Optimisation, Self-Healing and Self-Protection (Kephart and Chess 2003).

5.3.2 Activeness of changes

To achieve the dynamic evolution of those real-time systems operated in a highly dynamic world, their behaviour must be adjusted automatically in response to changing environments. This highly dynamic world has shifted the human role from operational to strategic. Thus, (open) dynamic software systems must be instrumented with self-adaptation mechanisms to monitor their environmental conditions so as to assess the need for evolution and plan alternative changes, as well as to validate and verify the results. Humans define the evolution strategies and the system performs all the adaption steps autonomously at runtime.

Buckley and his colleagues (Buckley et al. 2005) analysed software evolution from several dimensions comprising, among others, the What, Why and How dimensions of the evolution process. The "activeness of changes" aspect is put under the What dimension as it concerns the ways from which the dynamic changes can be driven. Dynamic changes can be reactive (the changes are driven externally) or proactive (the changes are driven autonomously by the system itself).

Reactive changes are part of the traditional type of evolution in which the evolution is needed after an evolution event has occurred. They are often driven by an external agent through a user interface or an external tool. Support for reactive changes allows the system to embrace the unforeseen changes which are not initially anticipated during the design phase.

Proactive changes are those changes driven autonomously by the system when some specific conditions or events occur. Thus, the system must contain a management system for controlling the self-change mechanism. There are two types of proactive changes:

- **Programmed changes**: those changes designed into the system, which can be triggered when a certain event or condition occurs. This mechanism has been applied at the level of the software architecture and is known as programmed reconfiguration. Gomes et al proposed an extension of the ADL language, ACME, in order to express programmed reconfiguration (Gomes et al.).
- **Non-programmed changes:** those changes that are automatically generated when a certain condition or event occurs during the runtime. The system analyses the situation and synthesises the solution and plans when and how to execute it.

Both types of change, reactive and proactive, can be used to introduce unforeseen changes and they are complementary. For example, proactive change can be the goal and the reactive change can be used as a back-up mechanism (if the proactive adaption fails).

5.4 Dynamic software evolution issues

Irrespective of the dynamic evolution mechanism used to reconfigure the software architecture at runtime, there are some issues (Oreizy 1998), (Kramer and Magee 1990) which should be addressed by any approach in order to carry out this process efficiently. Some of these issues are: 1) Safe stopping (consistency); 2) State transfer (Integrity); 3) Change management; and 4) Dynamic evolution scheduling.

The first point, safe stopping, deals with maintaining the application consistency and refers to the safe stopping of a running software artifact, thus minimising the impact of changes by leaving the system in a consistent state after a change is performed. The objective is that the system can be changed with minimal disruption of the normal operation of the system. The second issue, integrity (the transferring state), refers to the transformation of the internal structure of the information from the current artifact to the new artifact at runtime. The third issue is change management which concerns the way in which changes are handled and the interaction between evolution activities, as well as their access to the artifacts. Finally, the dynamic evolution scheduling deals with timing constraints, e.g. how the safe stop can be guaranteed in bounded time and how evolution tasks and system tasks can be jointly and dynamically scheduled at runtime. This section presents these issues in more detail in order to extract the required information that should be annotated with MES in order to fulfil the requirement of dynamic evolution modeling.

5.4.1 Safe stopping

One of the main issues that must be considered when handling dynamic evolution is the necessity of leaving systems in a consistent state after a change is performed. Evolving an artifact at runtime without considering its thread/task may disrupt or suspend its service for an arbitrarily long time, which can lead real-time tasks to miss some deadlines. Detecting when it is safe to evolve the artifacts is the key to guaranteeing that the system will not encounter an inconsistent state. Therefore, several strategies have been defined in order to tackle this issue, namely Quiescence (Kramer and Magee 1990) and Tranquility (Vandewoude et al. 2007).

These strategies differentiate the passive state from the active state of the software artifact and assume that an affected artifact should be placed into a passive state before the evolution operation is performed. Figure 36 illustrates the possible state of a software artifact and shows when the changes can be performed.

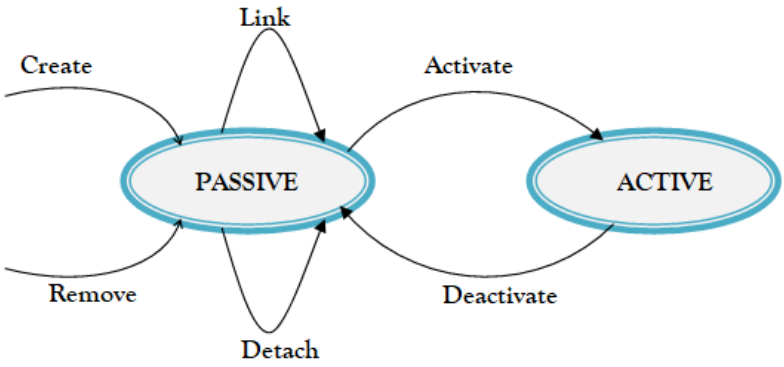

Figure 36: Artifact state transactions

For example, in the Quiescence strategy for safe stopping the definition of the active and passive state is as follows:-

- An artifact can be in the ***active state*** when it can initiate, accept and service transactions.
- An artifact can be in the ***passive state*** if it must continue to accept and service transactions, but
  (i) it is not currently engaged in a transaction that it initiated and
  (ii) it will not initiate new transactions.

Generally, a real-time system consists mainly of a set of elements which provide and /or create real-time services (threads). These real-time tasks can be periodic, aperiodic or sporadic (Li and Yao 2003), depending on how the corresponding job is activated. The passive elements in a real-time system are those that do not have any execution thread, but typically provide services for other elements. The Quiescence and Tranquility techniques can fit the dynamic evolution of these passive elements. The active elements are those that have active real-time threads; these techniques (Kramer and Magee 1990), (Vandewoude et al. 2007) require the elements to be shifted into a passive state in order to be modified. This means that the real-time threads in the element would need to be suspended, thus potentially resulting in missed deadlines for the threads of the element under evolution. This behaviour is, of course, undesirable for hard real-time systems.

Therefore the evolution operation should respect the timing constraints (deadline/delay constraints) of the active element that is subject to change. In this respect, the evolution-operation execution time is part of the timing constraint of the real-time system itself. Therefore it must not exceed the safe-state time of this element, e.g. the maximum duration of the evolution operation should be less than the minimum separation between two consecutive jobs of this element's task.

5.4.2 State Transfer

Another crucial issue of system consistency that should be considered when addressing a dynamic evolution is that of handling stateful artifacts (components that may have internal information, or connectors that may have buffers full of messages(García-Borgoñon et al. 2014)). 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 of the data structure being different), to the new element. Meanwhile, this step is not required for the replacement of stateless elements. This activity can be more complex if the internal structure of the two elements is different, which requires identifying and extracting the relevant data from the old element. This data must be modified in order to fit the new element.
Practically, it is difficult to develop a generic abstract that can fit the internal data structure for all the system elements in order to store the state of any element during its evolution. Therefore, preserving and transferring or transforming the internal state of the elements is a specific step. Thus, if a transfer state is required, this process should be specifically remedied with/for each operation.

5.4.3 Change management

Another issue in dynamic evolution is relevant to the mechanism performing this process and to how the changes are driven (activeness of change), how the evolution events can be detected and then how the suitable reactions can be effected. More accurately, it relates to how this process can be managed.

Generally, the software system can be reactive (the changes are driven externally), or proactive (a self-managing system which drives the changes by itself) as mentioned previously in Section5.3.2.
Thus, either the system is instrumented (intertwined) with change management, or with an interface to allow an external agent (controlling system) to introduce the changes dynamically.
In dynamic evolution, management can be represented as an evoluter (Role) who is responsible for dynamically performing the evolution. This can be formulated as a controlling system that monitors a controlled system (software system) in order to detect and analyse an evolution event when it occurs on the controlled system or its environment to select or synthesize the appropriate course action or scenario of evolution.
The dynamic MES should provide a modeling concept to express both the controlling (management) techniques for proactive and reactive changes.

5.4.4 Dynamic evolution scheduling

The issue of the timing constraints is more important when we handle a dynamic evolution of a hard real-time system, which needs to maintain high levels of application availability. In fact, whatever the change-management system used, the dynamic evolution operation is considered as a real-time task and, once this unscheduled task (unexpected event) occurs, it should not affect the timing constraints of a system's tasks.
System tasks are scheduled and executed according to their dynamic priorities. Indeed, whatever the system tasks are, the evolution operation should behave (interact) without compromising the tasks' completion. Generally, tasks in a hard real-time system have a higher priority than the

evolution operation which is usually handled as background tasks (lesser priority). In this aspect, if a background-priority task is used to evolve an element, this evolution task can be preempted by any higher-priority task (including a task from the element under evolution), which can lead to an unsafe state or loss of the internal state of the element.
Therefore, an evolution task should directly derive its priority from the element undergoing change. Thus, the management change should be able to handle the evolution tasks safely while still guaranteeing the timing constraints of the system, e.g. it should dynamically prioritise this unexpected event (evolution operation) within the system threads.

Furthermore, in the replacement and addition operations, it is necessary to guarantee that the new element threads have taken over the role of the old element threads without deadline violation, which also requires dynamic rescheduling in order to integrate the new element threads with the rest of the system's threads in the scheduler.

5.5 Dynamic evolution of software architecture

Software architecture cannot be immune from changes. Architecture is an artifact, like the code, needs to evolve in order to satisfy the new requirements of the ever-changing environments and technologies in which the system is working. Moreover, architecture must change to maintain consistency between the architecture and the implementation.
Buckley el al. in their taxonomy of software change (Buckley et al. 2005), state that evolution can be performed at different granularity levels. Granularity is an evolution aspect under the *Where* dimension, which refers to the scale of the software's artifacts that are to be changed. Granularity can range from very coarse, through medium, to very fine. The Coarse granularity refer to those changes that are performed at the software-architecture level (coarse-grained artifact), which may have an impact at the system, subsystem or package level. The dynamic reconfiguration of software architecture can take place under this category. However, medium granularity might include changes that can affect class, component or composition level. These kind of changes as an architecture-centric evolution might be handled by a dynamic evolution of software-architecture elements.
Kramer and Magee define the dynamic changes as "*an Evolutionary process which may involve modifications or extensions to the system which were not envisaged at design time. Furthermore, in many application domains there is a requirement that the system accommodate such change dynamically, without stopping or disturbing the operation of those parts for system unaffected by the change.*" *(Kramer and Magee 1990)*
Many reasons have necessitated the need for dynamic evolution (*e.g.*, economy, safety). The main goal is to be capable of changing a running system without stopping and restarting the whole application.
Software architecture brings modularity to a software system (composition), separate the concerns and minimises the complexity. Thus, architecture facilitates the identification of the pars (concerns) which are subject to changes. In this respect, architecture plays an important role in

dynamic evolution, especially in reducing the impact of code (regeneration) to the rest of the system (modules) unaffected by the changes. The parts (components) which are subject to change can be modified (regenerated), while the rest maintain the provision of services.

5.5.1 Dynamism in Architecture Description Languages (ADLs)

Dynamic software architecture focuses more on the dynamism aspect of software architectures. There are architecture in which interaction among components might change during the execution time. Dynamic ADLs are languages used to describe these architecture models. The ISO/IEEE 42010 specification defines and standardises the minimum requirements for ADLs. Software architecture captures both structural and behavioural design decisions of the system under development. The aim of using the ADLs is to help development teams to understand and to communicate early design decisions.

System structure is the main aspect that most architecture models capture. All ADLs model the structural aspect as a configuration of components and connectors (interface, link, attachment, binding, etc).

The most the first generation of ADLs includes UniCon (Shaw et al. 1995), xADL (Khare et al. 2001) and COSA (Smeda, Oussalah, and Khammaci 2004), which support the structural architecture where the interaction between component or the dynamic behavioural of the components cannot be captured. Afterwards, formal behavioural modeling methods are used to express the behavioural aspect in several (dynamic) ADLs. Process algebra (Milner 1989) was among these methods, used in Darwin (Magee and Kramer 1996), LEDA (Canal, Pimentel, and Troya 1999), and Dynamic Wright (Allen, Douence, and Garlan 1998). Graph-based methods have been used by other ADLs, whereby graph grammars are used to define software architecture and architecture style. Software configuration is represented as a graph, whereby the dynamic reconfiguration is specified through graph re-writing rules. Hirsch et al. (Hirsch, Inverardi, and Montanari 1998) propose an approach which represents software architecture as graphs and architectural style as graph context-free grammars. The construction and the dynamic evolution of architectural style are represented as context-free productions and graph rewriting. Baresia et al. (Baresi et al. 2004) used typed graph transformation to describe architectural styles encompassing a type graph to define the architectural elements, a set of constraints to restrict the valid models further, and a set of graph transformation by which a dynamic reconfiguration can be specified.

Reflection-based methods have been also used to formulate the dynamic aspect in software architecture and are used by some reflective ADLs. Fahrmair et al. (Fahrmair, Salzmann, and Schoenmakers 1999) have proposed an extension Jini™ platform within reflection mechanisms in order to develop distributed systems where the participating clients, services and their interactions can adapt dynamically to a changing environment. This was done by developing a meta architecture upon a Jini system, which reflects the running Jini system.

Cuesta et al. (Cuesta, de la Fuente, and Barrio-Solárzano 2001) developed a formal ADL called PiLar, whereby the developer can model hierarchical systems which can be dynamically reconfigured and evolved. Architecture can be defined in meta-layers, for example meta-components can be reified and be modified by meta-meta-components. This reflective capability of PiLar permits to describe dynamic reconfigurations and dynamic evolution of architectural types.

Archware architecture-description language (Verjus, Cîmpan, and Alloui 2012) is an architecture-centric software engineering environment based on π-calculus, which provides reflective capabilities by means of hyper-code and the concepts of dynamic system composition and decomposition.

PRISMA (Benedí 2006) is an Aspect-Oriented Architecture Description Language ADL, its concept based on integrating Aspect-Oriented Software Development (AOSD) and Component Based Software Development (CBSD) in order to describe software architectures of complex software systems. To this end, it defines aspect as a first-class entity, which is used to specify functional or non-functional properties for components and connectors. PRISMA divides the architecture into different levels of abstraction (type definition and configuration) and provides supporting for the dynamic evolution of software architectures using a reflective mechanism.

In addition to what has already been mentioned, other formal methods have also been used such as, Petri nets (Murata 1989), Actor model (Agha 1985) and Z notations (Spivey and Abrial 1992) in supporting ADLs to express the dynamic behaviours of their architecture elements. Other languages which support the expression of dynamic reconfiguration are: π-ADL (Oquendo 2004), C2 SADL (Medvidovic 1996) and Rapide (Luckham et al. 1995).

All these formal ADLs handle architectural dynamism from the specifications perspective. None of them provide the primitives to allow the architect to define how safe stop can be ensured, or how the internal state can be transferred from the old and the new element. ArchJava and Dynamic Wright are the ADLs which are considered as the most dynamic evolution aspect of the software architecture.

5.5.2 Dynamism in component-based software engineering

Software architecture is concerned with the ways of structuring software systems via the representation of their components and interactions and with specifying the rules that govern their development and evolution. Many aspects of design decisions can be expressed in an architecture model, including structure, behaviour, non-functional properties and configurations.

The component-based method is an architectural programming methodology which considers a component as a main module of the system. Component-Based Software Engineering (CBSE) is an approach to software development, the aim of which is to increase reusability and reduce complexity in the development of software-intensive systems. The dynamic evolution aspects have also been considered in the development of many component-based models and frameworks. Most of these extended components allow developers to **add**, **remove or reconfigure components at**

compile or runtime. Given this, the section summarises a selection of interesting research on component-based frameworks which support dynamic evolution.

- ***Simplex architecture*** is an approach introduced by Sha et al. (Sha, Rajkumar, and Gagliardi 1996), the aim of which is to provide fault-tolerant, dynamic upgrades for real-time systems. It was used in some highly responsive systems (e.g. Boeing 777 flight control) in which faults can be detected and fixed during runtime without jeopardising safety. Each complex component uses three units: safety, complex and decision. The safety implements system performance, safety controller and monitoring functions. The complex drives the system operations which can be changed or updated while the system is running. The decision unit determines which controllers should be used.

- ***ArchJava*** (Aldrich, Chambers, and Notkin 2002) is an Architectural Programming Language (APL) which extends the Java programming language with component classes, connector and ports. A component in ArchJava is a special kind of object whereby its code is defined using component classes. Components communicate through explicitly declared ports whereby the language provides constructs for each port to define a set of required and provided methods. The connector binds each required method to a provided method. ArchJava guarantees communication integrity between an architecture and supports dynamic architectures where components can be created and connected at runtime.

- ***Draco*** is a component-based framework which supports dynamic reconfiguration and runtime change of component. It handles how the state can be transferred from the old component to the new one and how the consistency can be preserved during runtime change as a means of achieving a safe state. Draco is implemented as a middleware platform in the toolchain which consists of a custom component language, a pre-processor which translates this language into standard Java, an Eclipse-based environment that assists the programmer with component development and the component framework (Draco). In order to realise runtime evolution of component-oriented applications, four primitives for live update are supported: loading, unloading, connection and disconnection of components.

- ***Fractal*** (Bruneton et al. 2006) is a hierarchical component model that provides reflective capabilities for the development of highly reconfigurable distributed systems. Fractal itself is not a component framework, but it describes how components should be specified and connected, along with the implementation. Components are encapsulated with controls which manage their interactions with the external environment and provide the feature that yields components with different introspection and intercession capabilities (including, for example, access and manipulation of component contents, control over components' life-cycle and behaviour, etc.). Interfaces are used as communication points to the outside environment and the content components.

  Several implementations and extensions of the Fractal component model were introduced with the aim of increasing and enhancing the dynamism level. This including, the FraSCAti component model (Seinturier et al. 2009) and WildCAT (David and Ledoux 2005). The FraSCAti platform

supports an extended Service Component Architecture (SAC) component model, where components can be provided with reflective capabilities to allow their introspection, monitoring, control and dynamic configuration. WildCAT is an extensible Java framework, the aim of which is to support the creation of self-adaptive applications. It provides a simple API for programmers to access context information both synchronously and asynchronously.

- ***SOFA 2.0*** (Bures, Hnetynka, and Plasil 2006) is a hierarchical component model which has inherited the connectors and most of the other features from its predecessor, SOFA/DCUP (Plasil, Balek, and Janecek 1998). The main differences from the earlier system are the meta-model-based design of components, the support of dynamic architecture reconfigurations, the support of any communication style and clearly separated and extensible control parts of the components. A metamodel-based definition (Component Description Language [CDL]) is used to define SOFA 2.0 instead of an ADL-based definition which is used in the original SOFA. This approach allows to exploitation of advantages (MOF technology) such as automated generation of a repository with standardised interface, standardized XML-based interchange format, support for automated generation of models' editors, etc.

  A component is specified by frame and architecture constructs whereby the frame defines a set of provided and/or required interfaces and their properties. The architecture defines the implementation of the frame. The support of dynamic architecture reconfiguration is achieved by well-defined reconfiguration patterns. Factory pattern, removal pattern and service access (utility interface) pattern are the three configuration patterns provided by SOFA 2.0. The factory pattern serves as component factory and the removal pattern serves as destructor, while the service access pattern allows access to external services through utility interfaces.

- ***OpenCOM*** (Coulson et al. 2004) is a reflective (building on Microsoft's COM) component model which provides provisions for reconfiguration. Components are deployed in containing entities that offer a component runtime. These are called capsules which provide operations for dynamically loading/unloading components and also binding/unbinding interfaces and the components at runtime. A set of reflective meta-models manage both the evolution and consistency of the base-level system. These reflective meta-models are: the architecture meta-model which represents the compositional topology of the components in terms of an architecture graph within a capsule that can be inspected to discover the topology, and adapted to change the topology the interface meta-model which supports the dynamic exploration of the set of interfaces defined on a component and the dynamic invocation of methods defined on these interfaces, both of which enable the invocation of interfaces whose types were unknown at design time; and the interception metamodel which permits the dynamic adaptation of components by means of interceptors at bindings between interfaces.

- ***Gravity*** (Hall and Cervantes 2003) is a service-oriented component-based framework which supports a run-time adaptation in response to the dynamic availability of functionality provided by the components. It is implemented on top of the Open Services Gateway Initiative (OSGi) framework. The dynamic availability in gravity was mainly supported by introducing the Service

Binder as a mechanism to automate service dependency management in its component model. Gravity supports a number of dynamic component reconfiguration scenarios that are triggered by environments with dynamic evolution capabilities.

- ***iPOJO*** (Escoffier, Hall, and Lalanda 2007) is a flexible and extensible service-oriented component model, the aim of which is to simplify the development of dynamic applications over the OSGi service platform. The name iPOJO is an abbreviation for injected POJOs (Plain Old Java Object). It has been widely used in academic and industrial projects and has been developed to run modern applications that exhibit modularity and require runtime adaptation and autonomic behaviour. By separating the component class and the component metadata, the iPOJO is able to manage the runtime component (the instance), to manage its life-cycle, to inject required services, to publish provided services and to discover needed services. As iPOJO relies on the OSGi R4.1 framework, it can be used on any compliant OSGi implementation such as Apache Felix, Eclipse Equinox, or Knopflerfish.

- The ***Rainbow*** framework (Garlan et al. 2004) provides a reusable infrastructure for self-adaptive systems, based on the belief that an external control mechanism can provide a more effective engineering solution than internal mechanisms can . It allows reuse of the adaptation knowledge and practices across different systems. The framework integrates a system's architectural model within its runtime (controlled) system, which can be used by a feedback control loop for self-adaption. The MAPE control loop (controlling system) consists of four units: 1) a model manager which handles and provides access to the system's architectural model; 2) a constraint evaluator which observes the model periodically and triggers adaptation if a constraint violation (evolution event) occurs; 3) an adaptation engine which determines the set of actions and carries out the necessary adaptation to fit the occurred event; and 4) an adaptation executor which reflects the changes in the application layer (controlled system).

