ARIANE : Automated Re-Documentation to Improve software Architecture uNderstanding and Evolution

Alexandre Le Borgne

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
ÉCOLE NATIONALE SUPÉRIEURE DES MINES D'ALÈS (IMT MINES ALÈS)

En Informatique

I2S – Information, Structures, Systèmes
Portée par l'Université de Montpellier

Unité de recherche LGI2P

ARIANE: Automated Re-documentation to Improve software Architecture uNderstanding and Evolution

Présentée par Alexandre LE BORGNE
Le 24 Janvier 2020

Sous la direction de David DELAHAYE
et Marianne HUCHARD

Devant le jury composé de

Nicolas ANQUETIL, MCF HDR, INRIA, Univ. Lille	Rapporteur
Nicole LEVY, PR, CNAM Paris	Rapporteur, Présidente
Nicolas BELLOIR, MCF, IRISA, Ecoles de St-Cyr Coëtquidan	Examinateur
Mourad OUSSALAH, PR, LINA, Univ. Nantes	Examinateur
David DELAHAYE, PR, LIRMM, Univ. Montpellier	Co-directeur
Marianne HUCHARD, PR, LIRMM, Univ. Montpellier	Co-directrice
Christelle URTADO, MA IMT HDR, LGI2P, IMT Mines Alès	Co-encadrante
Sylvain VAUTTIER, MA IMT HDR, LGI2P, IMT Mines Alès	Co-encadrant

UNIVERSITÉ DE MONTPELLIER

IMT Mines Alès
École Mines-Télécom

"En raison d'un appel à la grève émanant de la CGT, nous ne sommes pas en mesure de diffuser l'intégralité de nos programmes habituels. Nous vous prions de nous en excuser."

Radio France

Abstract

All along its life-cycle, a software may be subject to numerous changes that may affect its coherence with its original documentation. Moreover, despite the general agreement that up-to-date documentation is a great help to record design decisions all along the software life-cycle, software documentation is often outdated. Architecture models are one of the major documentation pieces. Ensuring coherence between them and other models of the software (including code) during software evolution (co-evolution) is a strong asset to software quality. Additionally, understanding a software architecture is highly valuable in terms of reuse, evolution and maintenance capabilities. For that reason, re-documenting software becomes essential for easing the understanding of software architectures. However architectures are rarely available and many research works aim at automatically recovering software architectures from code. Yet, most of the existing re-documenting approaches do not perform a strict reverse-documenting process to re-document architectures "as they are implemented" and perform re-engineering by clustering code into new components. Thus, this thesis proposes a framework for re-documentating architectures as they have been designed and implemented to provide a support for analyzing architectural decisions. This re-documentation is performed from the analysis of both object-oriented code and project deployment descriptors. The re-documentation process targets the Dedal architecture language which is especially tailored for managing and driving software evolution. Another highly important aspect of software documentation relates to the way concepts are versioned. Indeed, in many approaches and actual version control systems such as GitHub, files are versioned in an agnostic manner. This way of versioning keeps track of any file history. However, no information can be provided on the nature of the new version, and especially regarding software backward-compatibility with previous versions. This thesis thus proposes a formal way to version software architectures, based on the use of the Dedal architecture description language which provides a set of formal properties. It enables to automatically analyze versions in terms of substitutability, version propagation and proposes an automatic way for incrementing version tags so that their semantics corrrespond to actual evolution impact. By proposing such a formal approach, this thesis intends to prevent software drift and erosion. This thesis also proposes an empirical study, to validate our approach named ARIANE, based on both re-documenting and versioning processes on numerous versions on an enterprise project taken from GitHub.

Remerciements

En premier lieu je tiens à remercier Jacky Montmain, directeur du Laboratoire de Génie Informatique et d'ingénierie de Production (LGI2P) de l'IMT Mines Alès, de m'avoir accueilli au sein de l'équipe. Son écoute, sa patience et son dévouement pour les doctorants du LGI2P m'auront permis de réaliser cette thèse dans les meilleures dispositions.

Je tiens ensuite à remercier l'ensemble des membres du jury qui ont accepté d'évaluer mon travail et qui, malgré un report impromptu de la soutenance dans un climat social tendu, se sont mobilisés pour que je soutienne dans les meilleures conditions possibles. Je remercie particulièrement Madame Nicole Lévy d'avoir présidé le jury et rapporté ma thèse. Je remercie également Monsieur Nicolas Anquetil d'avoir lui aussi accepté de rapporter ma thèse. De même, je remercie Monsieur Mourad Oussalah d'avoir accepté d'examiner mon travail de thèse. Enfin je remercie Monsieur Nicolas Belloir qui, en sa qualité d'examinateur et ancien de mes enseignants, m'aura accompagné pendant mes études d'informatique et ce jusqu'à ma soutenance de thèse.

Je tiens à présent à remercier mon équipe encadrante qui m'a fait confiance et sans laquelle je ne serais pas arrivé au bout de ce travail. Leurs qualités tant professionnelles qu'humaines m'auront porté tout au long de cette thèse. Ainsi je remercie mes directeurs de thèse : Marianne Huchard qui a su apporter sa sagesse dans les réunions de travail, et toujours beaucoup de bienveillance envers ses doctorants ; David Delahaye qui, malgré ou grâce à son esprit formel, a toujours agrémenté nos échanges d'une bonne dose d'humour. Bien sûr je remercie particulièrement mes encadrants de proximité qui m'ont suivi de près au LGI2P : Christelle Urtado d'avoir toujours su me motiver pendant cette thèse et dont la justesse des conseils m'aura aidé à y voir plus clair aux moments les plus déterminants ; Sylvain Vauttier pour les éclairages souvent techniques et théoriques qu'il m'a apporté pendant cette thèse. Aussi, je tiens à remercier mes encadrants pour leur disponibilité et la facilité que nous avons eu à pouvoir échanger et ainsi avancer ensemble.

Je remercie aussi l'ensemble du personnel technique et administratif du LGI2P et plus particulièrement Claude Badiou et Edith Teychené pour leur gentillesse et leur abnégation. Je remercie aussi l'ensemble des enseignants-chercheurs, doctorants, post-doctorants et ingénieurs avec qui j'ai toujours eu plaisir à échanger. De même je remercie l'ensemble des stagiaires, Paul Heidmann, Guillaume André et Valentin Colas qui m'auront aidé à mener mon projet à bien. Je tiens également à remercier mes collègues et amis : Behrang, Quentin, Pierre-Antoine, Thibault, Emilie, Clément, Pascale, Roland, Perrine, Lucie, Frank et tous ceux avec qui j'ai eu l'occasion de passer de bons moments, et qui ont été d'un grand soutien pendant ces trois ans.

Enfin je remercie ma famille, pour son amour et son soutien indéfectible qui m'auront porté jusque-là. Je remercie aussi plus particulièrement Cécile que j'ai le bonheur d'avoir à mes côtés depuis le début de ma thèse – cœur avec les doigts –.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives a brief introduction of the context this thesis stands in, the problem which is addressed, the proposed contributions, and presents the outline of the manuscript.

1.1 General context of component-based software engineering

Because of the constantly increasing complexity of software systems, new needs have appeared from early ages of software engineering for advantageously producing and maintaining software reducing costs. This is why component-based software engineering has emerged in late 1990's as a sub-discipline of software engineering, which promises to address those issues. Component-based software engineering advocates a specific software development approach centered on component reuse [Som11]. This discipline proposes a set of methods and models which aim at improving component-based software development (CBSD). CBSD approaches give a methodological support to enhance reusability by providing guidelines to assemble already developed decoupled software components. Those component are stored in repositories and referred as Off-The-Shelf components. It therefore avoids building entire systems from scratch, taking advantage of past developments. It significantly decreases development costs and time-to-market preserving quality of software components [CL02].

As an essential part of CBSD, software architectures give an abstraction of the software structure and expose the way that it is supposed to evolve [Gar00]. Software architecture models therefore contain the list of the elements that are part of the system and the information about how those elements are connected one to another. As software architectures are abstraction of software themselves, they capture the design decisions, which occur during the development process. This level of abstraction helps to reason in terms of architectural element evolution instead of source code evolution, which can be more difficult to understand [GDT06].

However, despite CBSD processes have been improved over years, some issues remain that concern software architecture maintenance and evolution [BCL12]. As part of those issues, we can highlight the evolution of component-based software architecture documentation,

which often becomes obsolete [DP09], and the versioning of software architectures, which is surprisingly not much discussed in literature.

1.2 Documenting and versioning component-based software architectures issues

Despite a lot of work [DP09] in the field of software engineering for improving the documentation of software systems, software evolution still leads to design decision loss. Software evolution without co-evolution of software models is but the ruin of architectures. Then, most of the time, design decisions are lost due to architectural drift or erosion [GMW97]. We argue that the information that has been lost during software evolution must therefore be recovered prior to performing any crucial evolution task. A lot of work exists that addresses this issue by generally proposing approaches that re-engineer software architectures [DP09]. Very few of these approaches only intend to re-document component-based architectures "as they are implemented". Moreover, none of these approaches consider more than two abstraction levels whereas previous work on Dedal ADL has shown that three abstraction levels are necessary to handle the global software life-cycle [Zha+12b]. However, such documentation is essential for performing software evolution.

Additionally, during its evolution, a software is subject to numerous changes that lead to numerous versions of the software. Then, in order to keep track of the software history, it is necessary to identify its successive versions. Despite abundant literature in the field of databases and software versioning, little work even intend to address this issues in the field of component-based software architectures. There are even fewer approaches that propose a way to version components and / or architectures by using verifiable semantics. However, such semantics are needed in order to well identify software versions since a wrong version identifier may misguide software architects.

1.3 Thesis proposal and contribution

As a contribution, this thesis proposes to answer the following research questions:

- **RQ1.** Is it possible to re-document multi-abstraction level component-based architectures from source code, and is it possible to retrieve abstract design decisions from this re-documentation?
- **RQ2.** How to introduce semantics in component and architecture versioning?
- **RQ3.** Are such re-documenting and / or versioning approaches suitable for large software systems?
- **RQ4.** Is it possible to identify drift and / or erosion situations by re-documenting and analyzing software versions?

In order to improve software evolution, documentation must remain consistent with the actual software implementation and deployment all along its life-cycle. Moreover, this documentation must cover the software development main steps, which are the specification, the implementation, and the deployment. However, despite the well-known benefits of an up-to-date documentation, it is often not consistent with the actual state of the software because of an undocumented evolution. Performing evolution tasks may therefore be difficult in this case. This is why this thesis proposes an approach to re-document software architectures from raw source code. This approach is based on Dedal [Zha+12b; Mok+16a], which provides three architecture levels for tracking main steps of software life-cycle. Moreover, it also provides a formalized basis for calculating automated evolution plan. This formalism especially ensures the three architecture level coherence. In other words, it ensures that the description of deployment is consistent with the description of implementation, and finally that the description of implementation is consistent with the description of the specification. Dedal therefore provides a good support for re-documenting software in order to retrieve suitable software evolution capabilities. This contribution answers research question **RQ1**.

Another contribution of this thesis consists in using formal rules based on type theory [Aré+07; Aré+09] to characterize component and architecture differences in terms of backward compatibility. The characterization of changes is made from a change analysis impact study. This study is based on formal Dedal architectural rules and allows us to derive a set of rules to characterize substitutable and not substitutable changes. Change impact analysis also makes it possible to derive rules of version propagation among the three Dedal architecture levels in order to preserve architectural consistency. This contribution also includes a proposal to automatically change version identifiers accordingly to the kind of version that is identified. This part of the thesis answers research question **RQ2**.

Finally, the last contribution of this thesis is the set of tools that have been developed in order to answer research questions **RQ3** and **RQ4**. Those tools can be fully integrated into the eclipse ecosystem and *DedalStudio*, which is our CASE (Computer-Aided Software Engineering) tool that supports the Dedal ADL. The tools have been released online (see GitHub[1] and LGI2P's web site[2]) and consist of the following components:

- *SpringDSL* is our implementation of XML Spring [Joh+04] grammar into the EMF[3] environment.
- *HierarchyBuilder* proposes to build the entire type hierarchy of a Java project including required libraries where traditional code parsers only consider source code.
- *component-based-hierarchy-builder* re-documents three leveled Dedal architectures from source code and Spring framework.
- *ProjectComparator* calculates and characterizes architectural differences between two versions of a Dedal architecture model.

[1] http://www.github.com/DedalArmy
[2] http://www.dev.lgi2p.mines-ales.fr/ariane/
[3] https://www.eclipse.org/emf/ [Last seen 2019-09-05]

- *DiffAnalyzer* analyzes found differences and checks for architectural deviation situations.

In addition to the main tooling contributions, we released the previously developed *DedalStudio* modules as eclipse plugin online to ease its installation into the eclipse environment.

Next section presents the outline of the thesis.

1.4 Outline of the thesis

The thesis is organized as follows:

- Chapter 2 introduces in detail the context of this thesis. It presents the component-based software development process and the component-based software architecture concept. It also introduces the Dedal architecture description language.
- Chapter 3 introduces the state of the art of this thesis. It consists of three parts. The first one is a survey of versioning approaches in literature in order to identify limits of these approaches especially in term of formalization and automation of version identification. The second one concerns formal architecture evolution approaches to highlight their limits in terms of component and architecture versioning. Finally, the last one is a survey that compares re-documentation and reconstruction approaches, and justifies our choice for re-documenting software as three-leveled component-based architecture with the Dedal ADL.
- Chapter 4 introduces the proposed approach and algorithm for re-documenting component-based architectures from source code. It defines the different steps that lead from an undocumented software to a three-level description of it.
- Chapter 5 introduces an approach for managing version identification in software histories. This identification is based on a formal architecture impact analysis and proposes rules for automatically characterizing component and architecture versions.
- Chapter 6 presents the implementation of our approach and introduces a study that has been conducted on more than 200 versions of an enterprise open-source project to re-document it and check the soundness of its version identification in terms of architecture erosion / drift situation identification.
- Chapter 7 finally summarizes the thesis contributions and discusses limitations and perspectives to this work.

Chapter 2

Context and motivations

Contents

As introduced in Chapter 1, the contribution of this thesis takes place in the field of Component-Based Software Engineering (CBSE). More precisely, this thesis focuses on Component-Based Software Architectures re-documention and versioning. This chapter is designed to give a deeper understanding of the context this thesis stands in. As it takes places in the continuity of Zhang's thesis [ZUS10] and Mokni's thesis [Mok15], it is positioned in the same context and therefore follows the same outline as Zhang's and Mokni's thesis context.

2.1 Component-based software engineering

Component-Based Software Engineering appeared in late 1990's as a subdiscipline of the wide Software Engineering field. CBSE provides developers with methods, models and guidelines oriented towards component-based systems [Pre97]. CBSE then emerged as a reuse-based approach to software development [Som11]. The goal of CBSE, is to provide keys for producing software from already developed components in opposition with developments realized from scratch. The motivation of such approach is to meet software industry concerns about the reduction of costs, development time to meet customer needs, software maintainability and reliability [SGM02]. Thus, CBSE quickly took a great place in the field of Software Engineering. This success is due to some important factors. First of all, software becomes more and more complex and provides more functionality. The use of software components makes it possible to meet the need of producing more functionalities with the same investment in terms of costs and time [Pre97]. Next, until CBSE, traditional approaches fail at supporting reuse. As stated by Sommerville [Som11], abstract unit descriptions such as components can be considered as standalone service providers while object classes are too much detailed and specific. Last but not least, software constantly evolves

and so do its requirements, which means that a support for easy change is needed. Council and Heineman [CH01] identified the three following major concerns of CBSE:

- to support reusable component entities,
- to support development of systems as component assemblies,
- and to ease maintainability and upgrading of such systems by being able to customize and replace their components.

However, despite the well founded benefit of such goals, achieving them in practice can be very challenging and then improving reuse processes is tedious [Pre97]. This difficulty is even emphasized in component-based software evolution practice, especially after software deployment. Thus, for further understanding, it is necessary to consider component-based software life-cycle.

2.1.1 Component-based software life-cycle

Before the concept of component-based software life cycle is introduced, it is important to understand most global software development approaches.

2.1.1.1 Traditional software development processes

The waterfall model is an historic development model that has been proposed by Royce in 1970 [Roy87]. Most of the iterative software development approaches are based on the same activities than the waterfall model [Som11]. Those activities are shown in Figure 2.1 and are as follows:

1. **Requirement definition:** During this phase, the goals of the system are established. At the end of this phase, a specification of the system that complies with the realization of the goals is proposed. The specification defines functional and non-functional requirements.
2. **System and software design:** During this phase, the software is designed accordingly to the specification that has been produced at the previous phase. This design implies to identify and describe a conceptual and technical solution for the software to develop. Thus an architecture that describes the software is proposed.
3. **Implementation and unit testing:** This phases intends to produce an executable software that corresponds to the previously proposed design. Implementation can be composed of smaller units. The units are then verified and tested to meet their specification.
4. **Integration and system testing:** During this phase, system units are integrated and the complete system is verified, validated, and finally released.

5. **Operation and maintenance:** A system in operation needs continuous support and maintenance. This continuous support may imply to loop on previous waterfall phases for adding new functionalities, fixing bugs...

6. **Retirement and disposal:** This phase is often omitted in life-cycle models since it is implicit. It consists in the phasing out of the system that can either be replaced or completely terminated.

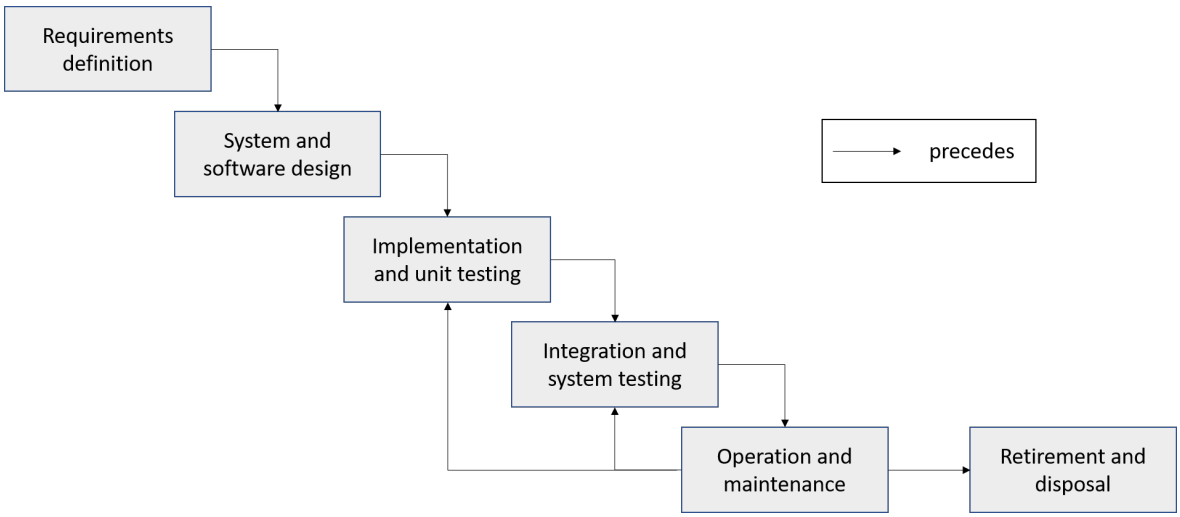


FIGURE 2.1: Waterfall development model

According to Sommerville [Som11], the waterfall model clearly separates the different phases of the software development process and the main advantage of its model is that phases do not overlap and documentation is incrementally produced and enriched. Such a development model is more suitable for small projects which requirements are well understood. However, such model is not appropriate to adapt to changing customer requirements. Such an approach is not responsive since commitments must be made early in the development process while in such models, results are produced very late [Som11].

2.1.1.2 Agile software development methods

In order to improve responsiveness of development processes, agile methods emerged in the 1990's. They support fast software development and are more adaptable to requirement change. They were primarily meant to support fast iterative development of business applications with short release cycles [Som11]. A group of practitioners has established in 2001 a consensus named the manifesto of agile software. This consensus sets the values of agile methods[1]. The manifesto argues for the following four values:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.

[1] https://www.agilealliance.org/agile101/the-agile-manifesto/ [Last seen 2019-08-31]

- Responding to change following a plan.

Agile methods are iterative methods that emphasize incremental development. They encourage active collaboration with customers into the development process for feedback and to ease requirement changes even after software delivery [Som11]. However, the main drawback of such methods resides in the fact that they depend too much on individual personalities. Developers may not be willing to bear the pressure of such processes, which may be intense. On the other hand, customers may no be willing to spend the time that is necessary to make those development processes valuable [Som11]. In addition, in such development processes, prioritizing changes might be a difficult task when the process involves too many stakeholders [Som11]. Thus, agile methods are well adapted to small and medium-sized systems without the risks associated to large, complex and critical systems [Som11].

2.1.1.3 Component-based software development processes

The main purpose of Component-Based Software Development (CBSD) is to build entire systems from preexisting components. Thus, there are two consequences on software development processes [CCL05]. The first consequence is that software development by component reuse is separated from component development. In CBSD processes, components need to be already developed at the start of the process. Second, the development process must include a component identification phase [Som11].

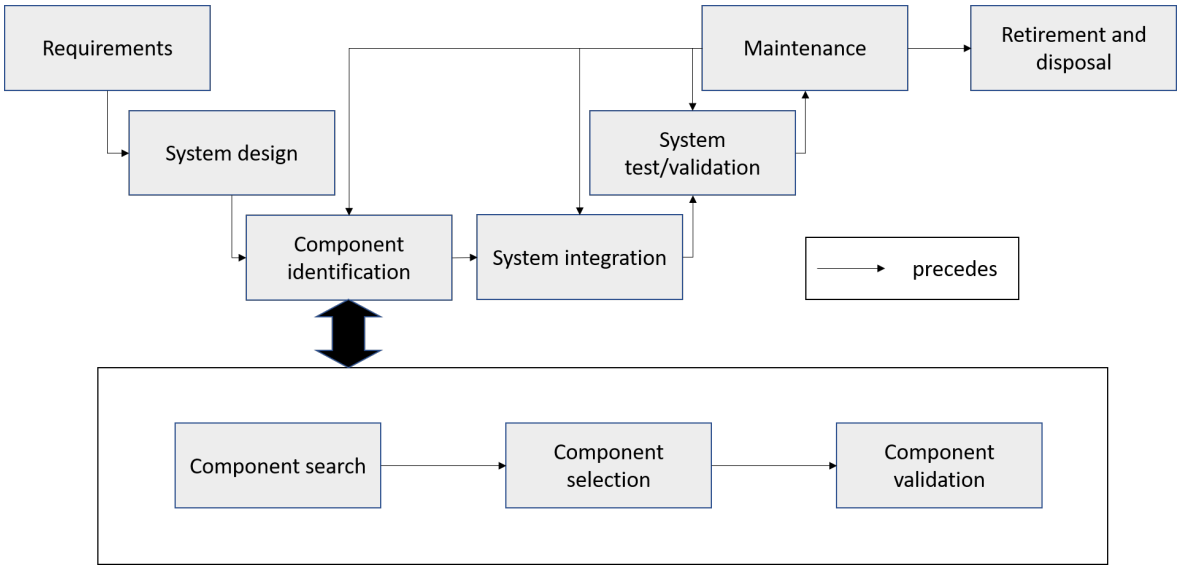


FIGURE 2.2: The CBSD process

CBSD activities are introduced in Figure 2.2 and are as follows:

1. **Requirements:** Requirements can be defined in the form of abstract component types that describe the functionalities of the system. In a component-based approach, the definition of requirements must take into account the ability to develop the system

with existing components. If possible, the system is realized using preexisting software components, otherwise, new components have to be developed or requirements might change to meet the available component resources [CCL05].

2. **System design:** This phase is designed to define a complete architecture of the system with refined component types that are fulfilled by existing software components. As in the previous phase, components are reused according to their availability. Components might need to be developed.

3. **Component identification:** This phase replaces the implementation phase of the traditional waterfall model. It consists in a combination of three activities:

   (a) **Component search:** During this activity, component repositories are browsed to identify suitable candidates to fit the architecture defined during the design phase.

   (b) **Component selection:** During this activity, a component composition is decided in order to offer the best coverage of system requirements [Som11]. This activity might be very complex since a perfect matching is often unrealistic.

   (c) **Component validation:** Once components have been selected, they need to be tested and validated in order to ensure that their behavior meets the system requirements.

4. **System integration:** This phase consists in deploying selected components into assemblies to constitute the executable system architecture.

5. **System test/validation:** This phase corresponds to the traditional test phase. It ensures that system requirements are met.

6. **Maintenance:** In the context of CBSD, this phase consists in keeping the system up-to-date by checking the availability of new component versions so that they can (if they meet system requirements) be identified, tested and deployed to replace older component versions.

7. **Retirement and disposal:** This phase has the same role as in traditional software development processes.

2.1.2 Summary

CBSE proposes a reuse intensive approach to software development. CBSD has several benefits compared to traditional software development processes as for instance, clear separation of concerns, reduced complexity, reduced development time, and increased software quality. However, it also comes with drawbacks. First of all, it can be difficult to identify trusty software components that perfectly match requirements and sometimes it is not even possible, which thus makes the adaptation of system requirements necessary. Second, managing evolution of such systems can be tricky. During maintenance phase, if the changes

that are caused by new component version deployment are not carefully handled, they may impact the whole system and compromise it. This issue is addressed in Chapter 5.

As an essential part of CBSE to handle components, the following section discusses the notion of component-based software architecture.

2.2 Component-based software architectures

This section gives an overview of basic concepts related to Component-Based Software Architectures (CBSA).

2.2.1 Basic concepts in software architecture

Software architectures are the outline of systems construction and evolution [TMD10]. They intend to provide an abstraction of the structure of software systems. They also expose the way systems are expected to evolve [Gar00]. Then, software architectures capture design decisions that are made during the system development. The structure of the system, its functional behavior, its interactions and its non-functional properties are design decisions. Perry and Wolf [PW92] identify three kinds of architectural elements which can be summarized into two major architectural concepts that are components and connectors. Those architectural elements are as follows:

- Processing elements are comparable to components that process data.
- Data elements are comparable to components that contain data to be processed.
- Connecting elements stand between components and hold connections.

Next section goes deeper and gives a more detailed overview of what components and connectors are.

2.2.1.1 Components

In order to define the concept of component, several definitions exist in the literature. A first definition of component has been given by Szypersky [SGM02]:

> *"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."*

Thus according to Szypersky, a component is a "black box", which hides details about code and implementation and which data is accessed through its interfaces. Components are decoupled entities, they are developed for reuse, and they comply with the principles of encapsulation, abstraction, and modularity.

A second definition has been given by Taylor *et al.* [TMD10]

> *"A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context."*

According to Taylor *et al.*, a component is a unit of composition, which encapsulates data and provides and / or requires (from other components) services. Thus, the notion of component is very wide and can represent a simple operation or an entire system according to the architecture.

Thus, mixing those two definitions, a component is made of a set of interfaces, an implementation and a specification.

Interfaces typically are the communication channels of components. They manage the component interactions with other components [SGM02]. An interface can be either provided or required. A provided interface exposes a set of services, which are provided to other components of the environment. In other words, other components may require services through other component provided interfaces. Unlike provided interfaces, required interfaces define services that are required by a component from other components for its execution. Thus, using such interface mechanisms, components are decoupled entities, which hide their complexity behind the exposure of provided and required interfaces and then are highly reusable units.

On the contrary, component implementation refers to the internal definition of a component, which includes the source code. However, the implementation of a component is only considered at development time and is quickly hidden to be integrated to a CBSD process. Doing so, no particular knowledge about their inner structure is needed to build component-based architectures.

According to Crnkovic and Larsson [CL02], the specification of a component is the definition (type) of its interfaces. In early stages of CBSE, interface specification was only a syntactical definition of the sets of signatures that are either provided or required. Then, some Interface Description Languages (IDLs) were proposed to specify component interfaces. On this basis, the notion of contract was proposed by Meyer [Mey92]. The concept of contract extends the purely syntactical information contained in interface specifications by adding the notion of behavior. This early notion of behavior focused on the definition of pre- and post-conditions on the interface operations. It has later been enriched with the concepts of synchronization and service quality [Beu+99; BJP10].

2.2.1.2 Connectors

The second major architectural elements are connectors. Connectors intend to manage communications between software system building blocks. In the context of CBSE, those blocks are components. Thus, connectors are meant to bind components together so that a component can invoke a service from another component and vice versa. Thus, connectors

are mediators between components [TMD10] and connect components through their interfaces. Computation concern (handled by components) and interaction concern (handled by connectors) are then well decoupled. This separation of concerns thus emphasizes reuse processes. According to Taylor *et al.* [TMD10], component interaction may become a very serious and challenging concern in the context of large and long time support systems. A connector may be of eight types, which have been identified by Mehta *et al.* [MMP00]: procedure call, event, data access, linkage, stream, arbitrator, adapter and distributor.

In some approaches such as C2-SADEL [MRT99] and Wright [AG97], connectors are considered as specific components with two communication points: the provided and the required connector ends. They can also be represented as simple links between two component interfaces [Mag+95].

2.2.2 Architecture modeling

Architecture modeling consists in describing one or more aspects of a system architecture. To do so, a particular notation is used in order to standardize the description. Taylor *et al.* [TMD10] define an architecture model as an artifact that captures parts or all of the design decisions of the software architecture.

2.2.2.1 Architectural modeling notations

There are several levels of formalism in architectural modeling notations, which stretch from informal to highly formal. Taylor *et al.* [TMD10] introduce three categories of architectural modeling notations according to their level of formalism:

- **Informal models**: Those models do not have a formally defined syntax. They are most often designed for non-technical stakeholders and usually presented as boxes-and-lines diagrams.
- **Semi-formal models**: Those models have a formally defined syntax. They can be used for both technical and non-technical stakeholders and are intended to find a balance between formalism and expressiveness. UML[2] is typically a semi-formal modeling notation.
- **Formal models**: Those models have a formal syntax and also formally defined semantics. They are most often intended to be used by the system technical stakeholders. They are mostly used to address system criticality and their formalized semantics make automated analysis possible.

Next section focuses on the languages that are used to describe architectures.

[2] http://www.omg.org/spec/UML/2.5/ [Last seen 2019-08-31]

2.2.2.2 Architecture description languages

Architecture Description Languages (ADLs) are languages that are dedicated to architecture modeling. They provide all the necessary features for describing software architectures. The definition of ADL has been given by Medvidovic [MJ06]:

> *"An architecture description language is a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. An ADL must support the building blocks of an architectural description."*

Thus, an ADL must provide the vocabulary to describe components and their interfaces, connectors, and configurations.

ADLs can also be used for performing architecture analysis to support architecture evolution. Such an activity is directly related to the level of formalism of the ADL. Section 3.2 introduces few of them.

2.2.3 Architecture evolution

During its whole life-cycle, a software is designed to evolve, so does its architecture. This evolution is considered as one of the most challenging tasks of CBSE. In order to understand motivations and issues of architecture evolution, it is important to introduce software evolution in general and the concept of architecture-centric evolution.

2.2.3.1 Software evolution

It is now well identified that the software maintenance phase concentrates most costs and difficulties. As a proof of this statement, Lientz *et al.* [LST78] have shown in the 1970's that this phase costs about 60% of the global software production costs. The IEEE 1219 Standard for software maintenance [Iee] defines maintenance as follows:

> *"Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."*

Moreover, the traditional CBSD process is too rigid and not suitable for dealing with evolution. As a matter of fact, requirements are also subject to change during the entire software life-cycle. It is then not realistic to consider that requirements are all known and fixed before starting the software design. In addition, experience acquired at the later phases might need to be fed back to earlier phases [DM08]. This limitation was known a long time ago and a particular interest to software evolution raised when Lehman stated the "Laws of software evolution" [Leh79]. He defined software evolution as follows:

> *"Software evolution is the collection of all programming activities intended to generate a new version of some software from an older operational version. If these activities can be performed at runtime without the need for system recompilation or restart, it becomes dynamic software evolution."*

The real novelty of this definition is that it dealt with system evolution rather than only considering code evolution.

Bennett and Rajlich then proposed an evolutionary process model [BR00a] for coping with the waterfall model limitations. As introduced in Figure 2.3, their model does not omit the problem of software aging. Once it has been initially developed, the software may suffer changes, which can lead to a degeneration of the software. When the software loses its evolvability, it enters in the servicing phase, which is intended to keep the software alive by applying small patches on it [DM08]. Finally, when it gets too hard and / or too expensive to keep the software running, it enters in the "phase out" phase and is then terminated in the close down phase.

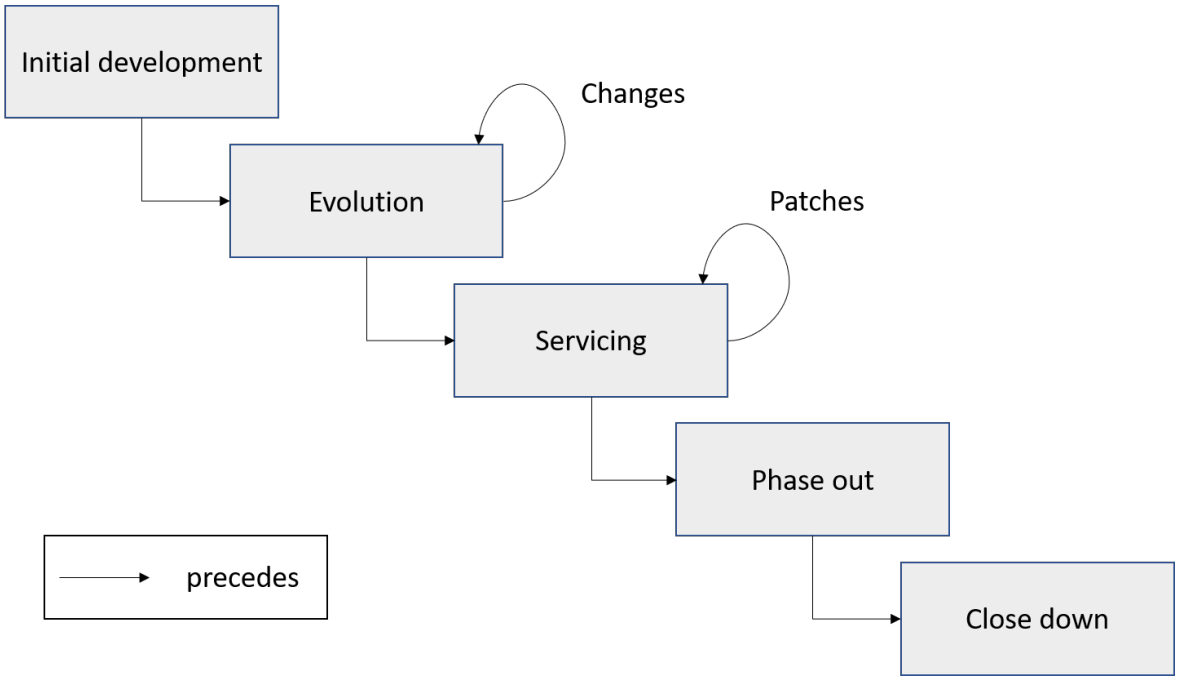


FIGURE 2.3: Benett an Rajlich process model for evolution [BR00a]

2.2.3.2 Architecture-centric evolution

Architectures present a great advantage for managing software evolution. As discussed before, architectures expose by nature the dimensions along which they are supposed to evolve [Gar00]. They make it possible to reason about evolution in an abstract manner that eases the understanding of change. They are also especially useful to estimate the cost that a change may have in terms of time and financial cost. By nature, components that are part of the architectures hide their complexity, then it is no more necessary to analyze source code that can be hard to understand and modify [GDT06]. Actually, architectures enable high level manipulations thanks to the nature of components, which can thus be added, deleted, or replaced, and their connection modified.

Nonetheless, issues exist in architecture evolution. Software architectures may be degraded

because of changes that lead to inconsistencies. Those inconsistencies may thus alter software architectures in such a way that they lose their evolvability, which then leads (according to Figure 2.3) to the progressive end of the software. It is then essential to prevent such inconsistencies in software architectures. However, it is one of the greatest challenge of CBSE and it can get very tricky to deal with such issue especially for reusing components [GAO09]. This issue is addressed in Chapter 5 in terms of architecture consistency recovering after versioning components (version propagation).

Next section introduces the notion of architectural inconsistency.

2.2.4 Architecture analysis

Architecture analysis intends to discover important system properties that are captured by its architecture models [TMD10]. Such an activity implies that analyzed models are formal as discussed in Section 2.2.2.1. Architecture analysis is typically useful to detect problems / inconsistencies in design decisions at early development stages.

According to Taylor *et al.* [TMD10], consistency is an internal architecture property that guarantees that elements in an architecture model do not collide. They identify five types of inconsistencies, which can appear in an architecture model:

- **Name inconsistency**: A name inconsistency occurs when several architectural elements have the same name and when an element that is not supposed to be accessed is actually accessed. It can also happen if a non-existing element is accessed.
- **Interface inconsistency**: This inconsistency can occur in case of a name inconsistency when a component requires a service which name does not match with a component provided service or when interface types do not match.
- **Behavioral inconsistency**: A behavioral inconsistency occurs between components that have services that do not match.
- **Interaction inconsistency**: An interaction inconsistency occurs when the interaction protocol between two components is not respected. For instance, not respecting a sequence for accessing a service may represent an interaction inconsistency.
- **Refinement inconsistency**: A refinement inconsistency occurs when architectural design decisions are changed, omitted or violated when an architectural description is refined.

In addition to those inconsistencies, we can cite two major architecture mismatches that have been introduced by Perry and Wolf [PW92]:

- **Drift**: Drift consists in the introduction of new design decisions at a low abstraction level, which are not described in higher abstraction levels. For instance, a new functionality is introduced in the implementation whereas it is not documented in higher abstraction levels. Drift can be considered as a refinement inconsistency.

- **Erosion**: Erosion is defined by de Silva and Balasubramaniam [DSB12] as follows:

  > *"Erosion is the phenomenon that occurs when the implemented architecture of a software diverges from its intended architecture."*

  In other words, erosion consists in the violation by an architecture implementation of design decisions described at higher abstraction level. As previously, erosion can also be considered as a refinement inconsistency. The definition given by de Silva and Balasubramaniam is interesting since it highlights the relation that exists between two architecture levels. Taylor *et al.* denote these levels as prescriptive and descriptive architectures [TMD10]. Little work has been lead to explicit these two levels and study their relationship. However, Zhang *et al.* [ZUS10; ZUV10; Zha+12a; Zha+12b; Mok+15] show that it is beneficial for architecture evolution and to control erosion to explicitely describe those two levels and even to take the runtime architecture level into account as dynamic changes can imply erosion. Considering erosion differently, the fact that the intended architecture diverges from its implemented architecture may be due to the addition of new requirements in the system's specification but some of them are not implemented. Zhang *et al.* define this issue as *pendency* which is the introduction of new design decisions into a higher architecture level that are not implemented by its lower architecture level.

Next section introduces the Dedal architecture model, which proposes to explicit the architecture abstraction levels that must be considered in CBSD processes in order to improve reuse and efficiently cope with architecture inconsistencies that may arise during architecture-centric evolution.

2.3 The Dedal architecture model

Figure 2.4 introduces the proposal of Zhang [Zha10] to explicit the architecture descriptions that are produced at each development step of a system. The process that is proposed focuses on three main development phases: specification (or design), implementation, and deployment. At the end of requirement analysis, an architecture specification is designed that defines the services that should be supplied by components. It also describes how components should be connected to one another in order to meet requirements. Thus, the architecture specification corresponds to the system intended architecture. The next step of the process consists in defining an architecture description (configuration) that implements the specification. This step is comparable to the component identification step of an usual CBSD: architects select suitable concrete components that match those specified. Architects then compose those concrete components to realize the complete architecture configuration. The configuration therefore corresponds to the implemented architecture that realizes the specification. The final life-cycle step described in this process consists in instantiating and deploying the configuration. The corresponding architecture that is documented at this step is called the assembly. It represents the architecture configuration as it is deployed.

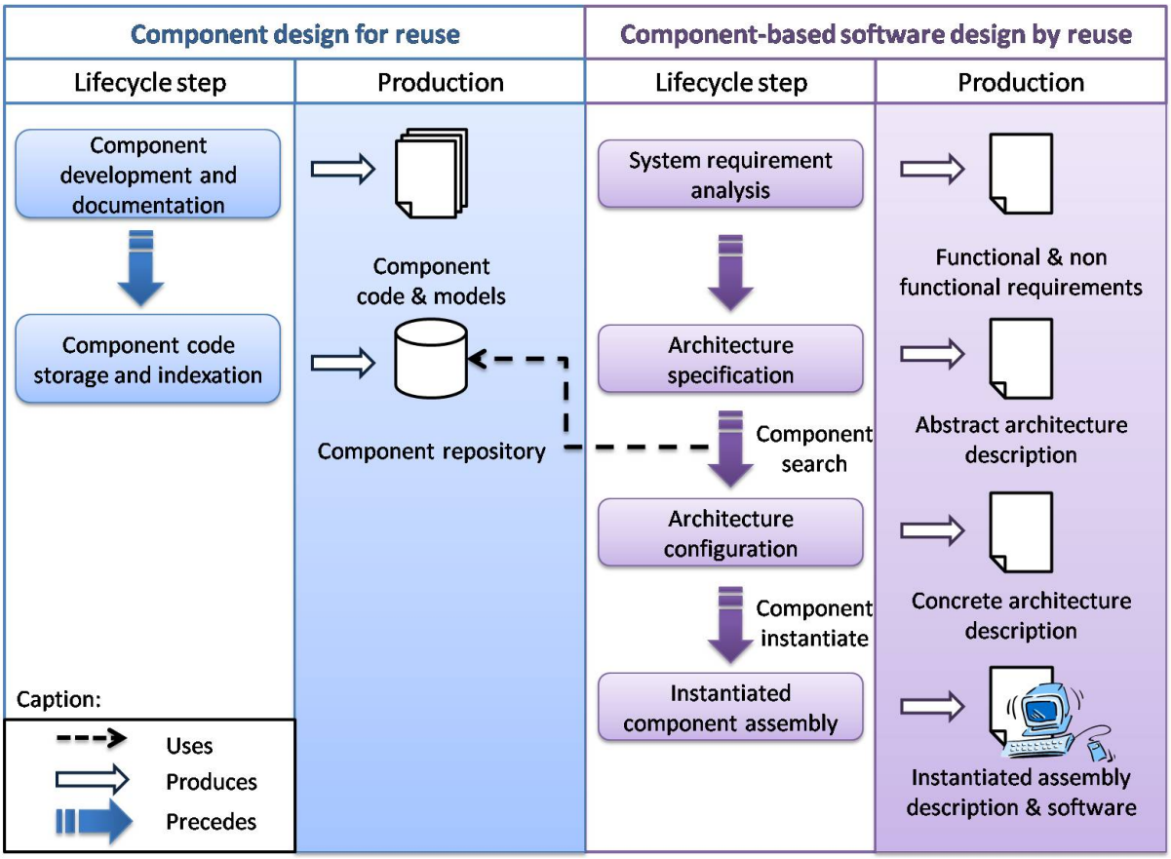


FIGURE 2.4: Reuse development process [Zha10]

Zhang *et al.* [ZUS10; Zha+12b] proposed an ADL and architecture model named Dedal, which supports such a process. This ADL clearly separates the three architecture definitions into three abstraction levels: specification, configuration and assembly. In order to ease the understanding, Figure 2.5 introduces the graphical notations of component interfaces that are used in the example that follows. The example is introduced in Figure 2.6 and illustrates the concepts of Dedal. This example is taken from Mokni's thesis [Mok15] and is a small architecture of a Home Automation Software (HAS). The HAS manages comfort scenarios by automatically controlling the building's lighting in function of the time. An orchestrator component interacts with the appropriate devices to play the desired scenario. This example is used in the following as a running example for improving our understanding of Dedal concepts.

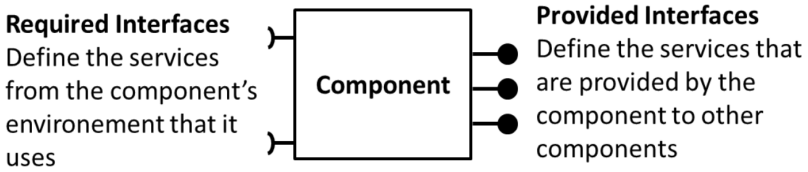


FIGURE 2.5: Component interfaces (adapted from [Som11])

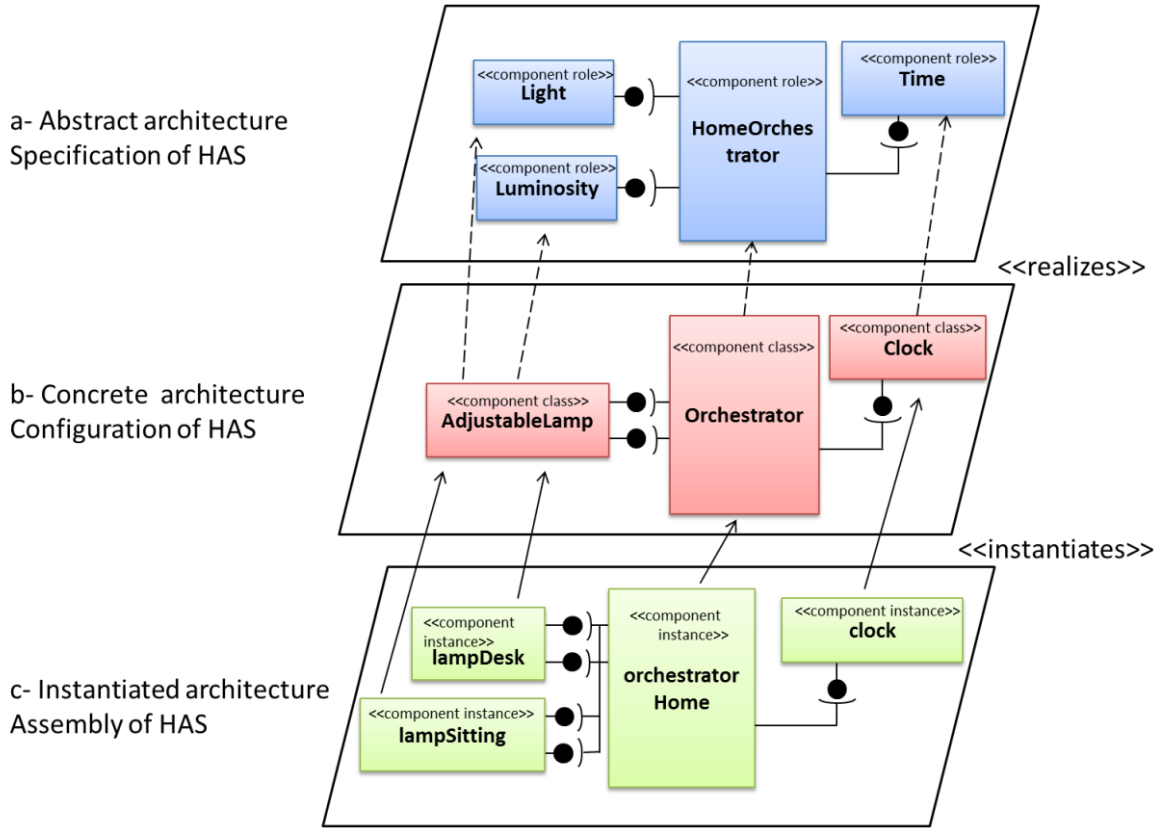


FIGURE 2.6: Dedal architecture levels for a Home Automated Software [Mok+16a]

2.3.1 The Dedal abstract architecture specification level

This architecture description level corresponds to the design phase of a CBSD process. It is designed to explicit the functional requirements of the software. Its purpose is to give an abstract view of the involved software elements (components). At this architecture level, design decisions consist in identifying abstract component types, which will be (re)used to operate the defined required functionalities. Those abstract components are called **component roles**.

Component roles are meant to declare the set of functionalities that are expected from available components. A component role declares a set of functionalities through the specification of interfaces. Doing so, it allows a wider set of components to match the specification and then be selected to implement the architecture. Component roles are thus guides for helping the concrete component search and selection process.

Following this principle, the specification of the HAS example (Figure 2.6) is made of the *HomeOrchestrator* component role, which handles lighting by using both *Light* and *Luminosity* component roles and also the *Time* component role.

2.3.2 The Dedal concrete architecture configuration level

The configuration architecture level represents the implementation phase of a CBSD process. It defines the concrete implementation that is adopted for the software system. A configuration is defined by the set of selected components (during the identification process) that best match the component roles defined in the architecture specification. These components are called **component classes** and their associated types are called **concrete component types**.

A component class therefore corresponds to an existing software component that has been stored in a repository. Dedal allows the definition of composite structures, which means that components can either be primitive or composite. A primitive component class encapsulates executable code whereas a composite component class encapsulates an inner architecture configuration. In a composite component class, the exposed set of interfaces corresponds to the set of unconnected interfaces of its inner components. Component classes may also contain observable attributes to allow parameterization.

A concrete component type is an abstract representation of a set of component classes. It declares a set of interfaces that a component class must define to be an implementation of this type. They are used to perform classification of component classes and build indexes in component repositories. Component roles are matched with concrete component types to find suitable component classes. The matching is performed by using specialization and substitution concepts inspired from those that have been defined by Arévalo *et al.* [Aré+07; Abo+09; Aré+09; Abo+19]. The particularity of Dedal *realization* relation is the fact that a component role can be realized by a single component class but also by a set of component classes.

Figure 2.6 shows an implementation (configuration) of the HAS that complies with the specification. In this configuration, *Orchestrator* realizes *HomeOrchestrator*, *Clock* realizes *Time*, whereas *AdjustableLamp* realizes both the *Light* and *Luminosity* component roles.

2.3.3 The Dedal instantiated architecture assembly level

The architecture assembly level is designed to capture decisions that are made at deployment time during the CBSD process. It corresponds to an assembly of instantiated component classes which have been selected to implement the software. Those components are then called **component instances**. The architecture assembly describes the software at runtime and holds information about its internal state. The assembly lists the component instances and their assembly constraints such as cardinality of connections, etc.

A component instance captures the decision that is made about how a given component class from an architecture configuration is instantiated at runtime. A component instance contains information about its initial and current states, defined and saved in a list of valued attributes.

Figure 2.6 shows the HAS example architecture assembly. This assembly instantiates two *AdjustableLamp* (*lampdesk* and *lampSitting*), one *Clock* (*clock*) and finally one *Orchestrator* (*orchestratorHome*). It is important to note that it is one of the possible instantiations of the configuration.

2.3.4 Dedal formal rules

Another contribution of Dedal is the set of relations that exist between components in each architecture description but also at different abstraction levels and also between architecture levels themselves. In addition to the Zhang's thesis [Zha10], Mokni implemented formal relations that exist between components and architecture levels [Mok+16a] thanks to the B language [Abr96]. Such a formalization makes it possible to perform automatic architecture analysis as discussed in Section 2.2.2.1. The formalization of the ADL is based on the use of type theory [LW94] generalized to components such as defined by Arévalo *et al.* [Aré+07; Abo+09; Aré+09; Abo+19] and defines a set of formal rules for automatically detecting architecture inconsistencies. Those relations are as follows:

- **Component connections**: The connection of components is verifiable thanks to the use of type theory. Mokni *et al.* [Mok+16a] define the concept of compatible component for a connection. This compatibility is calculated on the types of the interfaces that are involved in the connection. To be compatible, two interfaces must have an opposite direction (one must be a provided interface and the other one has to be required), and their types must match. It means that they must have the same type or that the provided interface is a specialization of the required interface. Those rules are applicable for any of the component connections at any of the architecture abstraction levels.

- **Component role realization**: This is the relation that exists between component roles and concrete component classes. This relation prevents refinement inconsistencies since it verifies that a component class is a specialization of its "realized" component role(s). In other words a concrete component class must be a subtype of its designed role to be a realization of this role.

- **Specification implementation**: This is the relation that relates the configuration with the specification. To be consistent with its specification, a configuration needs to define a realization of the component roles that constitute the specification. All the connections that exist into the specification must also be declared into the configuration. In other words, a configuration is a specialization (and thus a subtype) of its specification.

- **Component class instantiation**: This is the relation that exists between concrete component classes defined into the configuration and their component instances defined into the assembly. This relation verifies that component instances are instances of component classes. This relation verifies that component instances satisfy to the constraints defined in component classes (cardinality of connections, etc.).

- **configuration instantiation**: This is the relation that relates a configuration with one of its possible assemblies. As previously, this relation is formalized with type theory which means that it verifies that an assembly is an instance of its configuration. It verifies that all component classes are instantiated into the assembly and that all component connections are also instantiated into the assembly.

2.4 Motivations for re-documenting and versioning architectures

Mokni *et al.* [Mok+16a] address automatic evolution in Dedal using formal rules based type theory. They introduce a way for performing architecture analysis and automatic architecture consistency recovery at three abstraction levels. Doing so, they provide an approach for maintaining a long term evolution support to component-based software architectures. However, experience shows that in many cases, software documentation either does not exist or is not well maintained, which leads to drift and / or erosion of software [DP09]. It therefore appears as essential to be able to recover a documentation of such software in order to re-found long term evolution support. This is what Chapter 4 addresses by proposing a way to re-document three-level component-based architectures based on the Dedal formal rules previously discussed.

Moreover, another key point that has not yet been addressed by the ADL concerns versioning issues of components, and architecture descriptions. This is even rarely addressed concern in the field of CBSE. Still based on formal Dedal rules, Chapter 5 addresses versioning problems in the context of three-level architecture descriptions.

2.5 Conclusion

This chapter introduces the context of this thesis by generally introducing the context of Component-Based Software Engineering to accentuate the focus on Component-Based Software Architecture and then introduce the concepts of the Dedal ADL.

This chapter identifies the main issues of CBSE such as reuse of existing software components that is one of the main advantages of CBSE but still represents numerous challenges. Another issue that is presented in this chapter concerns the management of software architectures which, despite the progress in the field, represents a complex task to perform in particular because of the inconsistencies that may raise during the software life cycle. This chapter therefore introduces Dedal, which is an ADL that proposes to deal with those challenges by representing the entire life-cycle of software but also to perform formal architecture analysis, and automatic architecture evolution based on type theory. However, Dedal (and other approaches) still do not completely cope with those two challenges. This is what motivates this thesis since software evolution needs to address versioning problematics and also re-documentation perspective that can improve maintainability of software even after a long execution period.

Next chapter discusses the state of the art in the fields of component-based architecture versioning and software re-documentation.

Chapter 3

State of the art

Contents

Chapter 2 introduced the context of this thesis by presenting component-based development and component-based software architectures. This chapter relates to issues that concern version and documentation management of software architectures. For that reason, this chapter surveys existing versioning approaches in order to identify which are the most

suitable for versioning component-based software architectures. This study identifies limits of existing approaches in order to help us defining a new approach for component-based software architecture versioning. The second study surveys existing software evolution approaches. It classifies those approaches in terms of versioning management capabilities and implementation. Finally, the third study discusses and classifies software re-documentation and reconstruction approaches. It describes existing approaches and points out their limits. This study therefore help us to define a new re-documentation approach.

3.1 Study on component-based software architecture versioning

Initially, versioning activity came from the need of representing and retrieving the past states of a file through its evolution [EC95]. The existing literature is dense and addresses numerous issues such as difference discovery and characterization between two successive artifact versions. Most of the time, versioning relies on text-based mechanisms [CCL12] such as in very popular and used version control systems like Git [TH10] or CVS [Mor96]. In text-based versioning, deltas between versions are identified through basic operations on text in files. Text artifacts can be added to the file, they can also be deleted or modified which can correspond in some cases to a replacement where a text artifact is deleted and replaced by another one. Those text-based mechanisms are language-agnostic, which means that the differences that are observed do not embed any semantics. However, other approaches exist that come from the Software Configuration Management field. They define version models [CW98] that specify how versions are identified and which are the characteristics that are taken into account for the version identifier computation. Most of the time, version identifiers are $n-$tuples, which are meant to be human readable and give information about versions order. However this kind of information only provides information about artifact anteriority. This is why semantics must be added to this numbering especially in term of impact and compatibility with preexisting artifacts. This state of the art of versioning approaches aims at classifying versioning approaches. To classify component versioning approaches, we aim at answering following questions:

- What kind of identifier is used?
- What is the semantic behind version identifiers?
- How are identifiers attributed?
- Is the concept of backward compatibility formally checked?

In order to classify component-based architecture versioning approaches we aim at answering following questions:

- Is the concept of backward compatibility formally checked?
- Is the concept of version propagation addressed?

This section does not address versioning mechanisms in an exhaustive way. It focuses on approaches that can fit the context of a model-driven component-based software architecture evolution. Thus, it first surveys existing component versioning approaches. Second it surveys model evolution and versioning approaches to identify concepts that can be applied to component-based software architectures. Third it surveys component-based software architecture versioning approaches. Finally, it discusses the surveyed approaches to identify their strength and limits.

3.1.1 Versioning components

Initially, version control mechanisms were introduced in component-based software development in order to avoid recompiling unchanged components. This early change detection was supported by fingerprinting mechanisms such as discussed by Crelier [Cre94]. However this technique only makes it possible to provide the vague information that a change occurred without providing any other information. There exist several software component-based version management approaches which have been widely used over the decades. This subsection focuses on their main characteristics. The outline of this subsection is inspired by Stuckenholz *et al.*'s [Stu05] work.

3.1.1.1 Library interface in Unix systems

In the late 80's Sun Microsystems SunOS added dynamic shared libraries to Unix [Lev99]. This lead Sun to set up versioning mechanisms to support the evolution of the dynamic libraries. In this first versioning system, library versions were identified by two version digits ($X.Y$) for characterizing major (X) and minor (Y) releases. The semantics behind are as follows :

- major version number is incremented if the new version breaks backward compatibility
- minor version number is incremented if the new component compatibility is preserved.

These mechanisms enable the link editor to choose among the available versions, the latest compatible component to use being the component with the same major version number and the highest minor number as possible [Gin+87]. However, this mechanism could not ensure that the new version of the component would run on an earlier minor release level such as presented by Brown *et al.* [BR00b]. Thus Sun refined their versioning mechanism into an ELF-based binary format that contains libraries and executables. This new mechanism relies on the decoration of shared library symbols such as methods. Decorated symbols are then considered as required by the new version, thus the linker can dynamically search into libraries and more reliably identify compatible components. This mechanism makes it possible for Unix systems to maintain the parallel existence of multiple component versions. However, as the information about library version is not automatically attached it is subject

to the interpretation of the developer who releases the new version. Mistakes can occur during the versioning process which can be harmful for the system.

3.1.1.2 CORBA

CORBA component model is based on the Interface Description Language (IDL). Components are specified in IDL and their descriptions are mapped into a target programming language [OMG+02]. The change impact mechanism inherent to CORBA makes any change in a component visible into its IDL specification. However, CORBA does not embed any mechanism for managing component evolution and components cannot be enhanced by version information. Actually it is possible to define several versions of a component but they are actually separated by an explicit naming convention (version number is suffixed to the component type identifier). It means that a new version of a component is considered as a completely new component. This implies that all the components which depend on the old version must be rebuilt to use the new version.

3.1.1.3 Windows Dynamic Link Libraries (DLL)

In order to manage their libraries, Microsoft introduced the concept of DLL to their operating system. This concept allowed to dynamically load required libraries at runtime. However, at the beginning, the Microsoft's DLL concept did not embed any versioning mechanism. This led to numerous system crashes since applications could replace older DLL versions that could still be used by other applications. This problematic behavior was called DLL-Hell [Dev99; Pra01]. In next evolutions of Windows 98 SE and Windows ME operating systems, the possibility was added to better control the linkage of those DLLs through the mechanism of isolated applications. This isolation made possible to create and deploy COM components on one or more specific applications, avoiding the DLL-Hell problem. However, versioning mechanisms have properly been implemented only from Windows XP where component versions were specified into a manifest file. Thus applications can dynamically load specific versions of DLL components. In order to manage versions, Microsoft chose a version identifier composed of four digits $X.Y.Z.R$ where X is incremented for major releases that break backward compatibility, Y for identifying minor versions and, despite DLL versions are linked with X and Y, the two last digits correspond to Z the build number and R the revision number. Those two last digits enable quick fix engineering processes for a faster bug resolution [Bey01]. However, those labels are still attached manually which means that they are dependent of developer's assumptions.

3.1.1.4 COM/.NET

COM and .NET components have been introduced by Microsoft. COM component model came first and in order to avoid DLL-Hell problems discussed in 3.1.1.3, Microsoft policy was to forbid changes in existing components. Thus a COM component interface is given an unique identifier and instead of performing changes in the interface, a new interface needs to be created which is given a new interface identifier. This way, DLL-Hell is

avoided but it is impossible for clients to discover new component features without rebuilding them [Rog97]. With the upcoming of the .NET framework, Microsoft introduced the idea of assemblies which can be versioned. The version identifier is a four-digit number that is manually set by the developer. There exist two types of assemblies. Private assemblies only have a limited scope to isolated applications, and shared assemblies can be deployed to the Global Assembly Cache (GAC) to be shared by all the system applications [Low05]. Shared assemblies are identified by a strong name which corresponds to a unique identifier of the assembly. Thus multiple versions of shared assemblies can coexist, however there is no default mechanism for checking assemblies compatibility at runtime. Yet, approaches such as the one developed by Eisenbach *et al.* [EJS03] make it possible to check inconsistencies into assemblies, specifically when a component is upgraded. However, every component change mechanism relies on developers which can still lead to mistakes.

3.1.1.5 Java

Component-based technologies JavaBeans [Javb; Eng97] and Enterprise Java Beans (EJB) [Sha+01; RSB04; BMH06; MH97] are based on the Java language. As Java provides reflection mechanisms in its concepts, it is possible to analyze components at runtime in terms of type and exported interfaces. The default class loader mechanism prevents the system from loading several component versions at the same time. However, this default behavior can be changed by the implementation of hierarchical class loaders. In Java, version management has been introduced to handle object serialization. The serialization mechanism enables object persistence and their transmission between distant applications through Remote Method Invocation (RMI). In the context of distributed systems, where RMI mechanisms are used, a compatible change is defined as a change on a class which still allows to unserialize (using the new class version) data-streams that have been serialized from an older version of the class. Using custom class loaders also allows the definition of versioning policies. However, no standard rules exist to implement them which can lead to a loss of portability of a component to another system. Moreover, it is not mandatory to enhance component description with version information and no default version policy exists. Thus users have to analyze the components they want to use. However, Java does not provide any mechanism for automatically calculating the type of changes that occurred in components. In addition, more recent Java releases (from 9) introduce the concept of modules. However, version management still relies on tooling and frameworks (*i.e.*, OSGi) instead of internal Java mechanisms [MB17].

3.1.1.6 McCamant and Ernst approach

In their work, McCamant and Ernst [ME04] focus on semantic changes in components and their impacts on what they call their operational abstraction. They define the term of operational abstraction as follows: "An operational abstraction is a set of mathematical properties describing the observed behavior. An operational abstraction is syntactically identical to a formal specification – both describe program behavior via logical formulas over program

variables – but an operational abstraction describes actual program behavior and can be generated automatically."[ME04] By comparing the operational abstraction of old and new component versions, it is possible to automatically detect incompatibilities during the upgrade of a component and thus formally drive the evolution process of deployed component versions. However, this approach only addresses behavioral changes and do not address the source code change.

3.1.1.7 Brada and Bauml approach

Premsyl Brada's approach is based on the Architecture Description Language (ADL) named SOFA [PBJ98] which is discussed in 3.2.7. In his work [Bra99; Bra01a; Bra01b], he designed a scheme for automatically identifying component version backward compatibility thanks to automated tests using ELF-based component descriptions. The rules on which are based compatibility or incompatibility calculations are directly derived from type theory and subtyping rules. Thus, as he formalized component version compatibility, the automated version identification which is made after is much more reliable than manual approaches. Finally in further work (out of the SOFA approach) on automated versioning he defines first with Valenta [BV06] and after with Bauml [BB09; BB11] an automated versioning approach applied to OSGi world. In this work, authors question the reliability of component version identifiers and then define a pattern for characterizing and identifying component version and automatically assign them version numbers. Type analysis is the basis of version identification as in SOFA. They reuse the common version numbering $X.Y.Z$ and define strict rules for incrementing those numbers by analyzing type differences. If no type difference has been found then Z is incremented since the new type is compatible with the previous one. If the new type is a specialization of the old one, thus the change is compatible with the old version and Y is incremented. Finally, in case where the new type is a generalization or a mutation (*e.g.*, it is not comparable anymore to the ancient type) then it means that the new version is no more compatible with the old one and then X is incremented. This way, component version numbers are set automatically which ensures strong understanding of component version differences by developers that use and deploy them. This automated version identification is also useful for component upgrading at runtime for ensuring software consistency.

3.1.2 Model evolution and versioning

Considering that ADLs are defined by metamodels / grammars, versioning an instance of an ADL metamodel / grammar amounts to the versioning of the described architecture.

In the context of software engineering, versioning plays an important role since it maintains an historical archive of software past states and it also supports parallel evolution of artifacts by teams [Est+05]. These concerns are transposable to the field of MDE. However, due to the graph-based structure of models, already existing code versioning techniques are not suitable since they mostly rely on text-based mechanisms. Indeed, text-based versioning mechanisms fail at taking into account model structural information such as containment

references and multiplicities between several model artifacts. Thus it appeared that graph-based approaches were needed to manage model versioning.

This is why model evolution and versioning has been widely influenced by the huge experience in the field of database schema evolution. Indeed because of early needs to keep track of databases evolution, this field has already addressed numerous challenges that can be also identified in the field of model evolution and versioning. For instance, lot of work addressed the versioning of object-oriented database schemata from 80's [KCB86; Zdo87; BM88; CK88; Kat90; Lam92; TO93; UO96; UO98]. This came from the deep need of ensuring that the successive versions of a database schema would not be incompatible with previous ones and thus would ensure the consistency and coherency of schema with stored data. This research is thus highly based on the concept of type substitution that is inherent to object-oriented systems. Moreover, version models such as Iris [BM88], Encore [Zdo87], Lincks [Lam92], Mosaic [Lan86], Orion [CK88], Version Server [KCB86; Kat90] and the model proposed by Oussalah *et al.* [OTC93], Talens [TO93] *et al.* and Urtado *et al.* [UO96; UO98], propose mechanisms for propagating versions. Those mechanisms aim at automating version management by propagating version creation and destruction operations following dependency relations between versioned artifacts.

This section is organized as follows, first it presents model evolution approaches. Second it discusses version propagation in models.

3.1.2.1 Versioning models

A very common model versioning scenario involves parallel modifications of an unique artifact version v_0. The artifact has now a set of new versions $\nu = v_1, v_2, ..., v_n$ where n is the number of parallel versions of the artifact. Thus model versioning processes aim at consolidating and merging this set of versions into a unique version v_0 [PMR16].

As it has been identified by Altmanninger *et al.* [Alt+08; ASW09; Alt+09], a model versioning process occurs following three steps:

- **The change detection phase.** This is the phase where the changes that occurred between v_0 and the set of modified versions ν are detected and identified. This phase can be realized following two different types of approaches [PMR16]:
  - *State-based detection:* in a state-based detection approach, only the final states of the modified versions is taken into account for identifying model changes. Those approaches only support few operations which are additions, deletions and changes.
  - *Operation-based detection:* in an operation-based detection approach, the version history relies on the model editor which has to save all the operations that have been performed on a model. Those approaches make it possible to keep a record of the operation sequence that leads to a new version of an artifact such as introduced by Herrmanndoerfer *et al.* [HBJ09; Her09]. This can be useful for reverting

changes. However, those kinds of approaches are often editor dependent and language specific.

- **The conflict detection phase.** In the context of parallel versioning, conflicts may arise. For instance, in some cases, parallel changes are potentially overlapping or contradicting. Conflicts are detected by comparing all the changes that occurred on a model artifact so the overlapping contradicting ones are identified in order to be resolved. There are two ways for resolving a conflict:
  - *Manual approaches:* In those kinds of approaches, the user manually resolves conflicts between versions. Those kinds of techniques applied to modeling artifacts can be very challenging. Alanen and Porres [AP03] lead a seminal work on how to cope with identifying, classifying and reconciling conflicts. For textual conflicts, the conflicting versions are presented side by side to the user who chooses which action must be taken for resolving the conflict. However, in the case of model artifacts, the resolution can be challenging due to the nature of models that in many cases generates conflicts which cannot be resolved independently [Bro+12].
  - *Automatic approaches:* Another way for resolving conflicts is to automatically calculate all possible combinations of operations that can lead to a valid version. Thus, Cicchetti *et al.* [CDP09; CCL12] propose to define conflict patterns and resolution strategies through the use of a domain specific language (DSL) that they define. Moreover, they define patterns that resolve syntactic as well as semantic conflicts. Finally, they also define a versioning policy which still requires user intervention for cases where no policy is defined. In order to fully automate the process, Ehrig *et al.* [EET11] formalized a conflict resolution strategy. This strategy is especially tailored for conflicts that occur on graph modifications. As this is a formal approach, the obtained model is considered as consolidated (all the conflicts have been resolved) by construction. Finally, another approach to automatically address conflicts is to temporarily tolerate them. This is what is proposed by Nuseibeh *et al.* [NER01] who argue that it may be beneficial since those tolerated conflicts highlight parts of models that need to be further investigated and improved.
- **The inconsistency detection phase.** Inconsistencies may happen while merging concurrent versions of a model artifact. Those problems usually take place when the consolidated version violates metamodel validation rules. Those inconsistencies are in many cases resolved by users themselves. However Reder and Egyed [RE12] proposed a fully automated approach. Unfortunately this solution is language-specific.

In order to manage those model versions, several versioning systems have been proposed that use different combinations of discussed techniques. In their work on versioning UML models, Stevens *et al.* [SWB03] introduce algorithms for calculating differences, merging those differences and resolving conflicts. The differences are calculated from matching two

model versions and the unique identifiers of their elements. The approach is meta-model independent and is able to identify additions, deletions and changes of model elements. EMFStore has been proposed by Koegel *et al.* [KHS09] in the Eclipse[1] Modeling Framework (EMF[2]) ecosystem. It is a model repository which provides model version support. EMFStore is an operation-based tool which relies on the Eclipse IDE. Once modifications on the model are done, they are committed to the repository to save the new state of the model. Odyssey-VCS 2 that has been proposed by Oliveira *et al.* [OMW05] is a language specific version control system based on the UML language. As the tool of Stevens *et al.*, it is state-based and also takes advantage of model elements' unique identifiers. On the basis of the difference detection phase result, the tool automatically infers the operations that lead to the last version of the model. The tool is also able to raise conflict warning in case of contradictory changes. However, no inconsistency detection is performed after merging is done. Last but not least, Altmanninger *et al.* proposed their tool named AMOR (Adaptable MOdel veRsioning) [Alt+08] for managing model versioning. The tool provides capabilities such as an extensible conflict detection mechanism and resolver components which goal is to guide users during conflict resolution. Collaborative conflict resolution policies can also be supported by AMOR.

3.1.2.2 Models and metamodels co-evolution and version propagation

Systems may be defined not by a single model but by a set of models that document different abstraction levels or viewpoints. These models are then inter-related and versioning one model of the system may have an impact on its other representations. In the context of an ADL that documents software at each step of its life-cycle (see Chapter 2), the representation which is made at specification time can be considered as a metamodel of the one that is produced at implementation time. The same relation exists between the representation that is produced at implementation time and the one that is produced at deployment time. Indeed, those abstraction levels correspond to MOF[3] M0, M1 and M2 levels. In other words, such an ADL is described by a metamodel / grammar and an instance of its metamodel / grammar contains several abstraction levels. Moreover, in CBSD the implementation is influenced by choices that are made at specification time but the specification might also be influenced by choices that are made at implementation time (*e.g.*, component available for being reused). This is why top down and bottom up mechanisms are needed for propagating artifact versioning. Horizontal changes propagation are also needed in order to adapt each representation level (*e.g.*, a change at implementation time may influence other artifacts of the implementation). Research on model evolution and versioning has brought various approaches for managing co-evolution [PMR16]:

- **Manual approaches** such as Ecore2Ecore [HP06], Epsilon Flock [Ros+10; Ros+14] or Taentzer *et al.* [Tae+13] manually migrate models in order to make them compliant with their updated meta-model.

[1] https://www.eclipse.org
[2] https://www.eclipse.org/emf
[3] https://www.omg.org/mof/

- **Operator approaches** such as COPE / Edapt [Her09], MCL [Nar+09], Demuth *et al.* [Dem+16] or Rumbaugh *et al.* [RJB04] are based on patterns and are characterized by a set of predetermined strategies which can handle a step-by-step co-evolution of meta-models and models.
- **Inference approaches** such as Cicchetti *et al.* [CDP09] or AML [Gar+09] rely on meta-model comparison to generate a strategy for evolving models in order to conform to their updated meta-model.

Propagating changes to models from metamodels is the only way in the literature for performing version propagation in models. All those approaches are top down approaches that adapt models to their changed metamodels. None of them copes with a bottom up evolution approach or with inner model change propagation. This is due to the nature of models which cannot break the rules that are described by their metamodels. Yet the inconsistency detection phase discussed in 3.1.2.1 can be assimilated to horizontal version propagation, especially in Reder *et al.* approach [RE12] that defines validation trees for calculating the impact of changes. Moreover, all the discussed approaches are based on tooling and do not consider the semantics of model artifacts. Thus, although the concept of version propagation is interesting for versioning software architectures, it cannot be performed by using model-based approaches.

3.1.3 Versioning component-based software architectures

Only few work copes with versioning component-based software architectures. The few approaches that are presented here propose only basic mechanisms for architectural versioning that do not take into account the entire life-cyle of the software.

SOFA 2 [PBJ98; HP04; BHP06] which is discussed in 3.2.7 provides a way for formally define component-based architectures and manage their evolution. The language gives also the ability to calculate component version backward compatibility as discussed by Brada *et al.* [Bra99; Bra01a; Bra01b]. However, even if SOFA 2 describes multiple abstraction levels, and even if Brada [Bra03] describes a mechanism to propagate differences in order to recursively discover component differences, the concept of change propagation and version propagation is not addressed.

Mae [Ros+04] which is discussed in 3.2.6 is based on xADL 2.0 [DHT05] (discussed in 3.2.5) that provides two abstraction levels by distinguishing design-time and run-time. However, even if Mae introduces some mechanisms for calculating component compatibility and backward compatibility, this approach do not address any kind of version propagation.

Amirat *et al.* [ADO14] proposed a generic approach for evolving and versioning component-based software architectures using ATL transformations. However in their approach they do not address the key concept of version backward compatibility.

3.1.4 Discussion

This section covers a very wide research area in component, model and component-based architecture versioning. Table 3.1 summarizes the surveyed component versioning approaches. It appears that many works have already addressed the versioning of components and especially in the context of libraries that can be dynamically linked and used by third party applications which require them. Making applications use the last compatible component versions has been addressed in several ways. Those approaches sometimes failed at enabling several component versions to coexist. When approaches succeeded to manage multiple component versions, a naming convention is designed to help developers and automated processes to choose among versions for the last compatible one. To do so, a common template based on digit tuples is meant to carry information about the type of component versions. Nowadays, the most used naming pattern is described in the Semantic Versioning 2.0.0[4] approach and is very similar to what is proposed by Brada Valenta and Bauml [BV06; BB09; BB11]. It consists in identifying any versionable artifact with a triple $X.Y.Z$ where X represents the major release number that is incremented when the new version of the artifact is not backward compatible. Y is the minor release number. It is incremented when the new version is backward compatible. Finally Z is the build number that correspond to minor versions that may for instance correspond to bug fixes (the API and the observable/external behavior of the artifact is not impacted by the changes). The only addition of Semantic Versioning is the use of labels that can be passed as suffixes to provide additional information to developers. However, no mechanism ensures that version numbers are well identified and the semantics associated to the tags may be null and void. Indeed, most of the time those identifiers are set by developers themselves and mistakes can occur. This is why most of the surveyed approaches fail at ensuring component version backward compatibility. Only Brada and Valenta [BV06] propose a suitable approach based on SOFA [PBJ98] that formally identifies backward compatible components. Later Brada and Bauml [BB09] proposed to automate version identification using $X.Y.Z$ pattern in the OSGi world. As their methods is based on strict type-based rules, they ensure by design that the computed identifier conveys the right semantics. Finally, despite the Semantic Versioning 2.0.0 approach perfectly fits a human readable form and proposes a sound numbering scheme regarding backward compatibility, it does not explicitly identify the intention of the evolution, i.e. distinction between versions that are designed as revisions or variants. A variant is a version that is intended to co-exist with other versions of an artifact in order to provide alternative feature configurations (product line engineering). A revision is a version that is intended to replace the previous versions of an artifact (deprecation). As variant and revision semantics is orthogonal to backward-compatibility (revisions may not be backward compatible in order to allow re-engineering or technological breakthrough for instance), a specific scheme is needed to identify them.

In the field of MDE, a lot of research also exists that aims at maintaining, evolving and versioning models. To do so the domain has taken advantage of the already existing vast

[4] https://semver.org/

Approach	Identifier	Automatically incremented	Incompatibility check	Backward compatibility check
Unix	$X.Y$	–	–	–
CORBA	–	–	–	–
DLL	$X.Y.Z.R$	–	–	–
COM/.NET	$X.Y.Z.R$	–	–	–
Java	–	–	X	–
McCamant and Ernst	–	–	X	–
Brada and Bauml	$X.Y.Z$	X	X	X

TABLE 3.1: Component versioning approaches

research in the area of database schema evolution. Yet, some problems related to versioning are not addressed in the field of MDE. For instance compatibility of models is not addressed in a suitable way for components. Indeed, only the compliance of models to their metamodels is analyzed / verified. However a model can evolve in an ecosystem in relation with other models using it (for instance a model could be used by another in a transformation process). Yet no mechanisms for checking the compatibility of changes with the ecosystem seems to be addressed and then it gets difficult to ensure that a new model version is substitutable for its older version as this can be done for instance with Dedal architectures. It seems then very difficult in this context to formally identify model versions. Moreover, despite it is addressed in the field of databases schema versioning, the concept of version propagation is not well identified and addressed in the context of MDE. A lot of work deals with co-evolution of models but not directly with version propagation. The only activity of model versioning which can be assimilated to version propagation (or at least change propagation) is the inconsistency detection phase. Unfortunately, in this phase the change propagation occurs in a top down way. Actually, only changes that occur at metamodel level can be propagated to the models. However in a component-based software architecture evolution where, as in Dedal, several abstraction levels coexist in order to represent the global life-cycle of software, a change can occur at any of these abstraction levels and still may need to be propagated in a bottom up way as do the versions.

Table 3.2 summarizes approaches that address the concept of architecture versioning. Surprisingly, the field of component-based software architectures does not address component and architecture versioning in a very extensive way. The only approach which truly addresses component versioning is the one developed by Brada *et al.* [Bra99; Bra01a; Bra01b; Bra03; BV06; BB09; BB11], based of the SOFA ADL [PBJ98]. However, despite the concept of component version is very well developed, the approach does neither mention the versioning of architecture themselves nor mechanisms of version propagation within the SOFA architecture abstraction levels.

It appears that component-based software architecture field lacks semantics in component and architecture versioning. Within missing concepts, we identified no existing semantics for versioning CBSAs and thus, that version propagation is not addressed in CBSA field.

approach	backward compatibility check	version propagation
SOFA 2	X	–
MAE / XADL	–	–
Amirat *et al.*	–	–

TABLE 3.2: Architecture versioning approaches

In order to fill the gap, this thesis proposes component versioning mechanisms inspired by Brada *et al.* by identifying component backward compatibility from types and automatically increment their version identifiers following type differences. Moreover, this thesis proposes to extend such mechanisms to version entire architectures. Finally, it proposes to version multi-leveled software architectures by performing a change impact analysis to ensure the global architecture definition consistency.

Next section discusses architecture evolution approaches to identify an approach that meets versioning needs.

3.2 Architecture evolution approaches

The existing literature about architecture evolution is very abundant. This section only discusses the approaches which are the most relevant with this thesis (*i.e.*, component-based software architecture evolution approaches that propose formal evolution mechanisms). This survey intends to evaluate the capabilities of existing approaches to document software architecture evolution and to determine whether they provide a support for architecture versioning or not. To do so, we try to answer the following questions:

- What aspects of architecture life-cycle and evolution are supported by the approach? How many abstraction levels does the approach provide to model architectures?
- What paradigm does the approach use to model architectures? Are all the abstraction levels expressed with the same formalism?
- Does the approach provide a formalism that allows to derive specialization / substitutability rules between components / architecture levels?
- Does the approach supports component / architecture versioning? Does it provides version semantics?

3.2.1 C2 / C2-SADEL

C2-SADEL (Software Architecture Description and Evolution Language) has been proposed by Medvidovic *et al.* [MRT98; MJ06] on top of the C2 approach [Tay+96] which defines C2-style architectures. C2-style architectures are component- and message-based architectures especially tailored for GUI and distributed applications. Their dedicated ADL, C2-SADEL, provides sub-typing mechanisms that support architectural evolution. Moreover, this ADL provides two architecture points of view by clearly identifying component instances and

types. Thus it supports two abstraction levels expressed in a component-based paradigm that correspond to:

- the deployment of the software through the description of component instance and their connectors / dependencies
- and the implementation of the software through the description of the component classes and their connections.

Moreover, the C2-SADEL approach is based on strong subtyping mechanisms derived from Object-Oriented subtyping rules. Those subtyping mechanisms are embodied by the use of the Z language which is a first order logic language based on the Zermelo-Fraenkel Set Theory [Hay+68]. Thus this ADL also supports, by extension of its subtyping rules, formal architecture analysis such as detecting incoherency in component connections. However, it does not address the versioning problem and thus does not propose any semantics for identifying and producing component / architecture versions.

3.2.2 Darwin

Darwin [Mag+95] is an ADL which semantics are defined by $\pi-$calculus [MPW92]. It is designed to specify the structure of distributed systems and describes components through the use of a hierarchical decomposition scheme. Darwin only represents the implementation of the software. The $\pi-$calculus formalism enables analyze the architecture and guarantees the correctness of the component connections. Darwin also provides mechanisms to analyze the impact of architectural changes, however, in this approach, component / architecture versioning is not addressed and no semantics are proposed for identifying and producing versions.

3.2.3 Wright / Dynamic Wright

Dynamic Wright [ADG98] extends the previous Wright [AG97] ADL. This ADL especially supports evolution of distributed architectures and analyzes their behavior. Wright only provides one abstraction level through architecture descriptions which mixes implementation and deployment information. Indeed, a Style (architecture description) is defined by a list of components types, connector types and their instances. The concepts of implementation and deployment are not clearly decoupled. Wright models are based on the concept of CSP (Communicating Sequence Processes) [Hoa78] which formalizes component behavior and allows architecture analysis. In Dynamic Wright, only the implementation is represented. Finally, the concept of version is not addressed in the approach and thus no semantics are defined for component / architecture versions.

3.2.4 ArchWare

ArchWare [Oqu+04] is an European project which provides a set of languages and tools for engineering and evolving software system architectures. Architectures are modeled with

$\pi-$ADL which is the language that is proposed by the project. This language is also based on the $\pi-$calculus [MPW92]. $\pi-$ADL only represents the deployment of software by providing a single abstraction level in which component instances and their links are represented. ArchWare also provides an Architecture Refinement Language ($\pi-$ARL) which gives the ability to perform architecture change analysis since it relies on the $\pi-$calculus. However, this approach still does not address the versioning activity and does not provide any semantics for versions.

3.2.5 xADL

xADL [Kha+01] is an approach that aims at enabling architecture centric tool integration with XML. This ADL is based on C2-style architectures and enables the definition of component types and instances as well as connectors and connections to describe the implementation and the deployment of software. The xADL / xC2 Document Type Definition (DTD) of the language takes the concept of component and connectors compatibility into account. However, the concept of architecture compatibility is not addressed. Finally, the approach addresses the concept of version by making it possible to define independent version graphs for component, connectors and interface types. However, those mechanisms are not based on a strong typed semantics that would give information about version backward compatibility.

3.2.6 Mae

Mae [Ros+04] is an approach which is built on top of the xADL ADL. It provides an additional set of rules for checking compatibility among architectural elements. Those rules capture backward compatibility of architectural elements such as components and connectors with their predecessors. However, despite this upgrade that has been done to xADL, the concept of architecture version and architecture backward compatibility is still not addressed.

3.2.7 SOFA 2.0

SOFA 2.0 [HP04; BHP06] is an evolution of SOFA [PBJ98] which relies on the MOF (Meta Object Facilities) concepts. This language is designed to support distributed applications and distributed runtime environment. The language proposes two abstraction levels through one architecture description that describes the implementation and instantiation of software. Versioning is taken into account and histories of interfaces and components are decoupled. It provides mechanisms for checking component compatibility which can be used for taking backward compatibility of components into account such as discussed by Brada [Bra99] where versioning is made from the change analysis. Moreover, as the concept of architecture is modeled as composite component, the versioning of architectures can be taken into account and backward compatibility can also be calculated with predecessors.

3.2.8 Synthesis and comparison

The characteristics of studied adls are summarized in Table 3.3. Most of the approaches coping with software architecture evolution make use of ADLs for modeling architectures. Within those approaches, only a few are based on formal languages / concepts that make them component-compatibility aware. In the few approaches which have been discussed in this section, C2-SADEL, Darwin, Dynamic Wright and Archware enter into this category. Indeed, all of them are based on formal languages / concepts such as Z or $\pi-$calculus. However, despite the capabilities they provide for checking architecture consistency and component capabilities, they do not provide versioning processes for their components / architectures. Thus those approaches do not define component / architecture version backward compatibility. xADL neither addresses the concept of version backward compatibility even if its XML-based implementation of C2 (xADL / xC2 DTD) makes it aware of the concept of component compatibility (without capabilities for checking it). However, its evolution Mae, adds a set of rules for adding some semantics to the way components, connectors and interfaces are versioned. Finally, SOFA is based on Object-Oriented typing rules which make it aware of component / architecture compatibility and able to support version backward compatibility discovering. At the end, despite some encouraging approaches, none of them intends to formally represent the entire software life-cycle such as introduced in Chapter 2. Thus, in order to develop our approach, we choose to use the Dedal ADL, which is introduced in Section 2.3 since it provides all the mechanisms that are needed to version multi-leveled software architectures.