- ***Plastik*** (Batista, Joolia, and Coulson 2005) is a meta-framework for reconfiguration, which integrates an architecture description language (Armani ADL/ an ACME extension) and a component-based middleware (OpenCOM). The framework was developed to meet the aim of providing dynamic runtime reconfiguration while ensuring consistency. Armani is an extension of the ACME ADL which permits the description of architectural constraints over ACME architectures (e.g. constraints on system composition or system behaviour), but does not adopt the specification of dynamic reconfigurations. Therefore, the authors of in Plastik extend Armani with statements that define the reconfiguration triggers, the removal of elements and the detachment of the existence of dependencies among components. This extension allows expression of the reconfiguration actions which should be specified at design time/programmed reconfiguration. Meanwhile, for the unforeseen reconfiguration at runtime, the OpenCOM metamodels were used. These provide reflective capabilities, thus enabling dynamic reconfiguration. Dynamic reconfigurations can be described and executed, both reactively and proactively, by means of a centralised system configurator which is divided into two levels: an architectural configurator

responsible for accepting and validating reconfiguration requests at the ADL levels; and a runtime configurator responsible for managing the OpenCOM/runtime level.
There is an abundance of work on adaptive component frameworks for both industrial and research purposes, more than have been summarised in this section, and these include: Argus (Bloom 1983), K-component (Dowling and Cahill 2001), OpenREC (Hillman and Warren 2004), Java/A (Baumeister et al. 2006), Adapta (Sallem and e Silva 2007), Jade (Taton et al. 2005), CoBRA (Irmert, Fischer, and Meyer-Wegener 2008), StarMX (Asadollahi, Salehie, and Tahvildari 2009), iPOPO (Calmant et al. 2012) and several extensions of CORBA and Java which are extended to dynamic-reconfiguration capabilities and so forth.

5.5.3 Comparison of architecture-centric dynamic evolution approaches

J Buckley and others, in their paper "Towards taxonomy of software change" (Buckley et al. 2005) defined four themes of software change each: temporal properties (the change made), the object of change (where a change is made), the system properties (what is being changed) and the change support (how the change is accomplished). Each theme comprises some dimensions of software change. These dimensions (15) are classified in two categories, the Characterizing factor and the Influencing factor, as we can see in Figure 37.

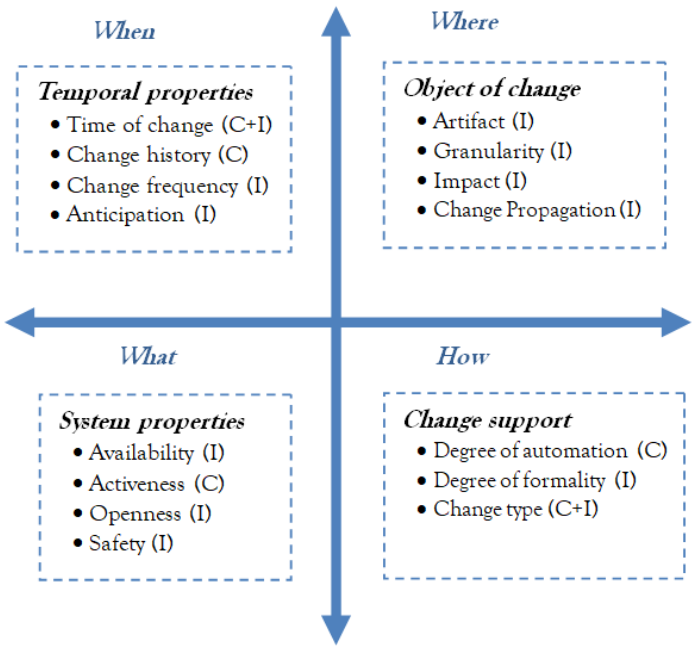

Figure 37: Themes and dimensions of software change

Based on this taxonomy, we can develop the model in Figure 38 to compare the previous dynamic evolution approaches. The Time of the change is one dimension of the temporal-properties theme which may take one of the following values: Static, Load-time and Dynamic. Since the Time of change in these approaches is fixed on the value "Dynamic", this theme is not considered as an axis

of the model (it originally forms the reference/base of comparison). The remaining three themes form the main axes of the comparison model. Each axis represents a different theme through an appropriate dimension. For example, from the system properties theme, the Safety dimension (consistency management) has been selected to represent this theme. This axis evaluates the presence of mechanisms or techniques that have been used to maintain the consistency of a system before and after a dynamic evolution. Meanwhile, availability is an important factor and is the main motivation behind the dynamic evolution, or the main requirement which must be satisfied. The activeness is relevant to the responsibility of automatically effectuating the evolution operation (system or environment). Thus the system should be in a working state in order to execute or respond to the evolution operations. Finally, the static evolution can be implemented on both the open and closed evolutionary system whereas the system must be openness to change in order to allow integrating the change at runtime.

In the change-support theme, the degree of formality was selected to determine what mechanism has been chosen to implement the dynamic evolution (ad hoc, mathematical formalism). Meanwhile, the automation (automated, partially automated and manual) of the changes must be executed in a bounded time (fast-time response) by an internal or external controller (automatic Role) in order to integrate change at runtime without halting the system (or at least the unaffected parts of the system).

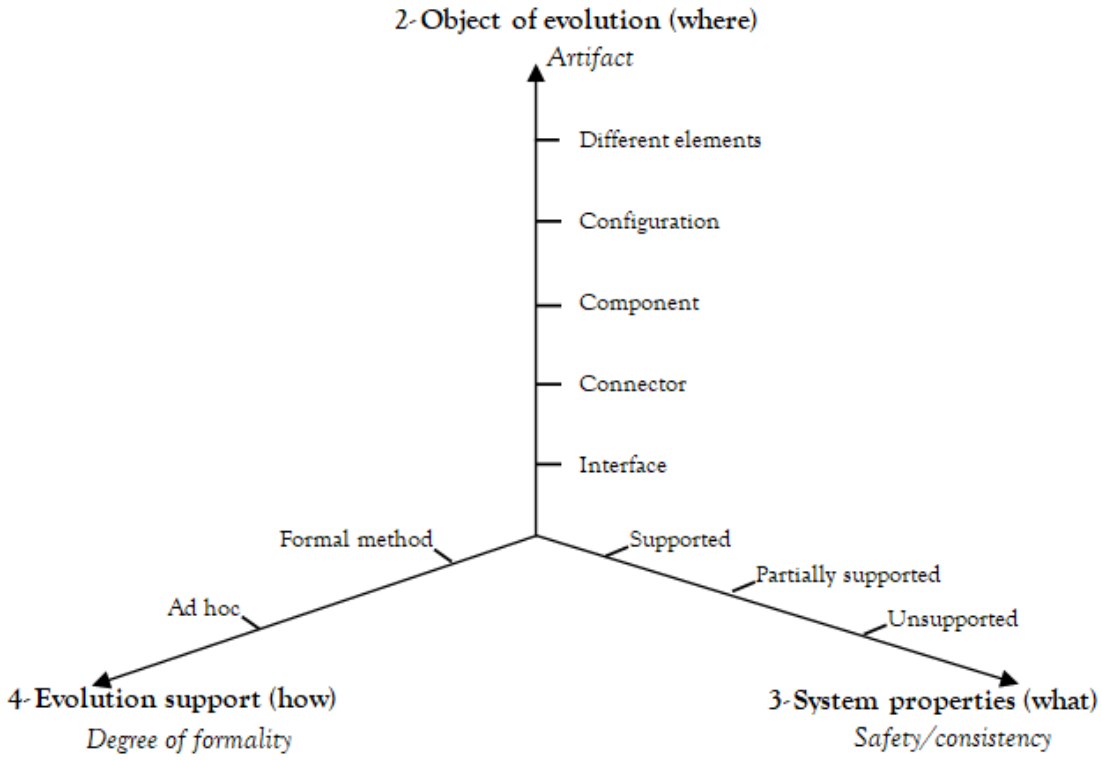

Figure 38: Reference model

Table 11 presents a comparison between these approaches (frameworks) in accordance with the reference model in Figure 38.

Table 11: Comparison of some dynamic component-base frameworks

ADL	Object of evolution (artifact)	System properties (Safety mechanism)	Evolution support (Degree of formality, Methods)
Darwin	Component, Binding, Configuration	The binding is only permitted if the service type of the requirement matches the service type of the provision.	The description of dynamic evolution is limited. Partial specification of the dynamic architecture in terms of component creation during the application execution. Lazy and direct dynamic instantiation. Process algebra (π-Calculus)
LEDA	Component, Connector	Role and adaptor are used to ensure the maintenance of compatibility in the interaction/compositio n of the new element.	It supports dynamic evolution that must be known at the design time (programmed). Process algebra (Communicating Sequential Processes)
Dynamic Wright	Component, Configuration	Connector deadlock-free. Port-Role consistency check.	It uses the reflective mechanism to introduce the dynamic change in the architecture. Process algebra (π-Calculus)
PiLar	Component	Not defined	It supports the dynamic evolution at the type level. π-Calculus (Maude/rewriting logic)
π-ADL	Component, Connector.	Not defined	It describes dynamic evolution in terms of addition, removal, upgrading of component, but does not specify the mechanism.
C2 SADL	Component, Connector, Configuration	Not defined	It allows description the dynamic of the architecture in terms of component creation (link and unlink operators). Process algebra (posts)
Rapide	Component, Configuration	conformance checking	It supports dynamic evolution that must be known at the design time (programmed). Process algebra (Communicating Sequential Processes)
Framework	Object of evolution (artifact)	Safety mechanism (consistency)	Evolution support (Method by which evolution is triggered)
Simplex	Component, Connector	Safety unit (n-version scheme)	It allows dynamic reconfiguration and component updating, by a user-supplied monitor
ArchJava	Component,	_	It allows the dynamic of the

	Connector, Configuration		architectures in terms of component creation, connection and reconnection.
Draco	Component, Connector	Tranquility	It allows loading, unloading, connection and disconnection of components at runtime, using Fresco methodology
Fractal	Component, Connector, Configuration	Quiescence	It supports dynamic reconfiguration and component creation, using API to allow both adhoc and limited programmed reconfiguration (reflective capabilities).
SOFA 2.0	Component, Connector, Interface	–	It supports dynamic reconfiguration, via evolutionary patterns
OpenCOM	Interface, Component, connection	Quiescence	It address the dynamic reconfiguration via reflective mechanism
Gravity	Component	–	It supports installation, update, activation, and removal of components at run time, which can be triggered via a dynamic environment.
iPOJO	Connector, Component	Inversion of control design pattern	It provides basic mechanisms to handle dynamically component via the OSGi framework.
Rainbow	Component Configuration	–	It allows dynamic reconfiguration via an outside control loop.
Plastik	Component, Links	–	It allow dynamic configuration (removal and creation of elements) via ACME ad-hoc reconfiguration.

5.6 Dynamic evolution Meta-style MES

MES consists of defining foundational meta-concepts for describing a software architecture evolution. These essential concepts were used in modeling and analysing static evolution styles (Hassan and Oussalah 2016). MES can be refined (specialised) for any other kind of architecture evolution. In this sense, the intent is not to define a new meta-style for modeling dynamic evolution of real-time systems, but to annotate MES with information required to analyse and model this process. Hence, it focuses on integrating the concepts of dynamic interaction and schedulability analysis into evolution styles.

This section presents how the concerns and issues of the dynamic aspect that have been introduced in Section 5.4 will be handled in order to enhance MES so as to analyse and model the dynamic evolution of software architectures. Figure 39 illustrates our proposition to extend MES with the necessary information to fulfil the requirement for modeling the dynamic evolution styles.

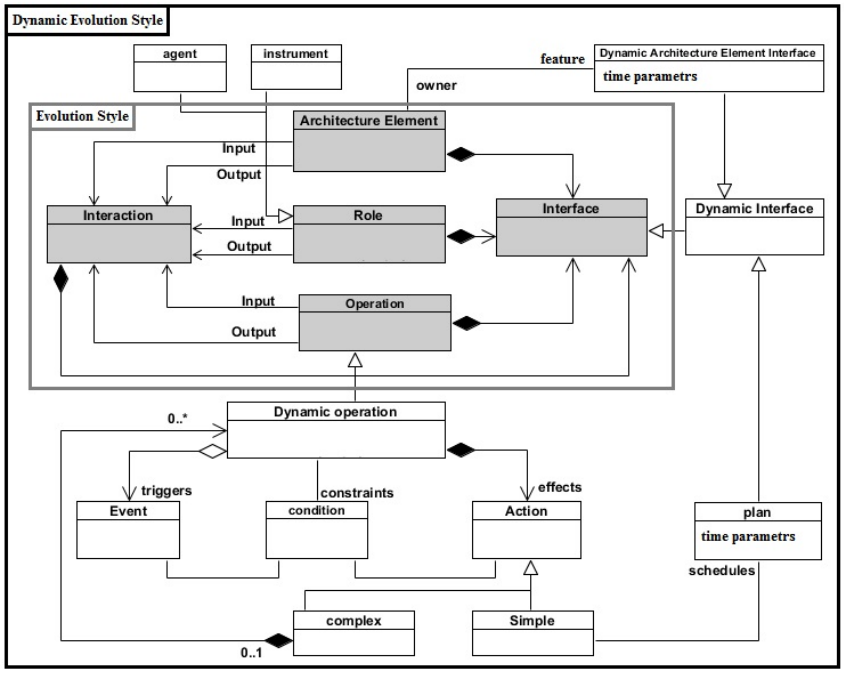

Figure 39: Dynamic evolution styles

Actually, the dynamic evolution of a real-time system requires the introducing changes in bounded time. Managing and performing this process without violating the timing constraints is more complex. This requires a fast, interactive Role (intelligent change management) which minimises or eliminates the human intervention Role and shifts it from operational to strategic. Thus, the Role in dynamic MES should support the concept of an automatic Role, either as an internal instrument or as an external agent.

Indeed, the needs of an architectural element for change are required when an evolution event has occurred. Therefore, the unexpected evolution events must be assigned with their potential scenarios of reactions (evolution paths). A strategy to synthesise the suitable reactions is defined such that all the affected elements are complete within their deadlines. In this sense, each evolution path consists of a series of evolution operations and represents one choice to evolve the architecture from the current state to the target state.

Therefore, each simple evolution operation must have a dynamic interface which provides the necessary parameters (priority, time execution/period) in order to be safely handled and not to cause timing misbehaviour (Denial of Service).

Several scheduling methods for the handling of unexpected events in real-time systems have been proposed in the literature in order to service aperiodic requests, where a set of hard aperiodic tasks is scheduled using the Earliest Dead-line First (EDF) algorithm (Liu and Layland 1973). Among them, the Total Bandwidth Server (TBS) (Spuri and Buttazzo 1994), (Spuri and Buttazzo 1996) and the Earliest Deadline as Late as possible server (EDL) (Chetto and Chetto 1989) both provide an efficient aperiodic service under EDF. In TBS, the worst-case execution time of aperiodic requests must be known in advance (which is not the case in the care evolution operation). This is why we

will turn to EDL to schedule this unexpected event dynamically and jointly with the system threads. Thus, the Operation should provide the necessary parameters that are needed by a dynamic scheduling algorithm in order to be scheduled.

The architectural element that can be changed in its active state must also have the suitable (dynamic) interfaces to provide the required parameters in order to be dynamically evolved.

An interface is needed to handle the internal state of the element during the replacement Operation (update). Another interface is also needed to provide the time parameters. These scheduling parameters are required by the scheduler to schedule the evolution operation and the threads of the new elements dynamically: Worst Case Execution Time (WCET), deadline and release time. These parameters allow the schedulability analysis of the dynamic evolution of hard real-time systems.

- ***Role***: Generally, a Role is responsible for the evolution operation that performs the changes. Managing and performing during the run-time requires a highly interactive Role. This automated Role can be an external agent or an internal instrument. The Role is responsible for ensuring that the execution of the evolution operation does not affect the integrity of the system. It works as a controller that supervises the model periodically and triggers evolution actions when certain events have occurred. These actions can be previously programmed at the design time to integrate introduce foreseen changes, or directly synthesised in order to respond to unforeseen evolution events (not initially predicted during the design of a system). There are two common ways of managing the evolution activities: either a top-down approach – which relies on a centralised management by which all the evolution decisions are taken and the changes are applied on the system globally – or a decentralized method (bottom-up approach) where the evolutions are locally managed by each element.

- ***Dynamic Operation***: A dynamic evolution operation can be a simple evolution, such as adding or deleting an architecture element, or a composite one, such as replacing an architecture element. A dynamic evolution process should be expressed in such a way that it supports both kinds of activeness, namely proactive and reactive. This can be achieved by separating evolution requests (Events) from the evolution mechanisms (Actions). Therefore, the construct of an evolution operation is based on the ECA rules: "On Event If Condition Do Action At Time" which means: when an evolution Event occurs, if the Condition is verified, then execute the suitable Action at an appropriate time. This means selecting the (suitable series of) evolution action(s) and putting them into execution in a proper sequence and with proper timing. The dynamic operation must offer a dynamic interface (plan) which provides relevant run-time parameters that are needed to schedule the operation as soon as possible within the system tasks.

- ***Dynamic Architecture Elements***: An architecture element must be evolutionar-open, which means that it has an interface with the necessary parameters (features) that enable it to react dynamically to evolution operation. An element should be able to provide its scheduling parameters to allow the Role to effect the changes dynamically without breaking the timing constraints of the system. These parameters also allow the dynamic scheduling of the new element's threads in the scheduler

(in order to allow the new version of the element to continue where the old version left off). The dynamic architecture element interface provides a port to pass the internal state in order to be converted to the structure of the new element.

- ***Interaction:*** In fact, the dynamic evolution is a real-time task, so the interaction element must guarantee that evolution Operations are subject to the timing constraints. The interaction element ensures the availability of required interfaces and parameters between the elements (Operation, Architecture Element, and Role) in the interaction. These interfaces are necessary for each element to participate in the dynamic evolution activities.

- ***Dynamic Interface***: A dynamic element should have an appropriate interface which provides the required parameters to interact efficiently at run-time. Such an interface is required, for example, to allow the Instrument (the Role in a self-managing system) to observe an architecture element in order to detect any evolution event or to determine the appropriate time to effect the changes.

- ***Dynamic process***: Represents the dynamic configuration of the evolution process that transfers the software architecture from the current architecture style to the target style. This configuration provides the temporal and topological (timing and sequence) organising of the evolution operations while respecting the consistency and integrity of the architecture elements.

For comparison, Table 12 has been developed from the issues of dynamic evolution embraced by MES and some of the dynamic evolution approaches analysed in the Section 5.5. This table contains two examples from each group: the formal approaches handling dynamic ADL and the technological approaches handling component-based models and frameworks for building systems with dynamic evolution property.

The formal ADL approaches provide a specification perspective to describe when and how the system architecture should be evolved (proactive changes). To be precise, they describe the programmed dynamism of a system (reconfigurable component). This gives the ability to rigorously specify the global architecture of the system which can be automatically analysed or which may support the automatic generation of parts of the software systems (Barais et al. 2008). They do not addressing the dynamism of an evolution operation, i.e. they do not completely specify the mechanisms of dynamically effectuating the evolution process nor the qualifications (constraints).

The technical approaches introduce some component-based models and frameworks which provide features for supporting dynamic evolution. With these features, a dynamically evolved software system can be built. Several mechanism and strategies to handle the dynamisms have been introduced, some of which manage to address different dynamic evolution issues. Generally, the dynamic-scheduling issue has been not completely handled, as most of them tackle the dynamic evolution of a firm real-time system. Thomas Richardson in his PhD thesis introduces an extension of the OSGi framework (RT-OSGi) to support the developing reconfigurable real-time system (component) in which the Real-Time Specification for Java (RTSJ) has been used to handle the dynamic rescheduling (Richardson 2011).

Table 12: Comparison of dynamic evolution approaches

		Safe-stopping	Safe transfer	Change management	Dynamic schedulability
ADL	Darwin	Supported	Unsupported	Distributed management	Not defined
	Dynamic Wright	Not-defined	Not defined	Centralised management	Not defined
Component-Based	Simplex	Safety unit	N-version scheme	Distributed: module management unit	Partially supported parallel running
	Draco	Tranquility and Quiescence	State transfer	Centralized evolution management	Not defined
Meta^2 model	MES	Supported dynamic interface	Supported dynamic interface	Supported Role: Centralised: agent Distributed: instrument	Supported dynamic interface (Plan)

The MES metamodeling framework provides a set of concepts (specifications) that can rigorously specify the dynamic architecture and dynamic evolution (process). This framework can offer the means of being used to develop an architecture language for modeling software-architecture evolution. These languages (evolution styles) can provide the means for architects to acquire and share architecture-evolution knowledge and practices (mechanisms, techniques, strategies) that have been used in developing technical approaches for dynamic component-based models and frameworks.

5.7 Conclusion

This chapter has handled the dynamic evolution of a software system, the reasons behind this process and the issues accompanying its implementation. We have reviewed some common architecture-centric approaches for dynamic evolution and looked at how they address these issues. Accordingly, we propose an extension of MES in order to satisfy the requirement of modeling this process at the architecture level. To realise this requirement in a better way, we integrated some behavioural concepts (that had been extracted from the reviewed implementation approaches) into MES. As a result, we can have a sound understanding of dynamic evolution issues and constraints, which is a prerequisite for modeling dynamic evolution of software architecture.

Chapter 6. Multi-View in Evolution Meta-Styles

6.1. Introduction

Planning the large-scale evolution is a complex activity that requires an understanding of the overall system structure and the rationale behind its design decisions before a set of non-destructive changes can be proposed. Software architecture is a system's artifact, the aim of which to capture the structure, behaviour and design knowledge of the system. Thus, architecture can be the optimal artifact whereby software-evolution alternatives can be predicated, understood, analysed, explored and verified. Re-architecting can be considered as a vehicle where different ways of evolution can be compared at a high level, far away from the complexity of low-level details.

Evolution style is a modeling approach, the aim of which is to capture best architectural knowledge and practice in a domain-specific software evolution. This body of knowledge can be used to guide the evolution team in the evolution process and to provide the rationale behind a possible alternative (Hassan and Oussalah 2016). Capturing all aspects of this process in one single model is almost impossible. Most design methodologies endeavour to develop an interlocking set of views, each of which intends to cover the stakeholder viewpoint (Boiten et al. 2000). Every viewpoint frames a specific stakeholder concern, and specifies what concepts and relations can be used to view the model in accordance with this viewpoint (in order to answer these concerns).
However, a multi-view style framework aims to define a methodology that can integrate a suitable set of views in order to offer a more complete picture (model) of the architectural evolution process, its alternatives, the decisions between these alternatives and the rationale behind those decisions. Here we investigate how (from where) the views of an evolution style model can be derived and how the meta-style (MES) can provide a common place (a meta-model) for instantiating or integrating these views.