Approach	Architecture Modeling				Architecture versioning support			
	ADL	CBSD support			Component versioning	Architecture versioning	Compatibility check capabilities	Backward compatibility support
		Specification	implementation	deployment				
C2 / C2-SADEL	C2-SADEL	X	X	–	–	–	X	–
Darwin	Darwin	–	X	X	–	–	X	–
Wright / Dynamic Wright	Dynamic Wright	–	X	–	X	–	X	–
ArchWare	$\pi-$ADL, $\pi-$ARL	–	X	X	–	–	X	–
xADL	xADL	–	X	X	X	X	–	–
Mae	Mae	–	X	X	X	X	X	–
SOFA 2	SOFA 2	–	X	X	X	X (composite components)	X	X

TABLE 3.3: Software evolution approaches: versioning integration

3.3 Retrieving architecture documentation and software maintainability

Architectures are a great way for representing software at a high-level of abstraction. Following Garlan [Gar00] software architectures are important in several software development aspects. They emphasize understanding, reuse, construction, evolution, analysis and management of software. However, in many cases this abstract documentation either does not exist, has been lost (the existing architecture is no more compliant with the actual one) or its quality is very poor [SAO05]. The maintainability of software is then threatened and it is necessary to retrieve this documentation in order to recover a suitable evolution and maintainability context. In literature, two types of approaches exist. First of all strict re-documentation approaches that aim at only recovering software documentation as it is (Section 3.3.1). As defined by Chikofsky *et al.* [CC90], "redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level". Thus, it consists in transforming the information contained in source code (and possibly other documents) into an updated documentation about code. Secondly, reconstruction approaches (Section 3.3.2) aim at recovering software documentation but also to interpret the way the software is implemented in order to perform some re-engineering. This section covers both approaches.

3.3.1 Software re-documentation approaches

As stated by Chikofsky *et al.* [CC90], re-documentation is the primal form of reverse engineering and is widely considered as a non-intrusive way for restructuring code information. There exist several approaches for re-documenting software architectures. Among those approaches we can cite XML-based approaches such as introduced by Hartmann *et al.* [HHT01]. We can also cite incremental approaches introduced by Rajlich, Václav [Raj97; Raj00] which produce text-based documentation. The Island grammar approach [VDK99; Moo01] relies on a parser to analyze and textually re-document code. In the DMG (DocLike Modularized Graph) Sulaiman *et al.* [SIS03] provide an advanced visualization graph as software documentation. Finally and surprisingly, very few work has been lead on producing model-oriented documentation.

One of the only model-oriented re-documentation approach has been proposed by Feng and Hongji [CY07]. They base their approach on the Java language and use its object-oriented paradigm to map its concepts with the OMG standards. They propose a tool, MOREDOC (Model Oriented REDOCumentation), that automatically re-documents Object-oriented software, taking advantage of MDE concepts. This re-documentation approach is the closest to context that is introduced in Chapter 2. However, it does not address component-based software architecture. Next section discusses architecture reconstruction approaches where more work has been led for component-based software architectures.

3.3.2 Software architecture reconstruction approaches

Many works have been led for reconstructing software architectures. This section focuses on the most similar to our proposed approach: approaches which extract component-based based architectures descriptions have been targeted. Moreover, retro-engineering approaches that aim at retrieving initial design decisions are differentiated from re-engineering approaches which reorganize the extracted information and / or software artifacts. Table 3.4 gives an overview of the approaches that have been considered.

This section is organized following the five criteria that have been defined by Ducasse *et al.* [DP09] in their taxonomy. Those criteria are as follows:

- The **goals** of the approach which can be:
  (a) *Redocumentation* is probably the universal goal of every software architecture reconstruction process since they are used during software's life-cycle which probably suffered from drift of erosion during its evolution.
  (b) *Reuse* (*i.e.*, ARES [Eix+98], MAP [SO01], PuLSE / SAVE [Kno+06] and ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]), since recovering architecture descriptions may highlight reusable entities as software components, frameworks, etc.
  (c) *Conformance* checking (*i.e.*, Bauhaus [Kos02; EKS03; CKS05], DiscoTect [Yan+04], PuLSE / SAVE [Kno+06] and Tran *et al.* [TH99]) for comparing the conceptual description of the software and its actual implementation.
  (d) *Co-evolution* (*i.e.*, PuLSE / SAVE [Kno+06], Huang *et al.* [HMY06] and Tran *et al.* [TH99]) for maintaining several levels of abstraction at the same time (typically an architecture and its implementation) and avoiding drift and erosion.
  (e) *Analysis* (*i.e.*, PuLSE / SAVE [Kno+06] and Huang *et al.* [HMY06]) that can be made on architectural views, such as quality analysis to assist architects in their decisions.
  (f) *Evolution and Maintenance* that can be made easier by software architecture reconstruction.

  Our approach aims at improving component and architecture reuse, by extracting component-based multi-leveled architecture descriptions for the Dedal [ZUS10; Mok+16b; Mok+16a] ADL, but also at giving capabilities for managing conformance checking, evolution, co-evolution and maintenance using the formal rules that have been defined in Dedal.
- The kind of **process** that is used:
  (a) *Bottom-up* approaches (*i.e.*, ARES [Eix+98] and ArchVis [Hat04]) use the lowest-level information for creating the model.

(b) *Top-down* approaches (*i.e.*, PuLSE / SAVE [Kno+06]) start with the highest-level information. Hypotheses are made which are next checked to make sure that they comply with the source code.

(c) *Hybrid* approaches (*i.e.*, Tran *et al.* [TH99], X-Ray [MK01], MAP [SO01], Alborz [Sar03], Focus [DM01; MJ06], Bauhaus [Kos02; EKS03; CKS05], DiscoTect [Yan+04], Pashov *et al.* [PR04], Huang *et al.* [HMY06], ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]) aim at combining *bottom-up* and *top-down* approaches. In those kinds of approaches, hypotheses are made that are refined using what is abstracted from low-level information.

We define our approach as bottom-up since it is exclusively based on source code and deployment descriptor files that are present in projects.

- The **inputs** of the software architecture reconstruction:

  (a) *Non-Architectural* inputs which may be the source code but also textual information like comments in the code (*i.e.*, ArchVis [Hat04]), dynamic information like logs and execution trace (*i.e.*, Alborz [Sar03], ArchVis [Hat04], Bauhaus [Kos02; EKS03; CKS05], DiscoTect [Yan+04], Huang *et al.* [HMY06] and Pashov *et al.* [PR04]). However those inputs may also be other kinds of information like the physical organization (*i.e.*, ArchVis [Hat04]) of the software. Human organization may also provide information about coding standards in a company that can make extraction easier and extraction rules more specific. *Non-architectural* inputs may also involve historical information and human expertise. Historical information can be used for instance for improving understanding of extraction results but is rarely used [DP09]. Besides human expertise is used in most of the software architecture reconstruction approaches since it is most of the time needed over reconstruction iterations for validating results. Human expertise is useful for the quality of the extraction but alter the automation of the process.

  (b) *Architectural* inputs like architectural styles or viewpoints for guiding extraction.

  Our approach is based on source code analysis coupled to the exploitation of primitive architectural information implemented by the Spring technology.

- The **techniques** that are proposed:

  (a) *Quasi-manual* (*i.e.*, MAP [SO01]) where the tool only assists the engineer in understanding the extracted information.

  (b) *Semi-Automatic* (*i.e.*, Tran *et al.* [TH99], DiscoTect [Yan+04] and PuLSE / SAVE [Kno+06]) where the software engineer guides the tool.

  (c) *Quasi-automatic* (*i.e.*, X-Ray [MK01], Alborz [Sar03], Bauhaus [Kos02; EKS03; CKS05], ArchVis [Hat04], Pashov *et al.* [PR04], Huang *et al.* [HMY06] and ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]) that are close to be fully automatic but still requires human expertise to ensure that the tool is going to the

right direction.

As Dedal [ZUS10; Mok+16b; Mok+16a] has been formalized and allows to perform automatic evolution calculation, we aim at providing an automatic extraction tool that would allow to fully automate the software architecture reconstruction process.

- The **outputs** that may be:
  (a) *Visual* software representations are the most common result of software architecture reconstruction processes.
  (b) *Architecture* description (*i.e.*, ARES [Eix+98], X-Ray [MK01], DiscoTect [Yan+04], ArchVis [Hat04], Huang *et al.* [HMY06] and ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]) for providing computable architectural information.
  (c) *Conformance checking* can be vertical (*i.e.*, Tran *et al.* [TH99], Bauhaus [Kos02; EKS03; CKS05] and PuLSE / SAVE [Kno+06]) for verifying whether the recovered architecture comply with the implementation, horizontal (*i.e.*, Huang *et al.* [HMY06] and ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]) for checking conformance between two architectures (conceptual vs concrete, two reconstructed views...) or both (*i.e.*, DiscoTect [Yan+04]).
  (d) *Analysis* (*i.e.*, ARES [Eix+98], Alborz [Sar03], PuLSE / SAVE [Kno+06], Huang *et al.* [HMY06]) that can be the result of any kind of analysis over the reconstructed architectures such as quality analysis (*i.e.*, ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]).

  Our approach produces a three-level formal Dedal architecture description and visualization thanks to the ADL.

However, all of those methods present several limitations. Indeed, they do not reconstruct architectures as they are implemented but perform some re-engineering. In addition, all of them deal with only two abstraction levels (implementation and deployment). Yet, those two levels might not correspond to the same paradigm (code Vs component-based architecture description) which is a drawback to co-evolve those abstraction levels. Then it is essential for maintaining, evolving and tracking software life-cycle to have three-leveled component-based architecture descriptions. Indeed, they give a more global but precise understanding to architects by providing a means to model architecture design, implementation and deployment decisions separately. Finally, even the approaches that seem close to ours either perform re-engineering such as ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17] that clusters classes into bigger components or DiscoTect [Yan+04] that reconstructs architectures from execution traces corresponding to their dynamic instantiation. There also exist approaches for re-documenting from object-oriented code through the use of UML[5]diagrams [GK00; SM07]. Unfortunately those kinds of re-documenting approaches do not perform component-based architecture reconstruction.

[5]http://www.uml.org/ [Last seen 04-13-2018]

On the basis of this observation, this thesis proposes a retro-engineering approach based on three-level component-based architecture description reconstruction. This approach eases evolution, co-evolution and life-cycle tracking processes by providing a complete re-documentation (*i.e.*, deployment through the ***Assembly*** level, implementation through the ***Configuration*** level, and design through the ***Specification*** level). Last but not least, this approach proposes to recover design decisions through the concept of ***Specification*** which is reconstructed from an abstraction of the concrete implementation.

TABLE 3.4: Existing software architecture reconstruction approaches

approach	process	paradigm	technique	retro-engineering	re-engineering	inputs	outputs
ARES [Eix+98]	bottom-up	procedural	-	x	-	source-code, expertise	*visualization, description, analysis*
Tran *et al.* [TH99]	hybrid	procedural	quasi-auto	-	x	source-code, expertise	*visualization, vertical conformance*
X-Ray [MK01]	hybrid	procedural, distributed	auto	x	-	source-code, expertise	*visualization, description*
MAP [SO01]	hybrid	procedural	manual	x	-	source-code, expertise, style	*visualization*
Alborz [Sar03]	hybrid	procedural	auto	x	-	source-code, dynamic, expertise	*visualization, analysis*
Focus [DM01; MJ06]	hybrid	object-oriented	manual	-	x	source-code, expertise, style	*visualization*
Bauhaus [Kos02; EKS03; CKS05]	hybrid	object-oriented	auto	-	x	source-code, dynamic, expertise	*visualization, vertical conformance*
DiscoTect [Yan+04]	hybrid	object-oriented	quasi-auto	x	-	source-code, dynamic, expertise, style	*visualization, description, vertical & horizontal conformance, analysis*
ArchVis [Hat04]	bottom-up	object-oriented	auto	x	-	source-code, textual, dynamic, physical, style, viewpoint	*visualization, description*
Pashov *et al.* [PR04]	hybrid	procedural, object-oriented	auto	-	x	source-code, dynamic, expertise, style	*visualization*
PuLSE / SAVE [Kno+06]	top-down	procedural, object-oriented	quasi-auto	-	x	source-code, expertise, viewpoint	*visualization, vertical conformance, analysis*
Huang *et al.* [HMY06]	hybrid	object-oriented	auto	x	-	source-code, dynamic, style	*description, horizontal conformance, analysis*
ROMANTIC [Cha+08; Keb+12; SS13; Als+16; Sha+17]	hybrid	object-oriented	auto	-	x	source-code	*visualization, description, horizontal conformance, analysis*

3.4 Conclusion

State of the art shows that versioning management and software re-documentation still lack concepts and techniques in the field of CBSE. Conceptual lacks concern the explicitation of precise semantics in component and architecture versioning. Moreover, existing architecture evolution approaches do not provide version mechanisms for propagating versioning between all the descriptions produced during the software life-cycle (*i.e.*, specification, implementation, deployment, components at each level). Another conceptual lack is the absence of approaches which re-document software as it is implemented. Moreover, none provides support to re-document all the software life-cycle. On the other hand, technical lacks concern the implementation of such approaches. Dedal ADL documents software life-cycle as component-based software architectures. Moreover, as it has been formalized with

the B language, it is thus fully adapted for adding semantics into versioning process. Finally, has it is implemented into the Eclipse ecosystem: it is a strong base for implementing our approach. On the versioning side, we chose to adapt Brada and Bauml approach to our component-based architectures since they proposed a formal way to version components and automatically increment their version identifiers.

Chapter 4

Re-documenting component-based software architectures

Contents

When it comes to software evolution, lack or loss of documentation may become a serious issue. Indeed, in many ways, a software may be subject to numerous changes during its whole life cycle. It is then necessary to maintain an up-to-date documentation of the software accordingly to it changes. However, in many cases, the software documentation either does not exists or is not well maintained. This may lead to drift and / or erosion of software [DP09]. Thus, in order to recover a good evolution management, the software must be re-documented in accordance with its actual state [Le +18].

4.1 Process overview

This section introduces the process that is proposed for reconstructing component-based software architectures. The reconstruction is made in five major steps which are as shown in Figure 4.1: (1) the Java object architecture recovery which results from the parsing of the deployment descriptor, (2) the transformation of the deployment descriptor to an incomplete Dedal Assembly, (3) the completion of the Assembly, (4) the extraction of the architecture Configuration level and (5) the extraction of the Specification level.

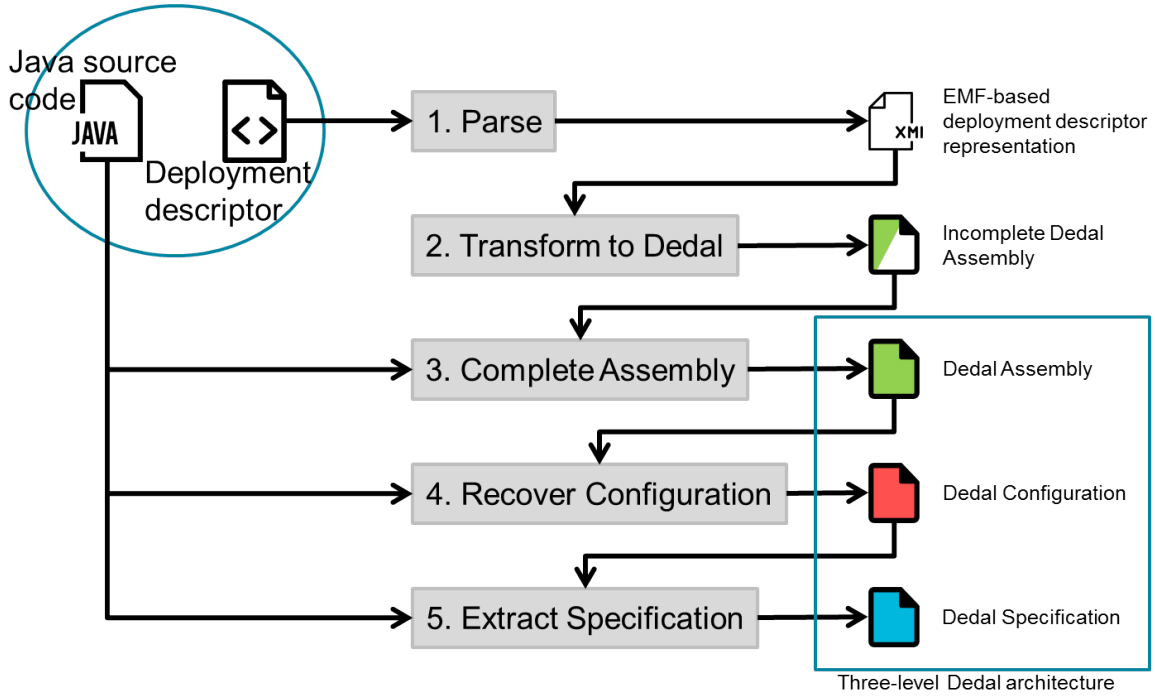


FIGURE 4.1: Process of Component-Based Software Architecture Reconstruction

4.1.1 Inputs

Figure 4.1 introduces the re-documentation process. In order to re-document the software architecture, the process needs two kinds of inputs. The first one is a deployment descriptor which describes how a given software is instantiated. The second input is the source code

```
<?xml version="1.0" encoding="UTF-8" standalone=no ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean class="AdjustableLamp" id="lampDesk" />
    <bean class="AdjustableLamp" id="lampSitting" />
    <bean class="AlarmClock" id="aClock1" />
    <bean class="SecurityManager" id="securityManager1" >
        <property name="alarm" ref="aClock1" />
    </bean>
    <bean class="HomeOrchestrator" id="orchestrator1" >
        <property name="lights">
            <set>
                <ref bean="lampDesk" />
                <ref bean="lampSitting" />
            </set>
        </property>
        <property name="clock" ref="aClock1" />
        <property name="securityManager" ref="securityManager1" />
    </bean>
</beans>
```

FIGURE 4.2: Home Automation Software (HAS): XML-based Spring configuration

of the software. Figure 4.2 introduces an example of an XML-based Spring [Joh+04] deployment descriptor. This file contains information about class instances and dependency injection which is the starting point of the re-documentation process. This example is a Java-based example which code structure is presented in Figure 4.3 as an UML diagram. The source code is not introduced since only its structure is necessary for re-documenting the structural aspect of software architectures.

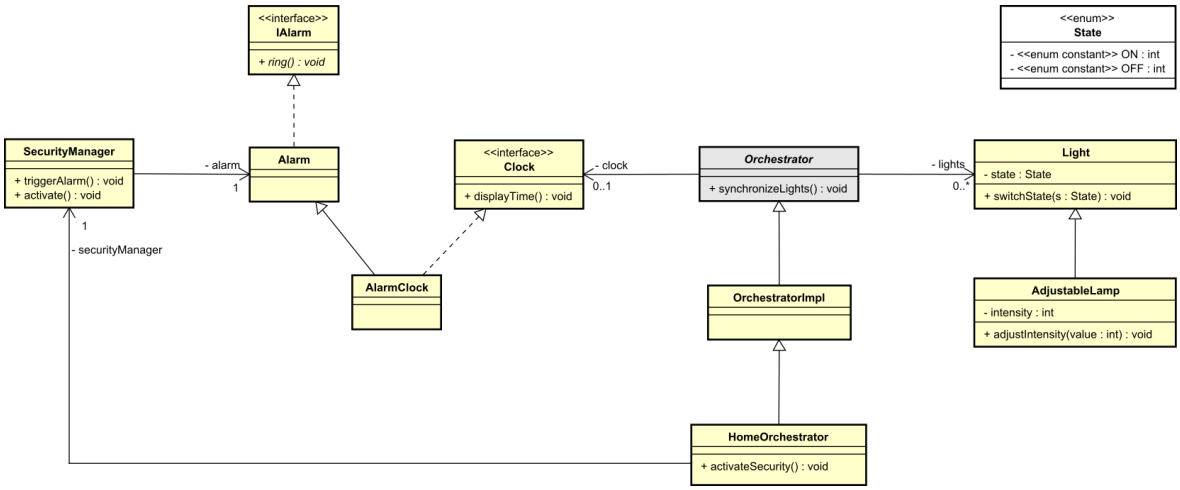


FIGURE 4.3: HAS: UML diagram

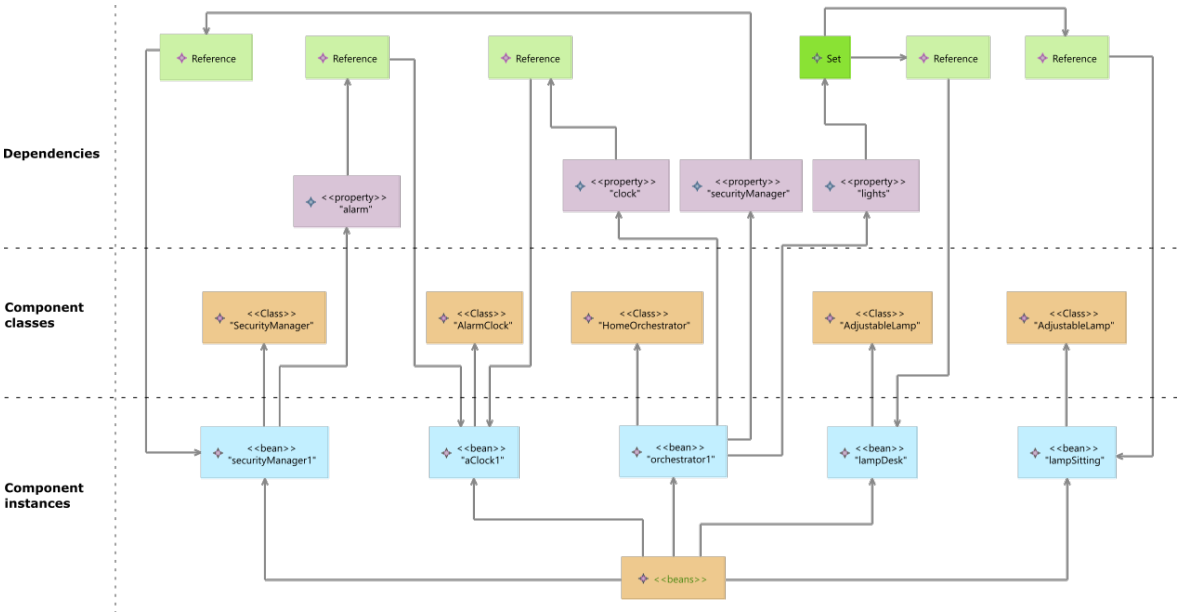


FIGURE 4.4: SpringDSL representation of HAS Spring deployment descriptor

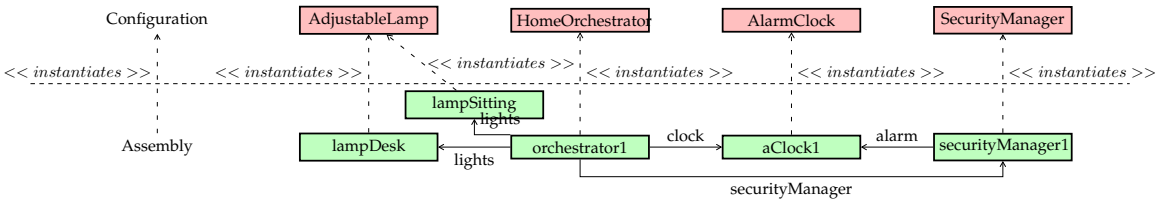


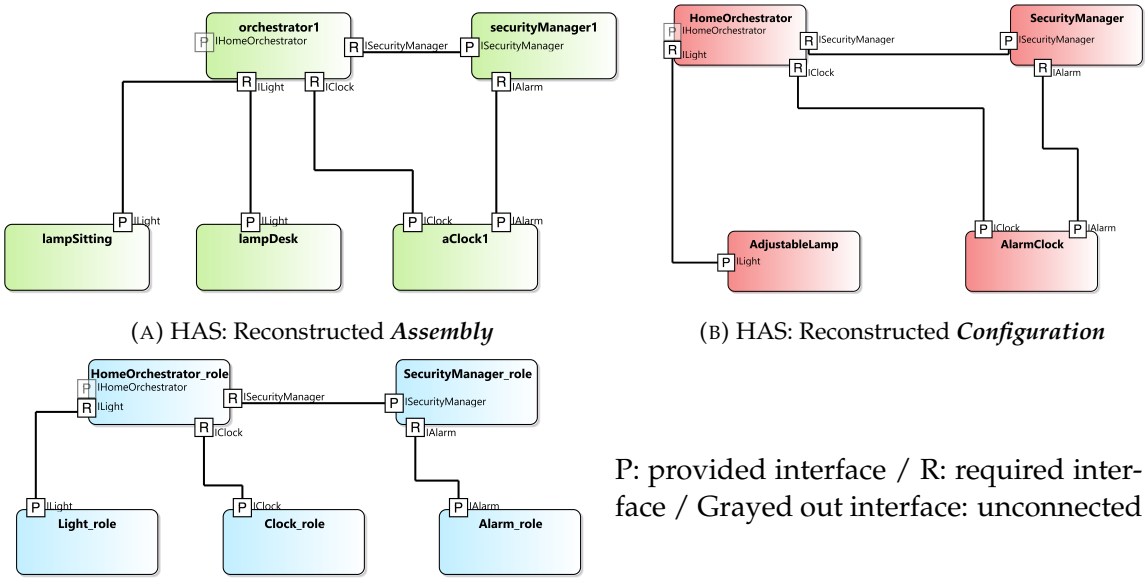
FIGURE 4.5: HAS: Dedal incomplete Assembly after step 2

4.1.2 Process

As the first step of the re-documentation process, the deployment descriptor file is parsed for building an EMF-based representation of the software deployment. Figure 4.4 introduces the result of parsing the deployment descriptor presented in Figure 4.2. Both representations contain strictly the same information about bean instances, instantiated Java classes and dependencies. For instance, Figure 4.4 shows a $<<\ bean\ >>$ named $securityManager1$ which $<<\ Class\ >>$ is $SecurityManager$ and that has a $<<\ property\ >>$ named $alarm$ which is set with a $Reference$ to the $<<\ bean\ >>$ named $aClock1$. This is equivalent to its XML version shown in Figure 4.7

Then, this model-based representation of the deployment descriptor is the input of a model to model transformation that transforms deployment descriptor information into partial architecture information concerning ***Assembly*** and ***Configuration*** levels in Dedal language. Figure 4.5 shows how Spring artifacts that are introduced in Figure 4.4 are arranged to fit Dedal concepts.

Then, this partial information is used, in combination with the Java source code, as the input of the third step that re-documents the ***Assembly*** level. Thus, missing information is extracted for re-documenting ***Assembly*** as presented in Figure 4.6a. At this step, ***Assembly*** is fully extracted.



(A) HAS: Reconstructed ***Assembly***

(B) HAS: Reconstructed ***Configuration***

(C) HAS: Reconstructed ***Specification***

FIGURE 4.6: HAS: Dedal Reconstructed Architecture Levels

```
<bean class="SecurityManager" id="securityManager1" >
    <property name="alarm" <ref="aClock1" />
</bean>
```

FIGURE 4.7: Example of bean declaration with dependency injection

This information is then reused at the fourth step which re-documents the ***Configuration*** level after the ***Assembly*** such as introduced in Figure 4.6b.

Finally, this information is used, still combined to the Java souce code, to re-document the ***Specification*** level as shown in Figure 4.6c.

4.1.3 Output

The output of the process is a complete three-level Dedal architecture composed of the ***Assembly***, ***Configuration*** and ***Specification*** that are presented in Figures 4.6a, 4.6b and 4.6c.

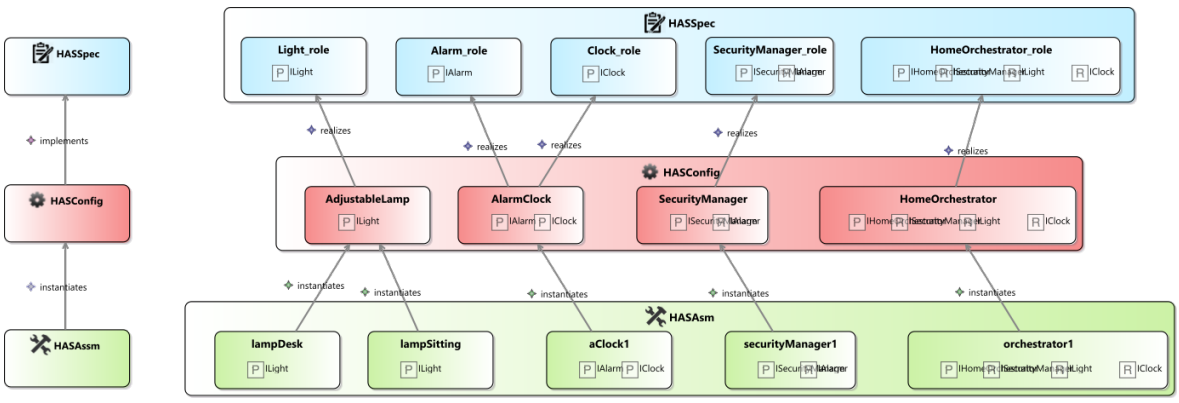


FIGURE 4.8: Three-level view of reconstructed Dedal architecture

Figure 4.8 shows the three-levels of the re-documented Dedal architecture. This view shows the relations that exist between the architecture levels and their respective components. Those relations are also re-documented during the process.

Next section discusses how the re-documentation is actually performed.

4.2 Re-documenting architectures

This section discusses how the different concepts are identified and mapped into architectural information, from the deployment descriptor to code analysis. Figure 4.9 represents the structure of the re-documentation module. As shown in this figure, the ***SAR_Module*** (Software Architecture Re-documentation module) is composed of three sub-modules (in grey):

- the ***DescriptorDSLModule***. This module parses and instantiates a model-based representation of the deployment descriptor.
- the ***DescriptorModelToDedalTransformation***. This module uses the previously generated model-based description and applies a model to model transformation to generate the very first model artifacts of the Dedal **Assembly** and **Configuration** architecture models.
- the ***CodeExtractionModule***. This module aims at completing the information that have

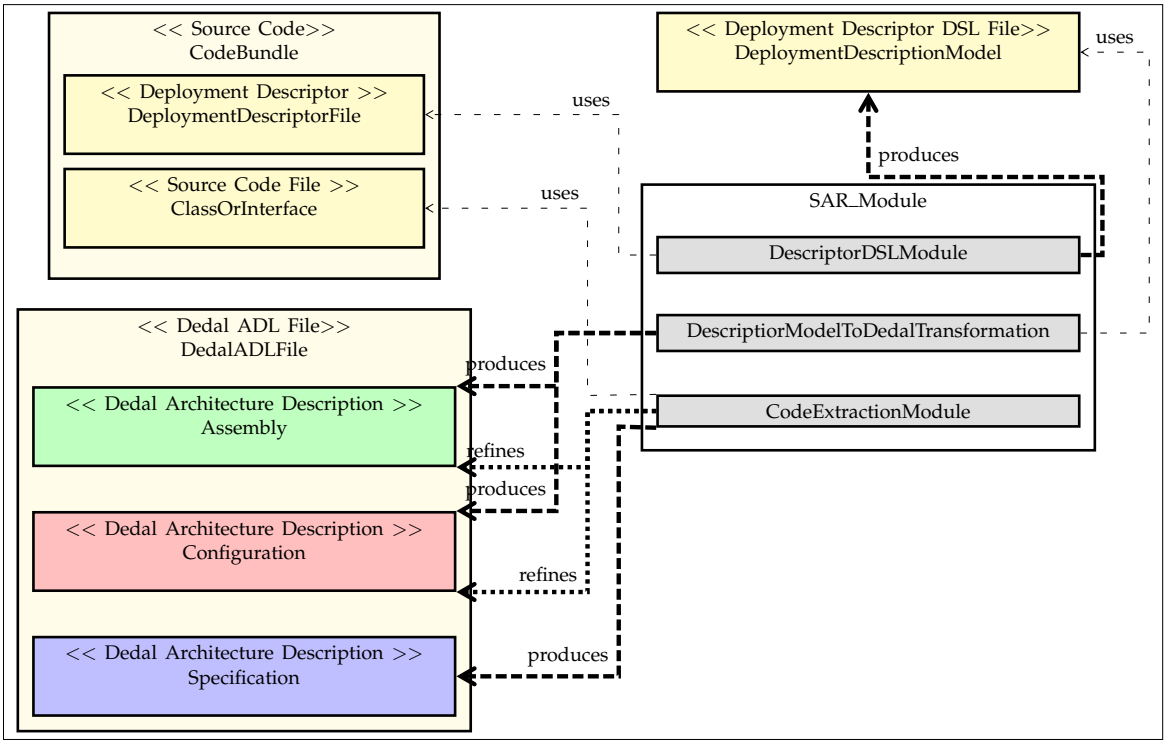


FIGURE 4.9: Structure of the re-documentation module

```
1  /*<beans/> */
2  Configuration returns Configuration:
3      {Configuration}
4      (
5      '<beans'>
6
7      [...]
8      (components+=Component
9          [...]
10     )*
11     [...]
12     ('</beans>');
```

FIGURE 4.10: Configuration Xtext-based implementation

been extracted from the deployment descriptors by exploring the code. It then re-documents the **Assembly** and **Configuration** levels and also generates the **Specification** level.

4.2.1 SpringDSL, a DSL for mapping Spring Concepts

As a natural consequence of a model-driven re-documentation process, a model-based representation of the deployment descriptor is a starting point for being able to re-document software as a component-based software architecture. Thus as a proof of concept, we implemented a part of the XML-based grammar of the Spring [Joh+04] deployment descriptor which we named SpringDSL. The Spring XML grammar has then been implemented using XText[1]. Thanks to Xtext, we direcly derived an EMF[2]-based metamodel of the Spring language, we as well automatically generated a parsing tool for XML-based Spring deployment descriptors. Thus, it is possible to automatically get a model of Spring deployment descriptors through a simple parsing by the Xtext-based tool.

In this part, only the concepts which are useful for the re-documentation process are presented, the full implementation of the Spring XML-based grammar is presented in Appendix A.

Figure 4.10 shows the a basic extract of the XML-based Spring grammar. An XML-based Spring development starts with a $< beans >$ tag and ends with a $< /beans >$ tag. The beans that constitute the description are declared in this scope. Figure 4.11 shows an extract of the implemented grammar that describes bean declaration. The bean is declared through a $< bean >$ tag and may contain one or several names, a class attribute that provides the information about the instantiated class and also additionnal features. Finally, Figure 4.12 presents the *Feature* rules which correspond to the set up of components (beans). In order to inject dependencies, a *Property* must be declared through the $< property >$ tag, with at

[1]https://www.eclipse.org/Xtext/ [Last seen 04-08-2018].
[2]https://www.eclipse.org/modeling/emf/[Last seen 04-08-2018].

```
1  /*<bean/> */
2  Component returns Component:
3      {Component}
4      '<bean'
5      ( ('id='name=ValidString)? & ('name='names+=ValidString)?
6      & class= CreationMethod
7      & [...]
8      )(
9          ('/>')
10         | ('>'
11         [...]
12         (features+=Feature | [...])*
13         '</bean>')
14     );
```

FIGURE 4.11: Component Xtext-based implementation

```
1  /**Abstract Class of elements present in bean */
2  AbstractArtefact returns AbstractArtefact:
3      Component |  AttributeTag  |IdRefTag| ReferenceTag  |Collection;
4
5  Feature returns Feature:
6      (Property |ConstructorArg);
7
8  /*<property/> */
9  Property returns Feature:
10     ('<property' ((
11         [...]|
12         (('name=' name=ValidString)'>' (description=Description)? (
               artefact=AbstractArtefact|NullTag) '</property>')
13         )
14     );
15
16 /**Reference create by a tag */
17 ReferenceTag returns Reference:
18  {Reference}
19  '<ref''bean='ref=([AbstractArtefact|ValidString]) ('/>'|'>''</ref>');
```

FIGURE 4.12: Reference Xtext-based implementation

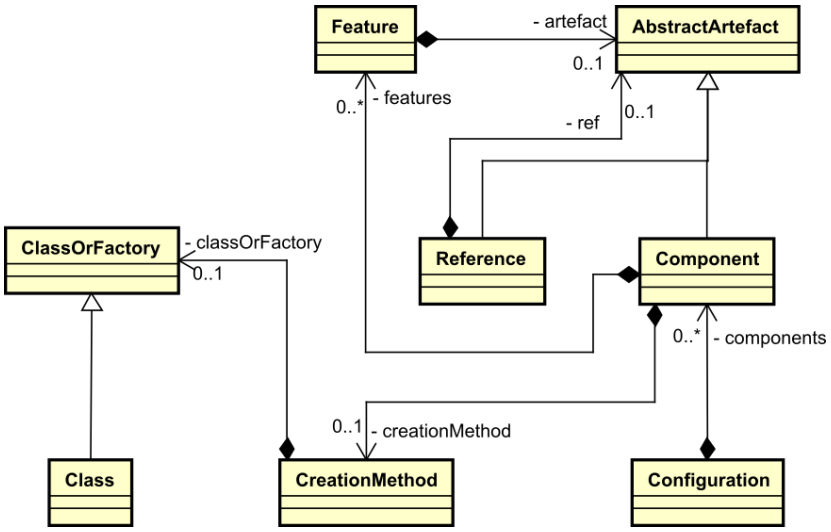


FIGURE 4.13: Excerpt of the SpringDSL Metamodel

least a name and an artefact. The artefact can be of several kinds through the AbstractArtefact rule. Then, the *Reference* rule describes a reference to a bean which corresponds to the declaration of dependency injection.

All the previously described grammar rules are then automatically derived as the metamodel extract that is shown in Figure 4.13. In SpringDSL, a *Configuration* corresponds to a Spring deployment and a *Component* corresponds to a bean.

The Spring deployment descriptor in Figure 4.2 is thus transformed into EMF objects such as introduced in Figure 4.4. $Bean$ tags are transformed into $<< bean >>$ objects, their $class$ attributes as $<< Class >>$ objects and their properties as Sets or $Reference$s according to the property kind.

Figure 4.4 presents an example of an XML-based Spring [Joh+04] deployment descriptor. This example is composed of five beans:

- *lampDesk* and *lampSitting*, instance of *AdjustableLamp*
- *aClock1*, instance of *AlarmClock*
- *securityManager1*, instance of *SecurityManager*, that has a dependency to *aClock1* declared in its tag *property* which name *alarm* is the name of the injected attribute in the *SecurityManager* Java class.
- *orchestrator1*, instance of *HomeOrchestrator* which refers *lampDesk* and *lampSitting* as its *lights* and *aClock1* as its *clock*.

The graphical representation is provided by Sirius [3]. This DSL thus eases the transformation which is made from Spring to Dedal. Indeed, it allows a direct mapping of XML-based descriptions as EMF objects which can then be mapped as Dedal artifacts through a Model

[3] https://www.eclipse.org/sirius/ [Last seen 08-23-2019]

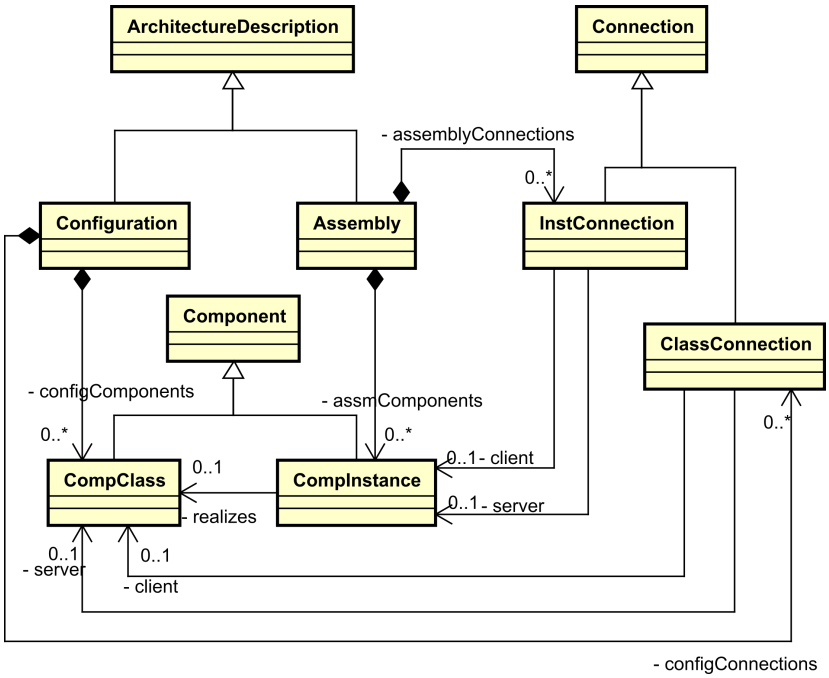


FIGURE 4.14: Dedal Metamodel Sub-part for M2M transformation

to Model transfomation implemented with the Operational QVT (QVTo[4]), which is the implementation of the Query View Transformation (QVT) language in the eclipse ecosystem. It is then possible to get the first Dedal artifacts as explained in next paragraph.

4.2.2 Model to model transformation: from descriptor model to partial Dedal architecture model

The second step of the proposed reconstruction process (Figure 4.1) consists in transforming the concepts of the SpringDSL model into Dedal artifacts. To do so, a simple mapping between the SpringDSL model and Dedal concepts is required.

As stated before, the transformation is implemented in QVT. The full implementation is not discussed here but can be found in Appendix B.

Comparing the fragment of the SpringDSL metamodel given in Figure 4.13 and the fragment of the Dedal metamodel given in Figure 4.14, it appears that a part of a Dedal model can be re-documented by only identifying concepts in SpringDSL that correspond to Dedal concepts. Thus, the mapping is realized as follows:

- a deployment (SpringDSL *Configuration*) is mapped as a Dedal *Assembly*, a Dedal *Configuration* is derived from the set of instantiated classes.
- a bean (SpringDSL *Component*) is mapped as a *CompInstance* since it typically is the declaration of an object instantiation.

[4]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Last seen 04-08-2018].