In general, the majority of modeling methodologies, either in the object-modeling discipline or in the software-architecture description discipline, represent the artifacts in accordance with two main viewpoints: structure and behaviour. For example, in the former discipline, the Unified Modeling Languages (UML) is considered the most common generic-purpose modeling language. It comes with different (13) diagrams classified under two categories: structural diagrams (Class Diagram, Composite Diagram and Object Diagram) and behavioral diagrams (Sequence Diagram, Communication Diagram, Activity Diagram, State Diagram, Timing Diagram and Interaction Diagram) each of which views software systems from one of these two viewpoints (Rumbaugh, Jacobson, and Booch 2004).
Meanwhile, another methodology in software architecture has been proposed in recent years, which considers the architecture model from the point of view of its stakeholders, looking at how they reason and make decisions. It is based mainly on ***Rationale*** in the definition of architecture that was proposed by Perry and Wolf. Jansen and Bosch also emphasised this view which was

manifested in their definition of software architecture as a composition of a set of architectural design decisions (Jansen and Bosch 2005).
Broadly, in software engineering, the aim of introducing the concept of views is the separation of concerns (reducing the complexity). Thus, multiple views are proposed as a means of managing the complexity of software artifacts and these include: requirements specifications, design models and programs. In what follows, we will first briefly conduct literature reviews on usage of the views in some areas of the software engineering discipline.

6.2. Multiple views in Software Requirements Specification (SRS)

The success of software system quantifies to what extent it satisfies the requirements (purposes) for which it was intended. The requirement specification is defined in the standard IEEE 830-1998 (Committee and Board 1998) as a description of the behaviour of a computer system and the interactions that users may have with it (more precisely, it is the document in which all requirements, functional and non-functional, are reported). Thus, requirements engineering is the process of discovering the software purpose, identifying its potential users and their needs and capturing these in a formal way that can be amenable to analysis, communication and subsequent implementation (Nuseibeh and Easterbrook 2000). Since the requirements represent the objectives behind which the software is developed and evolved, eliciting these requirements precisely represents a vital basis on which the right system will be constructed. To this end, verifying the correctness, completeness and consistency of the requirements is an important step in the software development, which confronts different problems inherited from the diversity of the stakeholders (their goals, perspectives… etc). Accordingly the multi-views methodology has been adopted by several approaches to tackle the issues of requirements specifications, as shown in the following list.

- ***SADT:*** Structured Analysis and Design Technique (Ross and Schoman 1977) is a toolchain for Structured Analysis and was the first graphic method developed for use in requirements specification. It was developed by Ross et al. as architectural documentation for large and complex systems and later used as a methodology to support the managing of the system's complexity via decomposition into subsystems, creating a hierarchical parent-child structure. SADT comprises two main components the diagramming language of structured analysis (SA) and the design technique (DT) that offers the required conceptions for using the SA language (Ross 1977). The graphical representations are based on a data-flow (actigram) diagram which expresses the system's activities and provides the basis for decomposition in the form of a rectangular box, representing the system's "most abstract activity", a set of data input, data output and control arrows. Requirement-tracing can be achieved by checking the consistency between abstraction levels through a comparison of their counterpart input and output. The multi-view is implicitly derived in SADT through the modeling technique (decomposing), whereby each actigram of the SADT model constitutes a view at a certain abstraction level.

- ***CORE*** Controlled Requirement Expression (Mullery 1979) is a formal method for requirement specialisation and the modeling of a complex software system. It was developed in the late 1970s for British Aerospace and is based on the idea of the decomposition of a system model in accordance with the conception of the viewpoints, each of which is associated within a certain client authority. CORE is one of the earliest approaches which provides prescriptive guidelines on specifying and analysing system requirements based on viewpoints which capture functional and non-functional aspects. The CORE methodology defines a multi-step modeling process in the production of the requirement specification, which comprises the definition of the problem, viewpoint identification and gathering and the documenting of information about viewpoints. In the first activity, the goal will be to propose possible viewpoints, classifying them as functional and non-functional for the current decomposition (iteration). The second activity is viewpoint structuring which provides a framework that captures the requirements and iteratively decomposing them into a hierarchy of functional subsystems, each of which represents a certain viewpoint relevant to the corresponding level in the target system. This hierarchy enables to the decomposition of a complex task into smaller ones for better analysis at a lower specification level where more details can be annotated. Finally, the third activity is the use of tabular, which allows the collection and organization of the information about viewpoints in accordance with the actions of their inputs and outputs data, which supports the verification of the reliability of these diagrams.
- ***VOSE*** The Viewpoint-Oriented Software Engineering framework was proposed by Finkelstein et al. It supports the multiple perspectives in system development and provides a means of developing and applying systems-design methods (Finkelstein et al. 1992). It utilises the concept of viewpoints in order to partition the system specification, the development method and the formal representations, each of which is used to express the system specifications. The development of an intensive system often involves several stakeholders, each with has/her own perspective which relies on his/her skills, role, knowledge and expertise. To this end, VOSE starts by identifying the issue of the multiple-perspectives problem; therefore, it introduces "viewpoints" as a framework for structuring, organising and managing these perspectives. A viewpoint represents a building block which can be thought of as a combination of an actor and his/her role and knowledge in the development process. This loosely coupled principle provides a way to encapsulate partial knowledge about the system and domain-specified in a particular, suitable representation scheme – and partial knowledge of the process of design. Each viewpoint has the following specific components: a representation ***style***, a scheme by which a particular viewpoint can be expressed; a ***domain*** which defines that part of the "world" delineated in the style; a ***specification***, the description of the particular domains offered by the style; a ***work plan*** which describes the specification process; and a ***work record*** which registers the history and current state of the development.
- ***VORD*** Viewpoints-Oriented Requirements Definition (Kotonya and Sommerville 1996) is a viewpoints-based method for requirement engineering that covers the system process from the initial requirement to its modeling. In software terms, it can be considered as a client-server model in which the viewpoints are considered to be clients. VORD relies on a concept of viewpoint based

on the entities whose requirements are responsible for, or may constrain, the development of the intended system. There are two types of viewpoint in VORD: ***direct viewpoints*** correspond to clients in that they exchange control information and services with the system; ***indirect viewpoints*** are for those not directly interacting with the system, but who may have an interest in the system services. There are three main iterative steps in VORD, which address the issues of viewpoint identification and structuring, viewpoint documentation and requirements analysis and viewpoint specification. In VORD there is no generic structure; each organisation should establish its specific hierarchy of viewpoints in accordance with its needs and particularity. Viewpoint identification starts with abstract statements of organisation requirements and then proceeds with irrelevant steps producing hierarchical viewpoint classes which identify the requirements resources (end-users, stakeholders etc), all of which may be associated with direct or indirect viewpoints.

- ***PREview*** The Process and Requirements Engineering Viewpoints method (Sommerville and Sawyer 1997) was proposed by Sommerville et al. in 1997 with the aim of addressing the early stages of the requirements-engineering process where requirement resource must be clearly discovered and defined before system analysis and design can launch. The author states that the quality of the requirements specification can be improved in two ways: by improving the requirements-engineering process or by improving the manner of organising and specifying the requirements engineering. In PREview, the notion of viewpoints is complemented by the organisational concerns. Thus, the concerns are identified at the early start of the requirements-engineering process and decomposed into questions, constraints, or requirements. During the step of requirement analysis, questions related to stockholder concerns must be linked to all viewpoints, whereby a set of related questions may be answered by viewpoint sources. The separated concerns can be used to support the early design decisions by the mapping of concerns to functional modules (architectural decisions).

6.3. Multi-view in system modeling

Modeling has always been at the core of information systems development especially with increases the acknowledgment of MDE, in which the model increases the productivity and accelerates the development time while maintaining the quality of the system. However, a large software-intensive system can hardly be described and understood by providing one complex model. The design of such a system must also include the materials to be used (hardware), the databases, the third-party platforms with which the system must use or interact and the programming interfaces that the system code must use, or offer. Thus, a number of separated facets/sub-models should be developed, each of which can be used to describe a set of relevant aspects of a software system in so-called views. Views in software systems can be oriented of different concerns or aspects such as data view, interface view, architecture view, process view, interaction view, deployment view, state transition view and so on. Therefore, in this section, we will handle methods or tools which address the system modeling by providing views (diagrams) that cover some of these common views.

- ***UML*** Unified Modeling Languages (Rumbaugh, Jacobson, and Booch 2004) from one of the most common tools embracing the nation of viewpoints. Indeed, UML comes in its stranded 13 types of diagrams, classified under two main categories: structural diagrams (Class Diagram, Composite Diagram, and Object Diagram) and behavioural diagrams (Sequence Diagram, Communication Diagram, Activity Diagram, State Diagram, Timing Diagram, and Interaction Diagram). Each diagram type confers an implicit viewpoint from which a complex system or process can be handled. This provides the capability to decompose a complex model of a software system into several complementary sub-models that are less complex and more understandable. Moreover, UML provides an extension mechanism (profile) that allows users to add or customise predefined diagram types. Thus, it provides an opportunity to add new viewpoints, or to cover specific characteristics of a particular problem domain. An example of this is the Systems Model Language (SysML), which is a UML profile and which adds some viewpoints not found in standard UML, such as the requirements diagram and the parametric diagram. There is also the UML profile for software-architecture analysis and design language (AADL), the so-called MARTE (Modeling and Analysis of Real-Time Embedded Systems).

- ***VUML*** View Based Unified Modeling Language is an approach developed by Nassar et al. (Nassar 2003) the aim of which is to introduce the nation of viewpoints explicitly into UML. VUML defines a class diagram which includes the standard class of UML and a new type of multi-view class which is defined by two stereotypes: base and view. The basic stereotype refers to the common part accessible from all viewpoints, while the stereotype view refers to the specific part of the base associated with a particular viewpoint. In order to like these two stereotypes a dependency relation (stereotype ViewExtension) was proposed. Another stereotype (relationship) was also proposed the so-called ***ViewDependancy*** in order to preserve consistency within a multi-view class – which can be expressed via Object Constraint Language (OCL). Moreover, a multi-view component is defined, which is an extension of the notion of the UML component, allowing a classical interface or multi-view interface (ViewInterface) to be assigned. These pointview-based modeling elements allow the storage and retrieval of information according to the profile of the user (viewpoint) and offer the possibility of dynamic change of view point.

- ***SysML*** System Modeling Language is an OMG specification that defines a general-purpose language (UML profile) for system-engineering applications which explicitly adopt the multi-view modeling conception. This profile defines new ModelElements, some of which represent the main concepts of multi-view methodology defined by IEEE-1471 standard, such as view, viewpoint, stakeholder, concerns, rationale and conform relationship. SysML 1.3 uses packages (instead of classes in V 1.4 and 1.5) to represent views. It also uses classes for the viewpoints and extends the dependency (generalisation association in V 1.4) to specify the conform relationship between the view and the viewpoint. The viewpoint class defines some string attributes (stakeholder, purpose, concern, method, language etc), all of which are used to specify the construction and usage of the related view. Thus, the view must conform to the associated viewpoint in terms of the stakeholders, concerns, method, language and presentation requirements. The view package elements can be comment, constraint elements, element import and package import; thus, the view elements must

be defined in a model to be imported. Since the viewpoint defines its properties as strings, no verification policy defined. The viewpoint does not define the language used to express the view. Moreover, SysML does not define explicit correspondence between views. Therefore, it can state that SysML does not completely apply the viewpoint conception as having the same meaning of the viewpoint as in IEEE-42010 or IEEE-1471.

6.4. Multi-view in Software architecture

The architecture model is one of the system artifacts (models), and multi-view in architecture is handled in a separate section as our approach is architecture-based modeling. As the software architecture represents the principal artifact, this provides a fertile source of information for architects to explain the design decisions to stakeholders. Thus, it is almost impossible to capture all the architectural knowledge of a large complex system in a single model which can fulfil its wide variety of stakeholders (users, acquirers, analysts, developers, testers, maintainers, inter-operators and others). Satisfying the concerns and aspirations of any stakeholder is the role of the architect, which must be answered within the architectural model. The recommended solution for developing a comprehensible model (architectural description) is to partition it into a number of separate and interdependent views. Each view represents different aspects of the system that fulfil functional and non-functional requirements (such as performance, robustness, availability, competition, distribution, etc.). Thus, an architectural view of a software system is a representation of the system from the perspective of a viewpoint. To this end, several architecture frameworks have been proposed, each of which defines or proposes several viewpoints from which an architecture model can be represented. The following paragraphs will briefly introduce some of the common frameworks.

- ***Zachman Framework*** (Zachman 1987) is the first architecture framework (Enterprise Architecture) introduced in 1987 by John Zachman, which paved the way for the emergence of other eminent enterprise architectures. It provides a set of vocabulary and collection perspectives for describing system development and enterprise engineering, the better to describe IT architectures across organisations. The framework was first defined as a bidimensional matrix (5x6), whereby the rows comprised five modeling layers (scope, business, logical systems, technical systems and detailed representations) and the columns contain six perspectives or views (data, function, network, people, time, motivation) based on the six basic questions (What, Where, When, Why, Who and How). Several amendments have subsequently been made to this. Table 13 indicates the architecture views in The Zachman Framework.

Table 13: The architecture views of the Zachman Framework

Views	Definition
Context	It is concerned with the basic requirements and represents the basis for estimates regarding the cost, scope and functionality of a system.

Business	It shows all the business entities and processes.
System	It determines the data and functions that realise the business model
Technology	It is concerned with the technological implementation of a system
Integration	It handles the system with regard to deployment aspects and configuration management.
Runtime	It covers the operation of a system within an organisation

- ***"4+1" Model*** This was proposed by Philippe Kruchten in 1995 of Rational Software Corp (Kruchten 1995b). It describes the concepts of an architecture model in five prime views: the logical view, process view, development view, physical view and scenario view. He suggests using a set of functional usage scenarios, the so-called "Four Plus One", to explain how the views (the first four) work together. The use case view provides scenarios for the developer to understand the other four views and also provides a means of reasoning about the architectural decisions. This approach became widely popular and Kruchten's view set is simple, logical and easy to explain. Moreover, it has also been accepted as a conceptual basis for the Rational Unified Process (Clements, Kazman, and Bass 2013). Table 14 illustrates the architectural views of the Kruchten model.

Table 14: The architecture views of Kruchten model

Views	Definition
Logical	It represents an object-oriented decomposition for the design, covering the functional requirement of the intended system.
Process	It represents the concurrency and synchronisation aspects of the design and some of the non-functional requirements.
Physical	It represents the mapping of the software into the hardware and reflects its distributed aspects
Development	It represents the statics organization of the software in its environment
Use Case	It represents an integration scenario of the element from the four previous views reflecting the process associated with a set of system requirements.

- ***Siemens model*** developed this vas developed in 1995 by Christine Hofmeister and some of her collogues at Siemens Research Centre (Soni, Nord, and Hofmeister 1995). They stated that the structures of an architecture description were grouped into four categories – conceptual, module, execution and code structures – each of which handles different stakeholder concerns. According to the authors, when these categories were separately handled, the implementation complexity was reduced and the reusability and reconfiguration of the software was enhanced. In order to separate these categories, the authors proposed that the architectural model should be expressed in four

views, each of which covered a different category of structures. In addition, the relationships between these four views had been defined. For example, the Modules in the module architecture were "assigned_to" runtime elements in the execution architecture, whereby each of the runtime elements was "implemented_by" specific modules in the module architecture. The Siemens four-view model is an architect-centered approach which ignores the explicit concerns of all stakeholders other than the software architect. However, the other stakeholders' concerns may be addressed implicitly by the views. This reflects the focus of the model being on the design and not on the documentation or communication purposes. Table 15 illustrates the different views of the Siemens model.

Table 15: The architecture views of the Siemens model.

View	Definition
Conceptual	It describes the system in terms of its architectural elements and the relation between them.
Module	It describes the system in terms of concrete structures decomposed into subsystems, modules and abstract program units.
Execution	It describes the dynamic structure of a system in terms of its runtime elements, threads and communication mechanisms and it also describes runtime topological design decisions.
Code	It describes the design-time layout of the system as a source code organised into language-level modules, directories, files and libraries.

- ***SEI Viewpoints (Views and Beyond/V & B)*** This is a set of viewpoints (viewtypes) developed by the Software Engineering Institute at Carnegie Mellon University for the documentation of software architecture (Clements et al. 2003). The set is comprised three viewtypes which are then specialised by a set of associated architectural styles within each viewtype. The approach is based on the principle that documentation of a software architecture begins with the documenting of the relevant views of this architecture. This is followed by the documenting of the information linking these views. The SEI viewtypes endeavour to address all the stakeholder concerns by providing software-architecture documentation that is decomposed into several views which address the needs of various stakeholders. Table 16 shows the architecture views of the SEI approach.

Table 16: The architecture views of the SEI

Viewtype	Definition
Module	It represents the principal implementation units, or modules, of a system, and the relations among them. There are several styles defined for the Module view type: Uses, Generalisation, Decomposition, and Layered.
Component and connector	It represents the set of interacting elements that have some runtime presence. There are some styles defined for this view type all of which relate to commonly occurring runtime system organisations: Pipe-and-Filter, Shared-Data, Publish-Subscribe, Client-Server, Peer-to-Peer,

	Communicating-Processes.
Allocation	It represents relationships between software elements and non-software elements. There are several styles defined for this view type: Deployment, Implementation and Work Assignment.

- ***Garlan and Anthony (Garland and Anthony 2003)*** propose a framework with the aim of defining a set of viewpoints in order to describe large-scale software architectures. They define a larger viewpoint set then the other models where each viewpoint has a narrow scope. This is an attempt to define specialised viewpoints, each of which is clearly focused, has a manageable size and plays an obvious role. Meanwhile, it has some drawbacks, such as the difficulty of managing the problem fragmentations in the architecture description and maintaining consistency among the views. The authors define (Garland and Anthony 2003) these viewpoints well, with purpose, applicability, stakeholder interest, models to use, modeling scalability and advice on creating the views. There are about 14 viewpoints which may be quite hard to use and document, especially for small and medium software architecture. Table 17 illustrates the architecture views of this approach.

Table 17: The architecture views of the Garlan and Anthony approach

Viewtype	Definition
Analysis Focused	It represents the way in which the elements of the system can work together in response to a functional usage scenario.
Analysis Interaction	It represents the interaction diagram used during problem analysis.
Analysis Overall	It combines the contents of the Analysis Focused view into a single model.
Component	It identifies the significant components of the system architecture and their connections
Component Interaction	It represents the way in which the components can interact in order to make the system work.
Component State	It illustrates the state(s) for a component or set of closely related components.
Context	It identifies the context within which the system exists, in terms of external actors and their interactions with the system.
Deployment	It illustrates how software components can be mapped to hardware entities in order to be executed.
Layered Subsystem	It presents the subsystems to be implemented and the layers in the software-design structure.
Logical Data	It shows the logical view of the significant architectural data structure.
Physical Data	It shows the physical view of the significant architectural data structure.
Process	It identifies the runtime concurrency structure.
Process State	It presents the state transition for the system's processes.
Subsystem Interface Dependency	It identifies the dependencies that exist between the subsystems and the interfaces of other subsystems.

- ***Rozanski and Woods*** proposed a set of viewpoints in their book (Rozanski and Woods 2005). They introduce a useful set of six viewpoints for documenting software architecture. The approach defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles and template models for constructing its views. These six viewpoints are: functional, information, concurrency, development, deployment and operational. The authors introduce the notion of architectural perspective to capture concerns that are relevant to many or all views, as "a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views". The perspectives comprise: accessibility, availability, evolution, internationalisation, performance and saleability, security and usability. In the second edition of their book in 2012, the seven viewpoints in the IEEE 42010 sense are introduced. These are illustrated in the Table 18.

Table 18: The architecture views of the Rozanski and Woods approach

Viewtype	Definition
Context:	It describes the relationships, dependencies and interactions between the system and its environment.
Functional	It describes the functional elements of the system during execution, as well as their responsibilities, interfaces and interactions.
Information	It describes how the system stores, handles, manages and distributes information and data.
Concurrency	It describes the concurrency structure of the system and assigns the functional elements to concurrency units in order to clarify the parts of the system that can be executed concurrently and to show where the concurrent operations might be.
Development	It describes the architectural aspects that support the development process.
Deployment	It describes the environment in which the system will be deployed, and the potential dependencies that the system may have on its run-time environment.
Operational	It describes how the system will be operated, managed and supported during its execution in the production environment.

- ***TOGAF*** The Open Group Architectural Framework was first developed in 1995 with the aim of specifying the process for developing (designing, evaluating and implementing) the enterprise architecture (Hornford 2011). It comprises two main parts: the Architecture Development Method (ADM) and the Architecture Content Framework (ACF). Through these parts, TOGAF provides guidelines, procedures and classifications for developing and using viewpoints and architectural views. It embraces the terminology of the ISO/IEC 42010:2007 standard for creating viewpoints and architecture views. TOGAF also defines architecture views for IT systems and supports decision-making and architecture principles for development and implementation. TOGAF's architecture views are described in Table 19.

Table 19 : The architecture views of TOGAF

Viewtype	Definition
Business Architecture	It focuses on addressing the concerns of users in order to fulfil a comprehensive understanding of the functional requirements.
Enterprise Security	It is concerned with the security aspects of the system.
Software Engineering	It is concerned with the development of new software systems.
System Engineering	It is concerned with assembling software and hardware components into a working system.
Communications Engineering	It is concerned with structuring communications and networking elements to simplify network planning and design.
Data Flow	It is concerned with the storage, retrieval, processing, archiving, and security of data.
Enterprise Manageability	It is concerned with the operation, administration and management of the system.
Acquirer	It is concerned with acquiring Commercial Off-The-Shelf (COTS) software and hardware.

- ***ISO/IEC/IEEE 42010-2011***[11] is the ISO standard Systems and Software Engineering Architecture description which is the latest edition of the original IEEE standard 1471:2000. It was developed by the IEEE Architecture Planning Group (APG) to formalise the software-architecture description and its main elements and also to provide a common standard for incorporating and embracing the best efforts in this area. ISO specifies architecture viewpoints, architecture frameworks and architecture description languages for use in architecture-descriptions. It defines the view as a work product representing the architecture of a system from the perspective of architecture-related concerns held by the system's stakeholders. It defines the viewpoint as a work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. The ISO does not restrict a particular set of viewpoints to be used, however; on the contrary, these should be a viewpoint for each view and each view should belong to a certain viewpoint which explains the conventions being used in that view. These architectural views are described in Table 20.

Table 20: The architecture views of ISO 42010:2011

Viewtype	Definition
Name	It records the name of the viewpoint and any synonyms or other common names for the viewpoint.
Overview	It is an abstract or brief overview of the viewpoint and its key features.

[11] http://cabibbo.dia.uniroma3.it/asw/altrui/iso-iec-ieee-42010-2011.pdf

Concerns and anti-concerns	It lists the architecture-related concerns to be framed by this viewpoint which can support decision on whether this viewpoint will be useful for a particular system of interest.
Typical stakeholders	It lists the system stakeholders expected to be users or audiences for the views prepared using this viewpoint.
Model kinds	It identifies each model kind specified by the viewpoint. For each type of model used, it describes the language, notation, or modeling techniques to be used. It may be documented in a number of ways, including: metamodel, templates, languages and operations.
Correspondence rules	They document any correspondence rules defined by this viewpoint or its model kinds. Usually, these rules will be "cross model" or "cross view" since constraints within a model kind will have been specified as part of the conventions of that model kind.
Operations on views	They define the methods to be applied to the views or to their models. Operations can be divided into categories: Creation methods, Interpretive methods, Analysis methods and Design or Implementation methods.
Examples	It provides examples for the reader.
Notes	It refers to any additional information that users of this viewpoint might need or find helpful.
Sources	It identifies the sources for this viewpoint, if any, including author, history, literature references and prior art.

- ***Common Architecture Viewpoint (CAV) Model (Vogel et al. 2011)*** This was proposed by Vogel et al. in 2011 with the aim of simplifying the handling of view models. This architecture-view model abstracts from the views of the architecture-view models and, thus, it handles and covers viewpoints that specify the names, the stakeholders and their concerns as well as the important artifacts for the architecture views used. Table 21 shows the architectural viewpoints of the common architecture-view model.

Table 21: The architecture views of the common architecture-view model

Viewtype	Definition
Requirements	It documents the architecture requirements.
Logical	It documents the architecture design.
Data	It documents aspects with regard to saving, manipulating, managing and distributing data.
Implementation	It documents the implementation structure and the implementation infrastructure.
Process	It documents the control and coordination of concurrent building blocks.
Deployment	It documents the physical deployment of software building blocks

- ***MoVAL***, The Model, View, and Abstraction Level-based software-architecture approach was proposed by A Kheir et al. in 2013 with the aim of defining a multi-level view methodology that guides the architect while developing his/her architecture (Kheir et al. 2014). MoVAL extends the IEEE 42010 standard and also complies with the multi-abstraction levels of the meta-object facilities (MOF). A view in MoVAL represents the system in accordance with a set of the development process's aspects and some problems associated with a specific category of stakeholders. This view is defined in multi-abstraction levels of two main types: achievement levels and description levels. The achievement level compiles a set of artifacts defining the architecture at a specific architecting phase, while the description level allows the architect to provide multiple descriptions with different granularity levels for each achievement level. Links form the architectural element which defines the relationships between views and abstraction levels as well as ensuring the consistency of the architecture.

Table 22: The architecture views of MoVAL.

Viewtype	Definition
Context	It describes the relationships, dependencies, and interactions between the system and its environment.
Functional Capabilities	It describes the functional capabilities of the system during runtime.
Functional Implementation	It describes the functional elements of the system during the execution time, as well as their responsibilities, interfaces and interactions.
Information	It describes the information the system needs to store
Logical Data	It describes the logical structure of the databases.
Physical Data	It describes the physical structure of the databases.
Deployment	It describes the environment in which the system will be deployed and the dependencies that the system may have on its elements

- The ***EPART E-Participation Architecture Framework*** was proposed by S. Scherer (Scherer 2016) in 2016 with the aim of filling the gap between the e-participation domain and the enterprise architecture-framework domain by exploring the methodical and technical guides that architecture frameworks can provide the better to design and implement successful e-participation. The framework combines an EPART-Metamodel with six EPARTViewpoints which frame the stakeholders' concerns: the Participation Scope, the Participant Viewpoint, the Participation Viewpoint, the Data and Information Viewpoint, the E-participation Viewpoint and the Implementation and Governance Viewpoint. Through the EPART-Method, the EPART can support the stakeholders in designing the E-Participation Enterprise (EE) and implementing e-participation and storing its output in an architecture description and a solution repository. The

EPART-Method is composed of five consecutive phases accompanied by requirements management: Initiation, Design, Implementation and Preparation, Participation and Evaluation.

Table 23: The architecture views of EPART

Viewtype	Definition
Participation Scope	It captures the objectives of e-participation and links them with the strategies to achieve them.
Implementation & Governance	It handles the operational management of the EE and the governance of architecture implementation. It addresses the concern of determining the managerial constraints of carrying out participation services and evaluating the outcomes.
Participant	It identifies, manages and engages the stakeholders actively and passively engaged in or affected by e-participation. It also supports the allocation of roles to stakeholders.
Participation	It links the objectives with the strategies to achieve them, through, the planning of the participation techniques, services, processes and activities necessary to carry out the participation undertaking.
Data & Information	It describes the data which performers produce or consume within e-participation.
E-participation	It represents the applications required to implement the participation architecture and to deploy and operate

We have presented a bibliographic synthesis on the application of view and viewpoint notions in some fields of computer science, such as requirements specification, systems modeling and especially software architecture. Even though each approach specifies its own semantics for the views, they share the basic idea of decomposing a complex artifact into multiple views and/or separating the stakeholder concerns.

6.5. Compression of multi-view architecture frameworks

Since our approach is an architecture oriented approach to model software-architecture evolution, this section present a comparative analysis between the architecture frameworks mentioned in the previous section. This study will compare these frameworks in accordance with some criteria relevant to the main characteristics of the architecture evolution process defined by MES and to the conception of multiple views. Table 24 presents some characteristics of certain frameworks with a similar purpose to MES.

Table 24: A comparative analysis between some multi-view frameworks of architecture approaches.

Frameworks	View realisation	Structural view	Functional view	Behavioural view	Role (Stakeholders)	Abstraction levels	Views' interaction
Zachman Framework	Concrete 6	Business view	Context view	Runtime view	Various stakeholders	_	_
"4+1" Model	Concrete 5	Development, Physical view	Logical view.	Process view.	Development team	_	_
Siemens model	Concrete 4	Conceptual view	Module view	Execution view	Development team	_	Assigned-to implemented-by located-in
SEI Viewpoints	Concrete 3	C&C viewtype	Module viewtype	Not available	various stakeholders	Level of Information view-packets	Limited interactions
Garlan and Anthony	Concrete 14	Component viewtype	Context viewtype	Component interaction viewtype	Various stakeholders	Layer of Subsystem	_
Rozanski and Woods	Concrete 7	Development viewtype	Context viewtype	Operational viewtype	Various stakeholders	_	_
TOGAF Framework	Concrete 8	Software & System engineering views	Business Architecture view	Not available	Various stakeholders	Granularity levels of architecture landscape	_
ISO/IEC/IEEE 42010-2011	Abstract metamodel	Can be instantiated	Can be instantiated	Can be instantiated	Various stakeholders	_	Correspondence concept
CAV Model	Concrete 6	Logical view	Not available	Process view	Various stakeholders	_	_
MoVAL	Abstract metamodel	Can be instantiated	Can be instantiated	Can be instantiated	Various stakeholders	Level of description	Inter and Intra Links
EPART	Concrete 6	E-participation	Implementation & Governance viewpoint	Participation viewpoint	Various stakeholders (Participants)	_	_

- **View realisation** answers how the concept of the view is realized, whether or not the view is a meta-concept element of a metamodel or a concrete element and whether there is a fixed number of predefined views or whether any view can be instantiated.
- The ***Structural view*** is the main aspect expressed by any architectural model. This criterion explains whether or not the framework is concerned with this aspect and which view covers this concern (and gives the name of the view covering the structural aspect).
- The ***Functional view***, it illustrates whether the framework defines a view that documents the system's functional elements. This view explains how the model/system is working and achieving the result for which the model exists.
- The ***Behavioral view*** shows whether the framework defines a view that clarifies how the model's elements are interacting (during the model execution-time) in order to achieve the goal (service or final product).
- The **Roles** illustrates whether the frameworks are designed for a specific group of stakeholders or for different categories of users.
- **Abstraction** levels illustrate whether the framework supports the view hierarchy, so the view can be represented in multiple levels of details.
- **Interaction** among views illustrates whether the framework explicitly defines a relationship (association) which links its views and maintains the consistency between them.

From the Table 24 we can see that some frameworks do not support all the criteria, or do not explicitly define views for certain aspects. It can be seen that all these frameworks support the structural view as no architecture oriented model can disregard this view. Meanwhile, the functional view is not explicitly defined by all the frameworks which the (re)-architecting activities/process did not cover. For example, Rozanski and Wood used an activity diagram to document the software Architecture Definition Process (ADP). The majority of studies consider the functional view for the system (Work Product) not for the process (task/architecting activity). IEEE 42010-2011 and MoVAL define a metamodel standard which provides a specification framework to develop the architecture description as an abstraction model, but they do not develop or clarify views (a concrete set). They merely provide a method of instantiating a view.

6.6. Multi-view & multi-abstraction evolution styles

It is almost impossible to capture all aspects of the design in a single description. Most design methodologies endeavour to develop an interlocking set of views, each of which is intended to cover one or more relevant aspects of the design (Boiten et al. 2000).

In general, there is no perfect solution because different solutions represent different views of the problem. Thus, we can state that, first, the evolution style model can represent one potential solution for a certain architecture-evolution context. Second, the evolution style approach represents a specific view and perception of the process. Finally, each evolution style can adopt a

certain development methodology that could meet its modeling requirements and, thereby, best depict its perception.

Here, with the multiple views, we aim to explore the main views of these issues which could be used in different approaches in order to meet their requirements. Ultimately, we aim to define a unified framework that could embrace this diversity.

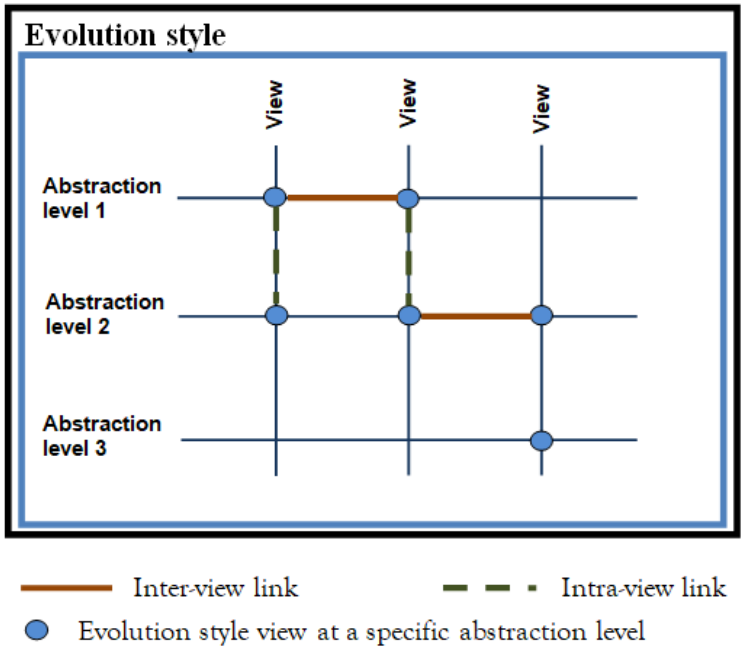

Figure 40: Multi-view and multi-abstraction evolution style

The result can serve as a road-map for the architects and developers in helping them to develop an evolution style, ranging from eliciting the process perception and the modeling process through to formulating their viewpoint to expanding the model into a set of views in accordance with the sets of stakeholder concerns (viewpoint). The result also uniformly describes this process of evolution-style development, which can be reused by different architects in order to develop their own evolution styles, each of which can express their perceptions with the appropriate views.

Our approach conforms to several standards (UML, IEEE-42010) and inherits from these standards the definitions of several basic concepts, such as the definitions of viewpoint, view and model. But it exploits and customises these standards to fit domain of evolution style. Figure 40 represents a matrix which clarifies the construction of a multi-view and multi-abstraction evolution style.

6.6.1 Evolution style (Model)

This refers to a modeling approach which aim to capture the main characteristics of a set of activities performed in a domain-specific software-architecture evolution and defines the vocabulary

of concepts necessary to model this process in accordance with a certain viewpoint (Hassan and Oussalah 2016).

Our approach is an architecture-centric approach as it is architecting (modeling) the process of software-architecture evolution. Thus, the main artifact (model) is an architecture model. In software architecture generally, the model can be considered as the most important component or element in a software-architecture description, the aim of which is to simplify communications between architect and stakeholders, as well as enable the stakeholder to understand and analyse the solution that the architect is trying to model. Moreover, the model can provide a common and precise means by which a set of relevant interlocking interests among different stakeholders can be formulated. It organises the evolution process via separating it into several tasks, identifying the (pre)requirements of each task and assigning it to the appropriate stakeholder.

In general, each group of stakeholders is concerned about certain aspects of the evolution process and the aim is that the model satisfies their concerns. To this end, the model must be a well-defined with a well-accepted formalism in the context of a certain viewpoint and abstraction level which details the process construction in accordance with this context. A set of models which represents the different viewpoints can be organised and evaluated to ensure consistency between different views.

MDA Methodology provides the fundamental concepts for developing different types of models and for how a generic model can be transferred into several customized models. It also provides the concepts for how Computation Independent Model (CIM) can be developed and then transferred into other models independent of the technology used (Platform Independent Model [PIM]). Finally, provides the specification for how to build from PIMs other specific models which fit a particular technology or platform (Platform Specific Model [PSM]) in order to implement the system in concrete manner. Therefore by applying the multi-view/multi-abstraction modeling methodology in MES, this can be transferred to whatever evolution style is compatible with MES.

6.6.2 Viewpoint and view

View refers to a facet of evolution styles from which a set of relevant aspects or certain stakeholders' concerns can be separately expressed. The viewpoint defines the concerns, concepts, semantics and methods relevant to its view.

Based on the fact that a software process and system can be analysed from different views, we can state that an evolution style consists of a set of distinct views, each of which reflects a certain relevant aspect. The most widely used views in the modeling methodologies are the structural view and the behavioural view. However, these do not preclude other choices, epically here in the architecture-centric methodology, and according to Bosch's definition, other views include choices of whether to consider the decisions' rationale viewpoints, or the physical aspect (topological view). Therefore, the suitable set of views to be established is not static; on the contrary, it is changeable and depends directly on the properties of the problem domain. Therefore, this work is

endeavouring to set up the way in which a multi-view model of evolution style can be instantiated by MES. The viewpoints of the evolution style can be defined by the architect (the evolution style's developer) in accordance with the list of associated stakeholders, the concerns or the requirements covered, and its formalisms.

In MES, a view is a first-class modeling entity for evolution style description. View can be consistent with one or more viewpoints in the sense that it uses the model conventions and correspondence rules defining by these viewpoints (for developing an instance of this view).

6.6.3 Abstraction levels

The abstraction level refers to the level of detail that the view provides. A view might be presented at several abstraction levels.

Despite the facilities offered by the multi-view technique, it is common for the development of a given view to be complicated and not easily implemented. Thus, in order to address this complexity in a gradual manner, it is essential to adopt a hierarchical approach that brings different understanding levels, or abstraction levels, into a view. Abstraction, in its broad sense is the process of removing detail to simplify and focus attention in order to identify the common core or essence of the model (Kramer 2007).

The multiple abstractions in MES aim to provide different detail levels, each of which provides different granularity in order to satisfy a certain group of stakeholders. These views are linked together in a certain manner **(Interaction relationship)** that maintains the consistency between them and facilitates their exploration.

Interaction refers to link that can express relations between different abstractions which can be classified in two categories as follows:

Inter-view link defines the relation between two distinct abstractions which belong to two different views.

Intra-view link defines the relationship between two distinct abstraction levels which belong to the same view.

6.7. Multi-view and multi-abstraction-level evolution meta-style

MES in (Hassan and Oussalah 2016) was defined as a Meta^2 model for the domain of architecture evolution. It is a component-based framework for architecture-evolution process description. The approach is compatible with the four modeling levels of the OMG, whereby, each evolution model is an instance of the model in the level above (its meta), including the meta-style MES which is an instance of itself.

An evolution style is an instance which is customised to fit a specific need of certain stakeholders. MES is an architectural-based modeling methodology which is in accordance with the software

architecture definitions, particularly that of Perry and Wolf which defines software architecture as a 3-tuple consisting of Elements, Form and Rationale (Perry and Wolf 1992). Since that time, three paradigms for describing an architecture model been emerged, each of which aims to represent different aspects of the systems, including structure, behavior and design decisions. The first paradigm represents the architecture in terms of the system-structure aspect whereby several approaches and modeling tools (ADLs) have been proposed that primarily focus on the capture of architectural structure. These include Aesop, Wright and UniCon. The subsequent paradigm represents the dynamic behaviors of the system elements whereby formal behavioural modeling methods have been used, including the process algebra, Petri nets and Z language (Bradbury et al. 2004). Darwin, Dynamic Wright and MARTE are tools that model the architecture behaviour. The third paradigm represents the software architecture in terms of design decisions, whereby different approaches are proposed to represent the architecture as a set of architecture-design processes (Capilla et al. 2016). In this vein, three prevailing viewpoints have been used to present the software-architecture model descriptions and several views (diagrams) have been developed in accordance with these viewpoints: structural views, behavioral views, and decision-rationale views.

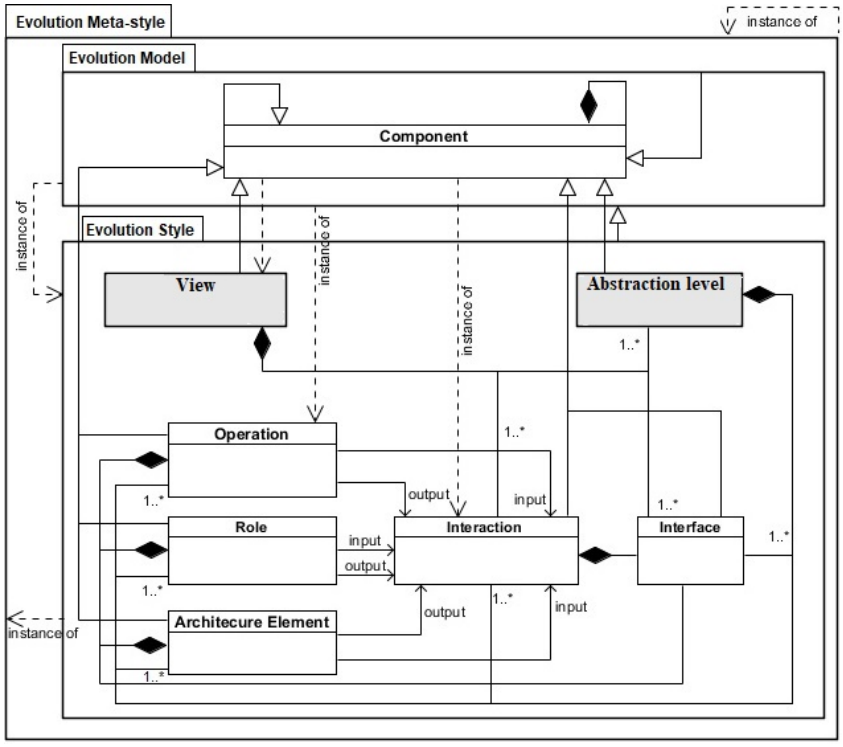

Figure 41: MES+

An evolution style model is an artifact that aims to capture some or all of the architectural views about the architecture-evolution process. These three views can be detailed at different levels (abstractions) to fulfil the modeling process paradigm for the intended view (artifact). MES can be refined to embrace the concepts of multi-view and multi-abstraction-level methodologies. Thus, the aim of this section is not to define a new meta-style, but to annotate MES with concepts of this methodology that are represented in the previous section.

Figure 41 shows MES-plus which is an extension of MES annotated with the concept necessary to satisfy the requirement of defining the multi-view modeling methodology in the domain of architecture evolution (the grey boxes refer to MES elements; the white boxes refer to proposed additional elements).

However, any evolution-style model can be expressed by using instances of the main concepts of MES. This style must have at least one view that covers one or more relevant concerns about the process. The view can also be expressed at different granularities. Therefore, an evolution style tool should come with one or more diagrams, each of which can be used to edit a view's abstraction of the evolution style.

6.8. Mapping MES+ to standard

In this section we aim to compare/map the conception of multiple views in MES+ with IEEE-42010 standard and to look at how these concepts can be represented as UML ModelElements. As mentioned above, IEEE-42010 is an updated version of IEEE-1471 standard, which proposes to formalise the software-architecture description and to provide a generic framework that will specify the system architecture in multiple views.

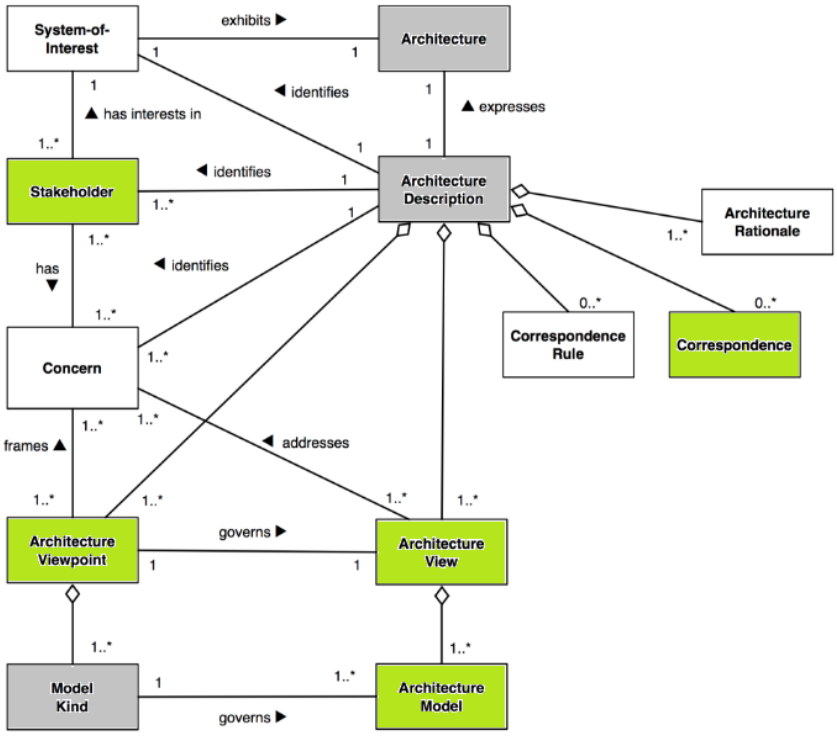

Figure 42: IEEE-42010 standard model

This standard was inspired by several multi-view modeling approaches, both industrial and academic such as TOGAF, the 4+1 view model and Zachman's framework. ISO defines a set of

concepts (a vocabulary) which can be used in the creation of an architecture description of systems and software. Thus, the objective of this comparison is to explore the extent to which extend MES+ follows this standard and to identify the differences between them in terms of the concept specifications.