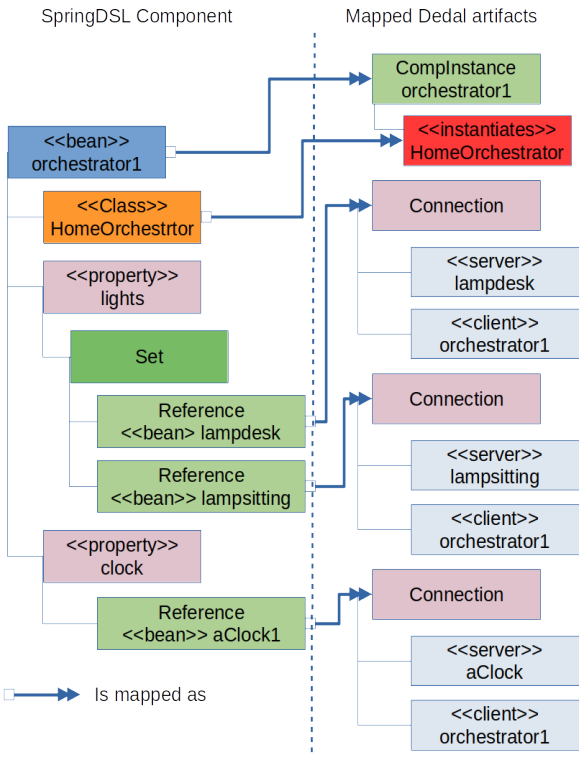


FIGURE 4.15: Mapping SpringDSL artifacts into Dedal artifacts

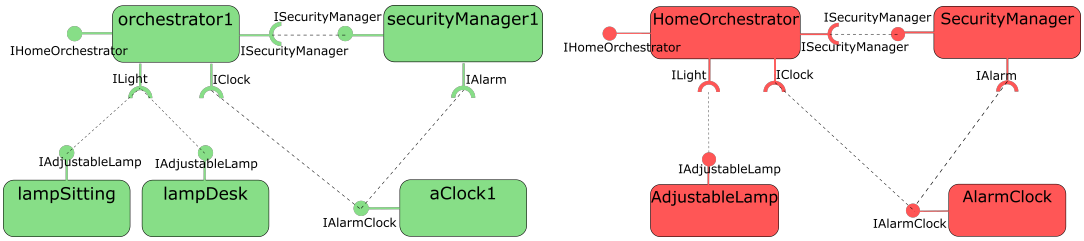
- a *Class* is mapped as a Dedal *CompClass*, it is the class that is instantiated by the component instance (*CompInstance*).
- a *Reference* is mapped into a connection. The bean that holds the reference is considered as the client while the one that is referenced is considered to be the required server. At this stage, component interfaces are not known, however the direction of connections can be deduced from dependency injections. Then, if a bean references another bean, it is considered as the client and thus the reconstructed component will be the owner of the required interface implied in the connection. On the other hand, the component that will be extracted from the referenced bean will own the provided interface that is the complementary part of the connection.

Figure 4.5 shows the partial Dedal architecture that is mapped thanks to QVTo. Figure 4.15 shows how the mapping is performed following the previously discussed mapping rules.

4.2.3 Extracting information from the object-oriented code

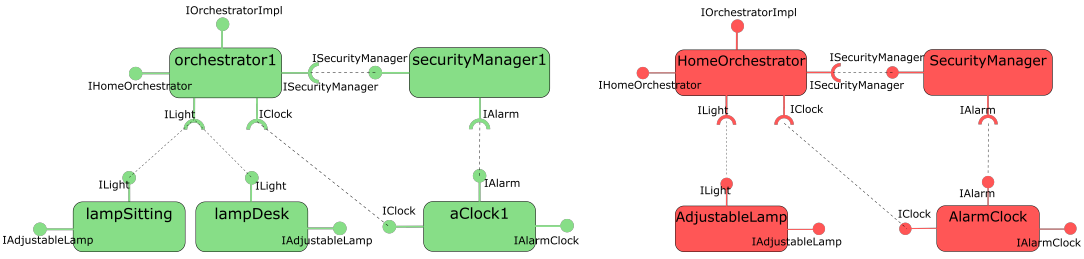
This subsection introduces the proposed methodology for generating missing architectural information from object-oriented code. The class hierarchy of the HAS example that is implemented in Java, is shown in the UML diagram of Figure 4.3. The HAS project is thus composed of seven concrete classes, an abstract class and two interfaces.

At the end of the model to model transformation step, the basis of the architecture has



(A) HAS ***Assembly*** (B) HAS ***Configuration***

FIGURE 4.16: A single provided interface is exposed



(A) HAS ***Assembly*** (B) HAS ***Configuration***

FIGURE 4.17: All provided interface are exposed

been re-documented. This step provides a very first and incomplete description of the ***Assembly*** level by exposing the deployed component instances, the instantiated component classes and finally the connections which exist between components. The inputs of the re-documenting algorithm consists in the three Dedal architecture descriptions that are being reconstructed as well as the set of classes of the project. At this point, information must have already been provided to the ***Assembly*** description through the deployment descriptor analysis (*e.g.*, Spring XML configuration files).

At this step re-documentation follows three main stages. First of all, the **Assembly** level is completed by identifying component interfaces from the source code, then the **Configuration** level can be re-documented both from the information that are contained in the **Assembly** and the source code. Finally a **Specification** is generated from the **Configuration** level, in accordance with components / classes type hierarchy so that it is consistent with the **Configuration** level.

4.2.4 Re-documenting Assembly

First of all, the **Assembly** level must be fully completed thanks to source code analysis.

4.2.4.1 Mapping component instances

As early reconstructed component instances from the **Assembly** are not complete, they must be refined from code inspection. To do so, required component interfaces are identified from the dependencies which are injected into deployment descriptors. As corresponding class attributes are known from the dependency injection, it is easy to get required types from code inspection and by extension, those types are seen as the required interface types.

Considering the *securityManager1* component instance in Figure 4.5, which instantiates the *SecurityManager* class (see Figure 4.3), provided interfaces will be calculated from the class itself. Its *alarm* attribute type, as it is implied in a connection, will be identified as a required interface. This will not be the case for the *AdjustableLamp* class, indeed it has an attribute named *intensity* which is not implied in any connection and thus which type will not be identified as a required interface.

As it is shown in Figures 4.16a and 4.17a there are several equivalent ways to re-document provided interfaces in **Assembly** component instances. The first option, shown in Figure 4.16a, consists in extracting only the coarser grained, and thus most specialized, provided interface. It means that the component will provide only one interface that will expose all the services the component provides. This interface corresponds to most universal one in terms of provided functionalities and connection capabilities. This is the way that guarantees maximal reusability capabilities. Indeed, as the provided interface corresponds to the more specialized type in the component type hierarchy, then it can be used by any component which would require any less specialized type. The second way for extracting provided interfaces, introduced in Figure 4.17a, consists in exposing several provided interfaces that correspond to each level of the component type hierarchy. For example the ***aClock*** component instance that derives from the ***AlarmClock*** class in Figure 4.3 exposes three provided interfaces which correspond to the types ***AlarmClock***, ***Alarm*** and ***Clock***. In this case, there is no need to identify an interface for the ***IAlarm*** Java interface since the class bellow (***Alarm***) barely implements the interface and does not define a new abstract type. This second solution is a refinement of the first since it still preserves a maximal reusability of the component class thanks to its explicit IAlarmClock provided interface, which holds the most specialized interface type and is thus the most generic provided interface. However, it makes it possible to handle finer grained connections between components. The goal is to expose all reference types that can be obtained on component instances as separate provided interfaces. These reference types correspond to the different abstract types that are implemented by the component class. As this approach only takes place in a re-documenting process, no interface types can be calculated in order to avoid functionality overlapping among them. Indeed, this would introduce new types that do not exist in the code, which is not possible since the purpose of this reconstruction is to expose how the software is actually deployed and implemented. However, to avoid useless redundancy, when the abstract type of a class is identical to the abstract type of the interface it implements or the superclass it specializes, only the most specialized provided interface is kept, corresponding to the most specialized class. This corresponds to classes with only one ancestor, that do not declare any new public functionality or overload any inherited public functionality.

From this point, the classes which are deployed by the deployment descriptor are seen as very fine grained components of the architecture, which is coherent with the willing of re-documenting architectures "as they are implemented". The mapping of the interfaces is discussed in next subsection.

4.2.4.2 Mapping component interfaces

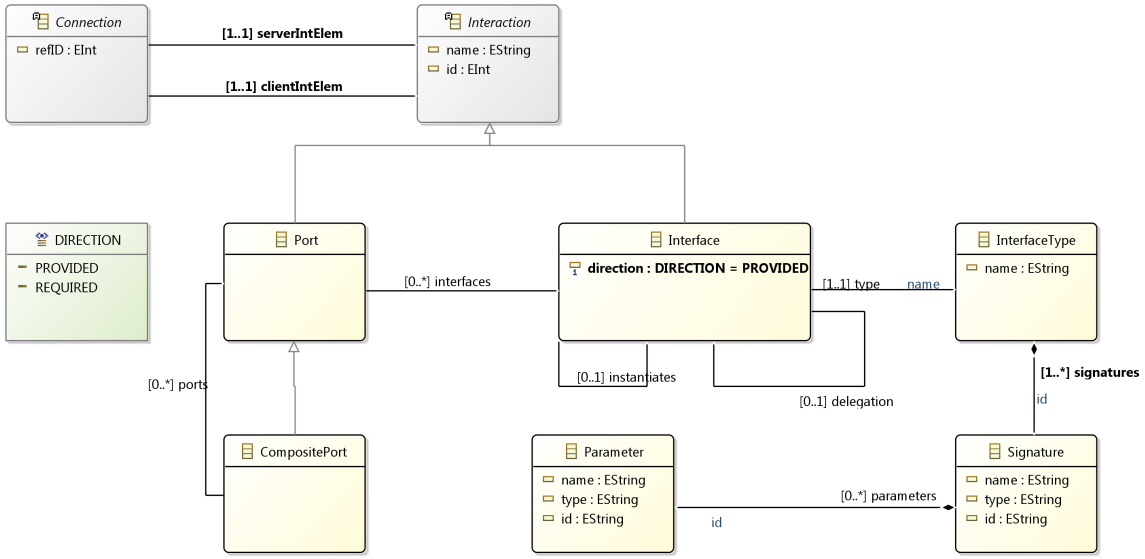


FIGURE 4.18: Dedal Interactions Meta-Model

Figure 4.18 is a subpart of the Dedal metamodel which introduces how interactions are handled by the ADL. An ***Interaction*** can be an ***Interface*** or a ***Port***. However, since this approach focuses on a re-documentation that occurs from raw source code and deployment descriptors, only ***Interface***s are considered. Thus, a Dedal ***Interface*** is very similar to the object-oriented concept of interface. An ***Interface*** is composed of a name to be identified within the component, a direction to provide information about whether it provides of requires functionalities and a type which is no more than the description of an interface as known in object-oriented languages in other words, it consists in declaring the set of methods that are part of the interface type. ***Signature***s correspond to the declared methods of the interface which themselves contain ***Parameter***s.

Algorithm 1 introduces the way interfaces are re-documented from the source code. The whole class hierarchy is explored in order to extract an interface for each of its super types and itself. Thus an interface is mapped as follows:

- the interface is been given a direction to identify whether it is required or provided,
- then, it is named for being identified,
- and then, its type is calculated thanks to the procedure *MapInterfaceType* which maps the signatures (*MapSignature* procedure) the interface is composed of from the public methods of the class.

Figure 4.19 illustrates the way interfaces are mapped from types in the code. Given a type *C*, assuming the type hierarchy which is proposed in Figure 4.19 and considering the re-documentation process as a strict retro-engineering process, each ***Interface*** that is re-documented must correspond to the granularity of the types into the hierarchy. Indeed it is the only way for ensuring the highest reusability and also the highest proximity to the

Algorithm 1 Mapping interfaces

Ensure:

$\forall\, interface \in \textbf{result}(\forall\, signature \in interface.interfaceType.signatures($
$\exists\, method \in class.methods, signature.type = method.type$
$\bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name))))$

function MAPINTERFACES($class : Class,\ direction : DIRECTION$)
  $result : Set(Interface)$
  $result \leftarrow \{$MAPINTERFACE($class,\ direction$)$\}$
  for all $super \in class.superTypes$ **do**
    $result \leftarrow result \bigcup$ MAPINTERFACES($super,\ direction$)
  end for
  return $result$
end function

Ensure:

$\forall\, signature \in \textbf{result}.interfaceType.signatures(\exists\, method \in class.methods,$
$signature.type = method.type \bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name)))$

function MAPINTERFACE($class : Class,\ direction : DIRECTION$)
  $result : Interface$
  $result.direction \leftarrow direction$
  $result.name \leftarrow$ "I" $+ class.name$
  $result.interfaceType \leftarrow$ MAPINTERFACETYPE($class$)
  return $result$
end function

Ensure:

$\forall\, signature \in \textbf{result}.signatures(\exists\, method \in class.methods,$
$signature.type = method.type \bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name)))$

function MAPINTERFACETYPE($class : Class$)
  $result : InterfaceType$
  $result.name \leftarrow$ "I" $+ class.name +$ "$\_type$"
  for all $m \in class.methods$ **do**
    $result.signatures \leftarrow result.signatures \bigcup$ MAPSIGNATURE(m)
  end for
  return $result$
end function

$$IC_4 = \{C_4.signatures\}$$
$$IC_{34} = \{C_3.signatures\} \bigcup IC_4$$
$$IC_{234} = \{C_2.signatures\} \bigcup IC_{34}$$
$$IC_1 = \{C_1.signatures\}$$
$$IC_{1234} = \{C.signatures\} \bigcup IC_1 \bigcup IC_{234}$$

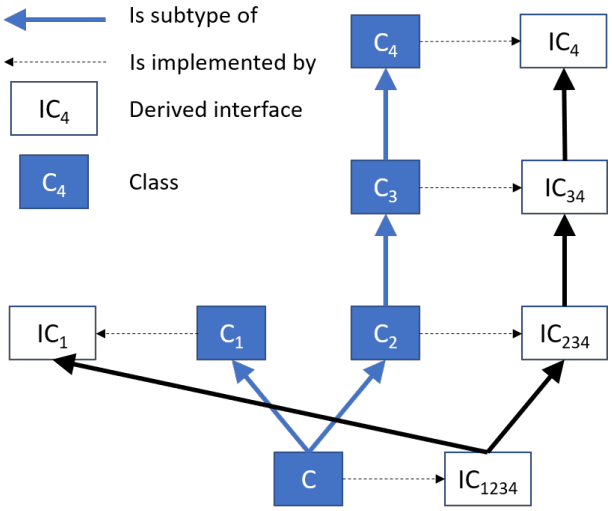


FIGURE 4.19: Mapping Dedal ***Interface***s from type hierarchy

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Y: Yes / N: No

TABLE 4.1: Java access level modifiers [Java]

source code and implementation decisions. This is the reason why each type of the hierarchy can be derived as an ***Interface*** which includes more generic ones. Thus, for instance a provided ***Interface*** which type is IC_{1234} can be required by an ***Interface*** which type is IC_4 or any of its super types. In the situation where an architect decides to reconstruct an architecture and to expose every component ***Interface***, each of the super types of IC_{1234} would be considered as a candidate ***Interface*** to replace IC_{1234} into a ***Connection***.

Discussion. The main intention behind re-documenting component interfaces is to expose all its API methods. In other words, an interface exposes methods that can be reached by other components. However, there is not a generic way for identifying methods that are reachable by other components since, for instance in Java, access modifiers can modify the way an interface is perceived by its environment and thus the concept of API method differs according to the point of view. For instance in Java, it is possible to use four different access modifiers (Table 4.1). Thus in the world point of view the API methods are the public ones and in the package point of view where API methods are public, protected or have default access modifier. Then, in the implementation of the algorithm we chose to consider only public methods in interfaces, however other visibility options are valid for re-documenting interfaces.

The completion of ***Connection***s between components is discussed in the following.

4.2.4.3 Completion of connections

As it has previously been discussed, the analysis of deployment descriptors is a good way for identifying connections between components. It is also a good way for identifying the direction of those connections. However, as it is presented in Figure 4.18, a Dedal ***Connection*** must be set between two ***Interface***s: a provided one and a required one. For instance, in Figure 4.5 two connections are identified. A connection exists from *orchestrator1* to *aClock1* and another one from *securityManager1* to *aClock1*. It means that both *orchestrator1* and *securityManager1* require an interface that belongs to *aClock1*. From an object-oriented point of view, such as UML (Figure 4.3), it means that both *HomeOrchestrator* and *SecurityManager* classes have a relation to the *AlarmClock* class or one of its super types, in this example they are respectively linked to *Clock* and *Alarm* through their respective *clock* and *alarm* properties. Moreover, those types are both substitutable by *AlarmClock*.

Then, in the case software re-documentation is performed choosing the first option, which only represents biggest provided interfaces, the connections can be set directly from required interface to a single ***Interface*** of type *IAlarmClock* such as proposed in Figure 4.16a. Required interfaces are derived from the component class type hierarchy, for instance as *SecurityManager* has an association to *Alarm*, *securityManager1* requires an ***Interface*** which type derives from the *Alarm* class. Finally, as it has been discussed in previous part, if an architect decides to expose every component ***Interface***, then candidates interfaces are explored for connecting the required interface to the most generic provided interface such as proposed in Figure 4.17a where *orchestrator1* and *securityManager1* are connected to *aClock1* through the exact type they require. This choice is proposed for simplicity and readability's sake (separate connections on separate interfaces as much as possible), as the many substitutable provided interfaces correspond to the same method implementation (we consider polymorphic substitution, as implemented in Java and defined in standard object-orientation).

4.2.5 Re-documenting Configuration from Assembly

The second step of the re-documentation process consists in deriving a ***Configuration*** from the previously re-documented ***Assembly***.

4.2.5.1 Identifying component classes from component instances

In the first place, all concrete component classes in the **Configuration** must be identified from the **Assembly** level. This identification comes naturally since deployment descriptors embed the names of the classes that are instantiated. Thus component instances from the **Assembly** also embed the reference to the **CompClasses** which are instantiated. For instance, Figure 4.5, exposes the instantiation relations between component instances from the **Assembly** level and the component classes which they instantiate. Those relations are directly identified from deployment descriptors.

Then component classes can be re-documented and their information completed through the analysis of source code and component instances from the **Assembly**. To do so, the component ***Interface***s of **Configuration** component classes are derived from the corresponding component instance ***Interface***s. Indeed, the relation between component classes and component instances is no more than an instantiation relation, thus component interfaces of component instances are instances of the ones belonging to component classes. As it is shown in Figures 4.16b and 4.17b the choice that is made at **Assembly** reconstruction step for exposing a single or several provided interfaces will impact the provided interface multiplicity of component classes into the ***Configuration***.

Identification of connections is discussed next.

4.2.5.2 Identifying Configuration connections

Once component classes are reconstructed, connections must be set. They are derived from the ones which exist in **Assembly** level. To do so, **Assembly** connections are simply copyied between instantiated component classes such as presented in Figures 4.16 and 4.17 where, for instance, the connection between *orchestrator1* and *securityManager1* (Figures 4.16a and 4.17a) is copied between *HomeOrchestrator* and *SecurityManager* (Figures 4.16b and 4.17b). However in some cases, a component instance might be connected, through a same required interface, to multiple component instances that belong to the same component type (the component type used to define the required interface). This case is illustrated in Figures 4.16a and 4.17a with *orchestrator1* that is connected to two component instances, *lampSitting* and *lampDesk* that both instantiate the *AdjustableLamp* component class (Figures 4.16b and 4.17b). In such a case, these multiple connections between component instances are modeled as one connection between component classes into the **Configuration** (in architecture configuration models, connections correspond conceptually to connection classes that are instantiated in **Assemby** models).

4.2.6 Re-documenting Specification

Finally the last step of the re-documentation process consists in deriving a **Specification** from the previously re-documented **Configuration**.

4.2.6.1 Mapping component roles

As discussed before, in Dedal, the **Specification** level corresponds to an abstract description of the implementation. It is a more generic description that is derived from the **Configuration** level. Thus, in order to re-document a more generic architecture level, it is necessary to get the more generic component roles that are realized by component classes from the **Configuration**. To do so, the type hierarchy of those component classes must be traversed. In other words, component roles are generated by analyzing the type hierarchy of the classes which are present in the source code.

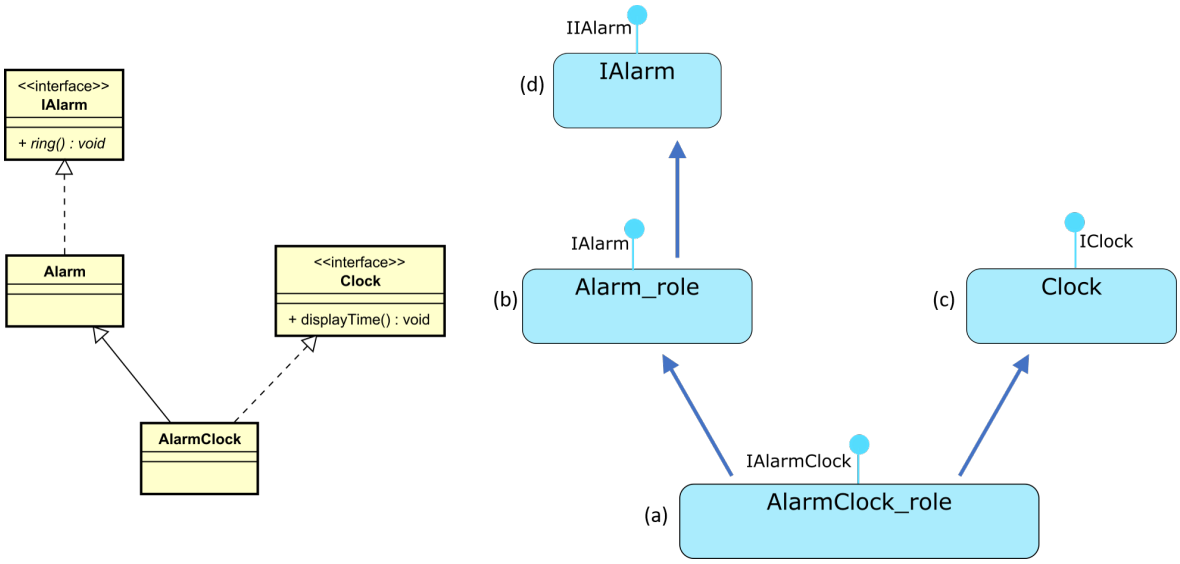


FIGURE 4.20: Example of Role Hierarchy based on the HAS Example

Thus in the first place, all potential component roles need to be mapped. They will next be analyzed to get the restrained set of component roles which are actually realized by the component classes from the **Configuration**. Component ***Interface***s at **Specification** level are then mapped following the same principles than component ***Interface***s in **Configuration** and **Assembly**. Figure 4.20 shows which roles can be reconstructed from *AlarmClock* class and its type hierarchy. Then four component roles can be reconstructed from the *AlarmClock*. This figure also introduces how those component roles are named. In order to avoid name confusion with **Configuration** component classes which correspond to the actual implementation of component roles, if a role is derived from a concrete type which means it derives directly from an actual implementation then the name of the type from which derives a component role must be completed by a suffix : "_role". However if it derives from an abstract type such as an interface or an abstract class, then there is no need to modify the type name since those types cannot correspond to component classes at **Configuration** level.

4.2.6.2 Identification of component roles

The next step of the **Specification** reconstruction consists in determining among all potential component roles for each component class in the configuration which ones are realized (or not) by the component class. A component role or a set of component roles is realized by a component class if and only if all its required interfaces are preserved as well as all its provided interfaces which are involved into connections with other components.

Algorithm 2 introduces the mapping and identification of component roles. In order to extract component roles from a component class, the type hierarchy of the component class is traversed and a role is mapped for each of these types. Then, at each type hierarchy level and for each potential component role, a "contract" is calculated. A "contract" is composed of

Algorithm 2 Mapping Component Roles

Ensure:

1: $\forall role \in$ **result**$((\forall interface \in role.componentInterfaces($
$\forall signature \in interface.interfaceType.signatures($
$\exists method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall parameter \in signature.parameters($
$\exists p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name)))))$

2: $\forall reqInterfaceType \in contract.first(\exists role \in$ **result**,
$role.getAllRequiredTypes().contains(reqInterfaceType))$

3: $\forall provInterfaceType \in contract.second(\exists role \in$ **result**,
$role.getAllProvidedTypes().contains(provInterfaceType))$

4: **function** MAPCOMPONENTROLES(
$contract : Pair(Set(InterfaceType), Set(InterfaceType)),$
$class : Class$) : $Set(ComponentRole)$

5: **if** $\neg class.$**hasSuperType**() **then**

6: **return** {MAPCOMPONENTROLE($class$)}

7: **end if**

8: $result : Set(ComponentRole)$

9: **for all** $superType \in class.superTypes$ **do**

10: $result \leftarrow result \bigcup$ MAPCOMPONENTROLES(
SPLITCONTRACT($superType, contract$), $superType$)

11: **end for**

12: **if** SATISFYCONTRACT($roles,\ contract$) **then**

13: **return** $result$

14: **else**

15: **return** {MAPCOMPONENTROLE($class$)}

16: **end if**

17: **return** $result$

18: **end function**

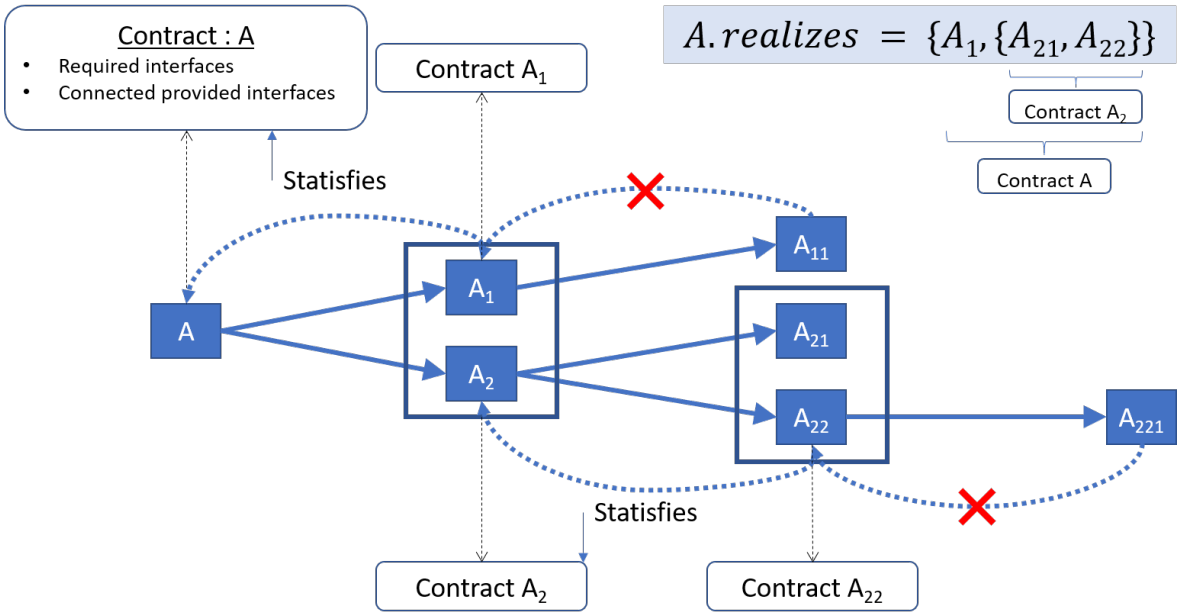


FIGURE 4.21: Identifying realized Component Roles

the component interfaces that must be preserved to ensure the coherence between the **Configuration** and **Specification** levels (see Section 2.3). Then, the set of realized component roles contains the only component types which comply with the initial contract calculated from the component class. Figure 4.21 illustrates the process. The component class type from which the component role is extracted is component *A*. A contract is calculated for *A* that corresponds to the set of interfaces to preserve. Then, its hierarchy is traversed, contracts are recalculated at each type level and finally the set of realized component roles is extracted. For instance, as it is introduced in Figure 4.17b, *AlarmClock* is connected through *IAlarm* and *IClock* then when identifying its realized component roles, those two ***Interface***s need to be preserved. Now considering Figure 4.20, component role (a) satisfies the contract through its ***Interface*** *IAlarmClock* which specializes *IClock* and *IAlarm*. At the next hierarchy level, component role (b) realizes a part of the contract through its *IAlarm* ***Interface*** and component role (c) realizes the other part through *IClock*. Component roles (b) and (c) satify the contract. Thus component role (d) still needs to be checked, then a new contract is calculated from the subpart of the contract which is realized by component role (b). This contract is composed of the *IAlarm* provided ***Interface***. As component role (b) is an implementation of the abstract type (d), then component role (b) is replaced by component (d) in the set of realized components. It cannot be the same with *HomeOrchestrator_role* (see Figure 4.23) that requires *SecurityManager_role* and which has no super type that preserves this required interface.

4.2.6.3 Identification of connections and representation of the Specification

Finally, the last step of the **Specification** reconstruction consists in connecting component roles. Figure 4.22 introduces how connections are identified from the **Assembly** level and

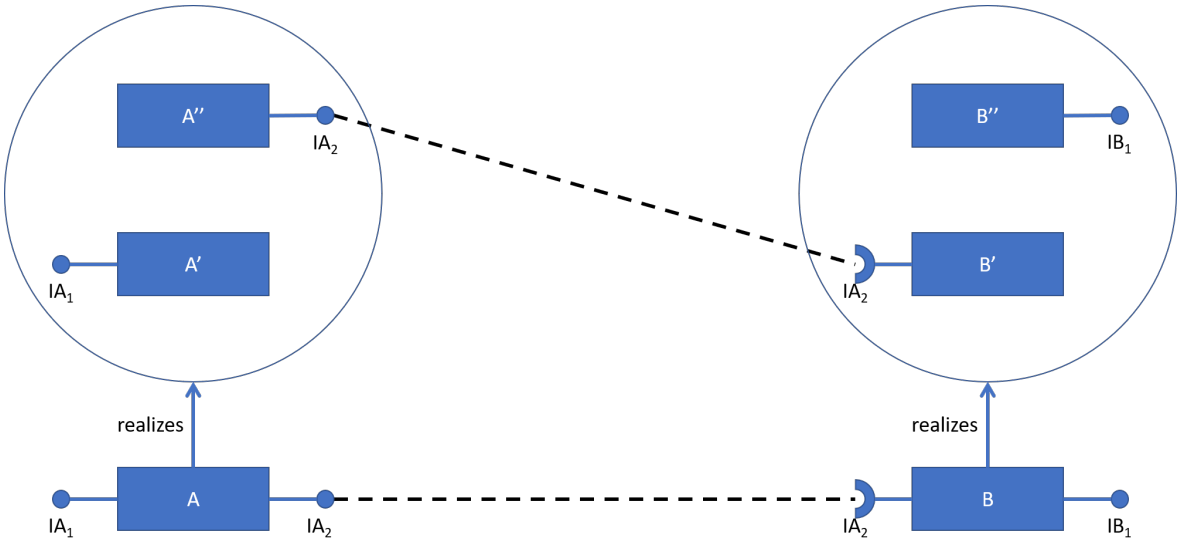
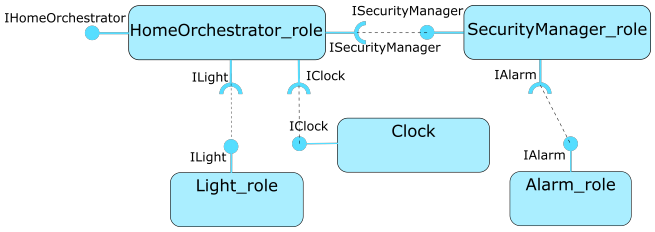
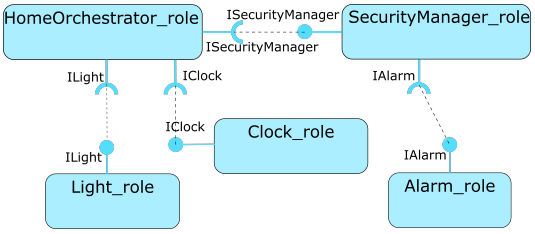


FIGURE 4.22: Connecting Component Roles

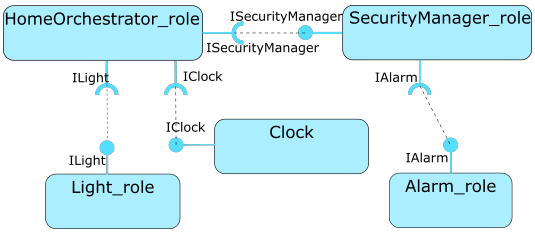
copied into the **Specification**. A basic mapping is done between the realized components roles of two connected component classes, using the types of their interfaces.



(A) Considering Only Existing Types in the Source Code



(B) Avoiding to Use Existing Types, Even Abstract Types



(C) Mixing Both Representations

FIGURE 4.23: HAS: Reconstructed **Specification**

Figure 4.23 introduces three different possible and equivalent **Specification** re-documentation results. Indeed, as it is the case for **Configuration** and **Assembly** re-documentation, it is possible to parameterize the re-documentation process of the **Specification**. In Figure 4.23a, the architect has chosen to re-document the **Specification** level by only using the types which exist into the source code. With this option, all the provided ***Interface***s (even unconnected) are kept in the component so it strictly corresponds to a type into the source code. However, both following propositions (Figures 4.23b and 4.23c) consider that, as component roles are meant to be the most abstract component types, it is not always possible to find them into

the source code so they describe the essential of the **Specification**. This is why those options get rid of unconnected provided ***Interface***s. Figure 4.23b introduces the case where all component roles must be as abstract as possible, thus types from source code are not reused as they are to define component roles (*e.g.*, Clock that becomes Clock_role). Finally, Figure 4.23c introduces the inbetween representation which states that if the most abstract type for describing a component role exists in the code then it must be used as component role, however if it does not exist then it is calculated as it is done in Figure 4.23b. However, in this final representation, if the most abstract component corresponds to a concrete type, then it cannot be kept as is, in order to avoid type confusion with component classes from the **Configuration**.

4.3 Generalization

This section discusses how the approach can be generalized.

4.3.1 Discussion

Despite the fact that the previous description of the re-documentation process is oriented on Java and Spring framework, it is actually suitable for other technologies since this approach is meant to be as generic as possible. Indeed, only few features are required for being able to re-document an architecture.

First of all the approach needs at least a kind of deployment descriptor. In the running example, Spring framework is used, however this technology is not mandatory for describing a deployment. As discussed previously a deployment descriptor must provide three kinds of information which are:

- the instances that compose the architecture,
- the classes which are instantiated,
- dependency injections.

Then as long as a deployment descriptor provides these information, it is suitable for the re-documentation process.

Secondly, the language is not fixed, the example targets Java but any other typed object-oriented language would fit the re-documenting process since object-oriented technologies provide inheritance mechanisms and type hierarchies that are highly used for retrieving abstract component roles.

Re-documenting algorithm is discussed in next subsection.

4.3.2 Algorithm

As the run of the algorithm has been described through the HAS example, this subsection does not introduce the whole algorithm which can be found in Appendix C. However, it discusses the generic specification of the main algorithm.

As it has been discussed in previous parts, the input of the re-documentation algorithm is the result of a model to model transformation from a deployment descriptor model to a Dedal model. Also, it has been discussed that a deployment descriptor must provide at least few information which are the instances, the modules which are instantiated and dependency injections. Thus, after a model to model transformation to the Dedal ADL, the resultant Dedal model must contain an **Assembly** that contains component instances, a **Configuration** that contains component classes and a set of classes from the source code which must not be empty. The **Assembly** and **Configuration** architecture descriptions must also contain connections for which source (client attribute) and target (server attribute) are known. However, this input Dedal model, cannot contain neither an interface into component instances / classes nor a role into its **Specification** level. Those are the pre-conditions of Algorithm 3. Those pre-conditions also specify that each component of the given **Configuration** must be instantiated (see line 5) and each connection that exists in the **Assembly** must exist into the **Configuration** (see line 6). Those last two rules ensure a part of the coherence of the Dedal model which are described in previous works [Mok+16b]. Indeed, as it has been described earlier, to be coherent, all component classes from the **Configuration** level must be instantiated into the **Assembly** level, and each connection that exists into the **Configuration** must also be instantiated into the **Assembly**.

Finally at the end of the re-documentation, the obtained Dedal model must comply with coherence principles between the three architecture levels. To do so, it must ensure that component classes interfaces are instantiated into component instances so the component instances are specialized component classes. All roles from the specification must be realized by component classes, connection must be set with the involved interfaces in addition to their client and server components and finally all the connections which exist into the **Specification** must be implemented into the **Configuration**.

Thus, the re-documented three-leveld component-based architecture satisfies all the type constraints which make it consistent and thanks to this it is now possible to maintain this documentation all along the evolution of the software. More over, the strict type theory on which Dedal ADL is based makes it possible from now to detect defaults such as drift or erosion of the software as early as possible. Indeed those defaults imply a lose of coherence and thus architects can be warned and act in consequence, for instance by stopping their change or even propagating them among the three architecture levels so as the documentation to remain consistent [Mok+16b].

Algorithm 3 Main re-documentation algorithm

Require:

1: $\forall\, s \in \{specification.componentRoles, specification.roleConnections\},\ s = \emptyset$

2: $\forall\, s \in \{classes, assembly.componentInstances, assembly.instanceConnections, configuration.componentClasses, configuration.configConnections\},\ s \neq \emptyset$

3: $\forall\, compInst \in assembly.componentInstances,\ compInst.compInterfaces = \emptyset$

4: $\forall\, compCl \in configuration.componentClasses,\ (compCl.compInterfaces = \emptyset) \wedge (compCl.attributes = \emptyset)$

5: $\forall\, compInst \in assembly.componentInstances, (configuration.componentClasses = \bigcup_i compInst_i.instantiates)$

6: $\forall\, ac \in assembly.assemblyConnections,\ (\exists\, cc \in configuration.configConnections \mid ac.clientElem.instantitates = cc.clientElem \wedge ac.serverElem.instantitates = cc.serverElem)$

Ensure:

7: $\forall\, compInstance \in assembly.compInstances($
$(\forall\, instanceConnection \in instanceConnections,$
$instanceConnection.serverInterface \preceq instanceConnection.clientInterface)) \bigwedge$
$(\forall\, interface \in compInstance.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, class \in classes($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name)))))))$

8: $\forall\, compClass \in configuration.compClasses($
$(\forall\, interface \in compClass.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, class \in classes($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name))))))$
$\bigwedge(\forall\, attribute \in compClass.attributes(\exists\, class \in classes($
$\exists\, attr \in class.attributes, attribute.type = attr.type \bigwedge$
$attribute.name = attr.name)))$

9: $\forall\, compClass \in configuration.compClasses,\ (compClass.realizes \neq \emptyset)$
$\wedge(\forall\, role \in compClass.realizes,\ (compClass \preceq role)$

10: $\forall\, role \in specification.specComponents($
$(\forall\, interface \in role.componentInterfaces($
$\exists\, class \in classes($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name))))))$
$\bigwedge\ (\exists\, c \in classes \mid \forall\, interface \in role.componentInterfaces,$
$interface.signatures \subseteq c.methods))$

```
11: procedure MapArchitectureLevels(assembly : Assembly,
                configuration : Configuration,
                specification : Specification,
                classes : Set(Class))
12:     MapComponentInstances(classes, assembly.assmComponents)
13:     MapAssemblyConnections(classes, assembly.assemblyConnections)
14:     MapComponentClasses(classes, assembly.assmComponents,
            configuration.configComponents)
15:     MapConfigConnections(classes, configuration.configConnections
            assembly.assemblyConnections) /* identified during M2M transformation */
16:     specification.componentRoles ← BuildComponentRoles(
            classes, configuration.componentClasses,
            configuration.configConnections)
17:     specification.specConnections ← MapSpecConnections(
            configuration.configConnections)
18: end procedure
```

4.4 Conclusion

This chapter introduced a generic way for re-documenting three-leveled component-based software architectures from object-oriented code and deployment descriptors. The re-documenting architecture is based on Dedal ADL which is especially tailored for managing software evolution. Moreover the type theory on which it is based ensures a good management of the architecture consistency. On a conceptual point of view, it proposes to re-document software as they are implemented through three-leveled component-based software architectures. On a technical point of view, it proposes an algorithm to re-document object-oriented software that use deployment descriptors. The proposed algorithm is meant to be generic and adaptable to any object-oriented technologies and deployment descriptor technologies. It answers research question **RQ1**: It is possible to re-document multi-abstraction level component-based architectures from source code, and it is possible to retrieve abstract design decisions from this re-documentation. However this approach presents several limitations. First, it only re-documents static aspect of software. In future work, the process should include also include a dynamic analysis to consider all the architecture aspects. Second, from a technical point of view, we chose to identify smallest roles as possible. However, this choice can be discussed. Dynamic analysis is needed in order to make more accurate choices about type decoupling. Future work should address this issue. Finally, SpringDSL is not taking all the Spring language into account for now. Thus, future work should enable to take all the language into account.

The next step is now to re-document the history of software project using Dedal and to formalize the concept of version so it can be extended with semantics which consider how the architecture evolved in the past.

Chapter 5

Versioning component-based software architectures

Contents

During its evolution, a software is subject to numerous changes. Those numerous changes lead to numerous versions of the software. In order to keep track of this evolution and record a trace of software history, it is necessary to identify its versions. As it has been discussed in Section 3.1, lot of work has been lead on versioning software. However, few work deals with adding verifiable semantics in version identification and versioning component-based software architectures. Finally, none of those work deal with propagating versions in component-based software architectures. Thus this chapter proposes an approach for identifying and characterizing versions using type-based semantics and also proposes an approach for performing version propagation in the context of multi-leveled architecture descriptions.

5.1 Semantics in versioning

This section presents versioning as it is performed by most common version control systems and discusses a coarse grained approach, based on history, for representing component and architecture evolution.

5.1.1 Definitions and notations

Notations. The used notation is as follows.
$T_1 \prec T_2$: T_1 is a subtype of T_2.
$T_1 \preceq T_2$: T_1 is a subtype of T_2 or equal to T_2.
$T_1 \succ T_2$: T_1 is a supertype of T_2. It is equivalent to $T_2 \prec T_1$
$T_1 \succeq T_2$: T_1 is a supertype of T_2 or equal to T_2. It is equivalent to $T_2 \preceq T_1$
$T_1 \parallel T_2$: T_1 is not comparable to T_2.
$(T_1 \not\preceq T_2) \Leftrightarrow \neg(T_1 \preceq T_2) \Leftrightarrow ((T_1 \succ T_2) \vee (T_1 \parallel T_2))$: T_1 is either a supertype of T_2 or not comparable to T_2.
$(T_1 \not\succeq T_2) \Leftrightarrow \neg(T_1 \succeq T_2) \Leftrightarrow ((T_1 \prec T_2) \vee (T_1 \parallel T_2))$: T_1 is either a subtype of T_2 or not comparable to T_2.
$T_2 \looparrowright T_1$: T_2 replaces T_1.

Version: A version of an evolving artifact represents a particular state of this artifact at a given time [CW98]. The term artifact covers any versionable object such as a file, software object, component, architecture. A versioned artifact is an artifact which is put under versioned control and thus which states are maintained. In contrast, an unversioned item only considers its last state and performs changes by overwriting its current state. The difference between two versions is called a *delta*. An item version must be identifiable.

History: An artifact history is the record of all the states / versions of this artifact through its evolution.

Backward compatible version: An artifact version is said backward compatible if it can substitute its older version. For a new component version C_{new}, being backward compatible with its older version C_{old} means that $C_{new} \preceq C_{old}$. In other words, if C_{new} is subtitutable for C_{old} then it is backward compatible.

5.1.2 Traditional versioning

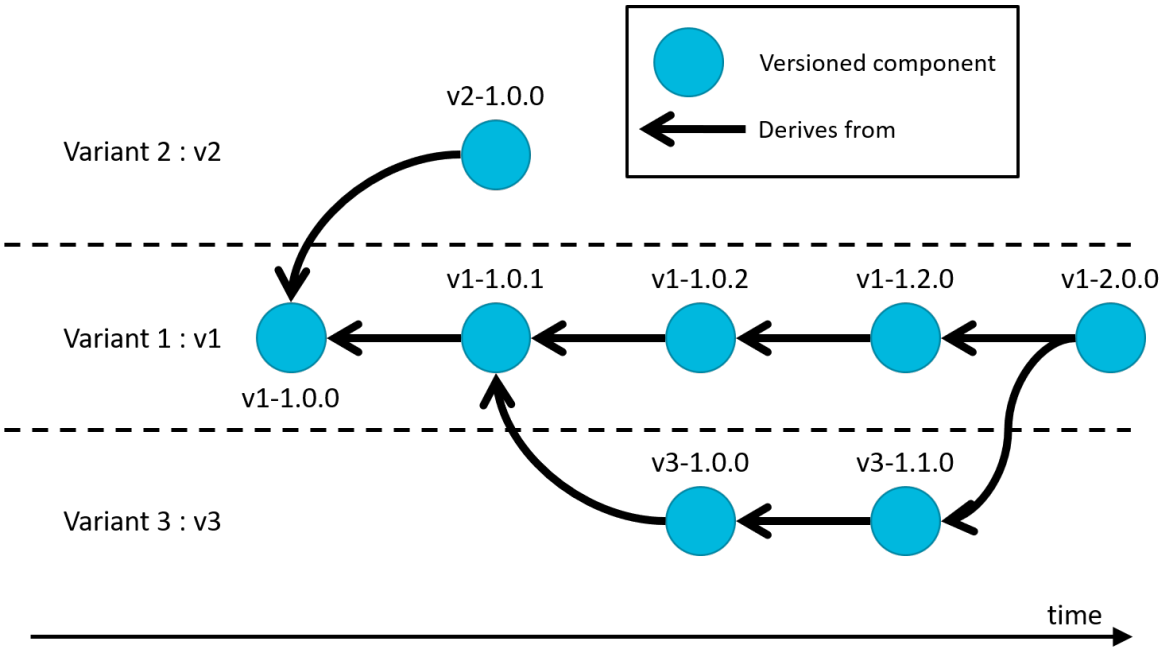


FIGURE 5.1: Traditional versioning

Traditionally versions are represented as a directed acyclic graph [CW98]. Nodes are artifact successive versions and edges represent the derivation relations that show the anteriority of artifact versions. Three kinds of versions can be identified [CW98]:

- **revisions** which intend to replace previous versions,
- **variants** which intend to be alternative versions that are not meant to replace previous versions but to provide alternative versions of the same component according to the developer / architect needs.
- and **cooperation** versions which intend to support collaborative work.

Then those versions must be identified. To do so versions are tagged (see Section 3.1) with a textual identifier since it is an easy and human readable way to identify versions. A very common way for tagging versions is the one proposed in Semantic Versioning 2.0.0[1]. Versions are labeled with a sequence of three numbers <**major**.**minor**.**build**> that corresponds to the type of version that is released. The number meanings are as follows :

- <**major**> stands for the major version identifier and is incremented when version changes make them incompatible.

[1]https://semver.org/#semantic-versioning-200.[last seen 2019.07.03]

- <**minor**> stands for the minor version identifier and is incremented when version changes preserve backward compatibility, such as addition of functionality.
- <**build**> version identifier is incremented for backward compatible bug fixes, patches...

Thus developers / architects can at first sight know whether a new artifact version is backward compatible with its older version and which kind of modifications were made. Additional labels may be used to identify version branches, pre-releases etc.. Such version tagging then makes it possible to precisely identify compatibility of artifact versions. Following the version graph introduced in Figure 5.1, variant branches are identified into the version tag with a prefix (*e.g.*, v1, v2, v3) and versions are tagged independently on their own branch. Successive versions on a single branch are revisions, while leaves of branches are variants. Thus artifact versions *v2-1.0.0* and *v1-2.0.0* are variants since they are leaves on their branches, however *v3-1.1.0* is not a variant since it as been revised by *v1-2.0.0* by a merge operation between *v1-1.2.0* and *v3-1.1.0*. Finally, following Semantic Versioning 2.0.0, it is possible to identify with its label that *v1-2.0.0* is not backward compatible with *v1-1.2.0*. However no information is provided about backward compatibility with *v3-1.1.0* which is a predecessor but on another variant branch.

5.1.3 Problems of current version management systems

As it has been stated before, despite an abundant literature in the field of versioning (see Section 3.1), no approach exists that aims at versioning multi-leveled component-based architectures. Moreover nowadays most important version manager systems such as GitHub[2], SVN[3] etc.. do not consider any kind of semantics in the identification of versions. Indeed, current version management systems only consider textual changes of versioned artifacts. It is useful for being able to version a large amount of different artifacts, however it gets hard for developers / architects to ensure the meaning of their version tags. In other words, despite the fact that the common version tagging identifies build, minor and major versions, there is no formal way to verify that a version is well identified since it is up to the developer / architect to state about it. This lack of semantics may lead to numerous problems which especially relate to the backward compatibility of the new version.

In the context of component-based software architectures, a new version of a component may be backward compatible with its old version. It means that the change to this new component version may not critically impact the architecture. However, if the new component version is not backward compatible, then it may raise several issues on the existing software. The architect could, for instance, release a build of a component (after fixing a bug for example), believing it is backward compatible, and introduce without knowing it a technology gap that can break the architecture. This is why versioning needs verifiable semantics.

[2]https://github.com [Last seen - 2019/08/29]
[3]https://subversion.apache.org/ [Last seen - 2019/08/29]

5.1.4 Substitutability-based versioning

In the context of software components, it is possible to enhance Semantic Versioning approach with the automatic versioning approach developed by Bauml and Brada [BB09]. In their approach they also use the same version tag pattern, however they base the automatic increment on the effect of the source code change on the type of the component. Thus they define four differences between component types which are:

- *None* : C_{new} is of same type as C_{old} thus C_{new} is backward compatible with C_{old}. In this case, the <**build**> number is incremented.
- *Specialization* : $C_{new} \prec C_{old}$ which means that C_{old} is specialized by C_{new} and thus C_{new} is backward compatible with C_{old}. In this case, the <**minor**> number is incremented.
- *Generalization* : $C_{new} \succ C_{old}$ which means that C_{old} is generalized by C_{new} and thus C_{new} is not backward compatible with C_{old}. In this case, the <**major**> number is incremented.
- *Mutation* : $C_{new} \parallel C_{old}$ which means that there is no type relation between C_{old} and C_{new} due to a mutation and thus C_{new} is not backward compatible with C_{old}. In this case, the <**major**> number is incremented.

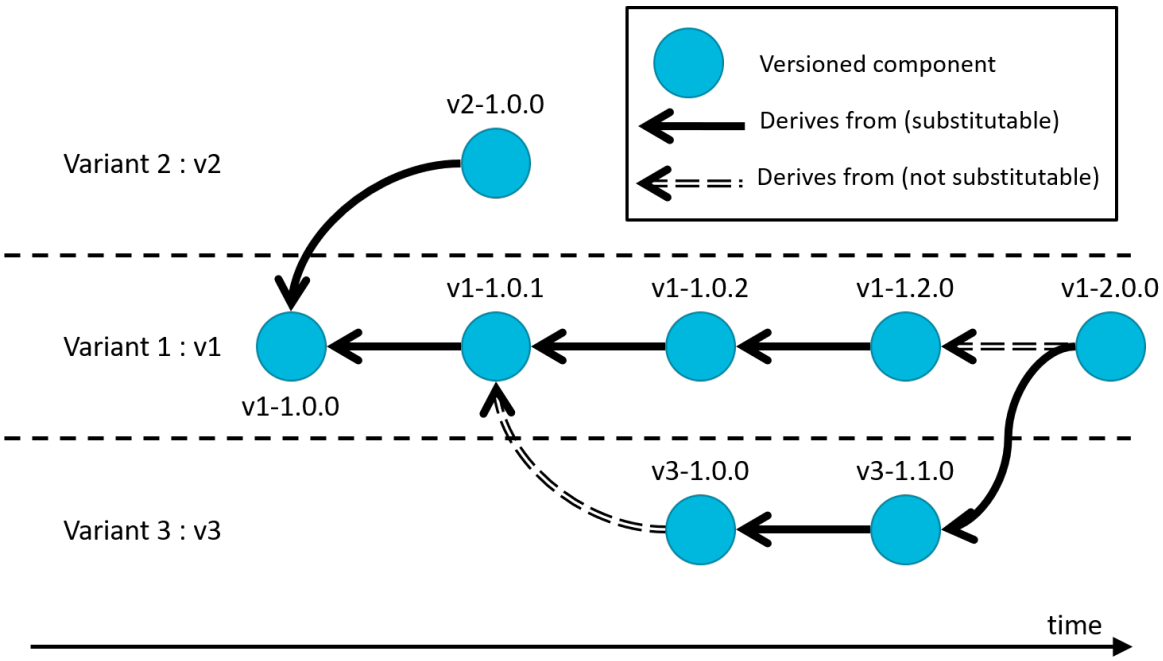


FIGURE 5.2: Substitutability-aware versioning

Thus the work of Bauml and Brada is directly applicable to components at each of the Dedal architecture description level (**Assembly**, **Configuration**, **Specification**). Indeed, Dedal contains a set of strict and formal rules [Mok+16b] which are based on type theory [Aré+07] as discussed in Section 2.3. Those rules ensure that the three architecture description levels of the ADL are coherent at each level and consistent to one another. Thus those type-based rules formally frame the concept of component and architecture substitutability and make

it possible to define version graphs such as presented in Figure 5.2. In this figure, a classical version graph is presented since versions are defined from derivation relations. However, this graph introduces the notion of substitutable derivation. A component version that is substitutable for its previous version must verify that:

- its required interface types are either super types or equal to the previous required interface types,
- its provided interface types are either subtypes or equal to the previous provided interface types,
- its own type is either a subtype or equal to the previous component type.

Moreover, component versions are numbered following Brada and Bauml principle, which ensures the meaning of version tags for further understanding by developers or architects.

This simple version graph and concepts thus make it possible to version components at a single architecture description level. Such a version graph can be reused then for versioning components at multiple abstraction levels.

5.1.4.1 Versioning components at multiple abstraction levels

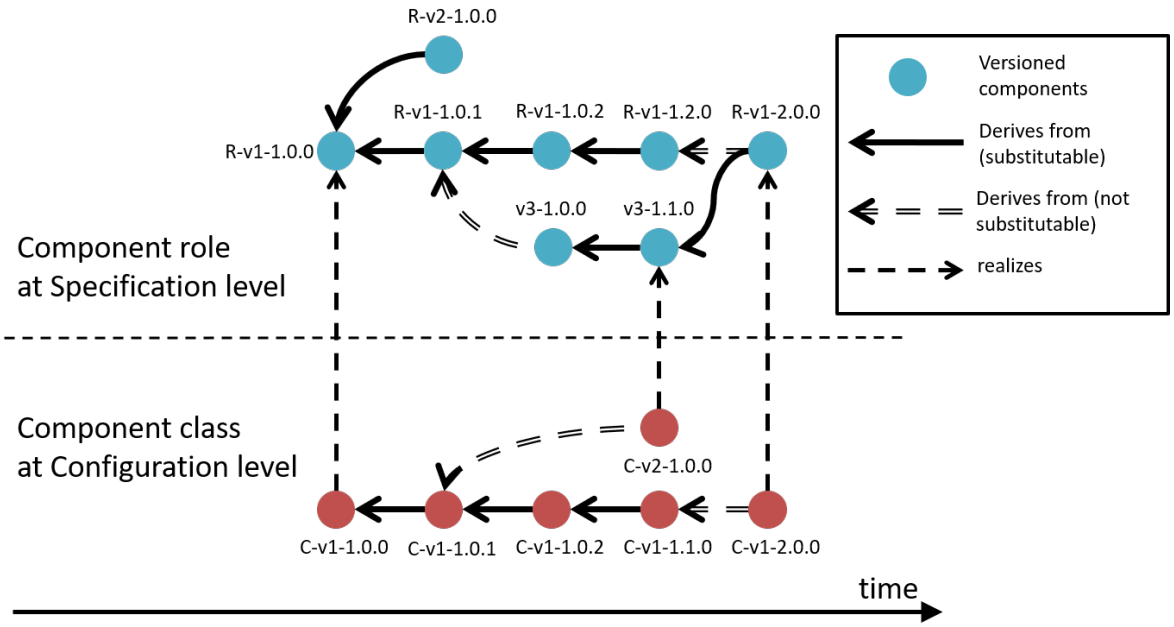


FIGURE 5.3: Multilevel component versioning

Figure 5.3 shows two version graphs that correspond to two abstraction levels of component descriptions (a component role at the Specification level and a component class at the Configuration level). As it is shown in this figure, components follow their own evolution process and are not necessarily versioned neither in the same time, nor in the same way. Thanks to the notion of substitutability, it is possible to navigate into the version graph and to instantly know if some version of a component role is realized by a component class or if a component class realizes a component role.

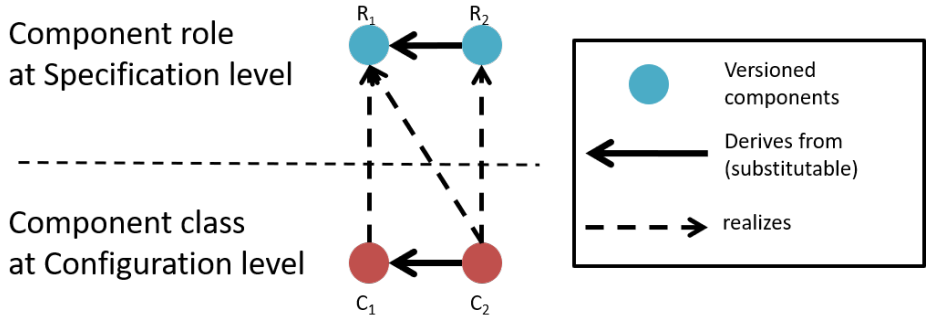


FIGURE 5.4: Finding transitively realized roles using substitutability

Considering the transitive substitutability relation, as it is shown in Figure 5.4:
Given two component roles R_1 and R_2 and two component classes C_1 and C_2, and given the following subtype relations : $R_2 \preceq R_1$ and $C_2 \preceq C_1$.
If C_1 realizes R_1 then C_2 also realizes R_1.
If C_2 realizes R_2 then C_2 also realizes R_1.

For instance in Figure 5.3, component class *C-v1-1.0.0* realizes component role *R-v1-1.0.0*, moreover, it is easy to find out that component class versions *C-v1-1.0.1*, *C-v1-1.0.2*, *C-v1-1.1.0* also realize the same component role since they all are substitutable for component class *C-v1-1.0.0*. In the opposite direction, component class *C-v1-2.0.0* realizes component role *R-v1-2.0.0* but also its previous versions *R-v3-1.1.0* and *R-v3-1.0.0*. Thus, adding information about version semantics into the version graph makes it possible to infer compatibility properties of given elements with all their predecessors. Now we can transpose this concept at architecture description levels.

5.1.4.2 Versioning multiple component-based architectures description levels

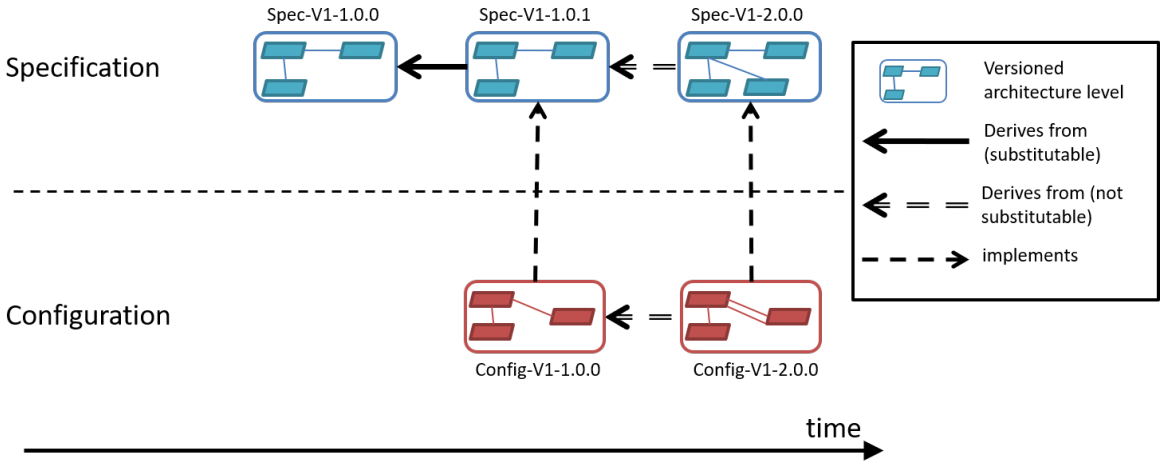


FIGURE 5.5: Multilevel architecture versioning

In addition to components, it is also possible to version architecture levels of Dedal. Indeed, as architecture levels are related to types, it is also possible to apply substitutability

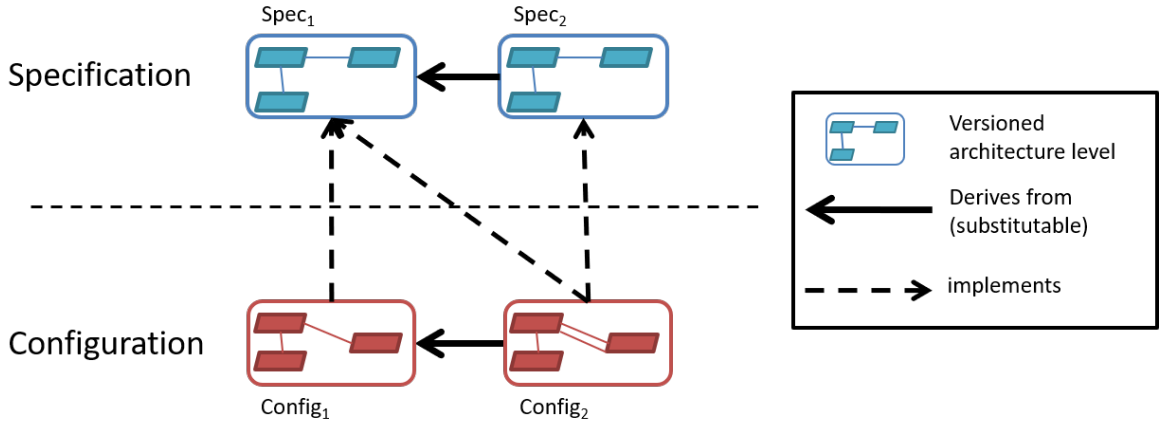


FIGURE 5.6: Finding transitively implemented Specifications using substitutability

principles between two architecture levels. Therefore, the same semantics centered on retro-compatibility of architecture levels can be applied between their successive versions. As it is shown in Figure 5.5, versions of architecture levels may evolve independently, however as previously presented with realized component roles, all the implementations of a **Specification** version can be identified automatically as well as all the **Specification** versions that are being implemented by a **Configuration** version.

As previously and because substitutability principle also applies to architecture description levels (as they are also types) implementation relations can be transitively discovered as shown in Figure 5.6.
Given two Specifications $Spec_1$ and $Spec_2$ and two Configurations $Config_1$ and $Config_2$, and given the following subtype relations : $Spec_2 \preceq Spec_1$ and $Config_2 \preceq Cconfig_1$.
If $Config_1$ implements $Spec_1$ then $Config_2$ also implements $Spec_1$.
If $Config_2$ implements $Spec_2$ then $Config_2$ also implements $Spec_1$.

Thus, in Figure 5.5, because Specification $Spec - v1 - 1.0.1$ is substitutable for Specification $Spec - v1 - 1.0.0$, $Spec - v1 - 1.0.0$ is implemented by $Config - v1 - 1.0.0$.

Then it is time to consider whole three leveled component-based architectures.

5.1.4.3 Versioning three-leveled component-based architectures

Finally, as it is shown in Figure 5.7, a third point of view may be versioned, which corresponds to the whole architecture itself composed of the three architecture description levels. As previously, version substitutability can be applied. In this point of view, each architecture version is composed of three architecture level descriptions that follow their own version history while they are also composed of components that can have their own histories. Version tags are not necessarily related one to another between each point of view (component, architecture level, whole architecture).

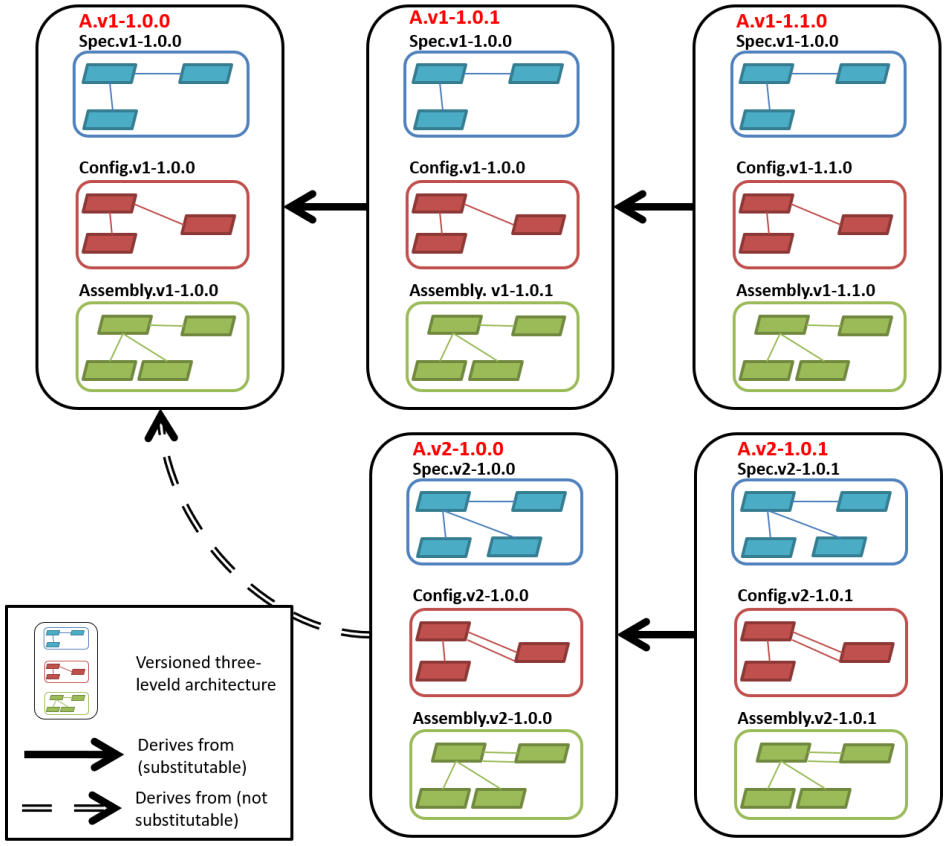


FIGURE 5.7: Dedal three-leveled architecture versioning

5.2 Identification of architectural changes, version characterization

The previous section introduces a very generic way for identifying and representing component / architecture versions. It also shows how it is possible to automatically visualize backward compatibility of those versions all along their history. Thus in order to ensure the meaning of such representation, the derivation relation identification as well as version numbering must be supported by a strong, automatic and verifiable process. Then this section discusses which are the architectural changes that are relevant for studying substitutability of components / architectures. This identification of architectural changes is based on previous work of Mokni *et al.* [Mok+16a] in the context of a formal architecture evolution implemented with language B [Abr96].

5.2.1 Identifying and categorizing component-based architecture changes

First of all, it is necessary to identify which architectural changes may occur. There are three types of changes: changes that occur at the component level, changes that occur at one of the three Dedal architecture description levels and changes that occur in the global architecture.

Changes at component level: At component level, relevant changes concern interfaces and component attributes (for component classes). Component attributes may be added, deleted, or replaced as may be component interfaces. Moreover, changes may occur into

interface type descriptions which are signature addition, deletion and replacement. Finally, a parameter of a signature may also be added, deleted or replaced.

Changes at one of the three Dedal architecture levels: At Dedal architecture description levels, relevant changes concern components and connections that can be added, deleted or replaced. At this level, changes which occur at component level are considered as component replacement since a new version of a component replaces its previous version.

Changes at the whole architecture level: At global architecture level, relevant changes concern architecture description levels (**Assembly**, **Configuration** and **Specification**) replacements. As previously, changes that affect architecture levels are considered as replacements since a new version of an architecture level replaces its previous version.

5.2.1.1 Type-based architectural changes categorization

The substitutability relation has already been formalized in Dedal. Table 5.1 summarizes the changes that can affect the architecture and categorizes them according to the substitutability of the new artifact version (substitutable or non-substitutable). Thus, this table gives the rules that characterize versions for components, architecture levels and the whole architecture, according to the kind of change that is performed. As Dedal types are based on the work of Arevalo *et al.* [Aré+07] which derives from the type theory defined by Liskov an Wing [LW94], substitutability concepts that are defined in Dedal also derive from this theory. Thus parameters in interface signatures follow contravariance principles as do required interfaces, while for the other Dedal artifacts being substitutable for another artifact means being a subtype of the substituted artifact.

In Dedal, versionable artifacts are as follows:

- **Components** which can be of three different types (**CompInstance**, **CompClass** and **CompRole**),
- **ArchitectureDescriptions** which can also be of three types (**Assembly**, **Configuration** and **Specification**) and
- **DedalDiagram** which is the global model that contains the three architecture description levels.

With this in mind, the conditions that are listed in Table 5.1 make it possible to fully automate the component, architecture levels descriptions and even whole three-leveled architectures are versioned. Indeed, following the proposition of Bauml and Brada [BB09], by applying their approach to Dedal, the differences they identify can be identified as follow:

- *None*: means that the type has not changed, this can typically be the case when the core of a method has changed without changing its signature or when a component instance is initialized differently. The kind of change that leads to such outcome is not listed in Table 5.1 since it is not a structural change that affects the architecture.

	Substitutable	Non-substitutable
Component changes	Interface signatures	
	Parameter replacement: $param_{new} \succeq param_{old}$	Parameter replacement: $(param_{new} \parallel param_{old})$ $\bigvee(param_{new} \preceq param_{old})$
	Parameter deletion	Parameter addition
	Interface type	
	Signature replacement: $sig_{new} \preceq sig_{old}$	Signature replacement: $(sig_{new} \parallel sig_{old}))$ $\bigvee(sig_{new} \succeq sig_{old})$
	Signature addition	Signature deletion
	Provided Interface	
	Interface Type replacement: $Int\_type_{new} \preceq Int\_type_{old}$	Interface Type replacement: $(Int\_type_{new} \parallel Int\_type_{old})$ $\bigvee(Int\_type_{new} \succeq Int\_type_{old})$
	Required Interface	
	Interface Type replacement: $Int\_type_{new} \succeq Int\_type_{old}$	Interface Type replacement: $(Int\_type_{new} \parallel Int\_type_{old})$ $\bigvee(Int\_type_{new} \preceq Int\_type_{old})$
	Component	
	Attribute replacement: $attr_{new} \preceq attr_{old}$	Attribute replacement: $(attr_{new} \parallel attr_{old})$ $\bigvee(attr_{new} \succeq attr_{old})$
	Attribute addition	Attribute deletion
	Provided Interface addition	Provided interface deletion
	Provided Interface replacement: $Int_{new} \preceq Int_{old}$	Provided interface replacement: $(Int_{new} \parallel Int_{old}) \bigvee(Int_{new} \succeq Int_{old})$
	Required interface deletion	required interface addition
	Required interface replacement: $Int_{new} \succeq Int_{old}$	required interface replacement: $(Int_{new} \parallel Int_{old}) \bigvee(Int_{new} \preceq Int_{old})$
Architecture level changes	Architecture description level	
	component replacement : $component_{new} \preceq component_{old}$	component replacement: $(component_{new} \parallel component_{old})$ $\bigvee(component_{new} \succeq component_{old})$
	Component addition	Component deletion
	Connection replacement: $(Int_{serv-new} \preceq Int_{serv-old})$ $\bigwedge(Int_{client-old} \preceq Int_{client-new})$	Connection replacement: $(Int_{serv-new} \npreceq Int_{serv-old})$ $\bigvee(Int_{client-old} \npreceq Int_{client-new})$
	Connection addition	Connection deletion
Global Architecture changes	Global Architecture	
	Assembly replacement: $asm_{new} \preceq asm_{old}$	Assembly replacement: $asm_{new} \npreceq asm_{old}$
	Configuration replacement: $(config_{new} \preceq config_{old})$ $\bigwedge(config_{new} \succeq asm)$	Configuration replacement : $\neg[(config_{new} \preceq config_{old})$ $\bigwedge(config_{new} \succeq asm)]$
	Specification replacement: $(spec_{new} \preceq spec_{old})$ $\bigwedge(spec_{new} \succeq config)$	Specification replacement : $\neg[(spec_{new} \preceq spec_{old})$ $\bigwedge(spec_{new} \succeq config)]$

TABLE 5.1: Substitutability-based architectural changes

However, if such a change occurs then the <**build**> number is incremented to give developers / architects information about the change. And consequently if a component <**build**> number is incremented, then the containing architecture <**build**> number is also incremented and then the three-leveled architecture <**build**> number is also incremented.

- *Specialization*: means that the new artifact version is a subtype of its previous version. It happens in case of any substitutable change listed in Table 5.1. In this case, the <**minor**> number of considered versioned artifact is incremented.
- *Generalization*: means that the new artifact version is a super-type of its previous version. In this case, the <**major**> number of considered versioned artifact is incremented.
- *Mutation*: means that the new artifact version is neither a subtype nor a super-type of its previous version. In this case, the <**major**> number of considered versioned artifact is incremented.

Then, as identifying versions becomes possible following these rules, it is necessary to define a version meta-model that embeds those concepts.

5.2.2 Version meta-model

In the previously discussed versioning approach, it is easy to identify backward compatible components or architectures on a same variant branch on the basis of their version number. However, such identification of backward compatible artifacts between variants cannot be realized in the same way. An easy way to represent backward compatibility in this case is a visual representation such as introduced in Figure 5.2. To do so we define a meta-model that is designed to fully represent version derivation links.

Figure 5.8 presents the proposed meta-model for representing versions with their semantics. An **History** is composed of **AbstractBranch**es and **Precedence** relations. An **AbstractBranch** represents version branches. It can be of two types, **WorkingBranch** for representing branches that are not necessarily designed to contain releases, and **Variant** that is designed to represent a branch which last artifact version is a variant of another **Variant** branch last artifact version. An **AbstractBranch** is thus composed of **AbstractArtifact**s that can be any kind of versionable artifact and a **Tag** for identifying branches. A **Precedence** relation can be of two types. **BranchPrec** connects two **AbstractBranch**es that play the respective roles of *predecessor* and *successor*. It stands for representing precedence of branches in order to keep track of branch history itself. **Precedence** can also be of type **ArtifactPrec** for representing derivation relations between **AbstractArtifact** (versions). Moreover, in order to add semantic to the derivation relation between versions, **ArtifactPrec** can also be of type **RetrocompatibleArtifactPrec** that is designed to represent the precedence relation between a *successor* that is substitutable for its *predecessor*. In an **ArtifactPrec** the *successor* is considered as a revision of its *predecessor* if its belongs to the same **AbstractBranch**. Finally, an **AbstractArtifact** also contains a **Tag** for being identified within its **AbstractBranch**. Thus variants

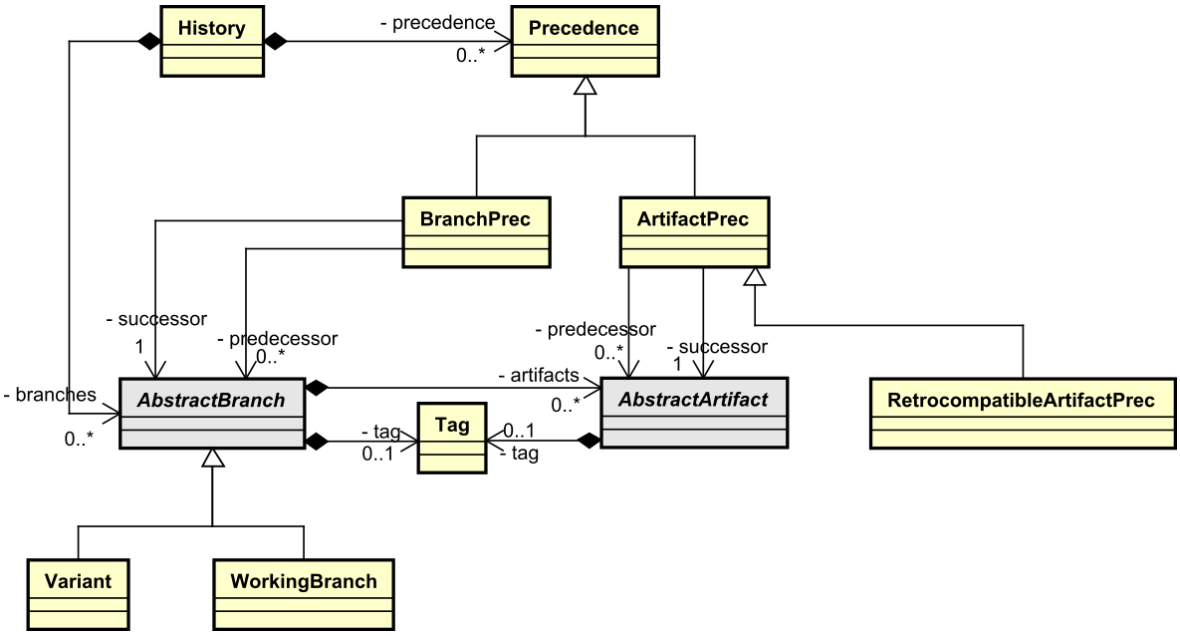


FIGURE 5.8: Metamodel for semantic versioning

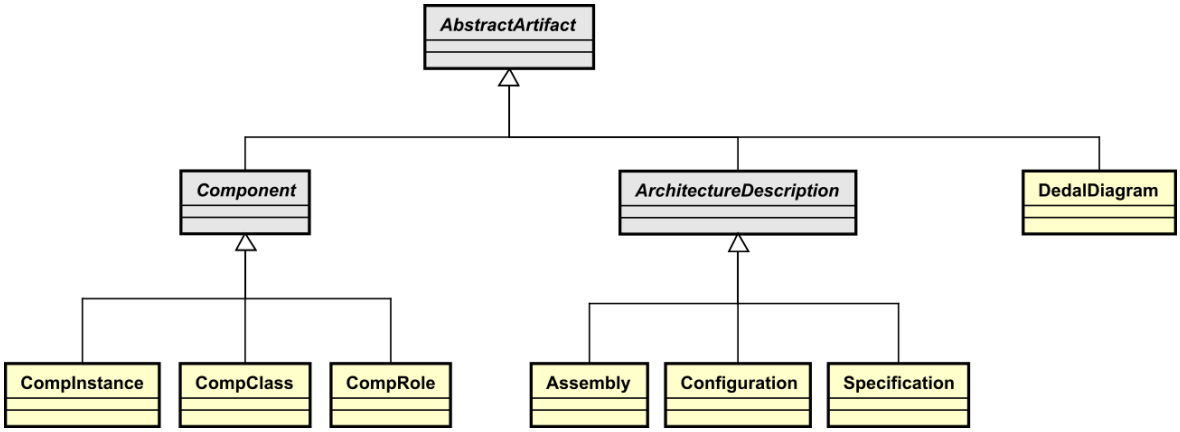


FIGURE 5.9: Dedal versionable artifacts

of an artifact are identified with a unique identifier composed of the **AbstractBranch Tag** as a prefix followed by the **AbstractArtifact Tag**. Thus, thanks to this metamodel, it is possible to semantically version three-leveled Dedal architectures.

5.2.3 Three-leveled version meta-model

As it has been discussed before, semantic versioning can be applied from several perspectives in Dedal. Thus Figure 5.9 completes the metamodel shown in Figure 5.8 and presents how it can be specialized to comply with the Dedal language. Thus, **Component**s, **ArchitectureDescription**s and the whole **DedalDiagram** (global Dedal architecture) can be versioned. Thus it allows to version architectures at three abstraction levels since a **Component** can be any of **CompInstance**, **CompClass** and **CompRole** types and in the same way, an **ArchitectureDescription** can be any of the three Dedal architecture description level.

However, in the context of a three-level component-based software architecture, artifacts cannot be replaced by newer version independently from their architecture siblings or containers. In some cases, a propagation of changes to other components or architecture levels is needed. Thus versioning an artifact may imply to version other artefacts. This is what is discussed in the next section.

5.3 Predicting version propagation

This section discusses the concept of version propagation within the three architecture description levels of Dedal [Le +17].

5.3.1 Typology of architectural change impact

Version propagation is inferred from Dedal substitutability concept. However, a substitutable change does not necessarily means that the coherence of the architecture is preserved. This is why this section introduces a typology of architectural change impact. This typology is based on component changes which are introduced in Table 5.1 at architecture level point of view. Indeed, component changes can be seen as a component replacement into an architecture description level. Moreover, component substitutability is sufficient for studying the impact of change in an architecture description level and its adjacent levels. The aim of this typology is to be able to differentiates and identify impacts change may have on architecture description and to decide wether a change is compatible or not with an existing architecture.

Thus, the relevant change operations are as follows :

- *Adding* new artifacts.
- *Removing* artifacts.
- *Replacing* artifacts with others that may be substitutable for the previous ones or not.

At architecture description level, a component may be either replaced by a component that is *substitutable* for the replaced component or by a component that is *not substitutable* for the replaced component.

Moreover, in a component-based architecture, several kinds of artifacts are subject to change:

- components themselves, this is the most coarse-grained change,
- the interfaces of a component, which is a finer-grained change,
- signatures that is the finest-grained change.

Finally, in a change analysis context, several outcomes can be expected:

- the change has no impact on the architecture,
- the change impacts its own architecture level,

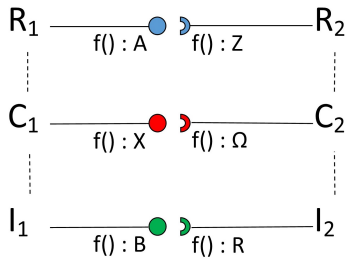


FIGURE 5.10: Base-Case: Functionality Connection Within a Three-Level Component-Based Architecture

- the change impacts adjacent architecture levels,
- the change impacts its own architecture level and adjacent architecture levels.

When studying architecture versioning in a Dedal development, it gets very relevant to study which is the initial level of change. Indeed, one of the most important aspects of using a three-level ADL is being able to perform co-evolution of those levels according to the origin of the perturbation. This is what it is discussed in the next section.

5.3.2 Change impact analysis

As Dedal is a three-level ADL, a change may occur (initiate) at any of its architecture levels. This study is based on replacement of provided / required functionality signature since such replacement is sufficient to analyze change impact. Indeed, component interactions are defined by their interfaces which are implied into connections. Thus, replacing a signature is equivalent to replacing its container interface and thus its container component.

Functionality substitutable for another. For a provided functionality sp_{new}, being substitutable for another functionality sp_{old} means that (1) the return type of sp_{new} is equal or subtype [LW94] of the return type of sp_{old} and (2) that the input parameters of sp_{old} are equal or subtypes of the ones of sp_{new} [Aré+07]. Conversely, for a required functionality sr_{new}, being substitutable for another functionality sr_{old} means that (1) the return type of sr_{new} is equal or a supertype of the return type of sr_{old},and (2) that the input parameters of sr_{old} are equal or supertypes of the ones of sr_{new}.

Figure 5.10 shows a basic example of a three-level architecture. It represents the three Dedal abstraction levels. The specification is composed of two component roles R_1 and R_2. They are realized respectively by the component classes C_1 and C_2, that are in turn instantiated by respectively I_1 and I_2.

5.3.2.1 Versioning at Specification Level

Table 5.2a summarizes the effects of role replacement on the architecture. Let us suppose that role R_1 is replaced by a new version R_1' which provides a functionality $f() : Y$. Several outcomes can be observed:

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow A$
Non-propagation
$X \preceq Y \preceq Z$
Propagation
Inter-level | Intra-level
$(Y \parallel X) \vee (Y \prec X)$ | $(Y \parallel Z) \vee (Y \succ Z)$
$(Y \parallel X) \wedge (Y \parallel Z)$

(A) Specification Level

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow X$
Non-propagation
$B \preceq Y \preceq A$
Propagation
Inter-level | Intra-level
$(Y \not\preceq A \Rightarrow \uparrow) \vee (Y \not\succeq B \Rightarrow \downarrow)$ | $Y \not\preceq \Omega$
$[(Y \not\preceq A) \vee (Y \not\succeq B)] \wedge (Y \not\preceq \Omega)$

(B) Configuration Level

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow B$
Non-propagation
$Y \preceq X$
Propagation
Inter-level | Intra-level
$Y \not\preceq X$ | $Y \not\preceq R$
$Y \not\preceq R$

(C) Assembly Level

TABLE 5.2: Replacing Components: Providing a Functionality

- **The change has no impact on the architecture (Table 5.2a.Non-propagation).** This case happens when the version is not propagated. The condition of non-propagation is given as follows, $X \preceq Y \preceq Z$ for any replacement type. Y can either be substitutable for A or not. It means that the new version of the role does not break architectural coherence since it is compatible with other roles within **Specification**, and all the component classes that previously realized the replaced role remain subtypes of the new role. The change is not propagated.
- **The change impacts its own architecture level (Table 5.2a.Propagation).** It is a case of **intra-level propagation**, which occurs if Y is a supertype of Z or if they are not comparable. This happens when the compatibility of component roles involved in a connection is broken but the new component role is still realized into the **Configuration** level.
- **The change impacts adjacent architecture levels (Table 5.2a.Propagation).** This is a case of **inter-level propagation**, which occurs if Y is a subtype of X or if they are not comparable. This happens when the component classes that previously realized the old component role do not realize the new component role.
- **The change impacts its own architecture level and adjacent architecture levels (Table 5.2a.Propagation).** It is a case of **inter and intra-level propagation**, that is a combination of both propagation conditions, which is Y is not comparable neither to X nor Z. This happens when both inter-level and intra-level coherence are broken.

5.3.2.2 Versioning at Configuration Level

Table 5.2b summarizes the effects of component class replacement on the architecture. Component class C_1 (Figure 5.10) is replaced by a component class C_1', which provides a functionality $f() : Y$. Then several outcomes can be observed:

- **The change has no impact on the architecture.** The change is not propagated. The condition of non-propagation is given by $B \preceq Y \preceq A$ for any replacement type. Y can either be substitutable for X or not. $(Y \preceq A)$ ensures C_1' realizes R_1 and $(Y \succeq B)$

ensures I_1 can be used as an instance of C_3. This happens when the change does not break neither intra-level nor inter-level architecture coherence.

- **The change impacts its own architecture level.** As previously, it is a case of **intra-level propagation**. This happens if Y is not a subtype of Ω. However, this condition also implies at least a propagation to the **Specification** since $(A \prec \Omega) \vdash (Y \not\preceq \Omega) \Rightarrow (Y \not\preceq A)$
- **The change impacts adjacent architecture levels.** Since **Configuration** is the intermediate architecture level, then change may be propagated:
  - **To the specification (↑)** if Y is not a subtype of A.
  - **To the assembly (↓)** if Y is not a supertype of B.
- **The change impacts its own architecture level and adjacent architecture levels.** The change may be propagated in every direction with any combination of the previously discussed conditions. The change may be propagated in one, two or three directions at a time.

5.3.2.3 Versioning at Assembly Level

Table 5.2c summarizes the impact of replacing I_1 (Figure 5.10) by a third component instance I_1', which provides a functionnality $f() : Y$. The possible outcomes that can be observed are as follows:

- **The change has no impact on the architecture.** As previously, **the version is not propagated.** It is the case when $Y \preceq X$ for any type of replacement (*substitutable* or *not-substitutable*). This condition ensures that I_1' instantiates C_1 and is compatible with I_2.
- **The change impacts its own architecture level.** There is an **intra-level propagation** if Y is not a subtype of R. However, this is also a sufficient condition for implying an inter-level propagation.
- **The change impacts adjacent architecture levels.** There is an **inter-level propagation** if Y is not a subtype of X.
- **The change impacts its own architecture level and adjacent architecture levels.** As said before the condition of **intra-level propagation** also implies **inter-level propagation**.