Table 25 presents the corresponding concepts. In the ISO specification these concepts are defined as follows:

- ***Architecture Description*** (AD) is a work product used to express the system architecture.
- ***Stakeholders*** are individuals, groups or organisations holding concerns about the system.
- An ***Architecture Viewpoint*** is a set of conventions for constructing, interpreting, using and analysing one type of architecture view. A viewpoint includes model kinds, viewpoint languages and notations, modeling methods and analytic techniques to frame a specific set of concerns.
- An ***Architecture View*** in an AD expresses the architecture of the system from the perspective of one or more stakeholders to address specific concerns, using the conventions established by its viewpoint.
- A view is comprised of ***Architecture Models***. Each model is constructed in accordance with the conventions established by its Model Kind, typically defined as part of its governing viewpoint. Models provide a means of sharing details between views and for using multiple notations within a view.
- A **Model kind** defines the conventions for one type of architecture model.
- ***Correspondences*** express a relationship between AD Elements.

Table 25: Corresponding elements between MES+ and IEEE-42010

IEEE-42010	MES+
System Architecture	*Evolution Architecture*
Architecture Description	*Process Description*
Stakeholder	Role
Architecture viewpoint	Evolution Style
Architecture View	Evolution Style Model
Architecture Model	Evolution Style View
Model Kind	*View Kind*
Correspondence	Interaction

From the specifications of MES+ and IEEE-42010, we can clarify these differences:

- As any system has an architecture, so the process also has an architecture.
- Process description is the source used to describe an evolution style (software architecture evolution).
- The stakeholder is the Role in MES+ of the person representing the software architect.
- The evolution style clarifies the concepts, methods, tools and techniques used to model the architecture evolution in accordance with the perspective of the architect.
- The evolution-style model expresses the process in accordance with the associated evolution style. This model can be expressed in different views, each of which covers one or more relevant concerns.
- The evolution-style view expresses the model in accordance with a certain View kind (from a certain aspect). The View consists of one or more abstraction levels.
- The View Kind is an abstract/conceptual element of MES, which represents the abstract view and which may be instantiated as structural, behavioural, or informational, or as another view.
- The interaction expresses the relationship between views, which may be inter-link or intra-link relationships.

6.8.1 UML concepts for MES+

The UML is the de facto standard for modeling a software system, which is defined and endorsed by the OMG. The UML specification provides a mechanism to define domain-specific languages by using the UML concepts as a basis on which to represent these languages via UML profiles. This mechanism facilitates the designer in developing tools by adopting a UML metamodel to fit the modeling requirement instead of developing a tool from scratch. Several domain-specific languages (UML profiles) have been developed for different domains.
Section 4.6.1 presents a mapping scenario between MES and MOF and this is the basis on which the mapping between MES+ and UML is conduction. With this mapping, MES+ can be defined as a UML profile, whereby each element in MES+ can be defined by extending (stereotyping) its corresponding UML ModelElement (super-class or meta-class). Table 26 represents the corresponding elements in MES and UML.

Table 26: Mapping MES+ - UML

MES Profile	UML
Style	Package
Model	Namespace
Component	Model Element
View	(sup-)Package
Abstraction (Sub-view)	(sup-)Package
Operation	Behavioural Classifier
Role	Behavioural Classifier

6.9. Conclusion

The primary aim of the evolution style approach is to provide an architecture-evolution model that can support the evolution team in understanding, planning and conducting this complex process. In the evolution process, multiple groups of stakeholders are involved, each of which seeks a view that covers his own concerns and satisfies his needs. This view is constructed in accordance with his viewpoint which defines the modeling concepts, the notation and the metaphor to express this view. Describing the architecture evolution process from multiple perspectives can provide greater simplicity in analysing, understanding and reusing the architectural-evolution knowledge represented in a process model. This chapter (6) introduces an extension of MES, the aim of which is to extend the conception of multi-view into software architecture-evolution modeling. To this end, MES+ was introduced with the notions of multi-view and multi-abstraction-level in software-architecture evolution modeling.

Chapter 7. Development of multi-view modeling tool for SAE

7.1. Introduction

In the previous chapter, we presented an extended version of MES, the aim of which is to integrate the multi-view modeling methodology into the domain of evolution style. In this chapter, we aim to investigate how a multi-view evolution style modeling method can be implemented. In order to validate and assess the technical feasibility of our approach, we develop an initial running prototype of a multi-view modeling tool for software-architecture evolution that is compatible with the conception of MES+. The intended tool is aimed at providing graphical editor for different diagram types, which can edit several views of an evolution style. With this implementation, we aim to bridge the cognitive and technical gap between the conceptual views and the diagrammatic views.

The tool provides different architectural views, the aim of which is to support the capturing of architectural-evolution decisions. Software-architecture evolution practices and knowledge in a particular domain can be represented as a multi-view model which expresses the potential ways of evolving this domain. Exploring these ways (paths) from different points of view can support the analysis and they are then compared in order to select the optimal path that best fits the current context. Different evolution style tools have been developed, as presented in the related works (Section 3.3). Here, however, we attempt to explore a different but complementary tool: rather than developing a new modeling tool, we aim to develop a multi-view editor that can visualise the architecture evolution process model from different perspectives. This prototype can serve as a road-map for the architects (and modelers) in helping them to develop a multi-view evolution style for their interests.

7.2. Motivation for new tool

Several general-purpose modeling languages (tools) exist such as UML, BPM and SysML, which can be used to express a process model. Some of the necessary concepts for modeling software-architecture evolution (in accordance with our approach) are missed in these languages. Hence, these general-purpose diagrams do not prevent us from developing a domain-specific modeling tool geared towards architecture evolution. The intended tool aims to provide graphical editors for different diagram types which can edit multiple views of an evolution style (model).

This section addresses the reasons why we developed a new tool instead of extending an existing tool and it can be summarised in the following points:

- There is a need for a tool supporting the meta-concepts of MES, which can express different approaches;

- Some tools are already extended tools which build on top of other (purpose) architecture tools, for example ATRIUM (Cuesta evolution style) which is developed for requirement traceability on the architectural level;
- There is need to develop an open-source tool that can be extended to embrace different viewpoints in modeling software-architecture evolution.

7.3. Domain-specific (Modeling) Languages DSLs

Domain-Specific Languages (DSLs) are programming or modeling languages dedicated to a particular domain. A generic approach usually presents a methodology, the aim of which is to provide solutions for many problems in a certain area, whereas a specific approach is closer to a certain domain and aims to provide easier solutions for set problems in this domain. In this vein, the General-Purpose Programming Languages, such as C, Java and Python, and general-purpose modeling languages, such as UML, are used to provide a solution for any domain. Numerous DSLs have been introduced for different domains such as software engineering, systems software, multimedia, telecommunication, and so on. Among the well-known languages are SQL, Java script, CSS and HTML (Van Deursen, Klint, and Visser 2000).

The advantages that can be gained from the use of DSLs are many. The most obvious are increases in productivity, maintainability, re-usability and quality, along with enhance communication and knowledge-share between development teams.

Domain-Specific Modeling Languages (DSMLs) are high-level languages specific to particular domain or set of problems. They provide programmers with a means to implement the solution domain at the abstraction level which enables them to develop programs quickly and effectively.

Several factors have strengthened DSMLs and among them is the emergence of MDD and MDA, both of which strongly rely on the model as a mean artifact in the software-development process in order to generate the final application automatically. To this end, much research has been conducted, all of which is concerned with methods, techniques, frameworks, workbenches, and Meta languages that support the DSML (Lara, Guerra, and Cuadrado 2015).

The software-engineering discipline has seen several modeling languages developed for different purposes, such as object modeling, architecture modeling and process modeling. Different tools have been developed to support these languages, whereby models can be edited and manipulated.

In the early days, the development of these domain-specific modeling tools (graphical tool) was hard work which requiring a lot of attention being paid to developing the editor and parser (interpreter or compiler). Meanwhile, the strong interest in model-driven engineering has yielded several technologies (IDE) that support the development of DSML tools easily and quickly.

The following section briefly presents some of the common frameworks (IDEs, workbenches) for developing graphical tools for domain-specific modeling languages.

7.3.1 Frameworks for Implementing DSLs

In this section we endeavours to provide a brief overview of some common workbenches that are appropriate for developing the intended tool. Each of these frameworks has been widely used to create several useful tools in different application domains.

Eclipse is arguably one of the most popular integrated development environments (IDEs). Several workbenches (based on the EMF metamodel) have been developed to create homogeneous graphical editors for domain modeling. The Eclipse Modeling Framework EMF (Baetens 2011) provides an object graph for representing models, as well as capabilities for serialising models in a number of different formats, checking constraints, and generating tree-based editors which can be directly executed in the Eclipse runtime-workbench. It relies on the Graphical Editor Framework (GEF) and Draw2D to provide the foundations (Model View Controller) for building graphical views, based on which the graphical editor is realised by hand-coding.

Graphical Modeling Framework (GMF) (Baetens 2011) by bridging between EMF and GEF, GMF provides a workbench whereby a diagram definition is linked to a domain model (EMF model) which serves as input to the generation of a graphical editor. In fact, developing an editor in GMF is often complex and depend highly on Java and Eclipse plug-in knowledge.

Graphiti (Liebenberg, Roßmaier, and Lakemeyer 2017) is an Eclipse-based graphics framework that bridges EMF and GEF and hides the complexities of GEF behind a lightweight and easy-t-use API. This allows us greater ease to speed up the development of homogeneous graphical editors. Thus, the developer does not have to deal with the complications of GEF and Draw2D.

Sirius (Liebenberg, Roßmaier, and Lakemeyer 2017) is an Eclipse project created by Thales and Obeo. Built on EMF and GMF, it creates a graphical modeling workbench and provides a generic workbench for model-based architecture engineering, which could be easily tailored to fit specific needs. Sirius simplifies the product, reduces the design time and rapidly increases the overall productivity. Based on a viewpoint approach, Sirius makes it possible to equip teams who have to deal with complex architectures on specific domains (Liebenberg, Roßmaier, and Lakemeyer 2017). On the Sirius online site[12], can be found several-modeling tools which created with Sirius platform for domains such as: Systems Engineering, Software Development, Business Configuration etc.

MetaEidt (Baetens 2011) is a metamodeling platform that allows the creation of visual domain-specific languages. It is based on the GOPPRR (graphical) meta-modeling language which is an acronym formed from *Graph*, *Object*, *Port*, *Property*, *Relationship* and *Role*. It provides a meta-modeling language and customised CASE tool functionality (i.e. diagramming editors, browsers, generators) for modeling, allowing designs to be edited as graphical diagrams, matrices or tables, switching between views according to user needs. The user can browse designs with filters and link models to other designs. The results of the modeling can be published as web pages or as a Word document.

[12] http:// www.eclipse.org/sirius/gallery.html

ADOxx *(Sinz and Bork)* is a metamodeling platform for developing modeling tools for domain specific languages which was been developed by BOC Group. It is an extensible, repository-based metamodeling platform which offers a generic tool with the ADOxx basic components for the representation and editing of diagrams, as well as analysis, simulation and comparison of models. These basic functions can be reused and customised by mapping the metamodel of the DSL onto the meta-meta-model of ADOxx. In other words, the meta-model of the DSL has to be specified using the concepts provided by the ADOxx meta-meta-model. In this methodology the ADOxx platform allows the DSL designer to design and implement a modeling tool that is both powerful and flexible. For example, Bork and Sinz, in *(Sinz and Bork)*, presented the design of a multi-view modeling tool on the ADOxx meta-modeling Platform for Semantic Object Model (SOM) business-process modeling.

Microsoft Visual Studio's Modeling and Visualization MSDK *(Cook et al. 2007)* is a software development kit that is integrated into the Microsoft visual studio (IDE). In the 2005's the visual studio version, the SDK was known as a DSL tool. It can be used to create powerful domain-specific tools that we can integrate into the Microsoft visual studio. The core of MSDK is the model definition (meta-model) of the DSL, which describes the domain concepts used to define the model. The IDE provides the model with a variety of tools, such as a graphical editor, a textual editor, a code generator, APIs etc. Several conceptions and patterns used in VMSDK stimulate those used in EMF, but with Microsoft technologies (for example, *T4* Text Templates can be used for code generation vs. Acceleo or Xpand). Figure 43 represents an example of a meta-model of the evolution-style representation in Visual studio 2015. Figure 44 illustrates the implementation model defined in accordance with this metamodel.

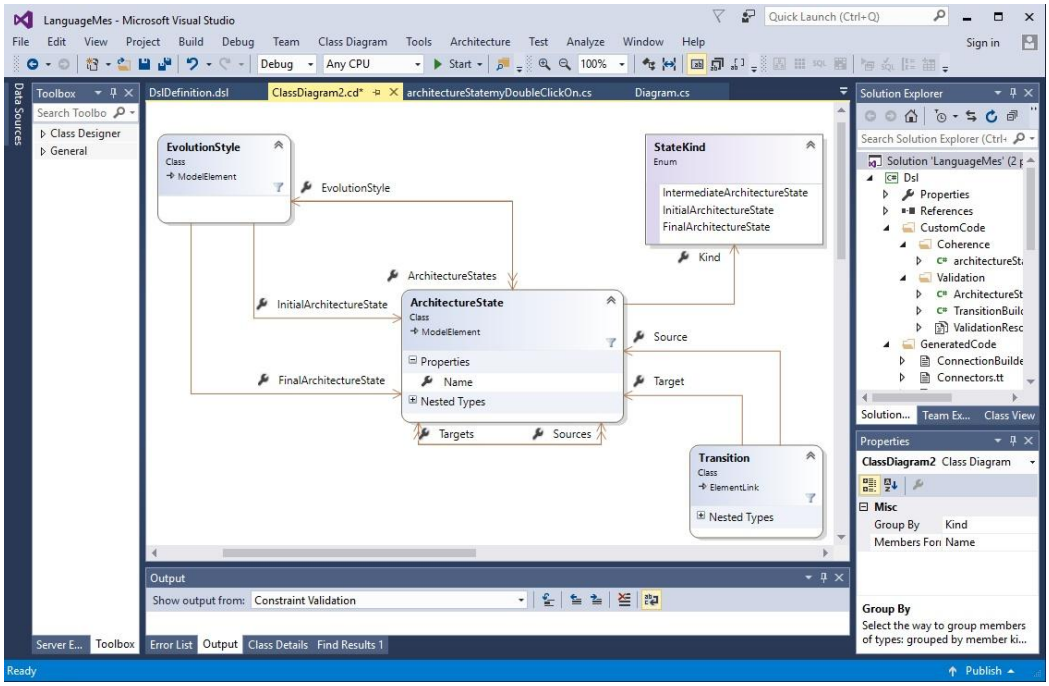

Figure 43: An evolution-style meta-model in MSDK

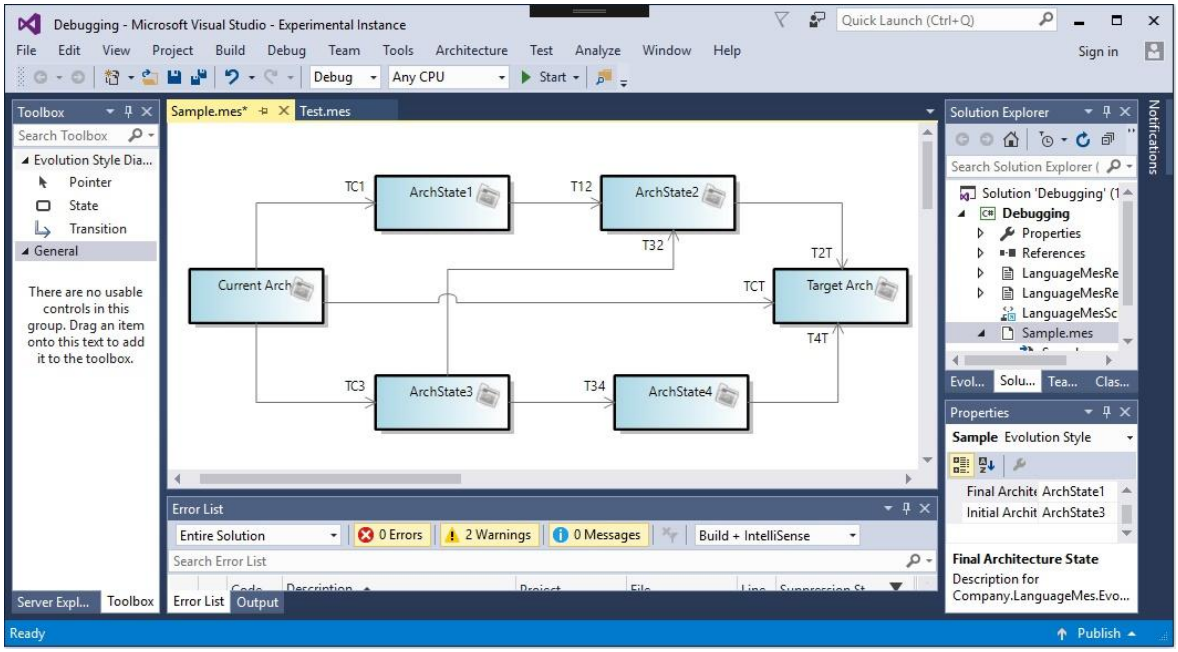

Figure 44: An evolution-style editor implemented on MSDK

7.4. Multi-view Evolution-Style Model

Our modeling approach is an architecture-centric framework, the aim of which is to capture the software-architecture evolution in an abstraction model which can express the process in different architectural views each of which provides the necessary information relates to a certain viewpoint.

As the software architecture refers to the high-level structure of a software system, thus, the **structural view** (structural aspects of the system) is the main viewpoint that an architectural model expresses. To this end, the evolution style's structural view aims to express the construction (topology) of the components element of the process and the way in which the whole process is constituted. Different abstraction-levels of the structural views can be extracted by consideration of a certain component (element) of the process (e.g. architecture component, operation component etc). This kind of extraction represents an intra-view relationship that can link between the different abstraction-levels of a view.

We chose the path diagram to represent the (main) structural view of the evolution style in our tool. The path diagram can provide an overall topology of the evolution-style model which represents the different ways of evolving a system. Each path can be represented as a sequence of architectural transitions between the current architecture of the system and the target architecture. A number of intermediate architectural states may be produced whereby every two successive states are linked by a transition. A transition is composed one, or a set, of evolution operations. Actually, the structural view comprises the different elements (notations) of the evolution style and, thus, other views or different abstraction levels can be derived (logically linked). For example, from the transition (click on the edge), we can get a behavioural view such as, activity diagram or a decision

view which provides the rationale behind this choice (evolution style path) and the possible alternatives. Figure 45 show a path diagram comprising five evolution paths which represent the different ways to of evolving the initial state (current architecture) to the final state (target architecture).

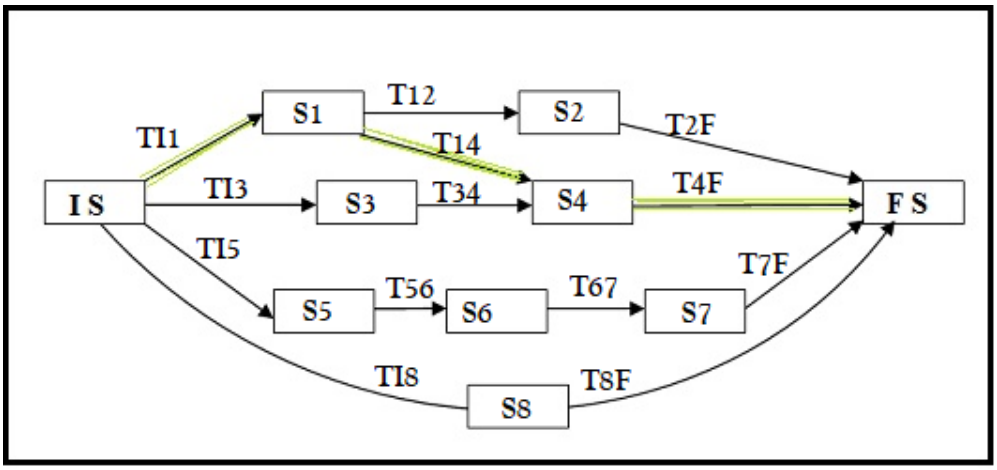

Figure 45: Evolution-path view

Other, more detailed, structural views which the intended prototype tool will represent are a component-and-connector view and a deployment view (***component diagram*** and ***deployment diagram***).

Decision-detail view: The architecture of a software system is the result of architectural decisions, which may concern various issues, such as, architectural styles, patterns, COTS components, programming languages, or infrastructure. To come to an architecture design (artifact), such a decision needs to be made. In fact, architecture evolution is about making new design decisions, or removing obsolete ones, in order to satisfy new requirements. Therefore, whenever the system needs to be changed, it is important, if not essential, to understand those decisions before (new) architecture evolution decisions are made. To this end, Kruchten et al. (Kruchten, Lago, and van Vliet 2006), in their paper *Building Up and Reasoning About Architectural Knowledge*, define architecture knowledge as follows "Architectural Knowledge = Design Decisions + Design". Consequently, the architectural design decision is the central concept in the evolution of software architectures. In this vein, architectural evolution decisions and their rationale, which are (tacit architectural knowledge) not explicitly expressed in the architecture design (artifact/ADL), needs to be documented, the better to understand the re-architecting process.

The nodes (architectural states) on a certain evolution path (in the path diagram) represent architectural milestones through which the current system's architecture must pass in order to get into the desired target architecture. These nodes are linked with an ADL editor, whereby the architectural designs can be displayed.

The nodes may have different outgoing edges, each of which represents one possible alternative evolution step towards the target architecture. Each edge is composed of a particular set of evolution operations, which introduces a new architectural instance (next node) and satisfies a particular context. Thus, an architectural decision must be made on each edge to explain which edge (step, alternative) should be taken. Maintaining the architectural decisions, their rationales and the possible alternatives for each choice can support the understanding of the evolution process. Thus, ***Decision-Detail View*** provides a means for the architects to maintain each decision and convey the idea behind each step in the evolution style.

Decision Template	
Name	A short name for the decision that can serve as an access key from other views.
Group	Decisions can be classified into groups, each of which shares specific characteristics. You can use a simple taxonomy, or a more sophisticated architecture-ontology grouping (e.g. Kruchten's ontology of architectural design-decisions (Kruchten 2004)).
Issue	Describe the architectural-design issue you are addressing, leaving no questions about why this issue is being addressed now and why this decision has been selected among one or more alternatives.
Decision	The outcome of the decision. Clearly state the solution you have selected.
Alternatives	List the set of potential alternative solutions considered when making the decision.
Argument	Outline the reasons behind selecting a solution, including properties such as estimated cost and time, and the availability of required resources (economic and technical feasibility).
Related decision	
Related architecture	State the source state (incoming architecture design) and the destination state (target architecture design) and list the architecture elements affected by implementing this step.
Notes	Record the notes and issues that accompanied the decision-making process.
Add-ons	List (hyper-link) the related artifact, design, or documents which can be used to convey the idea/rationale behind a decision to the stakeholders. (e.g. Gantt chart, a published process of EPF composer).

Figure 46: An architecture-evolution decision view

Architecture-decision documentation has attracted wide attention recently, in which different frameworks, models and tools for capturing and managing architecture decision have been proposed (Tang et al. 2010). There are three prevailing approaches to documenting architecture decisions: decision templates, decision models and annotations (van Heesch, Avgeriou, and Hilliard 2012). Figure 46 shows a template/model of an architectural-decision view, the form of which the prototype tool will adopt to express the architectural-evolution knowledge. This template is based on that defined by Tyree and Akerman in their article "Architecture Decisions: Demystifying Architecture" (Tyree and Akerman 2005).

7.5. Prototype implementation on the ADOxx platform

In this section, the conceptual design of a multi-view modeling methodology (MES+) for software architecture evolution is used to develop a corresponding modeling tool. The MES+ tool has been realized using the ADOxx[1] Meta modeling Platform.

The choice of using the ADOxx platform was motivated by two aspect concerns: first, the functionalities provided by the metamodeling platform (User, Library, Model, Attribute Profile and Component Management) which facilitate the management of developing the modeling method; second, the practical realisation of the modeling tools provided by the (modeling toolkit) which allow the user to test and debug the implementation. The modeling toolkit provides several components and modules to analyse, simulate and evaluate the implanted model. Besides other criteria, the choice of ADOxx as the website provides a large content of open-resource libraries (e.g. UML, BPMN) and code-repositories which can support and guide modelers in developing their methods. Furthermore, using ADOxx will not require advanced knowledge of programming languages such Java or C# where the modeler will use a scripting language (ADOxx Library Language (ALL)) as the implementation language.

7.5.1 ADOxx Metamodeling Platform

ADOxx[1] is a Metamodeling development and configuration platform, the aim of which is to facilitate the design and implementation of graphical modeling tools for domain-specific languages. It was developed by the BOC AG[2], a spin-off of the University of Vienna. The BOC was founded by Professor Dr Dimitris Karagiannis and their first product was the metamodeling ADONIS which was primarily used for Business Process Management and was evolved to result in the ADOxx platform. Several tools have been developed by the ADOxx platform in areas such as conceptual modeling, knowledge management, e-learning and many others. Moreover, some open source libraries (e.g. UML, BPMN) and code-repositories have been developed which can help and guide developers in designing their own tool.

The ADOxx is based on a meta-metamodel which can be customized (instantiated) in order to fit the specific needs of a modeling method. The ADOxx platform is built on an architectural

hierarchy with four modeling levels: original, model, metamodel and meta-metamodel, which are promoted by the OMG and Eclipse society.

Figure 47 shows the modeling hierarchy for the roles and languages which describes the classification of the meta-models's users (meta-modelers) in the ADOxx platform. An MM-tool developer can use a metamodel to instantiate his domain-specific meta-model which can be stored in a platform-specific language (ALL).

An MM-tool user can then create models in accordance with this domain-specific metamodel and the platform then stores these models in ADL. The ADOxx metamodel enables automatic generation of a modeling toolkit with built-in functionality that allows models to be created and modified according to the implemented metamodel, by using the Model.

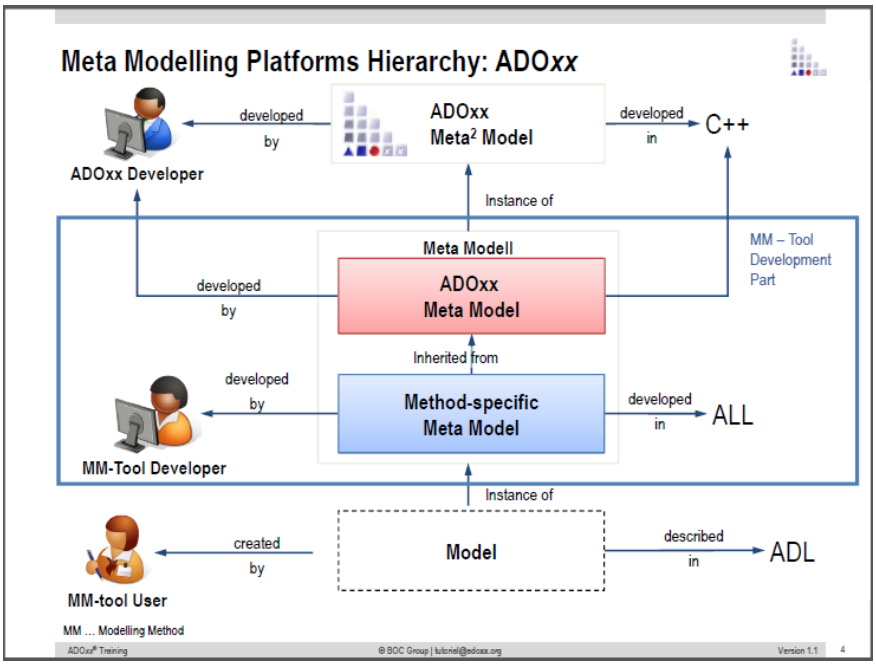

Figure 47: Roles and languages in the modeling hierarchy of ADOxx[13]

The ADOxx platform consists of two toolkits which are the ADOxx Development Toolkit and the ADOxx Modeling Toolkit. The ADOxx Development Toolkit (Administration Workspace) supports the user in developing his own metamodel or DSL. It includes the administration of ADOxx users and user groups, ADOxx libraries and ADOxx models and model groups. It also supports the Import/Export of elements (e.g. application libraries, repositories, models, objects, user).

The ADOxx Modeling Toolkit (Modeling Workspace) allows the user to create, test and debug the models in accordance with the DSL (metamodel) that is defined in the Development Toolkit. The

[13] https://www.adoxx.org

Modeling Toolkit provides a large number of editing facilities, which allows for the analyzing, simulating and evaluating of the models that have been created.

7.5.2 The MES+ multi-view/abstraction modeling tool development

The starting-point for the development of a modeling tool based on the ADOxx platform is the specification of the (user) domain-specific metamodel. The ADOxx platform provides a meta-meta-model, defining some generic modeling classes and relations as well as corresponding attributes and constraints (the most important classes are *Modeling Class, Relation Class and Model Type*). The user metamodel can be specified by relating (mapping) the concepts of DSL to the concepts of this meta-meta-model. In other words, the user metamodel has to be specified as an instance of the ADOxx meta-meta-model.

The realisation of the MES+ views

Section 7.4 provides a propositional design of the MES+ evolution-style tool with two views and two abstraction levels: the process context view, the decision view component-connector abstraction, and the deployment abstraction. The structural view (context view) presents the main view of the evolution style, from which the other views or abstraction-levels can be derived. The ADOxx meta-metamodel defines a ModelType element which comprises a set of Modeling Classes and Relation Classes, hereby realising the modeling language provided within an ADOxx model. The Modeling Class is the modeling element which can be used to represent any node of DSML. Thus, we can use it for the architecture state in path diagram (or for component in component-connector or deployment abstraction-levels). Relation Class represents edge between the nodes, thus, it can be used for the architecture transition in the path view. Thus, the first step in developing an evolution style's multi-view tool is to map the concepts of evolution-style views to the concepts of the ADOxx metamodel and Table 27 shows this mapping.

Table 27: Mapping of the MES concepts to the ADOxx concepts

ADOxx concepts	MES+ concepts	
Modeling Class	Architecture element, role	architecture state, component, node, interface, port, artifact, note, deployment specification, instance specification
Relation Class	Operation, interaction	architecture transition, connector, generalisation, realisation, dependency, association, anchor, requires interface, composition, aggregation

Figure 48 shows the realisation of the evolution style concepts within an ADOxx.

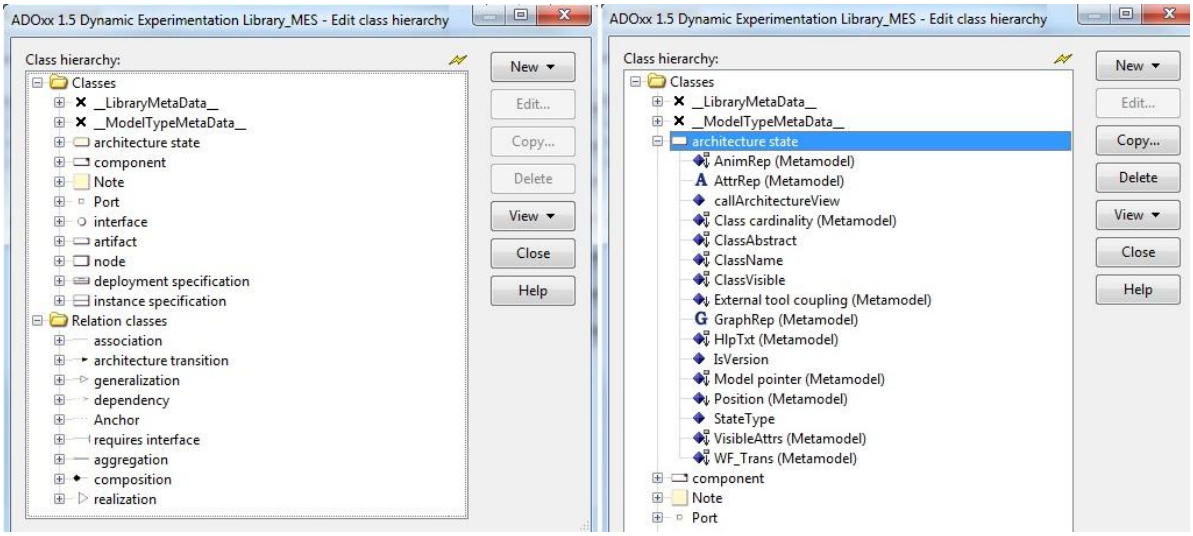

Figure 48: Modeling and Relation Classes definition in ADOxx

The next step, after the mapping between the metamodels has been conducted, is to specify the modeltypes so as to group the modeling and relation classes. Each group (modeltype) will be used to create a view (or an abstraction-level) of an evolution style. Table 28 shows the modeltypes and the evolution-style modeling elements used by them.

Table 28: Mapping of the evolution style modeling elements to the ADOxx modeltypes

ADOxx ModelType	Evolution style Modeling Classes and Relation Classes
Path view	architecture state
	architecture transition
Component-Connector abstraction-level	component, interface, port, artifact, note
	connector, association, generalisation, dependency, realisation, anchor, requires interface, composition, aggregation
Deployment abstraction-level	component, node, interface, port, artifact, note, deployment specification, instance specification
	association, generalisation, dependency, realisation, anchor, requires interface, composition, aggregation

Figure 49 illustrates the realisation of the evolution style's view and abstractions as ADOxx ModelTypes. The specification follows this syntax: first the keyword "MODELTYPE" refers to the start of a new modeltype (view). Subsequently, all its modeling classes and relation classes are added using the keyword "INCL" followed by the class name.

The decision is missed in Table 28 because it has been implemented as ADOxx *Notebook*. The ADOxx platform provides a specific syntax for describing Notebooks, whereby the attribute AttrRep of the modeling classes and relation classes can be used to represent selected attributes. The modeler can select which attributes should be visible, using a set of predefined representation types such as check-boxes, radio buttons and lists.

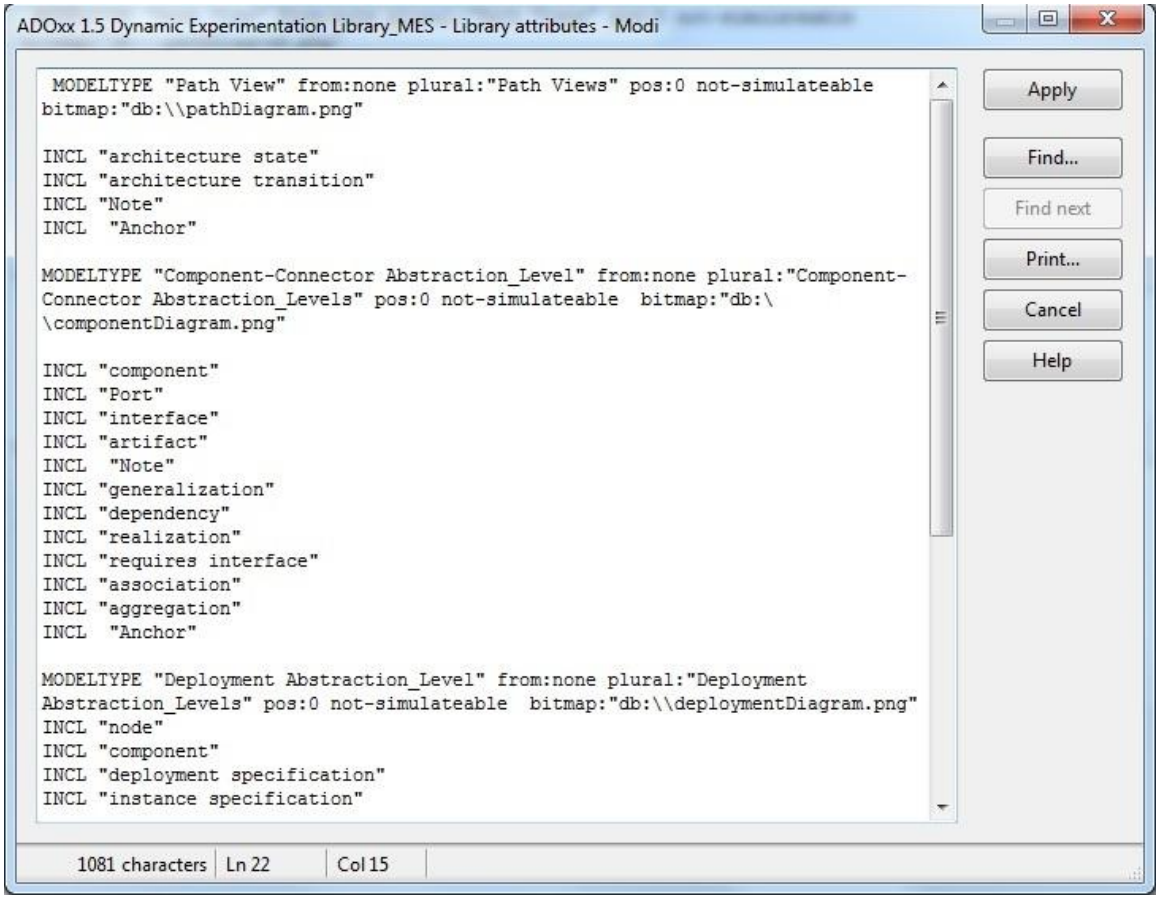

Figure 49: Realisation of evolution style's views

Figure 50 illustrates in the first part [1], the declaration of the attribute AttrRep (an abbreviation for attribute representation) in the relation class "architecture transition", the second part [2] shows the definition of the attribute which starts with the keyword "NOTEBOOK" and "CHAPTER", which are used to define the decision view, followed by all the attributes that should be included. Part [3] presents the visualisation of the notebook of the architecture transition which will be edited by clicking on the architecture transition in the path view.

Realisation of the evolution style modeling process

MES+ embraces the notion of the multiple views in modeling of software-architecture evolution in order to provide a comprehensive model. This model aims to satisfy the needs of the different stakeholders involved in this process. The types, or set of views, which may be needed to cover all the aspects or concerns are dependent on the set of stakeholders and on the system under evolution. Thus, in this prototype tool, we attempt to illustrate how a certain view, or a more detailed level, can be integrated with the path view (main view).

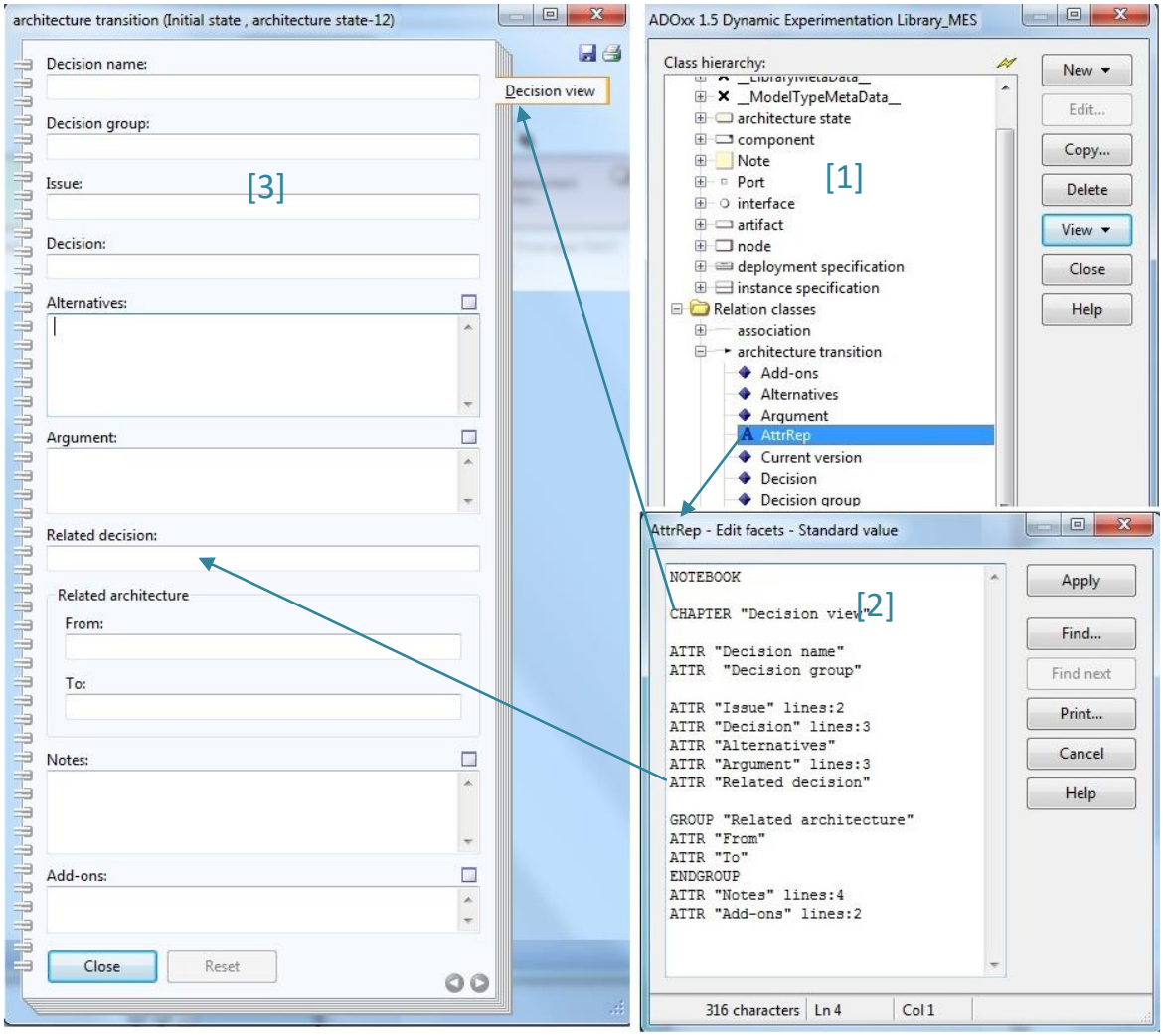

Figure 50: AttrRep definition

Figure 51 shows a screenshot of the prototype tool which visualises an evolution-style model. On the far left is the Model Groups Explorer which visualises all views within the evolution style and, as we can see, there are different view types. Three diagrams (views) are edited: the path view, the component-connector view and the deployment view. Each node in the path diagram represents a state which the system architecture may constitute on its way to the intended design. Thus, it can be linked with another view to provide more details about this node. This can be done by double-clicking on the node where a description dialogue (notebook) will be opened to link the related view. This associated view will appear as a link in the node to recall this view. This relationship between the evolution-path diagram and component, or the deployment diagram represents the intra-view link type that has been introduced in MES+.

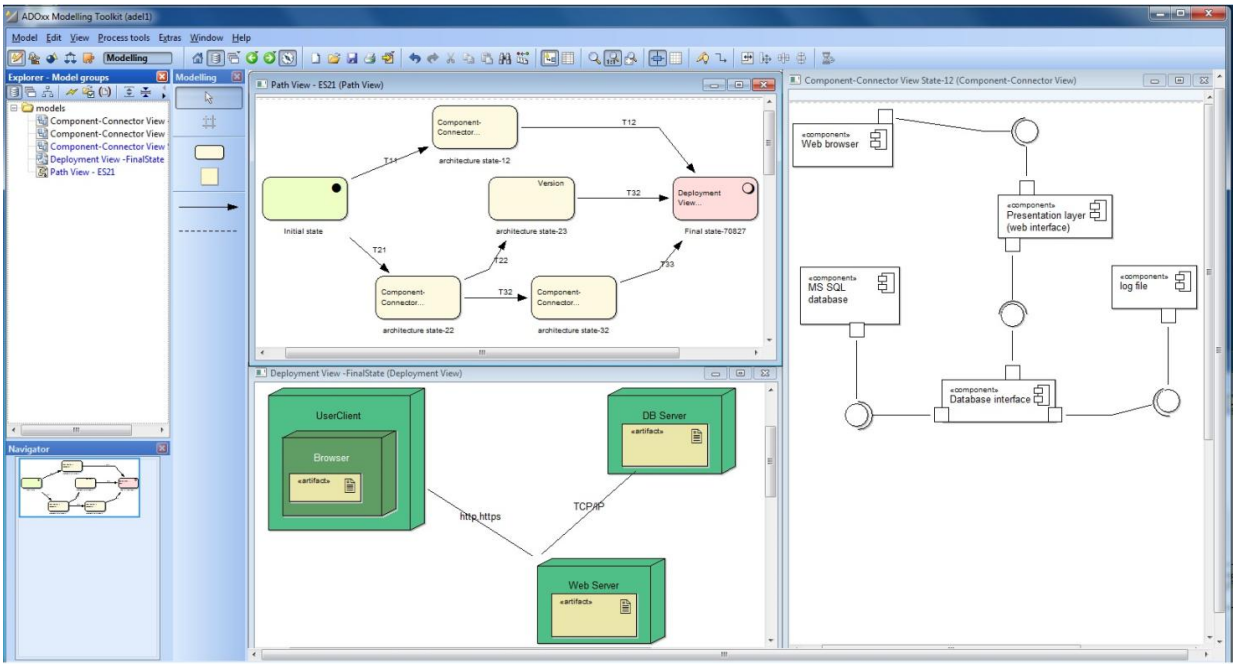

Figure 51: A screenshot of the evolution style editor based on MES +.