Tables 5.3a, 5.3b and 5.3c give the rules of the symetric change impact analysis that corresponds to the replacement of required functionality at the three architecture levels (R_2, C_2 and I_2 are replaced by a component that requires a functionality $f() : Y$).

5.3.2.4 Propagation example

Figure 5.11 introduces a simple case of three-level version propagation. In this example, the considered type hierarchy is $B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$ and the change that is considered

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow Z$
Non-propagation
$A \preceq Y \preceq \Omega$
Propagation
Inter-level | Intra-level
$Y \not\preceq \Omega$ | $Y \not\succeq A$
$(Y \parallel \Omega) \wedge (Y \parallel A)$

(A) Specification Level

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow \Omega$
Non-propagation
$Z \preceq Y \preceq R$
Propagation
Inter-level | Intra-level
$(Y \not\succeq Z \Rightarrow\uparrow) \vee (Y \not\preceq R \Rightarrow\downarrow)$ | $Y \not\succeq X$
$[(Y \not\succeq Z) \vee (Y \not\preceq R)] \wedge (Y \not\succeq X)$

(B) Configuration Level

Hypothesis on types
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$
$Y \nrightarrow R$
Non-propagation
$Y \succeq \Omega$
Propagation
Inter-level | Intra-level
$Y \not\succeq \Omega$ | $Y \not\succeq B$
$(Y \not\succeq \Omega) \wedge (Y \not\succeq B)$

(C) Assembly Level

TABLE 5.3: Replacing Components: Requiring a Functionality

happens on a provided functionality. X is replaced by Y with $(Y \preceq R) \bigwedge (Y \parallel \Omega)$. The result of the change impact analysis is thus given by $[(Y \not\preceq A) \vee (Y \not\succeq B)] \wedge (Y \not\preceq \Omega)$. This result corresponds to an intra-level and an inter-level (up and down) propagation of the version since the architectural coherence has been compromised.

The initial change occurs on component class C_1 which is replaced by C_1' at **Configuration** level. Thus, the type of the functionality of the new component is no more compatible with component C_2, then the initial change must be propagated in the new configuration to replace C_2 by a more suitable component version (C_2'). Next, as the coherence between the **Configuration** and the **Specification** has been broken, the change is propagated to the **Specification** by replacing both component roles that were previously realized by C_1 and C_2. In the same way, change is propagated to the **Assembly** by replacing the previous C_1 instance. At the end of this propagation, a new version of the architecture has emerged which is not substitutable for the previous one since at least one change is not substitutable. Indeed C_1' is not substitutable for C_1 since $(Y \parallel \Omega)$ while the previous provided interface type was X which is a subtype of Ω.

5.3.2.5 Generalization

1 to n replacement. The discussed analysis considers 1 to 1 replacement operations. However, this is sufficient to describe the propagation problem. Indeed, if a single component role is realized by multiple (n) component classes, then those component classes are considered together as a single composite component class that realizes a role. The same operation

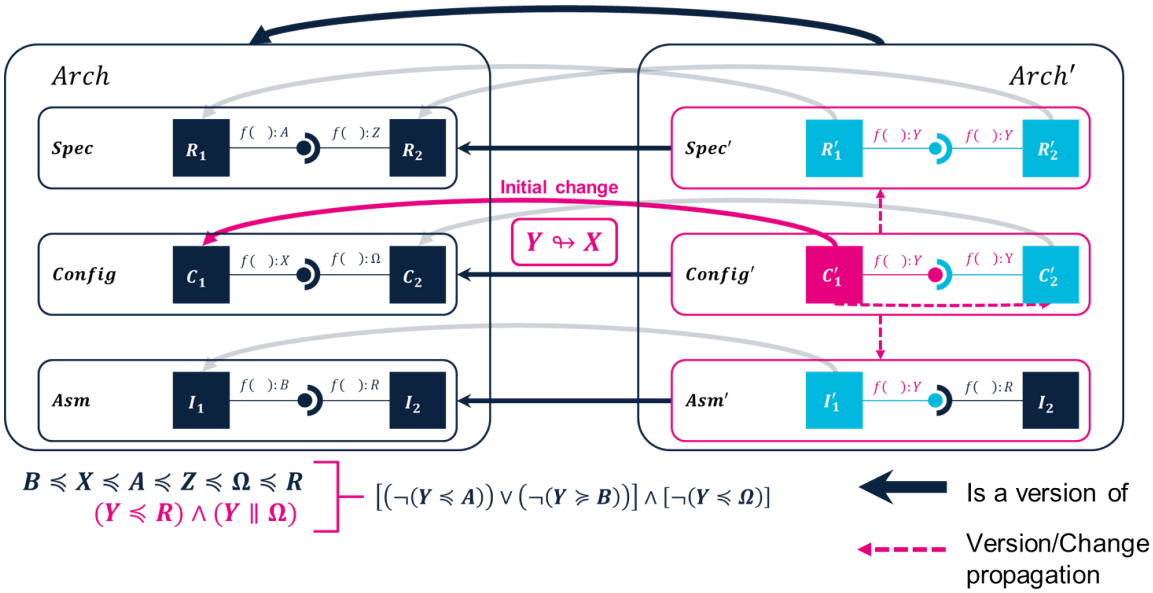


FIGURE 5.11: Propagating version at three architecture levels

can be realized in the case where a single component realizes multiple component roles. Those component roles are seen as a single component role which exposes the interfaces that describe the multiple roles.

Multiple connections. In a Dedal architecture, it is possible to connect a component interface to several interfaces. A solution to generalize such cases is to separately study change impacts on each connection.

As a result of this analysis, it turns out that substitutability is a good criterion for predicting impact on intra-level consistency. However, this is not a sufficient one and a more detailed approach is needed for studying impact on inter-level coherence as it is shown in the previously discussed tables.

5.4 Example of three-leveled architecture versioning

This section introduces an example of three-leveled component-based architecture versioning. This example is based on an excerpt of the HAS that is discussed in Chapter 4. Then Figure 5.12 introduces the type hierarchy that is used in the following example.

The components that are derived from this hierarchy are introduced with their version graph in Figure 5.13. Each version graph is represented as a three-level version graph so realization and instantiation relations are shown. Thus R_{Orch} is the role that corresponds to the abstract type *Orchestrator*, R_{Clock} is the role that corresponds to the type *Clock* and R_{Light} is the role that corresponds to the type *Light*. Following the same principle, C_{Orch}, C_{Clock} and C_{Light} respectively correspond to types *OrchestratorImpl* (that cannot be abstract), *Clock* and *Light*. I_{Orch}, I_{Clock}, I^1_{Light} and I^2_{Light} are respectively component instances of C_{Orch}, C_{Clock}

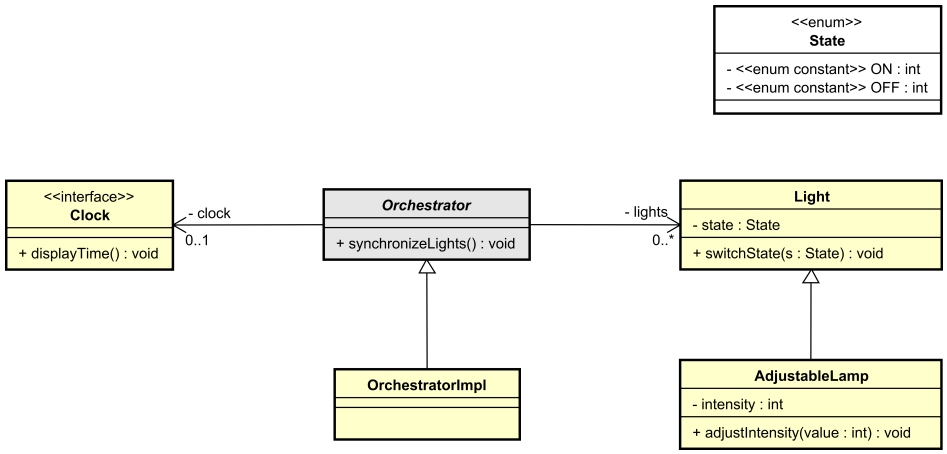


FIGURE 5.12: HAS type hierarchy extract

and C_{Light}. In order to ease the understanding of the example, all the component levels follow the same versioning process. In other words, $R_{Orch} - v1 - 1.0.0$ is an incompatible successor $R_{Orch} - v1 - 2.0.0$ so do $C_{Orch} - v1 - 1.0.0$ and $I_{Orch} - v1 - 1.0.0$ that also start with the same version number.

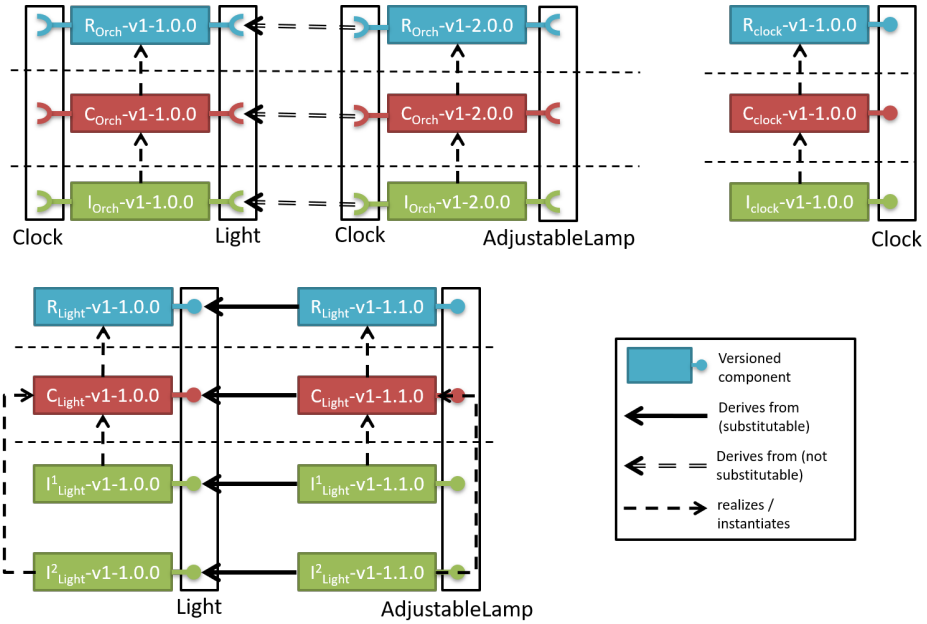


FIGURE 5.13: HAS components version graph

Scenario. Considering the initial HAS architecture that is introduced in Figure 5.14, the following changes occur. First of all, a new component instance ($I^2_{Light} - v1 - 1.0.0$) is added to the HAS **Assembly** and then the component role $R_{Orch} - v1 - 1.0.0$ is replaced by $R_{Orch} - v1 - 2.0.0$.

As shown in Figure 5.15 a component instance $I^2_{Light} - v1 - 1.0.0$ is added to the HAS Assembly. According to Table 5.1, the outcome of a component addition at an architecture description level is a new version of the architecture level that is substitutable and thus

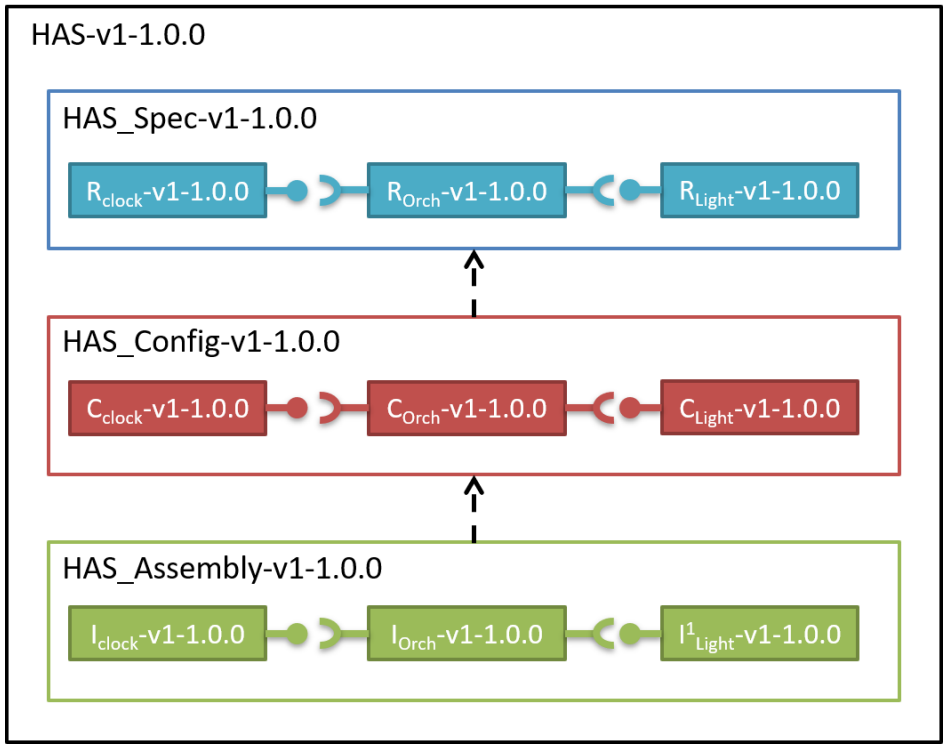


FIGURE 5.14: HAS initial architecture

backward compatible. Then the <**minor**> version number of $HAS\_Assembly - v1 - 1.0.0$ is incremented to become $HAS\_Assembly - v1 - 1.1.0$. The addition of the component instance does not break any connection that would imply to propagate the change within the architecture, then no version propagation is needed and only the <**minor**> version number of the global architecture is incremented to reflect the minor change that occurred at the Assembly level. Thus the new version of $HAS - v1 - 1.0.0$ is $HAS - v1 - 1.1.0$.

The second part of the scenario consists in performing an incompatible change. Thus, as introduced in Figure 5.16, component role $R_{Orch} - v1 - 1.0.0$ is replaced by $R_{Orch} - v1 - 2.0.0$ that is not backward compatible. Thus the <**major**> version number of $HAS\_Spec - v1 - 1.0.0$ is incremented to become $HAS\_Spec - v1 - 2.0.0$. It corresponds to the case of inter and intra-level propagation that is presented in Table 5.3a. Then the version must be propagated within the Specification level and also to the Configuration level. At Specification level, as the connection with $R_{Light} - v1 - 1.0.0$ changed (it requires a subtype of the interface type which is provided by the R_{Light} component) then another version of R_{Light} must be used, so $R_{Light} - v1 - 1.0.0$ is replaced by $R_{Light} - v1 - 1.1.0$ which provides a compatible interface type. Then this change is propagated to the Configuration level where, following the same principles, $C_{Orch} - v1 - 1.0.0$ is replaced by $C_{Orch} - v1 - 2.0.0$ and $C_{Light} - v1 - 1.0.0$ is replaced by $C_{Light} - v1 - 1.1.0$ to make Configuration consistent with Specification again. The <**major**> version number of the Configuration is also incremented. Finally, as the change that has been propagated to the Configuration broke the instantiation relation between the Assembly and the Configuration, then it is propagated to the Assembly level. Thus as previously, $I_{Orch} - v1v1.0.0$ is replaced by $I_{Orch} - v1 - 2.0.0$, $I^1_{Light} - v1 - 1.0.0$ and

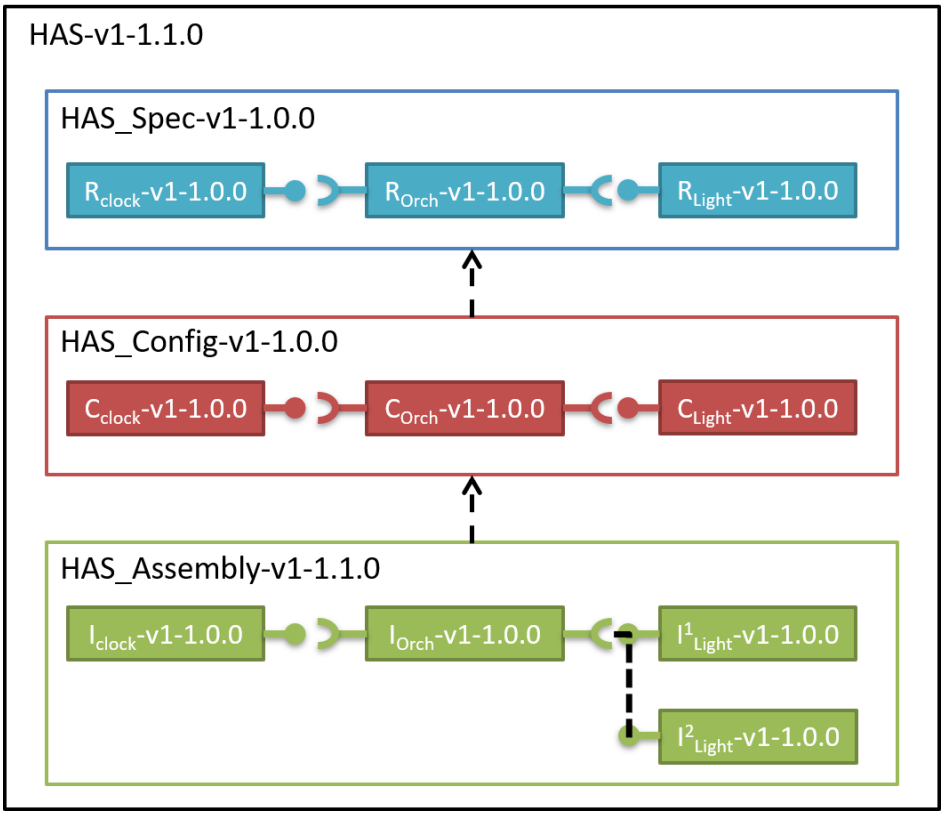


FIGURE 5.15: HAS: component instance addition

$I^2_{Light} - v1 - 1.0.0$ are respectively replaced by $I^1_{Light} - v1 - 1.1.0$ and $I^2_{Light} - v1 - 1.1.0$, and finally the <**major**> version number of the assembly is incremented as is the one of the global architecture.

Figure 5.17 introduces the version graph that is obtained during the HAS architecture evolution. It also shows the version propagation that occurs during the second step of the scenario. Then the HAS architecture has been versioned at three abstraction levels and following three points of view during its whole evolution.

Alternate scenario. Has one can see, at step two of the previously described scenario, if the change was on component role $R_{Light} - v1 - 1.0.0$ (replaced by $R_{Light} - v1 - 1.1.0$) then the change would have been substitutable into the Specification level but would still have implied a version propagation to the Configuration level. Indeed such change would have broken the realization relation with $C_{Light} - v1 - 1.0.0$. However the final architecture version would have been substitutable for $HAS - v1 - 1.1.0$.

5.5 Conclusion

This chapter discusses a generic approach for versioning component-based software architectures and providing semantics to versions. Semantics are derived from strict type-based

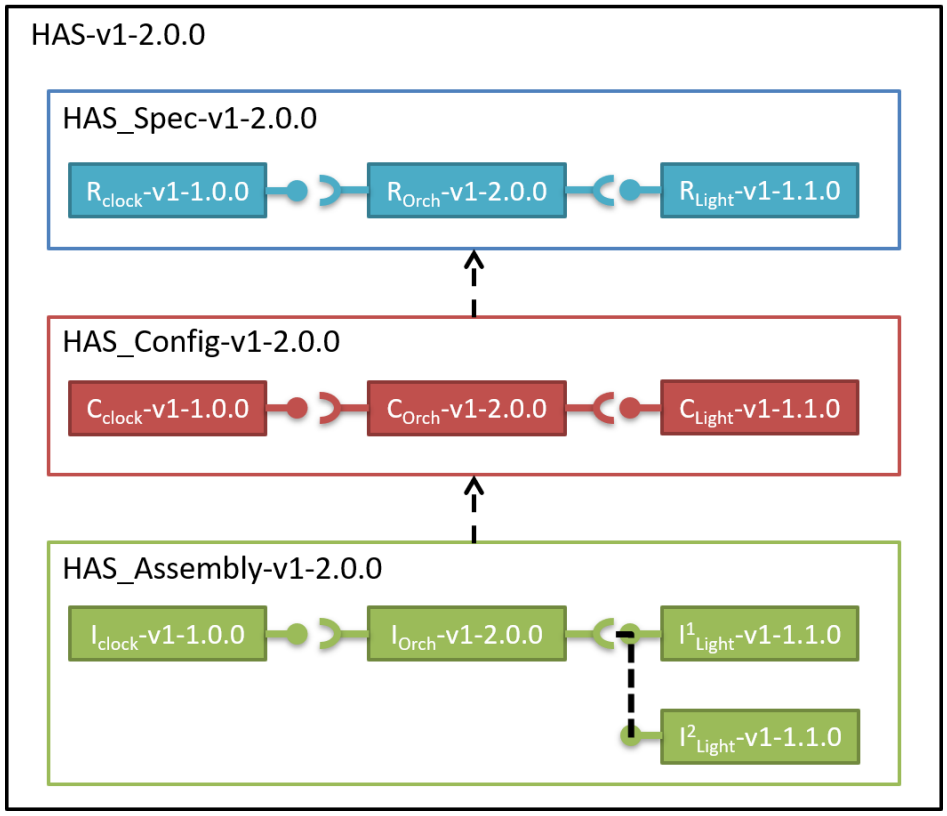


FIGURE 5.16: HAS: component role replacement

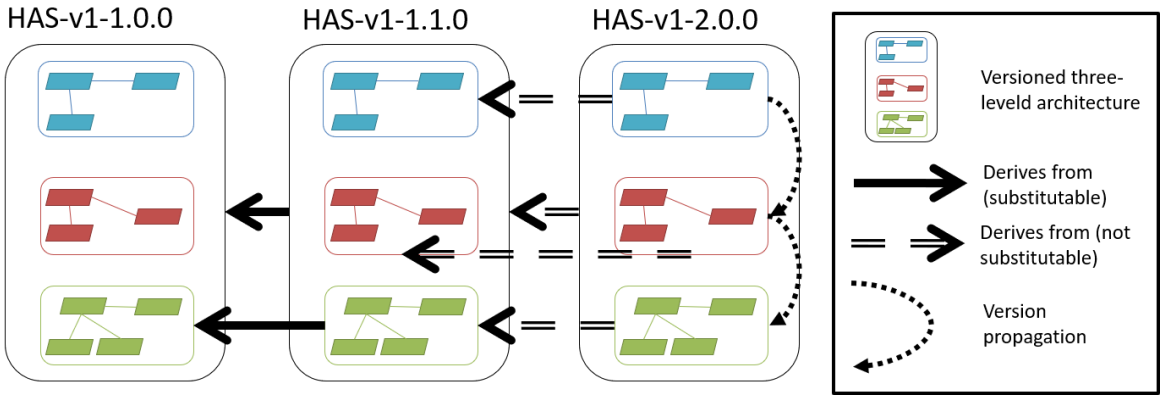


FIGURE 5.17: HAS: version graph

substitutability concepts that make it possible to formalize backward compatibility for component / architecture versions. Proposed semantics enables to classify architectural differences in terms of substitutability. This substitutability-based versioning is then applied to Dedal and used in a change impact analysis to determine change scenarii which imply or not a version propagation within multiple architecture levels. This analysis shows that substitutability is not sufficient for predicting version propagation and thus provides the conditions for such propagation. This chapter also proposes a metamodel for representing version histories considering backward-compatibility of artifacts at the three Dedal architecture levels. However, this metamodel can be adapted to other ADLs through the concept of *AbstractArtefact*. This chapter finally proposes a way for automatically increment version

identifiers considering version semantics. However, this chapter does not addresses comprehensively all the version management issues like for instance re-engineering histories in repositories for easing reuse processes. Such mechanisms will be studied in future work. This chapter nonetheless answers research question **RQ2** about how to introduce semantics in component and architecture versioning.

Next chapter introduces the implementation of our re-documentation and versioning approaches approaches and the experimentation which has been led for validating them.

Chapter 6

Case study and implementation

Contents

Chapters 4 and 5 introduced the foundations of our re-documentation and versioning approach. In our approach, we use the Dedal ADL, which has been defined in Zhang's thesis [Zha10]. This ADL has then been formalized and a CASE (Computer-Aided Software Engineering) tool has been developed in Mokni's thesis In order to validate our approach,

we implemented a re-documentation algorithm in *DedalStudio*. This chapter introduces the implementation which has been realized in order to apply the re-documentation approach on large projects and to re-document their history and thus analyze their evolution. It also describes an evaluation of the approach on a case study.

6.1 Implementation of re-documentation and versioning approaches

In order to take advantage of MDE-oriented tools that have been developed for years, the DedalStudio implementation leverages the Eclipse[1] ecosystem [Mok15]. Indeed, many tools in Eclipse emphasize MDE processes. EMF[2] (Eclipse Modeling Framework) allows the definition of meta-models and the generation of the corresponding Java code structures. Sirius[3] is a tool developped by Obeo. It is based on EMF and GMF[4] (Graphical Modeling Framework) and offers to create a graphical syntax as well as an editor for EMF models. Xtext[5] is a tool that enables to define textual grammars, export them as EMF meta-models and automatically generate a text editor for the specified language. The generated editor embeds a parser which is able to map the artifacts of the textual concrete syntax of the language with instances of the EMF metamodel that define its abstract syntax. QVTo[6] (Operational QVT) is the actual Eclipse-based implementation of the QVT[7] (Query View Transformation) language. QVT is a language that allows model to model transformations through the use of mapping rules. The re-documentation process has been implemented using these technologies provided by the Eclipse ecosystem.

Finally, in order to implement the versioning concepts presented in Chapter 5, we developed a module for finding differences between Dedal architecture versions and characterize them to propose semantic versioning. This module has also been implemented in the Eclipse world by using the EMF Compare API[8].

6.1.1 Overview of DedalStudio

DedalStudio has been first developped in Mokni's thesis [Mok15]. Figure 6.1 introduces the Eclipse product that we derived from Mokni's work. A first release can be downloaded at `https://github.com/DedalArmy/DedalStudio/releases`.

As shown in Figure 6.1 it embeds a textual editor and a graphical editor for Dedal architecture models. Dedal textual syntax was defined using Xtext while the graphical one was defined using Sirius. In addition, both syntaxes are defined to describe the same language

[1] https://www.eclipse.org/ [Last seen 2019-09-05]
[2] https://www.eclipse.org/emf/ [Last seen 2019-09-05]
[3] https://www.eclipse.org/sirius/ [Last seen 2019-09-05]
[4] https://www.eclipse.org/modeling/gmp/ [Last seen 2019-09-05]
[5] https://www.eclipse.org/Xtext/ [Last seen 2019-09-05]
[6] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Last seen 2019-09-05]
[7] https://www.omg.org/spec/QVT/ [Last seen 2019-09-05]
[8] https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html [Last seen 2019-09-05]

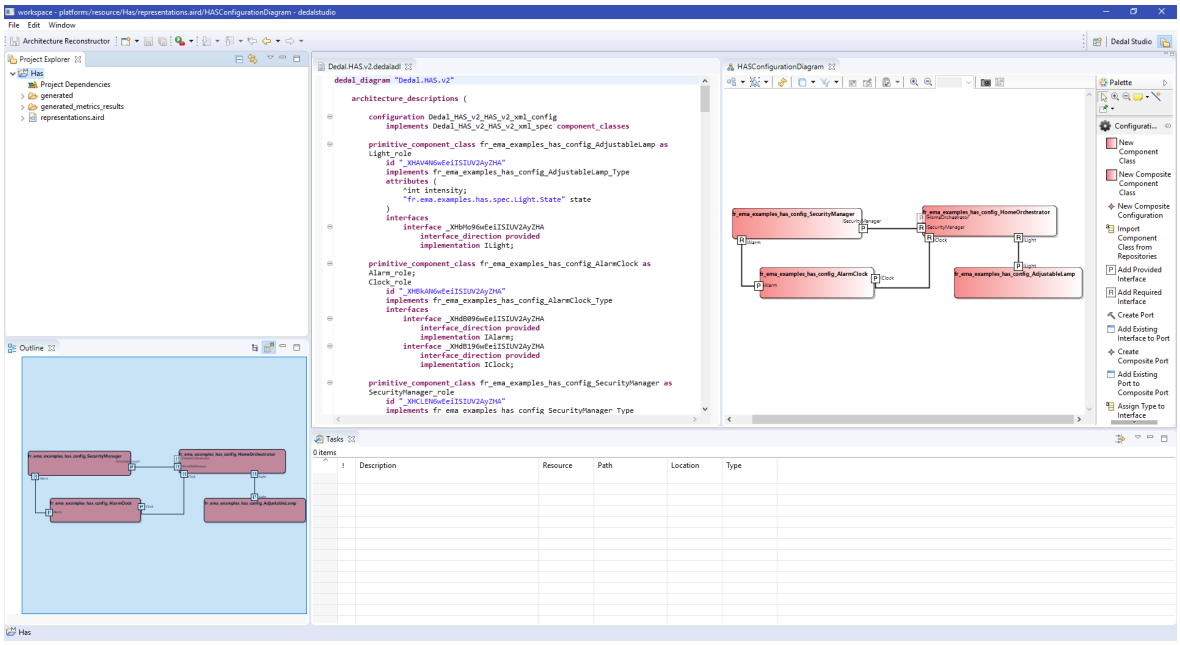


FIGURE 6.1: DedalStudio (and output of the *component-based-hierarchy-builder* module)

(meta-model) which allows to automatically modify the textual description of Dedal models through the modification of the graphical editor and vice-versa.

The first implementation of *DedalStudio* was a proof of concept for automated architecture evolution processes defined in Mokni's thesis [Mok15], thus the plugins that correspond to the Mokni's work have been refactored and are now released as easily installable Eclipse plugins at `http://www.dev.lgi2p.mines-ales.fr/ariane/p2/dedal/2019-06`. The plugins that have been developed in our approach have also been released at `http://www.dev.lgi2p.mines-ales.fr/ariane/p2/springdsl/2019-06` (for SpringDSL) and `http://www.dev.lgi2p.mines-ales.fr/ariane/p2/redoc/2019-06` (for the re-documentation tools). All the modules that have been developed during this work are available as Maven[9] modules at `http://www.dev.lgi2p.mines-ales.fr/ariane/mvn/`. In addition, source code repositories can be found at `https://www.github.com/DedalArmy`.

6.1.2 Implementation of the re-documentation module

Figure 6.2 introduces the structure of the re-documentation module. This module is composed of four sub-modules: *XMLMerge*, *JarLoader*, *HierarchyBuilder*, *SpringToDedal* and *component-based-hierarchy-builder*. *SpringDSL* that is the sixth module shown in Figure 6.2 has not been implemented as part of the re-documentation module but as a standalone plugin project.

Typically, as we plan to re-document large projects the process is separated in two phases.

[9] https://maven.apache.org/ [Last seen 2019-09-05]

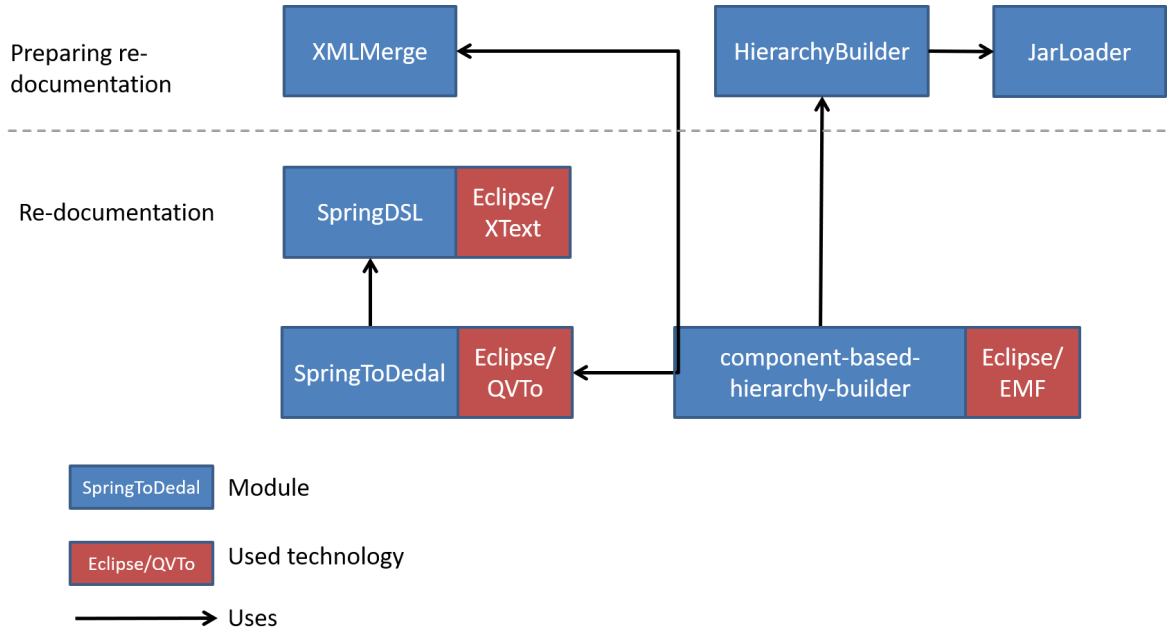


FIGURE 6.2: Re-documentation module structure

The first one is a preparation phase to the re-documentation and the second one is the re-documentation itself. Those phases and the modules that operate then are discussed thereafter.

6.1.2.1 Preparing re-documentation

The first phase before re-documenting software consists in preparing the inputs. Our approach focuses on Java projects that use deployment descriptors. Thus we need to perform some preprocessing operations.

XMLMerge. By nature, Spring projects can declare their deployment by using three architecture definition features:

- **XML descriptors**: In this case, architectures are defined by one or more XML descriptors. Those descriptors are parsed and interpreted by the Spring container at runtime. In this kind of descriptors, beans are defined by $< bean >$ tags and connections between them are defined by explicit dependency injection ($< ref >$ nested tags or ref attributes associated with property tags).
- **Configuration classes**: A configuration class is a specific Java class that is identified by an *@Configuration* annotation. Those classes are automatically processed by the Spring container to build the runtime architecture. This method enables pre- and post-processing of beans initialization. Beans are declared by methods holding *@Bean* annotations and connections between them are defined by passing bean references to other bean constructors or property setters (another kind of explicit dependency injection handled by the container).

- **Self-annotated classes**: A self-annotated class is identified in the code by the *@Component* annotation. Connections are identified by *@Autowired* annotations associated with attributes that define the dependencies to be initialized by the Spring container (IoC). They are automatically supplied by the container according to the available beans.

A difficulty for re-documenting such projects is that Spring configuration styles can be mixed up. Thus a part of the deployment may be described with XML descriptors while another part may be defined by configuration classes and / or self-annotated classes. Compared qualities of different approaches for the definition of architectures is out of this thesis scope and has been initiated by Perez *et al.* [Per+19]. Those styles are in practice equivalent for defining architectures (bean sets related by connections). We chose to consider projects that use only XML-based descriptors as an initial data source for the experimentation. Those descriptors correspond to external descriptions of software deployment and enable explicit and encapsulated architecture definitions. However, a future work perspective is to extend re-documentation for those styles. Then as stated before, XML-based Spring deployment descriptions are often distributed among several XML descriptors. Thus, this module (*XMLMerge*) aims at parsing projects to identify all the Spring XML files and then to merge them so they can be used later for re-documenting software. The idea of such merging is to extract a global XML descriptor of the whole architecture since Spring allows bean referencing from a descriptor to another.

JarLoader. As Spring is a technology that mostly focuses on web services, numerous infrastructure beans such as Java Database Connectivity (JDBC) components are deployed from imported libraries. Thus, we need to take into account the fact that those libraries are compiled and most of the time imported using Maven. Then after they have been imported in a project we need to load them with a custom class loader (*JarLoader*) that recursively parses a project to find all the libraries it contains. The classes that are part of these libraries can be then loaded at runtime on demand when the re-documentation process requires informations from them. This module is also used by the *HierarchyBuilder* module which is discussed next.

HierarchyBuilder. An essential part of the re-documentation process relies on the presence of a type hierarchy to manage a **Specification** re-documentation. Thus, in first intention, we tried to apply our approach to compiled Java projects which made it possible to recover a complete information about Java type hierarchy using the reflection capabilities that Java provides. However, we came to the conclusion that such an approach was difficult to implement since a lot of project compilations failed because of failing dependencies. Errors related to class loading was too much unpredictable depending on the re-documented project.

Thus we chose to build the class hierarchy directly from the project source code. To do so, we parse a project to list all the Java files. Then the hierarchy is built until the first library layer as shown in Figure 6.3 where *Boolean* (which has been added for example purpose) belongs to the *java.lang* library. Figure 6.3 is based on the HAS example. It is the output

of the *HierarchyBuilder* module. The figure exposes the hierarchy of all classes / interfaces (*i.e.*, *HomeOrchestratorImpl* inherits from abstract class *HomeOrchestrator*. However the hierarchy of libraries is not exposed since it can easily be explored through the Java reflect API.

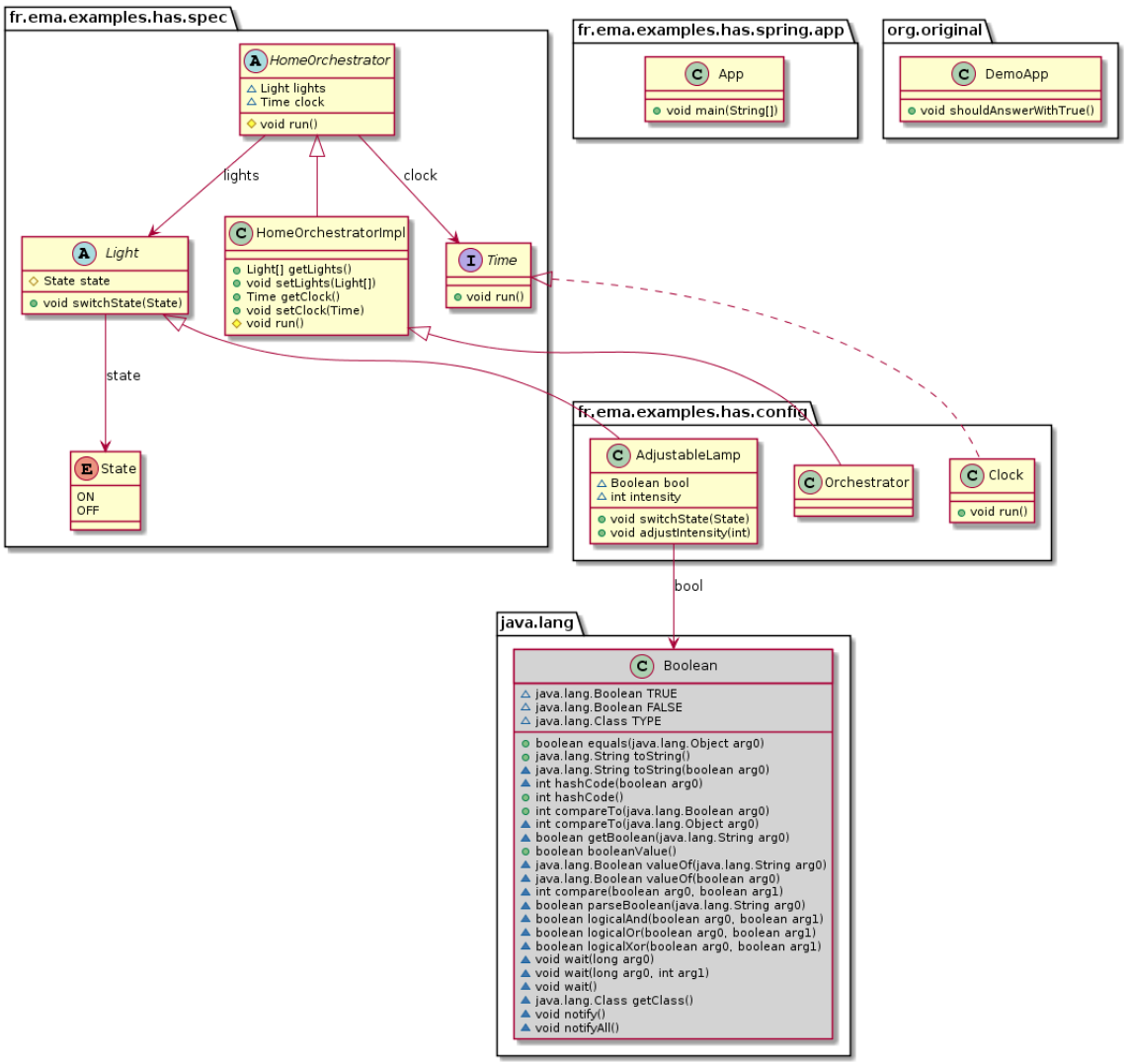


FIGURE 6.3: Example of built hierarchy from Java project (output of *HierarchyBuilder* module)

6.1.2.2 Re-documentation

The second phase consists in re-documenting component-based architectures as discussed in Chapter 4.

SpringDSL. Since this approach is model driven, we searched a way to abstract the Spring text descriptors and transform them into manipulable object models. A first option would have been to use an XML parser to generate DOMs corresponding to Spring deployment descriptors. However this does not allow to identify semantic model artifacts that can be easily transformed to re-document Dedal architectures based on EMF. Thus we implemented the

Spring grammar using Xtext (see Appendix A) in order to automatically generate a L* parser that would parse Spring descriptor and generate a model of the Spring descriptor and not simply a model of the XML document. As shown in Figure 6.4 the XML structure of the descriptor is transformed into an EMF model which eases the transformation to Dedal in next module.