The decision view can be edited by clicking on any architecture transition (relation class). For example, by clicking on the architecture transition T11, the decision view which is represented in part [3] in Figure 50 will be opened. The architect can fill in the details of this decision which provides the reasons (rationale) behind the selection of this step.

More views or abstraction details can be developed and integrated with this tool. For example different views of different evolution styles can be included. Moreover, supporting facilities, such as a paths evaluation, can be developed using graphic analysis methods in order to support the architect in analysing and comparing paths.

7.6. Conclusion

This chapter has described a validation scenario for the MES+ method by developing a prototype tool based on its conception of multiple views. In this vein, the motivation behind the tool implantation was introduced, followed by a brief review of some of the available workbenches which can be used to implement the intended tool.

A set of conceptual views to describe an evolution style have been proposed, the aim of which is to synthesise different aspects and provide different abstraction levels. This proposed scenario has been implemented on the ADOxx metamodeling platform. Thus, a multi-view evolution style modeling tool was realised according to the MES+ conceptual design.

Chapter 8. Conclusions & Future Work

This chapter concludes this thesis by summarising the main contributions (Section 8.1) and points the reader in the direction of future work (Section 8.2).

1.1 Conclusions

A software system may be subject to evolution several times throughout its lifespan in order to accommodate changes in requirements and technologies. The evolution of a software system is a complex process which consists of multiple interdependent tasks and involves multiple development teams. Every software has an architecture which must be restructured to keep abreast of the other system artifacts. Software architectures provide the most appropriate artifact for addressing the complexity of the evolution process as it enables development teams to work at a higher abstraction level. For this reason, software architecture has become an area in which a significant amount of research is being conducted. This research trend has relied on two different considerations of software architecture, the first being to tackle the architecture as an artifact for the evolution to guide the planning and restructuring of the system and the second considering the architecture as an artifact of evolution, which must itself be evolved to be consistent with the system change (Cuesta et al. 2013).

This thesis contributes to different topics in the area of software architecture evolution modeling and reuse. It proposes a multi-view/abstraction-level meta-style evolution for the domain of software architecture evolution. Evolution style is a domain-specific methodology for modeling software architecture evolution, the aim of which is to model the best architectural evolution practices and knowledge in a particular domain as a library encapsulates the set of possible evolution scenarios in this domain. Different evolution-style approaches have been developed, each of which defines his own vocabularies and metaphors (notions, syntax, semantics) to depict the evolution process in accordance with a certain viewpoint (Chapter 3). To this end, we adopt the metamodeling technique, acceding to the meta-meta-model level in the domain of architecture evolution and devising a meta-style evolution called MES in order to work as a unified solution for the domain (Chapter 4). This Meta-style exploits the conception of the domain object-oriented in the software process engineering modeling (such as SPEM, Essence and BPMN) so that it may reflect the architectural view on the modeling metaphor of the process. As SPEM defines the three main classes in the process modeling -: Task, Work Product and Role – MES also specifies the four modeling components: Operation, Architecture Element, Role and Interaction. The MES development methodology is component-based, reusability-oriented and compatible with the OMG specifications for process- and product-modeling approaches. However, in order to explore the ability of MES to specify a metamodeling language that has the necessary vocabulary to define an evolution style (i.e. it has the meta-elements which can instantiate any modeling element of an evolution style), mapping and comparison scenarios between MES and evolution styles, as well as standards models (MOG's MOF and SPEM), were carried out. Transformation rules were clarified,

which used to manage the transformation of an evolution style, for example to the Eclipse Process Framework (SPEM base framework).

In Chapter 5 we investigated the ability of extending MES in order to embrace the behavioural aspect of the evolution process. To this end, a literature review of the dynamic evolution of software architecture and a component-based framework has been conducted, and the relevant issues, such as safe-stopping, state transfer, change management and dynamic re-scheduling, were explored. In this way, some related concepts were extracted and annotated to MES in order to meet the requirements of modeling dynamic evolution in software architecture.

The emergence of diverse evolution-style approaches is the result of the diversity and difference of perspectives, concerns and interests among the stakeholders since the aim of the evolution style is to guide the planning and enacting of the evolution process in which a variety of stakeholder categories are involved. Each category has its own concerns regarding some relevant aspects of the process and has a need to access details to a certain level. Therefore, each group of stakeholders aims to have a process model (an evolution style) that satisfies its needs. Developing one single view (evolution style) that can satisfy all the different categories' concerns is very difficult, if not all but impossible. To this end, Chapter 6 was dedicated to investigating this issue. An extension of MES was developed to try to integrate the concepts of multi-view and multi-abstraction level into MES in order to fit the requirements of instantiating an evolution style that can comprise different views. Each view can cover a set of related aspects or satisfy the concerns of a stakeholder group.

In Chapter 7 we presented an attempt to evaluate the feasibility and applicability of our approach. To this end, an initial prototype of evolution-style editor, based on the MES methodology was developed. Some structural views (path diagram, component diagram and deployment diagram) were developed and integrated to cover different levels of detail about the process. Furthermore, a decision-detail view was developed to provide architectural-evolution knowledge about each architectural transition in the possible evolution scenarios. The initial tool attempts to cover different views often an architectural model: structural, behavioural and reasoning (decision rationale). A limitation of the implementation aspect of our approach is that it does not cover the behavioural view of the evolution process. This limitation can be addressed by adapting an ADDL which can support the behavioural and timing constraints. Such an implementation can support the evaluation of the ability of the MES to satisfy the requirements of the dynamic evolution of the software architecture that was proposed in Chapter 5.

Another limitation of this thesis is that it does not deeply investigate in the practical side of the proposed approach. A completed version of the tool can be used in an industrial case study in order to investigate what views are exactly can assist architects (stakeholders) in planning and conducting the evolution process.

8.2 Future Work

The perspectives of future work are directly related to the limitations of our proposal approach which can be classified into two categories: conceptual aspect and applicative aspect.

From the conceptual perspective, modeling the dynamic evolution still is an open challenge which needs more exploration, using the concepts of ADDLs to model the dynamic evolution. In this way, we can benefit from some facilities provided by these languages, for example the SAM (Schedulability Analysis Model) sub-profile of MARTE can support the scheduling analysis of dynamic evolution. This can augment the scope of research to include infrastructures (RE-OSGi and RT Operating System) that allow the dynamic evolution of a component-based software system.

Another point about this aspect which may need more exploration is the process methodology for process development. Like software process methodologies (e.g. Waterfall, Spiral, prototyping, V and so on), process methodologies are needed to develop specific methods that fit the process development where the main product is a process instead of a software. Such methodologies focus on issues related to processes and their fragments and on how to support process design and reuse.

Moreover, an interesting research area could lie in the emergence of the process models. Using the power of AI, cloud computing and big data, the development of a process product can be greatly enhanced. We can use machine-learning in scenarios such as process design in order to synthesise a process. This can help the developer by extracting a lot of previous process knowledge and practice, particularly with regard to domains, in order to estimate expected scenarios for the evolution process.

In the applicative perspective of our approach, the benefit obtained by MES can be wildly augmented only after an industrial case study which can assist in clarifying the optimal set of evolution-style views in order to develop the final version of the MES+ tool. However, achieving such a state requires wide experimentation and a very long time. We believe that many views can be further integrated into the initial conceptual design of MES+. For example, an object-oriented view (class) diagram can be associated as an abstraction level of component-connector view which can be used to fill the gap between the architectural artifacts and the code.

Another implementation can be archived on the editor where more facilities can be provided. For example, in the Ævol workbench, path analysis was provided. Such a facility can be used to support the tool in comparing evolution paths. Moreover, consistency among views should be taken into account.

Therefore, in the future, we plan better tool-support for MES+ the better to assist the architect, as it is an initial prototype and does not have all the intended facilities. A full version of this prototype would improve the capture and reuse of the architectural-evolution knowledge and provide a set of consistent views from different perspectives and within different levels of detail.

References

Agha, Gul A. 1985. "Actors: A model of concurrent computation in distributed systems." In.: MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

Aldrich, Jonathan, Craig Chambers, and David Notkin. 2002. "ArchJava: connecting software architecture to implementation." In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 187-97. IEEE.

Allen, Robert, Remi Douence, and David Garlan. 1998. 'Specifying and analyzing dynamic software architectures', *Fundamental Approaches to Software Engineering*: 21-37.

Allen, Robert, and David Garlan. 1997. 'A formal basis for architectural connection', *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6: 213-49.

Allen, Robert J. 1997. "A Formal Approach to Software Architecture." In.: CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

Amirat, Abdelkrim, and Mourad Oussalah. 2009. 'First-class connectors to support systematic construction of hierarchical software architecture', *The Journal of Object Technology*, 8: 107-30.

Aoussat, Fadila, Mourad Oussalah, and Mohamed Ahmed Nacer. 2011. "SPEM Extension with software process architectural concepts." In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, 215-23. IEEE.

Arnarsdóttir, K, Arne-Jørgen Berre, Axel Hahn, Michele Missikoff, and Francesco Taglino. 2006. "Semantic mapping: ontology-based vs. model-based approach Alternative or complementary approaches?" In *EMOI-INTEROP*.

Asadollahi, Reza, Mazeiar Salehie, and Ladan Tahvildari. 2009. "StarMX: A framework for developing self-managing Java-based systems." In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, 58-67. IEEE.

Baetens, Nick. 2011. 'Comparing graphical DSL editors: AToM3, GMF, MetaEdit', *University of Antwerp*.

Barais, Olivier, Anne Françoise Le Meur, Laurence Duchien, and Julia Lawall. 2008. 'Software architecture evolution.' in, *Software Evolution* (Springer).

Baresi, Luciano, Reiko Heckel, Sebastian Thone, and Dániel Varró. 2004. "Style-based refinement of dynamic software architectures." In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, 155-64. IEEE.

Barnes, Jeffrey M, David Garlan, and Bradley Schmerl. 2014. 'Evolution styles: foundations and models for software architecture evolution', *Software & Systems Modeling*, 13: 649-78.

Barnes, Jeffrey M, Ashutosh Pandey, and David Garlan. 2013. "Automated planning for software architecture evolution." In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 213-23. IEEE Press.

Bass, Len, Paul Clements, and Rick Kazman. 1998. "Software Architecture in Practice. 1998." In.: Addison-Wesley.

Batista, Thais, Ackbar Joolia, and Geoff Coulson. 2005. "Managing dynamic reconfiguration in component-based systems." In *European workshop on software architecture*, 1-17. Springer.

Baumeister, Hubert, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. 2006. 'A component model for architectural programming', *Electronic Notes in Theoretical Computer Science*, 160: 75-96.

Benedí, Jenifer Pérez. 2006. 'Prisma: aspect-oriented software architectures'.

Bézivin, Jean, and Reiko Heckel. 2006. 'Guest editorial to the special issue on language engineering for model-driven software development', *Software and Systems Modeling*, 5: 231-32.
Blom, Hans, De-Jiu Chen, Henrik Kaijser, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Sara Tucci. 2016. 'EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research', *International Journal of System Dynamics Applications (IJSDA)*, 5: 1-20.
Bloom, Toby. 1983. "Dynamic Module Replacement in a Distributed Programming System." In.: MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.
Boehm, Barry W. 1988. 'A spiral model of software development and enhancement', *Computer*, 21: 61-72.
Boiten, Eerke, Howard Bowman, John Derrick, Peter Linington, and Maarten Steen. 2000. 'Viewpoint consistency in ODP', *Computer Networks*, 34: 503-37.
Bosch, Jan. 2004. 'Software architecture: The next step', *EWSA*, 3047: 194-99.
Bradbury, Jeremy S, James R Cordy, Juergen Dingel, and Michel Wermelinger. 2004. "A survey of self-management in dynamic software architecture specifications." In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, 28-33. ACM.
Brand, Christian, Matthias Gorning, Tim Kaiser, Jürgen Pasch, and Michael Wenz. 2011. 'Development of high-quality graphical model editors', *Eclipse Magazine*.
Breivold, Hongyu Pei, Ivica Crnkovic, and Magnus Larsson. 2012. 'A systematic review of software architecture evolution research', *Information and Software Technology*, 54: 16-40.
Bruneton, Eric, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. 2006. 'The fractal component model and its support in java', *Software: Practice and Experience*, 36: 1257-84.
Buckley, Jim, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. 2005. 'Towards a taxonomy of software change', *Journal of Software: Evolution and Process*, 17: 309-32.
Bures, Tomas, Petr Hnetynka, and Frantisek Plasil. 2006. "Sofa 2.0: Balancing advanced features in a hierarchical component model." In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, 40-48. IEEE.
Calmant, Thomas, Joao Claudio Americo, Olivier Gattaz, Didier Donsez, and Kiev Gama. 2012. "A dynamic and service-oriented component model for python long-lived applications." In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, 35-40. ACM.
Canal, Calos, Ernesto Pimentel, and José M Troya. 1999. 'Specification and refinement of dynamic software architectures.' in, *Software Architecture* (Springer).
Capilla, Rafael, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. '10 years of software architecture knowledge management: Practice and future', *Journal of Systems and Software*, 116: 191-205.
Caplat, Guy, and J-L Sourrouille. 2005. 'Model mapping using formalism extensions', *IEEE Software*, 22: 44-51.
Cassou, Damien, Benjamin Bertran, Nicolas Loriant, and Charles Consel. 2009. "A generative programming approach to developing pervasive computing systems." In *ACM Sigplan Notices*, 137-46. ACM.
Chapin, Ned, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. 2001. 'Types of software evolution and software maintenance', *Journal of Software: Evolution and Process*, 13: 3-30.
Chetto, Houssine, and Maryline Chetto. 1989. 'Some results of the earliest deadline scheduling algorithm', *IEEE transactions on Software Engineering*, 15: 1261.

Clark, Tony, Paul Sammut, and James Willans. 2015. 'Applied metamodelling: a foundation for language driven development', *arXiv preprint arXiv:1505.00149*.
Clements, Paul C. 1996. "A survey of architecture description languages." In *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, 16-25. IEEE.
Clements, Paul, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. 2002. *Documenting software architectures: views and beyond* (Pearson Education).
Clements, Paul, David Garlan, Reed Little, Robert Nord, and Judith Stafford. 2002. 'Documenting Software Architectures: Views and Beyond'.
——. 2003. "Documenting software architectures: views and beyond." In *Proceedings of the 25th International Conference on Software Engineering*, 740-41. IEEE Computer Society.
Clements, Paul, Rick Kazman, and Len Bass. 2013. "Software Architecture in Practice." In.: Pearson.
Cockburn, Alistair. 2001. *Agile software development*.
Committee, IEEE Computer Society. Software Engineering Standards, and IEEE-SA Standards Board. 1998. "Ieee recommended practice for software requirements specifications." In.: Institute of Electrical and Electronics Engineers.
Cook, Steve, Gareth Jones, Stuart Kent, and Alan Cameron Wills. 2007. *Domain-specific development with visual studio dsl tools* (Pearson Education).
Coulson, Geoff, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. 2004. 'A component model for building systems software'.
Cuenot, Philippe, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, and Martin Törngren. 2010. '11 the east-adl architecture description language for automotive embedded software.' in, *Model-based engineering of embedded real-time systems* (Springer).
Cuesta, Carlos E, Pablo de la Fuente, and Manuel Barrio-Solárzano. 2001. "Dynamic coordination architecture through the use of reflection." In *Proceedings of the 2001 ACM symposium on Applied computing*, 134-40. ACM.
Cuesta, Carlos E, Elena Navarro, Dewayne E Perry, and Cristina Roda. 2013. 'Evolution styles: using architectural knowledge as an evolution driver', *Journal of Software: Evolution and Process*, 25: 957-80.
Dashofy, Eric M, André Van der Hoek, and Richard N Taylor. 2001. "A highly-extensible, XML-based architecture description language." In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, 103-12. IEEE.
David, Pierre-Charles, and Thomas Ledoux. 2005. "WildCAT: a generic framework for context-aware applications." In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, 1-7. ACM.
Debruyne, Vincent, Françoise Simonot-Lion, and Yvon Trinquet. 2005. 'EAST-ADL—An architecture description language.' in, *Architecture Description Languages* (Springer).
Dowling, Jim, and Vinny Cahill. 2001. "The k-component architecture meta-model for self-adaptive software." In *International Conference on Metalevel Architectures and Reflection*, 81-88. Springer.
Dumas, Marlon, Marcello La Rosa, Jan Mendling, and Hajo A Reijers. 2013. 'Essential Process Modeling.' in, *Fundamentals of Business Process Management* (Springer).
Emerson, Matthew J, Janos Sztipanovits, and Ted Bapty. 2004. 'A MOF-Based Metamodeling Environment', *J. UCS*, 10: 1357-82.
Erlikh, Len. 2000. 'Leveraging legacy system dollars for e-business', *IT professional*, 2: 17-23.

Escoffier, Clément, Richard S Hall, and Philippe Lalanda. 2007. "iPOJO: An extensible service-oriented component framework." In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, 474-81. IEEE.

Evensen, Kenneth D, and Kathryn Anne Weiss. 2010. "A comparison and evaluation of real-time software systems modeling languages." In *Aerospace Conference, Georgia, Atlanta*.

Fabry, Robert S. 1976. "How to design a system in which modules can be changed on the fly." In *Proceedings of the 2nd international conference on Software engineering*, 470-76. IEEE Computer Society Press.

Fahrmair, Michael, Chris Salzmann, and Maurice Schoenmakers. 1999. "Carp@—A Reflection Based Tool for Observing Jini Services." In *Workshop on Reflection and Software Engineering*, 209-27. Springer.

Fan, Xiaocong. 2015. *Real-time Embedded Systems: Design Principles and Engineering Practices* (Newnes).

Faugere, Madeleine, Thimothee Bourbeau, Robert De Simone, and Sebastien Gerard. 2007. "Marte: Also an uml profile for modeling aadl applications." In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, 359-64. IEEE.

Feiler, Peter H, David P Gluch, and John J Hudak. 2006. "The architecture analysis & design language (AADL): An introduction." In.: Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Finkelstein, Anthony, Jeff Kramer, Bashar Nuseibeh, Ludwik Finkelstein, and Michael Goedicke. 1992. 'Viewpoints: A framework for integrating multiple perspectives in system development', *International Journal of Software Engineering and Knowledge Engineering*, 2: 31-57.

Fowler, Martin. 2010. *Domain-specific languages* (Pearson Education).

Gacek, Cristina, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm. 1995. "On the definition of software system architecture." In *Proceedings of the First International Workshop on Architectures for Software Systems*, 85-94. Seattle, Wa.

García-Borgoñon, Laura, MA Barcelona, JA García-García, M Alba, and María José Escalona. 2014. 'Software process modeling languages: A systematic literature review', *Information and Software Technology*, 56: 103-16.

Garcia, Felix, Aurora Vizcaino, and Christof Ebert. 2011. 'Process management tools', *IEEE Software*, 28: 15-18.

Garlan, David. 2000. "Software architecture: a roadmap." In *Proceedings of the Conference on the Future of Software Engineering*, 91-101. ACM.

——. 2002. *Software architecture* (Wiley Online Library).

——. 2014. "Software architecture: a travelogue." In *Proceedings of the on Future of Software Engineering*, 29-39. ACM.

Garlan, David, Robert Allen, and John Ockerbloom. 1994. 'Exploiting style in architectural design environments', *ACM SIGSOFT Software engineering notes*, 19: 175-88.

Garlan, David, Jeffrey M Barnes, Bradley Schmerl, and Orieta Celiku. 2009. "Evolution styles: Foundations and tool support for software architecture evolution." In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, 131-40. IEEE.

Garlan, David, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. 2004. 'Rainbow: Architecture-based self-adaptation with reusable infrastructure', *Computer*, 37: 46-54.

Garlan, David, Robert T Monroe, and David Wile. 2000. 'Acme: Architectural description of component-based systems', *Foundations of component-based systems*, 68: 47-68.