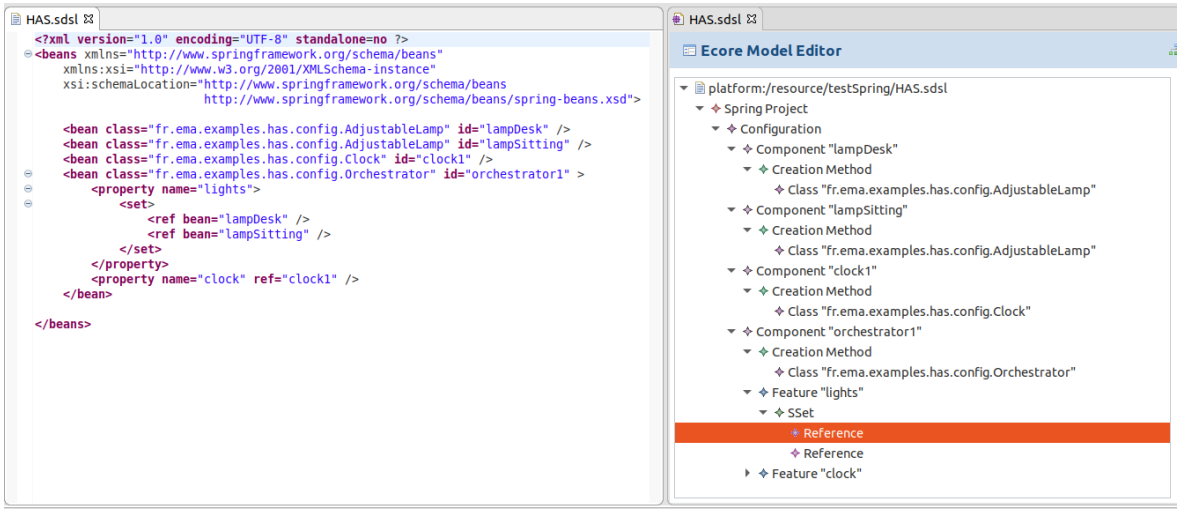


FIGURE 6.4: Example of SpringDSL file

SpringToDedal. In order to take advantage of the SpringDSL model to begin software re-documentation as a Dedal architecture, we need to generate an incomplete Dedal model from the Spring deployment descriptor artifacts. To do so it is only required to perform a basic mapping of SpringDSL artifacts to Dedal artifacts. Thus we used QVTo (see Appendix B) to write the transformation. For now the transformation is not complete since it does not cope with delicate situations that exist in Spring such as non-basic schemes (*i.e.*, $< mvc >$ tags, etc.). However, further development can deal with such difficulties. Additionally those limitations do not question the relevancy and generalization of our approach. Indeed, Spring is a source of projects that we want to re-document and track their evolution. We want to establish the principles of the approach and not release an industrial tool. Thus, the basic elements that are taken into account represent enough data for our experimentation. The extension to whole Spring is not a feasibility question but more a question of time.

component-based-hierarchy-builder. Finally this last module implements the re-documentation algorithm which is discussed in Chapter 4 and developed in Appendix C. This module is an Eclipse plugin and is intended (in further upgrades) to be dynamically used in Java Spring projects to make architects and developers able to keep an eye on the architecture they deploy since it can be fuzzy in numerous cases. Thus, this module uses the *SpringToDedal* transformation to transform the merged Spring descriptor from the *XMLMerge* module and obtain the incomplete Dedal architecture model. Then it uses the *HierarchyBuilder* module as a reflect-like API on Java source code and libraries to re-document software. Figure 6.1 shows *DedalStudio* after we re-documented our running HAS example. As the model

has been automatically generated, the textual and graphical forms of the language are also automatically generated by Eclipse tools (Xtext and Sirius) according to the model.

We used this implementation to re-document "real-life" projects extracted from GitHub. Next section introduces the implementation of the module we developed to analyze differences between architecture model versions.

6.2 Implementation of architecture versioning

Figure 6.5 shows the structure of the model comparator module. It is composed of two sub-modules which are **ProjectComparator** and **DiffAnalyzer**.

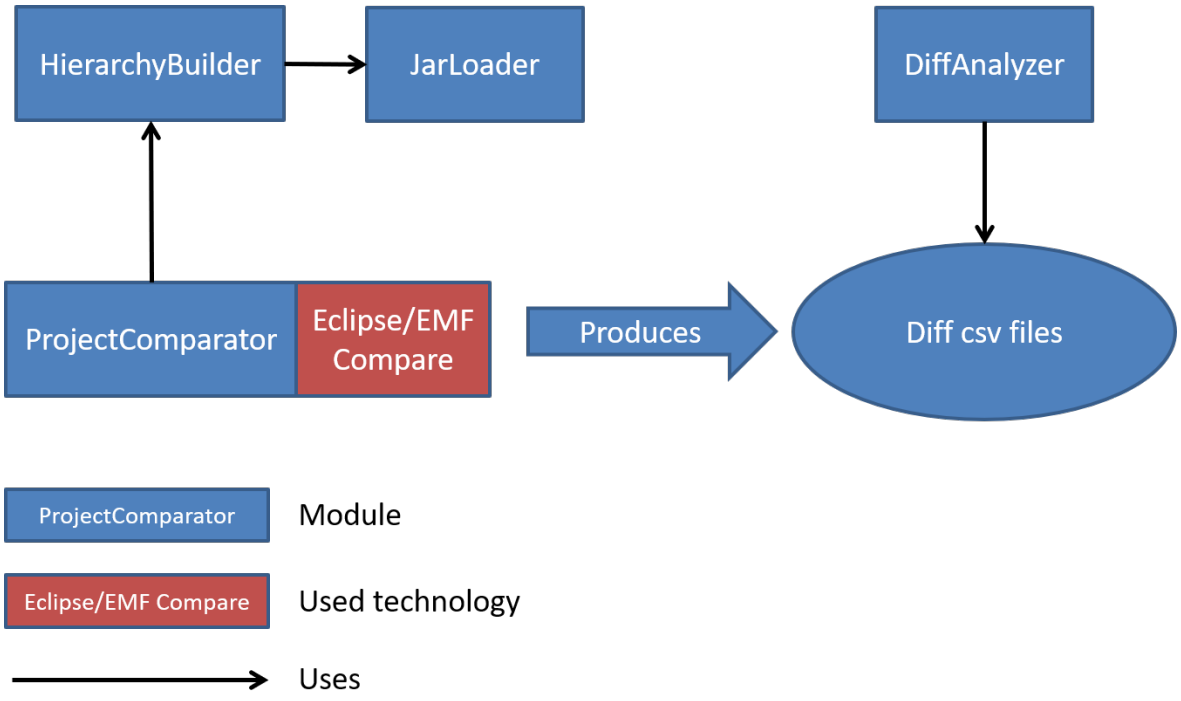


FIGURE 6.5: Dedal model comparison module

ProjectComparator. This sub-module is based on EMF compare. It compares Dedal architecture versions two by two and generates CSV files that describe the differences which are found between architecture versions. The advantage of using CSV files is that they are easily portable and analyzable with Python libraries. One CSV file is generated for every model comparison. Each generated CSV file contains information about all the differences that are observed between two model versions. This information is composed of five fields: the old model version tag, the new version tag, the object that differs, the kind of the difference and finally the backward compatibility of the change. The object field of the CSV file contains, for each difference in the file, the type and the name of the object for further analysis by the **DiffAnalyzer** sub-module. As it is discussed in Chapter 5 a difference can be of three types for a component-based architecture. It can be either an addition, a deletion or a change. Finally, as it is discussed in Chapter 5, substitutability of architecture artifacts is based on the

analysis of types. This is why the **ProjectComparator** sub-module uses **HierarchyBuilder** and **JarLoader** sub-modules from the Re-documentation module. In the resulting CSV file, the substitutability field may have three values :

- **true** means that the considered architecture artifact is formally identified as substitutable by analyzing types which are identified in EMF Compare Diff objects and according to the rules that are summarized in Table 5.1.
- **false** means that the considered architecture artifact is formally identified as not substitutable by analyzing types which are identified in EMF Compare Diff objects and according to the rules that are summarized in Table 5.1.
- **null** means that substitutability of the architecture artifact could not be calculated from the EMF Compare Diff object. It can be the case when the considered artifact is not directly concerned by the actual difference but is induced by other differences. Thus neither the old and new versions of the artifact are reachable which makes the substitutability impossible to calculate.

DiffAnalyzer. This sub-module is written in Python and interprets the differences stored in CSV files that are generated by **ProjectComparator**. Every difference of each CSV file is counted according to the objects that differ, their type, kind and substitutability. The **DiffAnalyzer** module produces a global CSV file where each line represents a difference that has been analyzed. The global CSV file is composed of 84 columns that contain all the information from individual files but also add some information about versions and architecture themselves which are as follows:

- **Intentional versioning accuracy:** The intentional versioning accuracy corresponds to the accuracy of the version tagging (by developers) according to the actual tagging that would be performed by following our approach. This analysis assumes that the considered project applies the semantic versioning 2.0.0[10] approach. The accuracy of the architecture intentional versioning can be evaluated by using some indicators contained in CSV file for answering two questions:
  - **Should the new architecture be substitutable?** This first question is answered by using version tags since they contain the developper analysis about backward compatibility. By comparing the old and the new version tag, it is easy to see if either the $< major >$, the $< minor >$, the $< build >$ number or the version suffix has been changed. Thus, if the $< major >$ version number has been incremented then the new architecture version should not be substitutable, otherwise it should. Moreover, a $< minor >$ increment indicates backward compatible changes. Finally $< build >$ increment and version suffix changes indicate backward compatible changes that should not have impact on the structure.

[10] https://semver.org/ [Last seen - 2019-07-03]

  - **Is the new architecture actually substitutable?** By using information from difference files, it is easy to find out if an architecture is actually substitutable or not for its previous version. Indeed, in case the difference file contains not substitutable changes then the new architecture version is considered as not substitutable.

  This makes it possible to propose a version increment according to the result of the difference analysis. And then to effectively evaluate intentional versioning accuracy.

- **Architecture degeneration:** Moreover, thanks to this information, it is possible to identify indicators that could be the sign of an architecture degeneration. Thus it is possible to partially answer the three following questions by analyzing the intention behind the version tag:
  - **Is the new architecture version subject to erosion?** Typically, it is possible to identify a situation that could indicate software erosion when:
    * The new version is denoted as substitutable by the new version tag.
    * The new version is actually given as not substitutable by difference analysis.
    * The new version derives from at least one artifact deletion which breaks the architecture substitutability.
  - **Is the new architecture subject to drift?** It is possible to identify a situation that could indicate software drift when:
    * The new version only has incremented $< build >$ number (or has changed version tag suffix) but the analysis reveals that a $< minor >$ increment would have been more accurate. It means that developers did not intend to change the structure of the software while they actually changed it in a substitutable way.
    * Otherwise it can also be the case if architecture substitutability is not preserved (wrong version increment and substitutability loss) and if no deletion is observed.
  - **Is the new architecture subject to both erosion and drift?** As this analysis is based on a bottom-up re-documentation, it is not possible to identify such mixed situations. Indeed, the erosion concept is based on deletions that break backward compatibility while the drift concept is based on changes that do not necessarily break backward compatibility. Thus, answering this question would require to know the initial willing of the architect, which is not possible in such approach.

Next section introduces the results of the experimentation we lead.

6.3 Experimentation and evaluation

In order to test our approach, we chose to apply it on "real-life" open source projects.

Data extraction. Data was extracted from GitHub[11] repositories. In order to target significant projects, the extraction was performed following the selection criteria proposed by Jarczyk *et al.* [Jar+14]. Thus we extracted the last commit of Java projects rated over 100 stars and which have been forked at least 10 times [Per+19]. The extracted projects also contained the "Spring" keyword. The last criteria of the extraction was the date of creation that needed to be after 2010-01-01 (after Spring 3 release). The extraction identified 524 projects. Extraction metadata are available online[12].

Project selection. As discussed in Section 6.1.2.1, our implementation only takes Spring XML files into account. Thus we needed to target projects that use Spring only in their XML form. Doing so, the data set was reduced to 63 projects. Then we analyzed projects to find a good candidate for applying our approach. A good candidate should be an industrial project with a version history where versions are identified following Semantic Versioning 2.0.0. We identified *BroadleafCommerce* which has more than 2200 classes in its 6.0.3-GA version (last version at the time of the extraction), 316 released versions and which applies Semantic Versioning 2.0.0 for tagging its versions.

6.3.1 Case study: Broadleaf Commerce

BroadleafCommerce[13] is an enterprise open source e-Commerce framework based on Spring that is available on GitHub[14]. It aims at providing a support for the development of enterprise-class, commerce-driven websites. The project is composed of 184 branches and 323 releases, it has 66 contributors, 1282 stars on GitHub and has been forked 1036 times (at the time of the 2019-09-21). The extraction occurred on the 2019-05-03, at this date we gathered 316 released versions (the amount of versions at the time of the extraction). The experimentation has been lead on the versions from 3.0.0.BETA1 to 6.0.3-GA.

6.3.2 Experimentation

As a first requirement of the experimentation, we had to import as much as possible of the project required libraries for each version. As *BroadleafCommerce* uses Maven, we could easily import dependencies. We thus applied the re-documentation process on the *BroadleafCommerce* history and thus obtained 236 Dedal architecture versions which is the amount of releases between version 3.0.0.BETA1 and version 6.0.3-GA. No architecture has been re-documented before version 3.0.0.BETA1 since, due to unknown factor, re-documentation failed only for those versions. Then we could calculate differences between successive versions and obtained 221 difference files because of some loss due to EMFCompare[15] API fails. The aim of this experimentation is to measure the intentional versioning accuracy and identify situations where drift and / or erosion of the architecture can be observed.

[11] https://github.com [Last seen - 2019-08-29]
[12] https://github.com/DedalArmy/MISORTIMA/tree/data-study-spring-deploy-features
[13] http://www.broadleafcommerce.com [Last seen - 2019-09-21]
[14] https://github.com/BroadleafCommerce/BroadleafCommerce [Last seen - 2019-09-21]
[15] https://www.eclipse.org/emf/compare/ [Last seen 2019-09-05]

6.3.2.1 Re-documenting *BroadleafCommerce* history

Figure 6.6 summarizes the output of the *BroadleafCommerce* history architecture re-documentation. It shows the evolution of the number of declared components in the architecture versions and the evolution of the number of classes that compose the project considering architecture versions which have been numbered from 1 to 236.

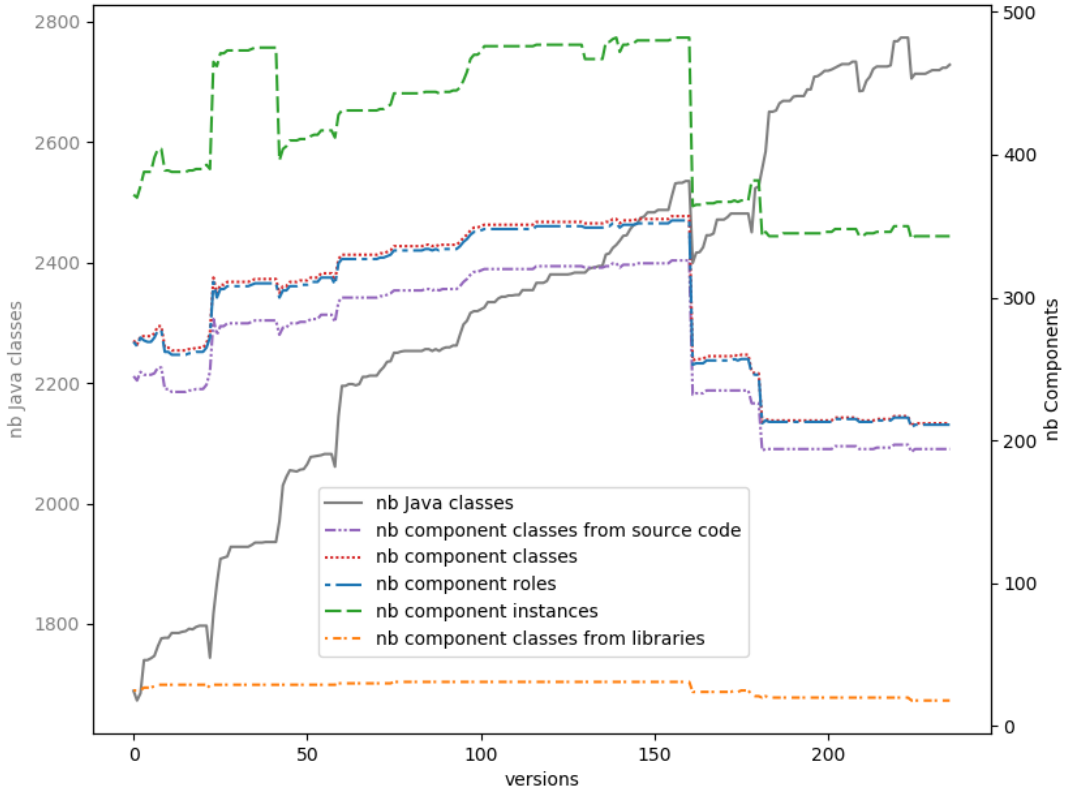


FIGURE 6.6: Re-documented components and Java classes in function of architecture versions

Five curves represent the number of components: one for each re-documented architecture level (*i.e.*, **Assembly**, **Configuration**, **Specification**), the two last ones represent the component classes that are declared in the source code and those which are declared in imported libraries. Observing Figure 6.6 we can identify following characteristics of the *BroadleafCommerce* project:

- The amount of components that are declared from libraries is almost constant and much lower than the amount of components which are declared in the source code of the project. This means that the project relies on a constant component layer whereas the real evolution of its architectures lies in the evolution of the source code. Thus it is a very suitable project for calculating architecture differences and characterizing them.
- The number of component instances is always greater than the number of component classes in architecture versions. This indicates that the implementation and the

deployment of the architectures are decoupled, favoring deployment reconfiguration and component instances parameterizing.

- The number of Java classes has increased all along the project evolution. Thus, it indicates that the complexity of the overall code structure also increases. Moreover, each class number growth or decrease seems to have an impact on the amount of components. Then, although deeper analysis of the project is required, it seems that the architecture versions are based on a policy which prevents them to grow too much and thus preserves more maintainable architectures. The architecture grows by stages until a drop in the amount of classes and components around the 150^{th} version. This drop is probably due to a simplification of the framework that aimed at reducing the size of architectures. Figure 6.7 reinforces this interpretation since it shows that component instances number clearly drops while the amount of XML files where they are declared suddenly grows. Deeper analysis would be needed to confirm this hypothesis.
- The amount of component roles is equal to the amount of component classes. The small gap between both (component classes and component roles) curves can be explained by issues which occur during the re-documentation if some external libraries that are needed in the project are missing, which means that the project may have internal dependencies that could not be resolved. It may also indicate that some component roles are realized by several component classes.

Another outcome of this re-documentation consists in comparing each of the re-documented **Specification** and **Configuration** to find out whether or not **Specification** is composed of more abstract types than the **Configuration**. 100% of the re-documented **Specification**s are more abstract than their respective **Configuration**s. In other words, at least one component role in each re-documented **Specification** is more abstract than the component role that realizes it. It may indicate a good respect of Java development good practices in terms of Java interface declaration and use.

The next step of the experimentation is the calculation of differences between architecture versions and their characterization.

6.3.2.2 Characterizing *BroadleafCommerce* versions

We calculated differences between previously obtained architecture versions to characterize them. Doing so, we obtained a total of 221 difference models that we could analyze. There were a loss during the comparison process because of unknown EMF issues probably due to some EMF format inconsistencies.

As it has been discussed before, we aimed at identifying situations that could induce drift or erosion. Figure 6.8 shows the architecture version increment accuracy measured on the project. Surprisingly only 2.71% architecture versions are rightly incremented (according to the intentional versioning accuracy). Moreover 10.41% of those version increments are correct according to the backward compatibility but are not correctly incremented. This means that in those cases, developers did not manage to identify structural changes while they

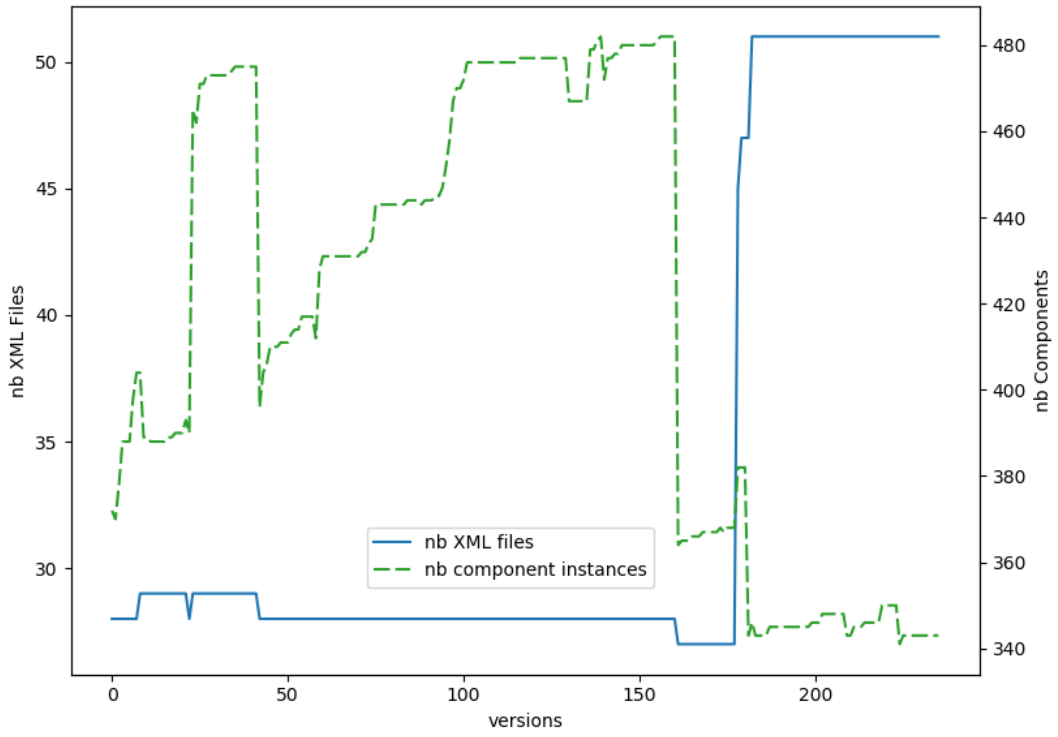


FIGURE 6.7: Component instances and XML Spring files in fonction of architecture versions

figured out the right backward compatibility properties. Finally, a lot of version (86.88%) increments are wrong. These results are completed by Figure 6.9 which shows the proportions of version increment mistakes according to their types. Thus 4.19% of the mistakes concern $< minor >$ increments that should be $< major >$ ones, 10.70% concern $< build >$ increments that should be $< minor >$ and the remainder (85.12%) concerns $< build >$ increments that should be $< major >$. Those results can be interpreted as follows:

- $< build >$ **for** $< minor >$ **increment**: They correspond to the inaccurate version increments of Figure 6.8. They correspond to a good backward compatibility analysis but to a lack of architectural impact perception since developers did not notice architectural change.
- $< build >$ **for** $< major >$ **increment**: They correspond to the majority of wrong version increments of Figure 6.8. However 53% of them correspond to suffix changes which can correspond to a release cycle where developers consider that suffixes should not necessarily be backward compatible since it is not clarified in Semantic Versioning 2.0.0 approach. Although still 86 of the $< build >$ increments are used to perform changes in a non-substitutable way which is a problem considering the loss of version tag accuracy that becomes useless.

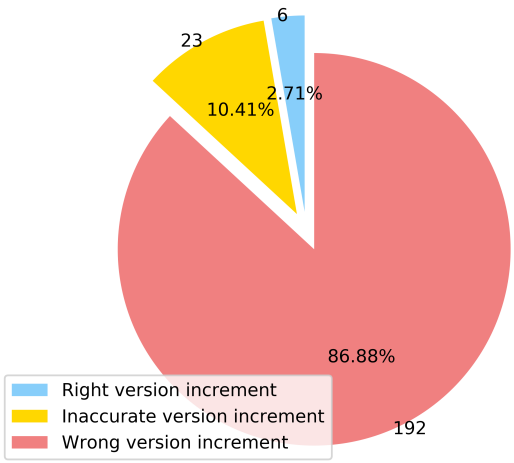


FIGURE 6.8: Version increment accuracy

- $< minor >$ **for** $< major >$ **increment**: As for the previous item, incompatible changes are indicated as compatible which leads to a fast version tags obsolescence.

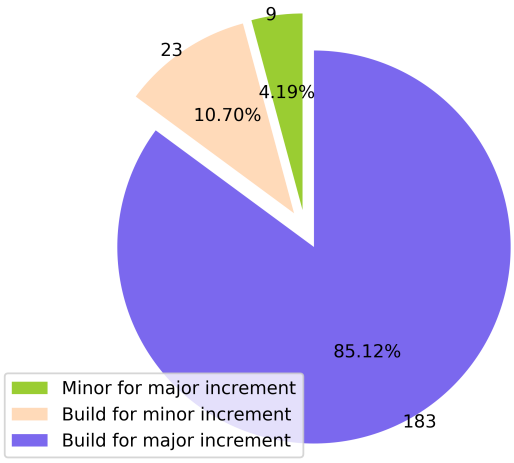


FIGURE 6.9: Version increment mistakes

Such version identification mistakes can progressively lead to a loss of architectural agility (considering component replacement, reconfiguration...) due to unexpected compatibility issues and then can lead to architecture degeneration.

Figure 6.10 introduces the proportion of situations that could lead to architecture degeneration situations that can be identified all along the *BroadleafCommerce* framework evolution.

Thus almost 40% of architecture versions are in situation of potential architecture degeneration:

- **Erosion.** 22.17% of *BroadleafCommerce* architecture versions show indicators of architectural erosion. This is mainly due to a bad version identification as it is discussed before. Then unwanted major changes such as undocumented artifact deletions may lead to software erosion. Then the erosion situations are identified mostly (45 / 49) from $< build >$ increments that should be $< major >$ which means that only few mistakes are made on $< minor >$ version increments. However such mistakes still occur.
- **Drift.** In the same way, drift situation could be observed in 16.74% of architecture versions. 23 of them correspond to $< build >$ increments that should be $< minor >$ and 14 correspond to $< build >$ increments that should be $< major >$. Thus it concerns both substitutable and not substitutable versions. This shows that developers are probably less sensible to drift than to erosion problems. Indeed, if we assume that developers do not consider that version suffixes must be backward compatible, then we identified more potential drift situation than erosion ones.

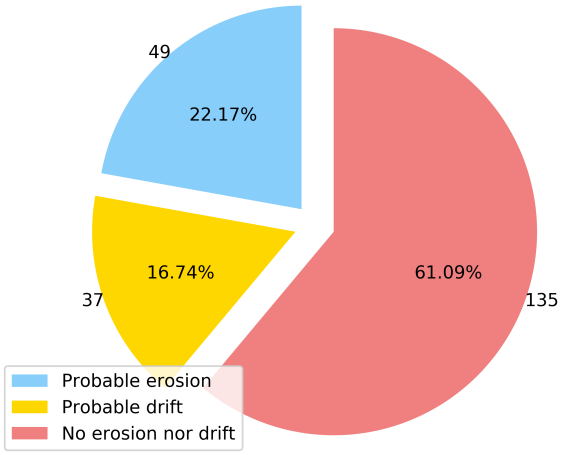


FIGURE 6.10: Architecture degeneration risks

Then this experimentation shows that an automated versioning process based on strict type rules is necessary to ensure a good consistency of version tags.

6.3.2.3 Threats to validity

Some threats to validity can be identified on this experimentation.

Threats to external validity. The first external validity threat relates to the way projects were selected. The selection itself may potentially bias the results. Indeed, we chose to

study a specific kind of projects which is not representative of all the ways Spring is used and thus this may not fully be representative of all the Spring developers habits. A second threat to external validity is the generalization. This threat is directly linked to the previous one. Indeed, more experimentation need to be led on other technologies than Spring to ensure generalization of the approach in practice.

Threats to internal validity. Due to the lack of existing workbench, it is not guaranteed that all the dependencies of the projects are imported and loaded into project directories which can lead to re-documentation imprecision. Re-documented architectures can thus potentially miss some architectural artifacts. The threat that is identified by Kalliamvakou *et al.* [Kal+16] which states that "many active projects do not use GitHub exclusively" can also imply a lack on the presence of all the project dependency.

6.3.2.4 Discussion

By being able to re-document three-leveled architectures and to calculate and characterize differences between their versions, this experimentation shows that such approach could be adapted to real projects. Indeed, the fact that it could be applied on more than 200 versions of an enterprise project is encouraging. Moreover, such approach can be coupled to other empirical studies for analyzing developer habits in term of architecture impact. Additionally, it can be coupled to more precise experimentation that would focus on other characteristics which could explain the evolution of architectures. For instance, analyzing commits could probably help to better identifying and explaining architecture evolution phases such as the drop of components that can be observed in Figure 6.6. Finally the implementation of the approach needs to be completed by further work to take into account more of the Spring framework, and also other technologies such as Enterprise Java Beans (EJBs).

6.4 Conclusion

This chapter presents our implementation and an experimentation of architecture re-documentation and versioning. The applicative contributions that are discussed in this chapter consist of the re-documentation and versioning tools. From the re-documentation perspective, this chapter introduces *SpringDSL* that is our Xtext-based implementation of the Spring XML grammar. *HierarchyBuilder* builds the entire type hierarchy of Java projects and generates UML models. The *component-based-hierarchy-builder* module then re-documents three-leveled Dedal architectures. In future work, improvement should be made to *SpringDSL* which does not take all the language into account and other technologies might be supported. Moreover, the implementation is Java-based, although, the approach should be implemented for other languages. From the versioning point of view, this chapter introduces *ProjectComparator* that compares architecture models and characterize their differences. *DiffAnalyzer* analyzes differences for identifying drift and erosion situations. To improve the version management, it is important to implement single component histories management in future work. Then the management of those histories needs to be integrated into IDEs

for easing reuse. The presented experimentation shows that it is possible to re-document large enterprise software and help to manage better versioning by automating the process. Indeed, this approach proves that it is suitable for re-documenting more than 200 versions of *BroadleafCommerce* framework which is a large enterprise open-source project. It is also suitable to characterize them in terms of substitutability in order to propose better version tags that are more consistent with their actual state. Thus by re-documenting and versioning architectures, it also shows that it is possible to support long time evolution thanks to the three Dedal architecture levels. Moreover, as it is discussed in this chapter, all the re-documented architectures have more abstract **Specification**s which is good for improving the reuse of software components and architectures as well as managing their evolution. This answers research question **RQ3**: such re-documenting and / or versioning approaches are suitable for large software systems. Finally, the fact that drift and erosion situations can be observed from this experimentation directly answers reserach question **RQ4**: it is possible to identify drift and / or erosion situations by re-documenting and analyzing software versions.

Chapter 7

Conclusion and Perspectives

Contents

This thesis presents our approach named ARIANE. This chapter summarizes the contributions of the thesis and discusses their limits, as well as some perspectives.

7.1 Contributions

This thesis contributes to the field of component-based software engineering. It especially addresses the problem of documentation loss during software evolution. It also addresses the problem of version identification in the context of component-based software architectures and more particularly software architectures that are described at multiple abstraction levels. It is organized around two main issues. The first one consists in recovering software documentation by a proposed re-documenting process. Moreover, the recovered documentation must describe the software at each step of its life cycle. Its deployment, implementation and specification must be re-documented in order to refound long-time evolution support. The second one consists in automating versioning of component-based architectures.

Moreover, in order to version each of the successive descriptions of software architectures, it is necessary to version them at multiple architectural levels. Then, they are versioned at component level, at architecture description level (*i.e.*, *Specification*, *Configuration*, and *Assembly*) and finally at a global level, which is composed of the three architecture levels.

7.1.1 Software re-documentation contributions

The first contribution of this thesis concerns software re-documentation. From a conceptual point of view, this thesis proposes an approach to re-document software architectures as they are implemented. In particular, it proposes to re-document them from object-oriented code and deployment descriptors. It is based on the type theory defined by Mokni *et al.* [Mok+16a], which provides a strong theoretical basis for analyzing component type hierarchies. The particularity of this approach is that it actually re-documents the software life-cycle. Re-documented architectures are composed of three description levels, which represent the specification of the software, its implementation, and its deployment. Those architecture levels are described in the Dedal ADL and are respectively the *Specification*, the *Configuration* and the *Assembly*.

The technical corresponding contribution is the re-documentation algorithm. This algorithm maps software concepts into Dedal architectures. The resulting architectures are composed of three abstraction levels, which are consistent to one another. This algorithm is generic and can be applied to any object-oriented and / or deployment descriptor technologies.

Finally, the applicative contribution of software re-documentation consists of the set of tools that have been released for implementing the approach. It has been implemented for re-documenting Java based projects that use the Spring [Joh+04] to describe their deployment. Thus this contribution is composed of the release of several tools as eclipse plugins and / or Maven[1] modules. *SpringDSL* implements the Spring XML grammar in the EMF ecosystem to automatically derive EMF-based Spring models. Derived models are directly usable as they are. In the case of this thesis, we perform a model to model transformation from *SpringDSL* to Dedal. The second released module is the *HierarchyBuilder* module. It builds the entire hierarchy of Java projects and also includes the type hierarchy of its dependencies. The module generates a UML description of the Java project based on the PlantUML[2] language. The *component-based-hierarchy-builder* module is the actual implementation of the re-documentation algorithm. This module generates Dedal architectures from Java source code and Spring XML deployment descriptors.

7.1.2 Software architecture versioning contributions

The second contribution of this thesis concerns the versioning of component-based software architectures. From a conceptual point of view, this thesis proposes an approach to formally analyze backward compatibility of component and architecture versions. In particular, it

[1] https://maven.apache.org/ [Last seen 2019-09-05]
[2] http://plantuml.com/fr/ [Last seen - 2019-09-26]

is based on the type theory defined by Arévalo *et al.* [Aré+07; Aré+09; Abo+19] to analyze substitutability of components and / or architectures after an architectural change occurred. This allows classification of changes in terms of substitutability. It is proposed as a formal change impact analysis based on the type theory developed by Mokni *et al.* [Mok+16a]. This change impact analysis relies on formal rules that make it possible to classify changes in terms of the impact they have on the existing architecture, considering it a three levels of abstraction. A change that is substitutable in its own architecture level may impact other description levels. Version propagation can be predicted from this change impact analysis.

From a technical point of view, this thesis proposes a metamodel for representing version histories considering backward compatibility at three abstraction levels. It also proposes to automatically increment version identifiers based on this analysis.

From an applicative point of view, the contribution consists of the implementation of two modules. The first one is the *ProjectComparator*, which compares architecture models using the EMFCompare[3] tool and characterizes their differences in terms of artifact backward compatibility. Those differences can be analyzed by the *DiffAnalyzer*, which finds out architectural derive situations such as drift and erosion. This module also calculates the best version identifier increment from the substitutability analysis of the new architecture version.

7.2 Limitations and perspectives

This section identifies limitations of this thesis and discusses some perspectives.

7.2.1 Software re-documentation perspectives

The presented approach for re-documenting software at three levels of architecture presents some limitations.

First, from a conceptual point of view, the approach only takes static information into account and then re-documents component-based architectures from deployment descriptors, which are static descriptions. In future work, this static re-documentation must be completed with dynamic re-documentation to consider all architecture aspects.

Second, from a technical point of view, we chose to identify the smallest roles as possible. However, this choice is based on type hierarchy and can be discussed. When several component roles can be identified for a single component class, they are automatically defined as the realized component roles. However, dynamic analysis would help to ensure that they are well identified. In fact, it could be more accurate to keep a more coarse grained component role in some cases. This issue should be addressed in future work. In the same way, component interfaces are filled with public class methods. However, as the visibility of class methods changes according to the point of view (*i.e.*, package, subclass…) future work should address this issue by proposing options to re-document interfaces.

[3] https://www.eclipse.org/emf/compare/ [Last seen 2019-09-05]

Finally, from the applicative point of view, several improvements should be made. First of all, the implementation of *SpringDSL* needs to be completed in order to deal with all the language. Additionally, more deployment descriptor languages should be implemented to broadcast this approach to other languages. Last but not least, the implementation is made for Java projects, but in future work, it needs to be extended to other object-oriented languages. Another improvement that can be done in further work, is to include the architecture re-documenting module directly into Integrated Development Environments (IDEs) to guide the architect in real-time.

7.2.2 Software architecture versioning perspectives

The presented versioning approach also presents some limitations.

From a technical point of view, the approach should include the management of architectural artifact histories into component repositories. This would highly enhance reuse processes. Moreover, as previous work on Dedal discussed automated evolution [Mok+16a], mechanisms for replacing, adding, deleting, and propagating versions would be easily handled.

From an applicative point of view, it is necessary to handle component versioning by not only tracking architecture evolution but also keeping a record of each component version. Then, the three-leveled version history management needs to be implemented from the proposed version metamodel. As previously, this approach gets valuable if it can guide software architects, and it needs to be integrated into IDEs.

7.2.3 Experimental perspectives

This thesis proposes to re-document and analyze versions on more than 200 architecture versions. However, this experimentation is not yet validated by experts. In future work, we plan to involve developers in a feedback process in order to further assess the accuracy and relevance of our approach.