Garlan, David, and Dewayne Perry. 1994. "Software architecture: practice, potential, and pitfalls." In *Proceedings of the 16th international conference on Software engineering*, 363-64. IEEE Computer Society Press.
Garlan, David, and Dewayne E Perry. 1995. 'Introduction to the special issue on software architecture', *IEEE Trans. Software Eng.*, 21: 269-74.
Garlan, David, and Bradley Schmerl. 2009. "Ævol: A tool for defining and planning architecture evolution." In *Proceedings of the 31st International Conference on Software Engineering*, 591-94. IEEE Computer Society.
Garlan, David, and Mary Shaw. 1993. "An introduction to software architecture: Advances in software engineering and knowledge engineering, volume I." In.: World Scientific Publishing.
Garland, Jeff, and Richard Anthony. 2003. *Large-scale software architecture: a practical guide using UML* (John Wiley & Sons).
Gilb, Tom. 1981. 'Evolutionary development', *ACM SIGSOFT Software engineering notes*, 6: 17-17.
Gomes, Antônio Tadeu A, Thais V Batista, Ackbar Joolia, and Geoff Coulson. 'Architecting Dynamic Reconfiguration in Dependable Systems'.
Gupta, Deepak, Pankaj Jalote, and Gautam Barua. 1996. 'A formal framework for on-line software version change', *IEEE transactions on Software Engineering*, 22: 120-31.
Hall, Richard S, and Humberto Cervantes. 2003. 'Gravity: supporting dynamically available services in client-side applications', *ACM SIGSOFT Software engineering notes*, 28: 379-82.
Hassan, Adel, and Mourad Oussalah. 2016. "Meta-evolution style for software architecture evolution." In *International Conference on Current Trends in Theory and Practice of Informatics*, 478-89. Springer.
Hassan, Adel, Audrey Queudet, and Mourad Oussalah. 2016. "Evolution Style: Framework for Dynamic Evolution of Real-Time Software Architecture." In *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28–December 2, 2016, Proceedings 10*, 166-74. Springer.
Haumer, Peter. 2005. 'Ibm rational method composer: Part 1: Key concepts', *IBM Report, December*.
——. 2007. 'Eclipse process framework composer', *Eclipse Foundation*.
Herraiz, Israel, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez Barahona. 2013. 'The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review', *ACM Comput. Surv.*, 46.
Hillman, Jamie, and Ian Warren. 2004. "An open framework for dynamic reconfiguration." In *Proceedings of the 26th International Conference on Software Engineering*, 594-603. IEEE Computer Society.
Hirsch, Dan, Paolo Inverardi, and Ugo Montanari. 1998. "Graph grammars and constraint solving for software architecture styles." In *Proceedings of the third international workshop on Software architecture*, 69-72. ACM.
Hornford, Dave. 2011. 'TOGAF Version 9.1', *Zaltbommel. Van Haren Publishing*.
Hutchinson, John, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. "Empirical assessment of MDE in industry." In *Software Engineering (ICSE), 2011 33rd International Conference on*, 471-80. IEEE.
Ilieva, Sylvia, Iva Krasteva, Gorka Benguria, and Brian Elvesæter. 2010. 'Deliverable D2. 8 REMICS Methodology with agile extension, Final Release'.
Irmert, Florian, Thomas Fischer, and Klaus Meyer-Wegener. 2008. "Runtime adaptation in a service-oriented component model." In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, 97-104. ACM.

ISO. 1999. "ISO/IEC 14764:1999. Information technology -- Software maintenance." In.
Ivers, James, Paul C Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Oviedo-Silva. 2004. 'Documenting component and connector views with UML 2.0', *Computer Science Department*: 665.
Jamshidi, Pooyan, Mohammad Ghafari, Aakash Ahmad, and Claus Pahl. 2013. "A framework for classifying and comparing architecture-centric software evolution research." In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 305-14. IEEE.
Jansen, Anton, and Jan Bosch. 2005. "Software architecture as a set of architectural design decisions." In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, 109-20. IEEE.
Jeusfeld, Manfred A, Matthias Jarke, and John Mylopoulos. 2009. *Metamodeling for method engineering* (the MIT Press).
Kajko-Mattsson, Mira, Michael Striewe, Michael Goedicke, Ivar Jacobson, Ian Spence, Shihong Huang, Paul McMahon, Bruce MacIsaac, Brian Elvesæter, and Arne J Berre. 2012. "Refounding software engineering: The Semat initiative (Invited presentation)." In *Software Engineering (ICSE), 2012 34th International Conference on*, 1649-50. IEEE.
Kent, Stuart. 2002. "Model driven engineering." In *Integrated formal methods*, 286-98. Springer.
Kephart, Jeffrey O, and David M Chess. 2003. 'The vision of autonomic computing', *Computer*, 36: 41-50.
Khare, Rohit, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. 2001. "xADL: enabling architecture-centric tool integration with XML." In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, 9 pp.: IEEE.
Kheir, Ahmad, Hala Naja, Kifah Tout, and Mourad Chabane Oussalah. 2014. 'MoVAL: Towards a Multi-views/Multi-hierarchy Software Architecture'.
Kiwelekar, Arvind W. 2013. "Architectural connectors." In.
Kleppe, Anneke G, Jos Warmer, and Wim Bast. 2003. "The model driven architecture: practice and promise." In.
Kotonya, Gerald, and Ian Sommerville. 1996. 'Requirements engineering with viewpoints', *Software Engineering Journal*, 11: 5-18.
Koudri, Ali, and Joel Champeau. 2010. 'MODAL: a SPEM extension to improve co-design process models', *New Modeling Concepts for Today's Software Processes*: 248-59.
Kramer, Jeff. 2007. 'Is abstraction the key to computing?', *Communications of the ACM*, 50: 36-42.
Kramer, Jeff, and Jeff Magee. 1990. 'The evolving philosophers problem: Dynamic change management', *IEEE transactions on Software Engineering*, 16: 1293-306.
Kruchten, Philippe. 1995a. 'Architectural blueprints—The "4+ 1" view model of software architecture', *Tutorial Proceedings of Tri-Ada*, 95: 540-55.
——. 2004. "An ontology of architectural design decisions in software intensive systems." In *2nd Groningen workshop on software variability*, 54-61. Citeseer.
Kruchten, Philippe B. 1995b. 'The 4+ 1 view model of architecture', *IEEE Software*, 12: 42-50.
Kruchten, Philippe, Patricia Lago, and Hans van Vliet. 2006. 'Building Up and Reasoning About Architectural Knowledge', *Quality of Software Architectures*: 43.
Küçükkeçeci Çetinkaya, D. 2013. 'Model Driven Development of Simulation Models: Defining and Transforming Conceptual Models into Simulation Models by Using Metamodels and Model Transformations'.
Lara, Juan, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. 'Model-driven engineering with domain-specific meta-modelling languages', *Software and Systems Modeling (SoSyM)*, 14: 429-59.

Lau, Kung-Kiu, Perla Velasco Elizondo, and Zheng Wang. 2005. "Exogenous Connectors for Software Components." In *CBSE*, 90-106. Springer.
Le Goaer, Olivier, Dalila Tamzalit, Mourad Chabane Oussalah, and Abdelhak-Djamel Seriai. 2008. "Evolution styles to the rescue of architectural evolution knowledge." In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, 31-36. ACM.
Lehman, M. M. 1974. ' Programs, Cities, Students, Limits to Growth?', *Inaugural Lecture*, 9: 18.
Lehman, Manny M. 1996. "Laws of software evolution revisited." In *European Workshop on Software Process Technology*, 108-24. Springer.
Lehman, MM. 2000. "Towards a theory of software evolution-and its practical impact." In.
Levendovszky, Tihamer, Gabor Karsai, Miklos Maroti, Akos Ledeczi, and Hassan Charaf. 2002. "Model reuse with metamodel-based transformations." In *International Conference on Software Reuse*, 166-78. Springer.
Li, Qing, and Caroline Yao. 2003. *Real-time concepts for embedded systems* (CRC Press).
Liebenberg, Martin, Kerstin Roßmaier, and Gerhard Lakemeyer. 2017. 'An iStar 2.0 Editor Based on the Eclipse Modelling Framework'.
Lientz, B.P., and E.B. Swanson. 1980. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations* (Addison-Wesley).
Liu, Chung Laung, and James W Layland. 1973. 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM (JACM)*, 20: 46-61.
Luckham, David C., John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. 1995. 'Specification and analysis of system architecture using Rapide', *IEEE transactions on Software Engineering*, 21: 336-54.
Luckham, David C., and James Vera. 1995. 'An event-based architecture definition language', *IEEE transactions on Software Engineering*, 21: 717-34.
Maciel, Rita Suzana Pitangueira, Bruno Carreiro da Silva, Ana Patrícia Fontes Magalhães, and Nelson Souto Rosa. 2009. "An integrated approach for model driven process modeling and enactment." In *Software Engineering, 2009. SBES'09. XXIII Brazilian Symposium on*, 104-14. IEEE.
Magee, Jeff, and Jeff Kramer. 1996. "Dynamic structure in software architectures." In *ACM SIGSOFT Software engineering notes*, 3-14. ACM.
Maier, Mark W, David Emery, and Rich Hilliard. 2001. 'Software architecture: Introducing IEEE standard 1471', *Computer*, 34: 107-09.
Malabarba, Scott, Raju Pandey, Jeff Gragg, Earl Barr, and J Fritz Barnes. 2000. "Runtime support for type-safe dynamic Java classes." In *European Conference on Object-Oriented Programming*, 337-61. Springer.
Martínez-Ruiz, Tomás, Félix García, and Mario Piattini. 2008. 'Towards a SPEM v2. 0 extension to define process lines variability mechanisms', *Software engineering research, management and applications*: 115-30.
Martinho, Ricardo, João Varajão, and Dulce Domingos. 2009. 'FlexSPMF: a framework for modelling and learning flexibility in software processes', *Visioning and Engineering the Knowledge Society. A Web Science Perspective*: 78-87.
Medvidovic, Nenad. 1996. "ADLs and dynamic architecture changes." In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, 24-27. ACM.
——. 1998. "An architecture-based approach to software evolution." In.

Medvidovic, Nenad, and Richard N Taylor. 2000. 'A classification and comparison framework for software architecture description languages', *IEEE transactions on Software Engineering*, 26: 70-93.
Mehta, Nikunj R, Nenad Medvidovic, and Sandeep Phadke. 2000. "Towards a taxonomy of software connectors." In *Proceedings of the 22nd international conference on Software engineering*, 178-87. ACM.
Mens, T., A. Serebrenik, and A. Cleve. 2014. *Evolving Software Systems* (Springer My Copy UK).
Milner, Robin. 1989. 'Communication and concurrency'.
Moad, Jeff. 1990. 'Maintaining the competitive edge', *Datamation*, 36: 61-&.
Montero, Francisco, and Elena Navarro. 2009. "ATRIUM: Software architecture driven by requirements." In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, 230-39. IEEE.
Mullery, Geoff P. 1979. "CORE-a method for controlled requirement specification." In *Proceedings of the 4th international conference on Software engineering*, 126-35. IEEE Press.
Münch, Jürgen, Ove Armbrust, Martin Kowalczyk, and Martin Sotó. 2012. *Software process definition and management* (Springer Science & Business Media).
Murata, Tadao. 1989. 'Petri nets: Properties, analysis and applications', *Proceedings of the IEEE*, 77: 541-80.
Nassar, Mahmoud. 2003. "VUML: a Viewpoint oriented UML Extension." In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 373-76. IEEE.
Nuseibeh, Bashar, and Steve Easterbrook. 2000. "Requirements engineering: a roadmap." In *Proceedings of the Conference on the Future of Software Engineering*, 35-46. ACM.
OMG. 2008. "Software & Systems Process Engineering Meta-Model Specification, Version 2.0." In.: Object Management Group (OMG).
OMG Submitters, O. 2012. 'Essence–Kernel and Language for Software Engineering Methods'.
Oquendo, Flavio. 2004. 'π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures', *ACM SIGSOFT Software engineering notes*, 29: 1-14.
Oquendo, Flavio, Brian Warboys, Ronald Morrison, Régis Dindeleux, Ferdinando Gallo, Hubert Garavel, and Carmen Occhipinti. 2004. "Archware: Architecting evolvable software." In *EWSA*, 257-71. Springer.
Oreizy, Peyman. 1998. "Issues in modeling and analyzing dynamic software architectures." In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, 54-57.
Oussalah, Mourad Chabane. 2014. *Software architecture 1* (John Wiley & Sons).
Oussalah, Mourad, Nassima Sadou, and Dalila Tamzalit. 2006. "SAEV: A model to face evolution problem in software architecture." In *Proceedings of the International ERCIM Workshop on Software Evolution*, 137-46.
Oussalah, Mourad, Dalila Tamzalit, Olivier Le Goaer, and Abdelhak Seriai. 2006. "Updating Software Architectures: A Style-Based Approach." In *International Conference on Software Engineering Research and Practice*.
Perry, DE. 1997. "Software architecture and its relevance to software engineering, invited talk." In *Second International Conference on Coordination Models and Languages*.
Perry, Dewayne E, and Alexander L Wolf. 1992. 'Foundations for the study of software architecture', *ACM SIGSOFT Software engineering notes*, 17: 40-52.

Plasil, Frantisek, Dusan Balek, and Radovan Janecek. 1998. "SOFA/DCUP: Architecture for component trading and dynamic updating." In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, 43-51. IEEE.

Rajlich, Václav T, and Keith H Bennett. 2000. 'A staged model for the software life cycle', *Computer*, 33: 66-71.

Recker, Jan, Michael Rosemann, Marta Indulska, and Peter Green. 2009. 'Business process modeling-a comparative analysis', *Journal of the Association for Information Systems*, 10: 1.

Richardson, Thomas. 2011. 'Developing Dynamically Reconfigurable Real-time Systems with Real-time OSGi (RT-OSGi)', University of York, UK.

Rombach, H Dieter, and Martin Verlage. 1995. 'Directions in software process research.' in, *Advances in computers* (Elsevier).

Ross, Douglas T. 1977. 'Structured analysis (SA): A language for communicating ideas', *IEEE transactions on Software Engineering*: 16-34.

Ross, Douglas T, and Kenneth E Schoman. 1977. 'Structured analysis for requirements definition', *IEEE transactions on Software Engineering*: 6-15.

Rozanski, Nick, and Eóin Woods. 2005. 'Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives'.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. 2004. *Unified modeling language reference manual, the* (Pearson Higher Education).

Sallem, Marcio Augusto Sekeff, and Francisco José da Silva e Silva. 2007. "The Adapta Framework for Building Self-Adaptive Distributed Applications." In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, 46-46. IEEE.

Scherer, Sabrina. 2016. 'Towards an E-Participation Architecture Framework (EPArt-Framework)'.

Segal, Mark E, and Ophir Frieder. 1989. 'Dynamic program updating: A software maintenance technique for minimizing software, downtime', *Journal of Software: Evolution and Process*, 1: 59-79.

Seinturier, Lionel, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. 2009. "Reconfigurable sca applications with the frascati platform." In *Services Computing, 2009. SCC'09. IEEE International Conference on*, 268-75. IEEE.

Sha, Lui, Ragunathan Rajkumar, and Michael Gagliardi. 1996. "Evolving dependable real-time systems." In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, 335-46. IEEE.

Shaw, Mary. 1993. "Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status." In *Workshop on Studies of Software Design*, 17-32. Springer.

Shaw, Mary, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. 1995. 'Abstractions for software architecture and tools to support them', *IEEE transactions on Software Engineering*, 21: 314-35.

Shaw, Mary, and David Garlan. 1996. *Software architecture: perspectives on an emerging discipline* (Prentice Hall Englewood Cliffs).

Sinz, Elmar J, and Domenik Bork. 'Design of a SOM Business Process Modelling Tool Based on the ADOxx Meta-modelling Platform'.

Smeda, Adel, Mourad Oussalah, AboBaker ElHouni, and El-Bahlul Fgee. 2008. "COSABuilder: an extensible tool for architectural description." In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, 1-6. IEEE.

Smeda, Adel, Mourad Oussalah, and Tahar Khammaci. 2004. 'A Multi-Paradigm Approach to Describe Complex Software System', *WSEAS Transactions on Computers*, 3: 936--41.

——. 2005. "Madl: Meta architecture description language." In *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*, 152-59. IEEE.
Sommerville, Ian, and Pete Sawyer. 1997. 'Viewpoints: principles, problems and a practical approach to requirements engineering', *Annals of software engineering*, 3: 101-30.
Soni, Dilip, Robert L Nord, and Christine Hofmeister. 1995. "Software architecture in industrial applications." In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, 196-96. IEEE.
Spivey, J Michael, and JR Abrial. 1992. *The Z notation* (Prentice Hall Hemel Hempstead).
Spuri, Marco, and Giorgio Buttazzo. 1996. 'Scheduling aperiodic tasks in dynamic priority systems', *Real-Time Systems*, 10: 179-210.
Spuri, Marco, and Giorgio C Buttazzo. 1994. "Efficient Aperiodic Service Under Earliest Deadline Scheduling." In *RTSS*, 2-11.
Stankovic, John A. 1988. 'Misconceptions about real-time computing: A serious problem for next-generation systems', *Computer*, 21: 10-19.
Suryadevara, Anuradha. 2010. 'Surveying and evaluating tools for managing processes for software-intensive systems', M. Sc Thesis, Mälardalen University, Sweden.
Szyperski, Clemens. 2002. *Component Software: Beyond Object-Oriented Programming* (Addison-Wesley Longman Publishing Co., Inc.).
Tang, Antony, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar. 2010. 'A comparative study of architecture knowledge management tools', *Journal of Systems and Software*, 83: 352-70.
Taton, Christophe, Sara Bouchenak, Fabienne Boyer, Noël De Palma, Daniel Hagimont, and Adrian Mos. 2005. 'Self-manageable replicated servers', *Implementation, and Deployment of Database Replication*: 55.
Taylor, R. N., N. Medvidovic, and E.M. Dashofy. 2009. *Software Architecture: Foundations, Theory and Practice* (Wiley).
Ter Hofstede, Arthur HM, and TF Verhoef. 1997. 'On the feasibility of situational method engineering', *Information Systems*, 22: 401-22.
Tyree, Jeff, and Art Akerman. 2005. 'Architecture decisions: Demystifying architecture', *IEEE Software*, 22: 19-27.
Van Deursen, Arie, Paul Klint, and Joost Visser. 2000. 'Domain-specific languages: An annotated bibliography', *ACM Sigplan Notices*, 35: 26-36.
van Heesch, Uwe, Paris Avgeriou, and Rich Hilliard. 2012. 'A documentation framework for architecture decisions', *Journal of Systems and Software*, 85: 795-820.
van Heesch, Uwe, Veli-Pekka Eloranta, Paris Avgeriou, Kai Koskimies, and Neil Harrison. 2014. 'Decision-centric architecture reviews', *IEEE Software*, 31: 69-76.
Van Ommering, Rob, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. 2000. 'The Koala component model for consumer electronics software', *Computer*, 33: 78-85.
Vandewoude, Yves, and Yolande Berbers. 2005. 'Fresco: Flexible and reliable evolution system for components', *Electronic Notes in Theoretical Computer Science*, 127: 197-205.
Vandewoude, Yves, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. 'Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates', *IEEE transactions on Software Engineering*, 33: 856-68.
Verjus, Hervé, Sorana Cîmpan, and Ilham Alloui. 2012. 'An Architecture-Centric Approach for Information System Architecture Modeling, Enactement and Evolution.' in, *Innovative Information Systems Modelling Techniques* (InTech).

Vogel, Oliver, Ingo Arnold, Arif Chughtai, and Timo Kehrer. 2011. *Software architecture: a comprehensive framework and guide for practitioners* (Springer Science & Business Media).

Wang, Qianxiang, Junrong Shen, Xiaopeng Wang, and Hong Mei. 2006. 'A component-based approach to online software evolution', *Journal of Software: Evolution and Process*, 18: 181-205.

White, Jules, Douglas C Schmidt, and Sean Mulligan. 2007. "The generic eclipse modeling system." In *Model-Driven Development Tool Implementer's Forum, TOOLS*.

Yau, Stephen S, James S Collofello, and T MacGregor. 1978. "Ripple effect analysis of software maintenance." In *Computer Software and Applications Conference, 1978. COMPSAC'78. The IEEE Computer Society's Second International*, 60-65. IEEE.

Zachman, John A. 1987. 'A framework for information systems architecture', *IBM systems journal*, 26: 276-92.

Zelkowitz, Marvin V, Alan C Shaw, and John D Gannon. 1979. *Principles of software engineering and design* (Prentice-Hall Englewood Cliffs).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre: Styles et Méta-Styles: Une Autre Façon de Réutiliser l'Evolution d'Architectures Logicielles.

Mots clés: Style d'évolution, Réutilisation de l'évolution, Architecture Logicielle.

Résumé: Au cours des dernières années, la taille et la complexité des systèmes logiciels ont considérablement augmenté, rendant le processus d'évolution plus complexe et consommant ainsi beaucoup de ressources. C'est pourquoi, l'architecture logicielle est devenue l'un des éléments les plus importants dans la planification et la mise en œuvre du processus d'évolution. Cette abstraction permet une meilleure compréhension des décisions de conception prises précédemment et un bon moyen d'explorer, d'analyser et de comparer des scénarii alternatifs de l'évolution. Fort de constat, nous avons introduit une approche de styles d'évolution afin de capitaliser les pratiques d'évolution récurrentes dans un domaine particulier et de favoriser leur réutilisation. Dans cette thèse, nous préconisons en spécifiant un cadre de modélisation standard conforme à différents styles d'évolution et pouvant satisfaire les préoccupations de différentes équipes impliquées dans un processus d'évolution. Afin de relever les défis de la réutilisation de l'évolution de l'architecture logicielle, nous nous fixons comme objectif: D'abord, d'introduire un style de méta-évolution qui spécifie les éléments conceptuels de base nécessaires à la modélisation de l'évolution; ensuite, de décrire une nouvelle méthodologie pour développer un style d'évolution selon plusieurs vues et plusieurs abstractions. Cette approche multi-vues/multi-abstractions permet de réduire la complexité du modèle de processus d'évolution en décomposant un style d'évolution en plusieurs vues et abstractions pertinentes. Enfin, pour la validité et la faisabilité de notre approche, nous avons développé un prototype basé sur la plateforme de méta-modélisation ADOxx.

Title: Style and Meta-Style: Another Way to Reuse Software Architecture Evolution

Keywords: Evolution style, Evolution reuse, Software architecture.

Abstract: Over the last years, the size and complexity of software systems has been dramatically increased, making the evolution process more complex and consuming a great deal of resources. Consequently, software architecture is becoming an important artifact in planning and carrying out the evolution process. It can provide an overall structural view of the system without undue focus on low-level details. This view can provide a deep understanding of previous design decisions and a means of analysing and comparing alternative evolution scenarios. Therefore, software architecture evolution has gained significant importance in developing methods, techniques and tools that can help architects to plan evolution. To this end, an evolution styles approach has been introduced with the aim of capitalising on the recurrent evolution practices and of fostering their reuse

In this thesis, we endeavour to tackle the challenges in software architecture evolution reuse by specifying a standard modeling framework that can conform to different evolution styles and satisfy the concerns of the different stakeholder groups. The primary contribution of this thesis is twofold. First, it introduces a meta-evolution style which specifies the core conceptual elements for software architecture evolution modeling. Second, it introduces a new methodology to develop a multi-view & multi-abstraction evolution style in order to reduce the complexity of the evolution model by breaking down an evolution style into several views, each of which covers a relevant set of aspects. The central ideas are embodied in a prototype tool in order to validate the applicability and feasibility of the proposed approaches.