Appendix A

XText-based Spring implementation

```
1  grammar org.xtext.spring.SpringConfigDsl hidden(WS, ML_COMMENT)
2  generate springConfigDsl "http://www.xtext.org/spring/SpringConfigDsl"
3  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4
5
6  SpringProject returns SpringProject:
7      {SpringProject}
8      '<?xml''version''='STRING'encoding''='STRING ('standalone''='('yes'|'
           no'))?'?>'
9       configurations+=Configuration;
10
11
12 AbstractKeyValue:
13     (AbstractArtefact|DataString)
14 ;
15
16 /**Abstract Class of elements present in bean */
17 AbstractArtefact returns AbstractArtefact:
18     Component | AttributeTag |IdRefTag| ReferenceTag |Collection;
19
20 /**Abstract Class of Collection */
21 Collection returns Collection:
22     Array|sList | sSet | Map | Props;
23
24 /**Abstract Class of Collection */
25 /*Util:
26     (UtilConstant|UtilPropertyPath|UtilProperties/* |UtilList|UtilMap|
           UtilSet)
27 ;*/
28
29 /*<beans/> */
30 Configuration returns Configuration:
31     {Configuration}
32     (
33     '<beans'
```

```
34
35      (('defaultautowire='defaultAutowire=AutoWiredType)?
36      &('defaultinitmethod='defaultInitMethod=ValidString)? //Method
37      &('defaultautowirecandidates='defaultAutowireCandidates=ValidString)?
            //REGEX
38      &('defaultdestroymethod='defaultDestroyMethod=ValidString)? //Method
39      &('defaultlazyinit='defaultLazyInit=DefaultableBoolean)?
40      &('defaultmerge='defaultMerge=DefaultableBoolean)? //Default is false
41      &('profile='profile=ValidString)?
42      & (IdDashAndColon'='ValidString)* )
43      '>')
44      description=Description?
45      (components+=Component|alias+=Alias|imports+=Import
46          |contexts+=Context
47          |mvcs += MVC
48          |aspects+=Aspect
49          |utilConstants+=UtilConstant|utilLists+=UtilList|utilMaps+=
                UtilMap
50          |utilProperties+=UtilProperties|utilSets+=UtilSet|
                utilPropertiesPath+=UtilPropertyPath
51          |txAdvices+=TxAdvise|txJtaTransactionManager+=
                TxJtaTransactionManager
52
53      )*
54      (ConfigurationComposite+=Configuration)*
55      ('</beans>');
56
57  MVC:
58      {MVC}
59      ('<mvc' ':' 'annotationdriven' '/>')
60  ;
61
62
63
64  /*<alias/> */
65  Alias returns Alias:
66      {Alias}
67      '<alias' 'name=' origin=[Component|ValidString] 'alias=' alias=
            ValidString ('/>'|('>''</alias>'));
68
69  /*<import/> */
70  Import returns Import:
71      {Import}
72      '<import''resource='resource=ValidString('/>'|('>''</import>'))
73  ;
74
75  Context:
76  '<context:'ContextType
```

```
77 ;
78
79 ContextType returns Context:
80     (AnnotationConfig|ComponentScan|LoadTimeWeaver|MbeanExport|
           MbeanServer|PropertyHolder|SpringConfigured)
81 ;
82
83 /** looks for annotations on beans in the same application context in
       which it is defined */
84 AnnotationConfig:
85     {AnnotationConfig}
86     ('annotationconfig' (('/>')|('>''</context:annotationconfig>')))
87 ;
88
89 /** Spring can automatically detect stereotyped classes and register
       corresponding BeanDefinitions with the ApplicationContext
90  * (implicitly enables the functionality of <context:annotationconfig>)
91  *
92  * basepackage  The comma/semicolon/space/tab/linefeedseparated list of
        packages to scan for annotated components.
93  * annotationconfig  Indicates whether the implicit annotation post-
        processors should be enabled. Default is "true".
94  * namegenerator  The fullyqualified class name of the BeanNameGenerator
        to be used for naming detected components.
95  * resourcepattern  Controls the class files eligible for component
        detection. "** /*.class"
96  * scoperesolver  The fullyqualified class name of the
        ScopeMetadataResolver to
97  * be used for resolving the scope of detected components.
98  * scopedproxy  Indicates whether proxies should be generated for
        detected
99  components
100 * usedefaultfilters  Indicates whether automatic detection of classes
        annotated with @Component, @Repository, @Service, or
101 @Controller should be enabled. Default is "true".
102 */
103 ComponentScan:
104     {ComponentScan}
105     'componentscan'
106     (('basepackage='basePackage=ValidString)
107     &('annotationconfig='annotationConfig=sBoolean)?
108     &('namegenerator='nameGeneratorBean=[Component|ValidString])?
109     &('resourcepattern=' resourcePattern=ValidString)?
110     &('scoperesolver='scopeResolver=[Component|ValidString])?
111     &('scopedproxy='scopedProxy=EnumScopedProxy)?
112     &('usedefaultfilters='useDefaultFilters=sBoolean)?
113     )
114
```

```
115
116     (('/>')|('>'(includeFilters+=IncludeFilter)*(excludeFilters+=
            ExcludeFilter)*'</context:componentscan>'))
117 ;
118
119
120 IncludeFilter:
121     '<context:include filter'
122         (('type='type=EnumTypeFilter)
123         &('expression='expression=ValidString)
124         )
125         (('/>')|('>''</context:include filter>'))
126 ;
127
128 ExcludeFilter:
129     '<context:exclude filter'
130      (('type='type=EnumTypeFilter)
131         &('expression='expression=ValidString)
132         )
133         (('/>')|('>''</context:exclude filter>'))
134 ;
135
136 /** loadtime weaving for Aspect class */
137 LoadTimeWeaver:
138     {LoadTimeWeaver}
139     'loadtimeweaver'  (('aspectjweaving='aspectjWeaving=ValidString)? & (
            'weaverclass='weaverClass=ValidString)?) (('/>')|('>''</context:
            loadtimeweaver>'))
140 ;
141
142
143 MbeanExport:
144     {MbeanExport}
145     'mbeanexport' (('defaultdomain='defaultDomain=ValidString)? & ('
            registration='registration=MbeanRegistrationEnum)? & ('server='
            server=[Component|ValidString])?) (('/>')|('>''</context:mbean-
            export>'))
146 ;
147
148
149
150 MbeanServer:
151     {MbeanServer}
152     'mbeanserver' (('agentid='agentId=ValidString)? & ('id='name=
            ValidString)?) (('/>')|('>''</context:mbeanserver>'))
153
154 ;
155
```

```
156 PropertyHolder :
157 (PropertyPlaceholder|PropertyOverride);
158
159 /** Placeholder for properties files */
160
161
162 PropertyPlaceholder:
163     'propertyplaceholder'
164     (    propertyfile=PropertyFile ?
165         &('ignoreresourcenotfound='ignoreResourceNotFound=sBoolean)? & ('
                ignoreunresolvable='ignoreUnresolvable=sBoolean)? &('local-
                override='localOverride=sBoolean)?
166         & ('order='order=ValidString)? &('propertiesref='propertiesRef=[
                Component|ValidString])?
167         & ('systempropertiesmode='systemPropertiesMode=ValidString )?//
                Depreciated since 3.1
168     )  (('/>')|('>''</context:propertyplaceholder>'))
169 ;
170
171 /** Activates pushing of override values into bean properties */
172 PropertyOverride:
173     'propertyoverride'
174     (propertyfile=PropertyFile
175     &('ignoreresourcenotfound='ignoreResourceNotFound=sBoolean)? & ('
            ignoreunresolvable='ignoreUnresolvable=sBoolean)? &('localoverride
            ='localOverride=sBoolean)?
176         & ('order='order=ValidString)? &('propertiesref='propertiesRef=[
                Component|ValidString])?
177         )  (('/>')|('>''</context:propertyoverride>'))
178 ;
179
180 /*Signals the current application context to apply dependency injection
       to nonmanaged classes that are
181  instantiated outside of the Spring bean factory (typically classes
       annotated with the @Configurable annotation). */
182 SpringConfigured:
183     {SpringConfigured}
184     'springconfigured'  (('/>')|('>''</context:springconfigured>'))
185 ;
186
187 /** Aspect main tags */
188 Aspect returns Aspect:
189 '<aop:'AspectType
190 ;
191
192 AspectType returns Aspect:
193     (AopAspectJAutoproxy|AopConfig)
194 ;
```

```
195
196 /** To enable @AspectJ support with XML based configuration */
197 AopAspectJAutoproxy:
198     {AopAspectJAutoproxy}
199     'aspectjautoproxy' (('exposeproxy='exposeProxy=sBoolean) ? & ('proxy-
        targetclass='proxyTrajetClass=sBoolean)?) (('/>')|('>'(aopincludes
        +=AopInclude)*'</aop:aspectjautoproxy>'))
200 ;
201
202 /** Include @AspectJ aspect  use with Spring AOP*/
203 AopInclude:
204     'include' 'name='aopInclude=[Component|ValidString] (('/>')|('>''</
        aop:include>'))
205 ;
206
207 AopConfig:
208     {AopConfig}(
209     'config' (('exposeproxy='exposeProxy=sBoolean) ? & ('proxytargetclass
        ='proxyTrajetClass=sBoolean)?)
210     (('/>')
211     |('>'((aopPointcuts+=AopPointcut)*(aopAdvisors+=AopAdvisor)*(aspects
        +=AopAspect)*)'</aop:config>')
212     )
213
214 )
215 ;
216
217 AopPointcut:
218     '<aop:pointcut'
219     (('expression='expression=ValidString) &('id='name=ValidString))
220     (('/>')|('>''</aop:pointcut>'))
221
222 ;
223 /*  (pointcut|pointcutref)*/
224 AopAdvisor:
225     '<aop:advisor'
226     (('adviceref='adviceRef=[TxAdvise|ValidString]) & ('id='name=
        ValidString)? & ('order='order=ValidString)? & ((pointcut=
        PointCutExpression)|('pointcutref='pointcutRef=[AopPointcut|
        ValidString])))
227     (('/>')|('>''</aop:advisor>'))
228 ;
229
230 PointCutExpression returns AopPointcut:
231     'pointcut='expression=ValidString
232 ;
233
```

```
234 /**(pointcut | declareparents | before | after | afterreturning | after-
       throwing | around)* */
235 AopAspect:
236     '<aop:aspect' (('id='name=ValidString) & ('order='order=ValidString)
          & ('ref='backingBeanRef=[Component|ValidString])) '>' (
          aopPointcuts+=AopPointcut|declaredParents+=DeclareParents|advises
          +=Advise)* '</aop:aspect>'
237 ;
238
239 /* Introductions
240  * Allows this aspect to introduce additional interfaces that the
241  advised object will transparently implement. */
242 DeclareParents:
243     '<aop:declareparents'
244     (
245           ('typesmatching='typeMatching=ValidString)
246          &(implementInterface=AopImplInterface)
247          &((defaultImplInterface=AopDefaultImplInterface)
248          |(delegateImplRef=AopDelegateImplRef))
249     )
250     (('/>')|('>''</aop:declareparents>'))
251 ;
252
253
254 /**The fully qualified name of the interface that will be introduced. */
255 AopImplInterface returns Interface:
256     'implementinterface='name=ValidString
257 ;
258
259 /*The fully qualified name of the interface that will be introduced. */
260 AopDefaultImplInterface:
261     'defaultimpl='name=ValidString
262 ;
263
264 /*A reference to the bean that will serve as the default implementation
       of the introduced
265  interface. */
266 AopDelegateImplRef:
267     'delegate ref='ref=[Component|ValidString]
268 ;
269
270 Advise:
271     (BeforeAdvise|AfterAdvise|AroundAdvise|AfterReturning|AfterThowing)
272 ;
273
274
275 BeforeAdvise:
```

```
276     '<aop:before' (('pointcutref='pointcutRef=[AopPointcut|ValidString])
            |('pointcut='PointcutExpression=ValidString)) & ('method='method=
            ValidString))
277     (('/>')|('>''</aop:before>'))
278 ;
279
280 AfterAdvise:
281     '<aop:after' (('pointcutref='pointcutRef=[AopPointcut|ValidString])
            |('pointcut='PointcutExpression=ValidString)) & ('method='method=
            ValidString))
282     (('/>')|('>''</aop:after>'))
283 ;
284
285 AroundAdvise:
286     '<aop:around' (('pointcutref='pointcutRef=[AopPointcut|ValidString])
            |('pointcut='PointcutExpression=ValidString)) & ('method='method=
            ValidString))
287     (('/>')|('>''</aop:around>'))
288 ;
289
290 AfterReturning:
291     '<aop:afterreturning' (('pointcutref='pointcutRef=[AopPointcut|
            ValidString])|('pointcut='PointcutExpression=ValidString)) & ('
            method='method=ValidString) & ('returning='returningValue=
            ValidString))
292     (('/>')|('>''</aop:afterreturning>'))
293 ;
294
295 AfterThowing:
296     '<aop:afterthrowing' (('pointcutref='pointcutRef=[AopPointcut|
            ValidString])|('pointcut='PointcutExpression=ValidString)) & ('
            method='method=ValidString) & ('throwing='throwingValue=
            ValidString))
297     (('/>')|('>''</aop:afterthrowing>'))
298 ;
299
300
301 TxAdvise:
302     '<tx:advice' (('id='name=ValidString) &
303     ('transactionmanager='transactionManager=ValidString)? //Default
            value="transactionManager"
304     )
305     '>'
306     (txAttribute=TxAttribute)?
307     '</tx:advice>'
308 ;
309
310 TxAttribute:
```

```
311     {TxAttribute}
312     '<tx:attributes''>'
313     (txMethods+=TxMethod)+
314     '</tx:attributes>'
315
316 ;
317
318 TxMethod:
319     '<tx:method'
320     (('name='name=ValidString)
321     &('isolation='isolation=EnumIsolation)?
322     &('norollbackfor='noRollBackFor=ValidString)?//ref Exeption
323     &('propagation='propagation=EnumIsolation)?
324     &('readonly='readOnly=sBoolean)?
325     &('rollbackfor='rollbackFor=ValidString)? //ref Exeption
326     &('timeout='timeOut=ValidString)? //int default value:1
327 );
328
329 TxJtaTransactionManager:
330     {TxJtaTransactionManager}
331 '<tx:jtatransactionmanager'
332 (('/>')|('>''</tx:jtatransactionmanager>'))
333
334 ;/*<bean/> */
335 Component returns Component:
336     {Component}
337     '<bean'
338     ( ('id='name=ValidString)? & ('name='names+=ValidString)?
339     &   class= CreationMethod
340     &  (features+=PNamespace)*
341     &  (features+= CNamespace)*
342     & ('abstract='abstract=sBoolean)? //bool
343     & ('autowirecandidate='autowireCandidate=DefaultableBoolean)? //enum
344     & ('autowire='autowire=DefaultableBoolean)? //enum
345     & ('dependson='dependsOn=[Component|ValidString])? //ref a bean
346     & ('destroymethod='detroyMethod=ValidString)? //ref a method
347     & ('initmethod='initMethod=ValidString)? //ref a method
348     & ('lazyinit='lazyInit=DefaultableBoolean)? //enum
349     & ('parent='parent=[Component|ValidString])? //ref a bean
350     & ('primary='primary=sBoolean)? //bool
351     & ('scope='scope=ValidString )?
352     )
353     (
354         ('/>')
355         | ('>'
356         description=Description?
357
```

```
358         (features+=Feature|lookupMethods+=LookupMethod|qualifiers+=
                Qualifier|meta+=MetaTag
359         |(aopScopedProxy=AopScopedProxy)|utilPropertiesPath+=
                UtilPropertyPath
360         )*
361         '</bean>')
362     )
363     ;
364
365
366
367
368 CreationMethod:
369  (('factorymethod=' factoryMethod=ValidString)?  & class=( Class|Factory)
        )
370 ;
371
372 /*If the bean is created by a factory */
373 ClassOrFactory:
374 (Class|Factory)
375 ;
376
377 Factory:
378     ('factorybean=' factoryBean=[Component|ValidString]);
379
380 Class returns Class:
381             ('class='/*(path=Path '.' )? classname=ID */  classname=
                ValidString)
382      ;
383
384
385 /*Path of the class */
386 /*Path returns ecore::EString:
387 (ID ('.'ID)*)
388 ;
389 */
390
391 AopScopedProxy:
392     {AopScopedProxy}
393  '<aop:scopedproxy'  ('proxytargetclass='proxyTargetClass=ValidString)?
        (('/>')|('>''</aop:scopedproxy>'))
394 ;
395
396
397 Feature returns Feature:
398     (Property |ConstructorArg)
399 ;
400
```

```
401 /*<property/> */
402 Property returns Feature:
403     ('<property' ((
404         (('name=' name=ValidString) & (artefact=(ReferenceAtt|
                AttributeAtt)) ) ('/>'|'>'(description=Description)?'</
                property>'))
405         | (('name=' name=ValidString)'>' (description=Description)? (
                artefact=AbstractArtefact|NullTag) '</property>')
406         )
407     )
408 ;
409
410 /** If Attribute is a attribute of <property/>/<Constructorarg/> */
411 AttributeAtt returns Attribute:
412     {Attribute}
413     ( ('value='value=ValidString))
414 ;
415
416 /**Attribute create by a tag */
417 AttributeTag returns Attribute:
418     ({Attribute}
419     '<value'('type='type=ValidString)? (('>'value=QSTRING '</value>')|('
            />')))
420     |(UtilConstant)
421 ;
422
423 AttributSimple returns Attribute:
424     value=ValidString
425 ;
426 AttributSimpleValue returns Attribute:
427     'value='value=ValidString
428 ;
429 /** <Idref/> */
430 IdRefTag returns Attribute:
431     {Attribute}
432     '<idref''bean='value=ValidString(('/>')|('>''</idref>'))
433 ;
434 /** If Reference is a attribute of <property/>/<Constructorarg/> */
435 ReferenceAtt returns Reference:
436     {Reference}
437     'ref='ref=[Component|ValidString]
438 ;
439
440 /**Reference create by a tag */
441 ReferenceTag returns Reference:
442  {Reference}
443  '<ref''bean='ref=([AbstractArtefact|ValidString]) ('/>'|'>''</ref>')
444 ;
```

```
445
446
447 ReferenceComponent returns Reference:
448     ref=[Component|ValidString]
449 ;
450
451 /** <constructorarg/>*/
452 ConstructorArg returns Feature:
453      '<constructorarg'
454      ( ( (ConstructorArgAtt)
455      (('>'(description=Description)?'</constructorarg>')|('/>'))
456      )
457      |
458      ({Feature}
459          ( ('index='index=ValidString)?&("name="name=ValidString)?&("type
                ="type=ValidString)?)'>'
460             (description=Description)?
461             (artefact=AbstractArtefact|NullTag)
462         '</constructorarg>'
463         )
464      )
465
466 ;
467
468 /*If the parameter is a attribute */
469 ConstructorArgAtt returns Feature:
470      (
471         ('index='index=ValidString)?
472         & ("name="name=ValidString)?
473          &  (artefact=AttributesCons)
474     )
475     ;
476
477 AttributesCons returns AbstractArtefact:
478         (
479          ({Attribute}('value='value=ValidString & ('type='type=
                ValidString)?))
480         |({Reference}('ref='ref=[Component|ValidString] /* &('type='
               STRING)? */))
481         )
482
483
484 ;
485
486 /*Attribute created in <constructorarg/> */
487 /*AttributeAttCons returns Attribute:
488     {Attribute}
489     ( ( ('value='STRING)& ('type='type=STRING)?))
```

```
490 ;*/
491
492
493 /*Attribute created in <constructorarg/> */
494 /*ReferenceAttCons returns Reference:
495     {Reference}
496     ( ( ('ref='ref=[Component|STRING]) & ('type='STRING)?))
497 ;*/
498
499
500 PNamespace returns Feature:
501 'p:'name=ID (('ref''='artefact=ReferenceComponent)|'='artefact=
        AttributSimple)
502 ;
503
504 CNamespace returns Feature:
505 'c:'name=ID (('ref''='artefact=ReferenceComponent)|'='artefact=
        AttributSimple)
506
507 ;
508
509
510 LookupMethod:
511 '<lookupmethod' (('name='name=ValidString) & ('bean='ref=[Component|
        ValidString]))(('/>')|('>''</lookupmethod>'))
512 ;
513
514 Qualifier:
515     '<qualifier'
516      (('type='type=ValidString)&('value='value=ValidString)?)
517     (('/>')|('>'(qualifierAttributes+=QualifierAttribute)*'</qualifier>')
          )
518
519 ;
520 MetaTag returns Meta:
521     '<meta'
522     (('key='key=ValidString)&('value='value=ValidString))
523     (('/>')|('>''</meta>'))
524
525
526 ;
527 QualifierAttribute:
528     '<attribute'
529     (('key='key=ValidString)&('value='value=ValidString))
530     (('/>')|('>''</attribute>'))
531
532 ;
533
```

```
534 /* <array> */
535 Array returns Array:
536     ({Array})
537     ('<array' (('valuetype='valueType=ValidString)? &('merge='merge=
            DefaultableBoolean)?) '>'
538         (description=Description)?
539         (artefacts+=AbstractArtefact|NullTag)*
540     '</array>')
541 ;
542
543 /*<list/> */
544 sList returns sList:
545     ({sList}
546     ('<list' (('valuetype='valueType=ValidString)? &('merge='merge=
            DefaultableBoolean)?) '>'
547         (description=Description)?
548         (artefacts+=AbstractArtefact|NullTag)*
549     '</list>'))|(UtilList)
550 ;
551
552 /*<set/> */
553 sSet returns sSet:
554 {sSet}
555     (('<set' (('valuetype='valueType=ValidString)? &('merge='merge=
            DefaultableBoolean)?) '>'
556         (description=Description)?
557         (artefacts+=AbstractArtefact|NullTag)*
558     '</set>'))|UtilSet
559 ;
560
561 /*<props/> */
562 Props returns Props:
563     ({Props}
564     ('<props' (('valuetype='valueType=ValidString)? &('merge='merge=
            DefaultableBoolean)?) '>'
565         (description=Description)?
566         (props+=Prop)*
567     '</props>'))|UtilProperties
568 ;
569
570 /*<prop/> */
571 Prop:
572     '<prop' 'key='key=ValidString'>'value=QSTRING'</prop>'
573 ;
574
575 /*<map/> */
576 Map:
577     {Map}
```

```
578     ('<map' (('keytype='keyType=ValidString)?& ('merge='merge=
            DefaultableBoolean)? & ('valuetype='valueType=ValidString)?) '>'
579          (description=Description)?
580     (entries+=MapEntry)*
581     '</map>')|UtilMap
582
583 ;
584
585 /*Entries for map  */
586 MapEntry returns MapEntry:
587 '<entry'
588     ( (MapEntryKey| MapEntryValue|MapEntryAtt)
589       | (
590      /*('valuetype='valueType=ValidString)?*/ '>'
591     key=Key (value=AbstractArtefact|NullTag)
592     (description=Description)?
593     ('</entry>')))
594
595
596 ;
597
598 /*If the key is a attribute */
599 MapEntryKey returns MapEntry:
600  (('valuetype='valueType=ValidString)? &
601  (key=MapEntryKeyAtt))'>'
602  (description=Description)?
603  value=AbstractArtefact
604 ('</entry>')
605 ;
606
607 /*If the value is a attribute */
608 MapEntryValue returns MapEntry:
609 (('valuetype='valueType=ValidString)?    &
610  (value=AttributSimpleValue|value=MapEntryValRef))'>'
611  key=Key
612  (description=Description)?
613  ('</entry>')
614 ;
615
616 /*If the key and the value are attributes */
617 MapEntryAtt returns MapEntry:
618  (('valuetype='valueType=ValidString)? key=MapEntryKeyAtt & (value=
        AttributSimpleValue|(value=MapEntryValRef)))
619  (('/>')|('>'(description=Description)?'</entry>'))
620 ;
621
622 /*<key/> */
623 Key:
```

```
624     {Key}
625     '<key''>'(description=Description)?(key=AbstractArtefact|NullTag) '</
            key>'
626 ;
627
628 /*if key is a attribute */
629 MapEntryKeyAtt returns Key:
630     {Key} /*('key='key1=ValidString|keyref=MapEntrykeyRef )*/
631     ('key='key=DataString|key=MapEntrykeyRef )
632 ;
633
634
635 MapEntrykeyRef returns Reference:
636     ('keyref='ref=[Component|ValidString])
637 ;
638 MapEntryValRef returns Reference:
639     ('valueref='ref=[Component|ValidString])
640 ;
641
642
643 UtilConstant:
644 {UtilConstant}
645 '<util:constant'
646 ( ('staticfield='StaticField=STRING)
647   &('id='name=ValidString)?
648 )
649 (('/>')|('>''</util:constant>'))
650 ;
651
652 UtilPropertyPath:
653 '<util:propertypath' (('id='name=ValidString)& ('path='path=ValidString)
        )
654 (('/>')|('>''</util:constant>'))
655 ;
656
657 UtilProperties:
658     {UtilProperties}
659 '<util:properties'
660 (
661      (propertyfile=PropertyFileSimple)?
662     &('id='name=ValidString)?
663     &('ignoreresourcenotfound='ignoreResourceNotFound=sBoolean)?
664     &('localoverride='localOverride=sBoolean)?
665     &('scope='scope=ValidString)?
666     &('valuetype='valueType=ValidString)?
667     (('/>')|('>'(props+=Prop)*'</util:properties>'))
668 )
669 ;
```

```
670
671 UtilList:
672     {UtilList}
673     '<util:list'
674     (('id='name=ValidString)?
675     &('listclass='listClass=ValidString)?
676     &('scope='scope=ValidString)?
677     &('valuetype='valueType=ValidString)?
678     )
679     (('/>')|('>'(artefacts+=AbstractArtefact)*'</util:properties>'))
680 ;
681
682 UtilMap:
683
684     {UtilMap}
685     '<util:map'
686 (   ('id='name=ValidString)?
687     &('keytype='keyType=ValidString)?
688     &('mapclass='mapClass=ValidString)?
689     &('scope='scope=ValidString)?
690     &('valuetype='valueType=ValidString)?
691
692 )
693 (('/>')|('>'(entries+=MapEntry)*'</util:map>'))
694 ;
695
696 UtilSet:
697     {UtilSet}
698     '<util:set'
699 (   ('id='name=ValidString)?
700     &('setclass='setClass=ValidString)?
701     &('scope='scope=ValidString)?
702     &('valuetype='valueType=ValidString)?
703
704 )
705 (('/>')|('>'(artefacts+=AbstractArtefact)*'</util:set>'))
706 ;
707 PropertyFileSimple returns PropertyFile:
708     ('location='location=ValidString)
709 ;
710 PropertyFile:
711     (('location='location=ValidString) & ('fileencoding='fileEncoding=
        ValidString)?)
712     ;
713
714
715 Description:
716     '<description>'QSTRING'</description>'
```

```
717 ;
718
719
720 DataString :
721     s=ValidString
722 ;
723
724 IdDashAndColon:
725     ID ( ('' ID)|(':' ID)*)
726 ;
727
728 IdWithDash:
729     ID ('' ID)?;
730
731
732 QSTRING hidden(ML_COMMENT):
733       (ID|INT|WS| ValidString
734           |','|'.'|';'|':'|''|'?'|'!'
735           |'+'|'*'|'='|'°' |'>'
736           |'/'|'|'|'\\'
737           |'('|')'|'{'|'}'|'"'|"'"|'['|']'
738           |'@'|'&'|'#'
739       )*;
740
741 NullTag:
742     '<null'('/>'|'>''</null>')
743 ;
744 ValidString:
745     (STRING|'"true"'|'"false"'|'"default"'|'"no"'|'"byName"'|'"byType"'|'
          "constructor"'|'"interfaces"'|'"targetClass"')
746 ;
747
748 /**********ENUM */
749 enum sBoolean:
750     FALSE='"false"'      |FALSE="'false'"
751     |TRUE='"true"'       |TRUE= "'true'"
752 ;
753 enum DefaultableBoolean:
754     DEFAULT='"default"' |DEFAULT="'default'"
755     |FALSE='"false"'     |FALSE="'false'"
756     |TRUE='"true"'       |TRUE= "'true'"
757 ;
758 enum AutoWiredType :
759     DEFAULT='"default"'          |DEFAULT="'default'"
760     |NO='"no"'                   |NO="'no'"
761     |BYNAME='"byName"'           |BYNAME= "'byName'"
762     |BYTYPE= '"byType"'          |BYTYPE= "'byType'"
763     |CONSTRUCTOR='"constructor"'|CONSTRUCTOR="'constructor'"
```

```
764 ;
765 enum EnumScopedProxy:
766     NO='"no"'                    |NO="'no'"
767     |INTERFACES='"interfaces"'   |INTERFACES="'interfaces'"
768     |TARGETCLASS='"targetClass"'|TARGETCLASS="'targetClass'"
769 ;
770 enum EnumTypeFilter:
771     ANNOTATION= '"annotation"'   |ANNOTATION="'annotation'"
772     |ASSIGNABLE='"assignable"'   |ASSIGNABLE="'assignable'"
773     |ASPECTJ= '"aspectj"'        |ASPECTJ= "'aspectj'"
774     |REGEX= '"regex"'            |REGEX= "'regex'"
775     |CUSTOM ='"custom"'          |CUSTOM ="'custom'"
776 ;
777
778 enum MbeanRegistrationEnum:
779     FAILONEXISTING='"failOnExisting"'    |FAILONEXISTING="'failOnExisting'
        "
780     |IGNOREEXISTING='"ignoreExisting"'   |IGNOREEXISTING="'ignoreExisting'
        "
781     |REPLACEEXISTING='"replaceExisting"'|REPLACEEXISTING="'
        replaceExisting'"
782     ;
783 enum EnumIsolation:
784      DEFAULT='"DEFAULT"'                          |DEFAULT="'DEFAULT'"
785      |READ_UNCOMMITTED='"READ_UNCOMMITTED"'       |READ_UNCOMMITTED="'
         READ_UNCOMMITTED'"
786      |READ_COMMITTED ='"READ_COMMITTED"'          |READ_COMMITTED ="'
         READ_COMMITTED'"
787      |REPEATABLE_READ ='"REPEATABLE_READ"'        |REPEATABLE_READ ="'
         REPEATABLE_READ'"
788      |SERIALIZABLE='"SERIALIZABLE"'               |SERIALIZABLE="'SERIALIZABLE'
         "
789 ;
790
791 enum PropagationEnum:
792      REQUIRED='"REQUIRED"'                       |REQUIRED="'REQUIRED'"
793      |SUPPORTS='"SUPPORTS"'                      |SUPPORTS="'SUPPORTS'"
794      |MANDATORY='"MANDATORY"'                    |MANDATORY="'MANDATORY'"
795      |REQUIRES_NEW='"REQUIRES_NEW"'              |REQUIRES_NEW="'REQUIRES_NEW'"
796      |NOT_SUPPORTED='"NOT_SUPPORTED"'            |NOT_SUPPORTED="'NOT_SUPPORTED'"
797      |NEVER='"NEVER"'                            |NEVER="'NEVER'"
798      |NESTED='"NESTED"'                          |NESTED="'NESTED'"
799 ;
800 /***TERMINAL RULES */
801 terminal ID          : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|
        '0'..'9')*;
802 terminal INT returns ecore::EInt: ('0'..'9')+;
803
```

```
804 terminal STRING :
805             '"' ( '\\' . | !('\\'|'"') )* '"' |
806             "'" ( '\\' . | !('\\'|"'") )* "'"
807         ;
808
809 terminal WS         : (' '|'\t'|'\r'|'\n')+;
810
811 terminal ANY_OTHER: .;
812
813 terminal ML_COMMENT:
814     '<!'>'>';
```

Appendix B

SpringToDedal QVTo transformation

```
1  transformation springToDedal(in spring : SpringModel, out dedal :
       DedalModel);
2
3
4  modeltype SpringModel "strict" uses springConfigDsl('http://www.xtext.org
       /spring/SpringConfigDsl');
5  modeltype DedalModel "strict" uses dedal('http://www.dedal.fr/metamodel')
       ;
6
7
8
9
10
11 //
       ////////////////////////////////////////////////////////////////////////
12 //
       ////////////////////////////////////////////////////////////////////////
13 //
       //
14 //                                MAIN
       //
15 //
       //
16 //
       ////////////////////////////////////////////////////////////////////////
17 //
       ////////////////////////////////////////////////////////////////////////
18
19
20 main() {
21      spring.rootObjects()[SpringProject]>map toDedalDiagram();
```

```
22 }
23
24
25
26
27
28 //
       ////////////////////////////////////////////////////////////////////////
29 //
       ////////////////////////////////////////////////////////////////////////
30 //
       //
31 //                                MAPPINGS
       //
32 //
       //
33 //
       ////////////////////////////////////////////////////////////////////////
34 //
       ////////////////////////////////////////////////////////////////////////
35
36
37 ////////////////////////////////////////////////////////////////
38 //                        DedalDiagram                        //
39 ////////////////////////////////////////////////////////////////
40
41 mapping SpringModel::SpringProject::toDedalDiagram() : dedal::
       DedalDiagram {
42     name := "Generated Diagram";
43     architectureDescriptions := self.configurations.
           getArchitectureDescriptions();
44 }
45
46
47 ////////////////////////////////////////////////////////////////
48 //                        DedalConfiguration                  //
49 ////////////////////////////////////////////////////////////////
50
51 mapping SpringModel::Configuration::toDedalConfiguration() : DedalModel::
       Configuration {
52     if(self.alias>notEmpty())
53     {
54         result.name := self.alias>first().alias + "Configuration";
55     }
```

```
56      else
57      {
58          result.name := spring.toString() + "Configuration";
59      };
60      configComponents := self.getConfigComponents();
61      configConnections := self.getConfigConnections();
62      //TODO
63  }
64
65
66  ////////////////////////////////////////////////////////////////
67  //                        DedalAssembly                       //
68  ////////////////////////////////////////////////////////////////
69
70  mapping SpringModel::Configuration::toDedalAssembly() : DedalModel::
        Assembly {
71      name:="defaultName";
72      if(self.alias>notEmpty())
73      {
74          result.name := self.alias>first().alias + "Assembly";
75      }
76      else
77      {
78          result.name := spring.toString() + "Assembly";
79      };
80      assmComponents := self.getAssmComponents();
81      assemblyConnections := self.getAssemblyConnections();
82      //TODO
83  }
84
85
86  ////////////////////////////////////////////////////////////////
87  //                        CompInstance                        //
88  ////////////////////////////////////////////////////////////////
89
90  mapping SpringModel::Component::toCompInstance() : DedalModel::
        CompInstance {
91      var instanciatedCompClass := resolveone(compClass:dedal::CompClass |
            compClass.name.=(self._class.getComponentClassName()));
92      instantiates := instanciatedCompClass;
93
94
95      var numbId : Integer;
96      numbId := 0;
97      (DedalModel::CompInstance).allInstances()>forEach(ci)
98      {
99          if(ci.instantiates = instantiates)
100         {
```

```
101                 numbId:=numbId+1;
102             }
103         };
104
105         if(self.name.<>(null))
106         {
107             name := self.name
108         }
109         else if (self.names.length().<>(0))
110         {
111             name := self.names>at(0)
112         }
113         else
114         {
115             if(self._class._class.oclIsTypeOf(SpringModel::Class))
116             {
117                 name := self._class._class.oclAsType(SpringModel::Class).
                        classname + numbId.toString();
118             }
119             else if(self._class._class.oclIsTypeOf(SpringModel::Factory))
120             {
121                 name := self._class._class.oclAsType(SpringModel::Factory).
                        factoryBean.name;
122             }
123             else
124             {
125                 name := "default";
126             }
127         };
128 }
129
130
131 ////////////////////////////////////////////////////////////////////
132 //                        CompClass                               //
133 ////////////////////////////////////////////////////////////////////
134
135 mapping SpringModel::Component::toCompClass() : DedalModel::CompClass {
136     var tempname : oclstdlib::String;
137     if(self._class._class.oclIsTypeOf(SpringModel::Class))
138     {
139         tempname := self._class._class.oclAsType(SpringModel::Class).
            classname;
140     }
141     else
142     {
143         tempname := self._class._class.oclAsType(SpringModel::Factory).
            factoryBean.name;
144     };
```

```
145     name := tempname;
146 }
147
148
149 ////////////////////////////////////////////////////////////////
150 //                        ComponentClass                      //
151 ////////////////////////////////////////////////////////////////
152
153 mapping SpringModel::CreationMethod::toComponentClass() : DedalModel::
        CompClass {
154     var compClass : DedalModel::CompClass;
155     var tempname : String;
156     if(self._class.oclIsTypeOf(SpringModel::Class))
157     {
158         tempname := self._class.oclAsType(SpringModel::Class).classname;
159         compClass := self.resolveone (compC:DedalModel::CompClass | compC
                .name = tempname);
160         if(compClass = null)
161             name := tempname;
162     }
163     else
164     {
165         name := self._class.oclAsType(SpringModel::Factory).factoryBean.
                name;
166     }
167 }
168
169
170 ////////////////////////////////////////////////////////////////
171 //                        AssemblyConnection                  //
172 ////////////////////////////////////////////////////////////////
173
174 mapping SpringModel::Component::toAssemblyConnection(name:String, r:
        SpringModel::Reference) : DedalModel::InstConnection {
175
176     var source := self.name;
177     var sourceRef : CompInstance;
178     _property := source.replaceAll("\"", "")+"."+name.replaceAll("\"", ""
            );
179     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
180     var target := r.ref.oclAsType(SpringModel::Component).name;
181     var targetRef : CompInstance;
182     targetRef := r.ref.oclAsType(SpringModel::Component).map
            toCompInstance();
183     clientInstElem := sourceRef;
184     serverInstElem := targetRef;
185 }
```

```
186
187 mapping SpringModel::Component::toAssemblyConnection(name:String, c:
        SpringModel::Component) : DedalModel::InstConnection {
188
189     var source := self.name;
190     var sourceRef : CompInstance;
191     _property := source.replaceAll("\"", "")+"."+name.replaceAll("\"", ""
            );
192     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
193     var targetRef := c.map toCompInstance();
194     clientInstElem := sourceRef;
195     serverInstElem := targetRef;
196 }
197
198 mapping SpringModel::Component::toAssemblyConnection_set(name:String, r:
        SpringModel::Reference) : DedalModel::InstConnection {
199
200     var source := self.name;
201     var sourceRef : CompInstance;
202     _property := "set:"+source.replaceAll("\"", "")+"."+name.replaceAll("
            \"", "");
203     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
204     var target := r.ref.oclAsType(SpringModel::Component).name;
205     var targetRef : CompInstance;
206     targetRef := r.ref.oclAsType(SpringModel::Component).map
            toCompInstance();
207     clientInstElem := sourceRef;
208     serverInstElem := targetRef;
209 }
210
211 mapping SpringModel::Component::toAssemblyConnection_set(name:String, c:
        SpringModel::Component) : DedalModel::InstConnection {
212
213     var source := self.name;
214     var sourceRef : CompInstance;
215     _property := "set:"+source.replaceAll("\"", "")+"."+name.replaceAll("
            \"", "");
216     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
217     var targetRef := c.map toCompInstance();
218     clientInstElem := sourceRef;
219     serverInstElem := targetRef;
220 }
221
222 mapping SpringModel::Component::toAssemblyConnection_list(name:String, r:
        SpringModel::Reference) : DedalModel::InstConnection {
```

```
223
224     var source := self.name;
225     var sourceRef : CompInstance;
226     _property := "list:"+source.replaceAll("\"", "")+"."+name.replaceAll(
            "\"", "");
227     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
228     var target := r.ref.oclAsType(SpringModel::Component).name;
229     var targetRef : CompInstance;
230     targetRef := r.ref.oclAsType(SpringModel::Component).map
            toCompInstance();
231     clientInstElem := sourceRef;
232     serverInstElem := targetRef;
233 }
234
235 mapping SpringModel::Component::toAssemblyConnection_list(name:String, c:
        SpringModel::Component) : DedalModel::InstConnection {
236
237     var source := self.name;
238     var sourceRef : CompInstance;
239     _property := "set:"+source.replaceAll("\"", "")+"."+name.replaceAll("
            \"", "");
240     sourceRef := resolveone(compInst:DedalModel::CompInstance | compInst.
            name=source);
241     var targetRef := c.map toCompInstance();
242     clientInstElem := sourceRef;
243     serverInstElem := targetRef;
244 }
245
246 ////////////////////////////////////////////////////////////////
247 //                      ConfigConnection                      //
248 ////////////////////////////////////////////////////////////////
249
250 mapping SpringModel::Component::toConfigConnection(name:String, r:
        SpringModel::Reference) : DedalModel::ClassConnection {
251
252     var source := self._class.getComponentClassName();
253     var sourceRef : CompClass;
254     _property := name;
255     sourceRef := resolveone(compClass:DedalModel::CompClass | compClass.
            name=source);
256     var target := r.ref.oclAsType(SpringModel::Component).name;
257     var targetRef : CompClass;
258     targetRef := r.ref.oclAsType(SpringModel::Component).map toCompClass
            ();
259     clientClassElem := sourceRef;
260     serverClassElem := targetRef;
261
```

```
262 }
263
264 mapping SpringModel::Component::toConfigConnection(name:String, c:
        SpringModel::Component) : DedalModel::ClassConnection {
265
266     var source := self._class.getComponentClassName();
267     var sourceRef : CompClass;
268     _property := name;
269     sourceRef := resolveone(compClass:DedalModel::CompClass | compClass.
            name=source);
270     var targetRef := c.map toCompClass();
271     clientClassElem := sourceRef;
272     serverClassElem := targetRef;
273
274 }
275
276
277
278
279
280 //
        ////////////////////////////////////////////////////////////////////////
281 //
        ////////////////////////////////////////////////////////////////////////
282 //
        //
283 //                              QUERIES
        //
284 //
        //
285 //
        ////////////////////////////////////////////////////////////////////////
286 //
        ////////////////////////////////////////////////////////////////////////
287
288
289
290
291
292 ////////////////////////////////////////////////////////////////
293 //                      getDiagramName                        //
294 ////////////////////////////////////////////////////////////////
295
```

```
296 query getDiagramName(configs : Set(SpringModel::Configuration)) : String
    {
297     var name := "DefaultName";
298     if(configs = null)
299     {
300         name := "null";
301     }
302     else
303     {
304         name := configs->size().toString() + "_in_it";
305     };
306     configs -> forEach(c)
307     {
308         name := "GeneratedDiagram";
309     };
310
311     return name;
312 }
313
314
315 ////////////////////////////////////////////////////////////////
316 //                      getArchitectureDescriptions           //
317 ////////////////////////////////////////////////////////////////
318
319 query SpringModel::Configuration::getArchitectureDescriptions() :
    OrderedSet(DedalModel::ArchitectureDescription) {
320     var architectureDescriptions : OrderedSet(DedalModel::
        ArchitectureDescription);
321     architectureDescriptions += self.map toDedalConfiguration();
322     architectureDescriptions += self.map toDedalAssembly();
323     return architectureDescriptions;
324 }
325
326
327 ////////////////////////////////////////////////////////////////
328 //                      getAssmComponents                     //
329 ////////////////////////////////////////////////////////////////
330
331 query SpringModel::Configuration::getAssmComponents() : Sequence(
    DedalModel::CompInstance) {
332     var assmComponents : Sequence(DedalModel::CompInstance);
333     assmComponents := self.components -> map toCompInstance();
334     return assmComponents;
335 }
336
337
338 ////////////////////////////////////////////////////////////////
339 //                      getConfigComponents                   //
```

```
340 ////////////////////////////////////////////////////////////////////
341
342 query SpringModel::Configuration::getConfigComponents() : Sequence(
        DedalModel::CompClass) {
343     var configComponents : Sequence(DedalModel::CompClass);
344     self.components>forEach(c)
345     {
346         var s := c._class.getComponentClassName();
347         var cc := resolveone(compClass:DedalModel::CompClass | compClass.
              name = c._class.getComponentClassName());
348         if(cc = null)
349             configComponents += c.map toCompClass();
350     };
351     return configComponents;
352 }
353
354
355 ////////////////////////////////////////////////////////////////////
356 //                        getComponentClass                       //
357 ////////////////////////////////////////////////////////////////////
358
359 query CreationMethod::getComponentClassName() : String {
360     var name : String;
361     if(self._class.oclIsTypeOf(SpringModel::Class))
362     {
363         name := self._class.oclAsType(SpringModel::Class).classname;
364     }
365     else
366     {
367         name := self._class.oclAsType(SpringModel::Factory).factoryBean.
              name.toString();
368     };
369     return name;
370 }
371
372
373 ////////////////////////////////////////////////////////////////////
374 //                        getAssemblyConnections                  //
375 ////////////////////////////////////////////////////////////////////
376
377 query SpringModel::Configuration::getAssemblyConnections() : Sequence(
        DedalModel::InstConnection) {
378     var assemblyConnections : Sequence(DedalModel::InstConnection);
379     self.components>forEach(c)
380     {
381         assemblyConnections += c.getInstConnections()
382     };
383     return assemblyConnections;
```

```
384 }
385
386
387 ////////////////////////////////////////////////////////////////
388 //                        getConfigConnections                //
389 ////////////////////////////////////////////////////////////////
390
391 query SpringModel::Configuration::getConfigConnections() : Sequence(
        DedalModel::ClassConnection) {
392     var configConnections : Sequence(DedalModel::ClassConnection);
393     self.components>forEach(c)
394     {
395         configConnections += c.getClassConnections()
396     };
397     return configConnections;
398 }
399
400
401 ////////////////////////////////////////////////////////////////
402 //                        getInstConnections                  //
403 ////////////////////////////////////////////////////////////////
404
405 query SpringModel::Component::getInstConnections() : Sequence(DedalModel
        ::InstConnection) {
406     var connections : Sequence(DedalModel::InstConnection);
407     self.features>forEach(f)
408     {
409         if(f.artefact.oclIsTypeOf(SpringModel::Reference))
410         {
411             if(f.artefact.oclAsType(SpringModel::Reference).ref.
                    oclIsKindOf(SpringModel::Component))
412                 connections += self.map toAssemblyConnection(f.name, f.
                        artefact.oclAsType(SpringModel::Reference));
413         }
414         else if(f.artefact.oclIsTypeOf(SpringModel::Component))
415         {
416             connections += self.map toAssemblyConnection(f.name, f.
                    artefact.oclAsType(SpringModel::Component));
417         }
418         else
419         {
420             if(f.artefact.oclIsKindOf(SpringModel::Collection))
421             {
422                 connections += f.artefact.oclAsType(SpringModel::
                        Collection).getInstConnections(self, f.name);
423             }
424         }
425     };
```

```
426     return connections;
427 }
428
429 query SpringModel::_Collection::getInstConnections(c:SpringModel::
    Component, name:String) : Sequence(DedalModel::InstConnection) {
430     var connections : Sequence(DedalModel::InstConnection);
431     if(self.oclIsTypeOf(SpringModel::sSet))
432     {
433         (self.oclAsType(SpringModel::sSet)).artefacts>forEach(f2)
434         {
435             if(f2.oclIsTypeOf(SpringModel::Reference))
436             {
437                 if(f2.oclAsType(SpringModel::Reference).ref.oclIsKindOf(
                        SpringModel::Component))
438                     connections += c.map toAssemblyConnection_set(name,f2
                            .oclAsType(SpringModel::Reference));
439             }
440             else if(f2.oclIsTypeOf(SpringModel::Component))
441             {
442                 var exists : Boolean;
443                 exists := false;
444                 dedal.objectsOfType(DedalModel::Assembly)>forEach(asm)
445                 {
446                     asm.assmComponents>forEach(comp)
447                     {
448                         if(comp.name = f2.oclAsType(SpringModel::
                                Component)._class.getComponentClassName())
449                         {
450                             exists := true
451                         }
452                     };
453                     if(not exists)
454                     {
455                         asm.assmComponents += f2.oclAsType(SpringModel::
                                Component).map toCompInstance();
456                     };
457                 };
458                 connections += c.map toAssemblyConnection_set(name, f2.
                        oclAsType(SpringModel::Component));
459             }
460         }
461     }
462     else if(self.oclIsTypeOf(SpringModel::sList))
463     {
464         (self.oclAsType(SpringModel::sList)).artefacts>forEach(f2)
465         {
466             if(f2.oclIsTypeOf(SpringModel::Reference))
467             {
```

```
468                 if(f2.oclAsType(SpringModel::Reference).ref.oclIsKindOf(
                        SpringModel::Component))
469                     connections += c.map toAssemblyConnection_list(name,
                            f2.oclAsType(SpringModel::Reference));
470             }
471             else if(f2.oclIsTypeOf(SpringModel::Component))
472             {
473                 var exists : Boolean;
474                 exists := false;
475                 dedal.objectsOfType(DedalModel::Assembly)>forEach(asm)
476                 {
477                     asm.assmComponents>forEach(comp)
478                     {
479                         if(comp.name = f2.oclAsType(SpringModel::
                                Component)._class.getComponentClassName())
480                         {
481                             exists := true
482                         }
483                     };
484                     if(not exists)
485                     {
486                         asm.assmComponents += f2.oclAsType(SpringModel::
                                Component).map toCompInstance();
487                     };
488                 };
489                 connections += c.map toAssemblyConnection_list(name, f2.
                        oclAsType(SpringModel::Component));
490             }
491         }
492     }
493     else if(self.oclIsTypeOf(SpringModel::Map))
494     {
495         (self.oclAsType(SpringModel::Map)).entries>forEach(e)
496         {
497             if(e.key.key.oclIsTypeOf(SpringModel::Reference))
498             {
499                 if(e.key.key.oclAsType(SpringModel::Reference).ref.
                        oclIsKindOf(SpringModel::Component))
500                     connections += c.map toAssemblyConnection(name, e.key
                            .key.oclAsType(SpringModel::Reference));
501             }
502             else if(e.key.key.oclIsTypeOf(SpringModel::Component))
503             {
504                 connections += c.map toAssemblyConnection(name, e.key.key
                        .oclAsType(SpringModel::Component));
505             }
506             else if(e.key.key.oclIsKindOf(SpringModel::Collection))
507             {
```

```
508                 connections += e.key.key.oclAsType(SpringModel::
                        Collection).getInstConnections(c,name);
509             };
510             if(e.value.oclIsTypeOf(SpringModel::Reference))
511             {
512                 if(e.value.oclAsType(SpringModel::Reference).ref.
                        oclIsKindOf(SpringModel::Component))
513                     connections += c.map toAssemblyConnection(name, e.
                            value.oclAsType(SpringModel::Reference));
514             }
515             else if(e.value.oclIsTypeOf(SpringModel::Component))
516             {
517                 connections += c.map toAssemblyConnection(name, e.value.
                        oclAsType(SpringModel::Component));
518             }
519             else if(e.value.oclIsKindOf(SpringModel::Collection))
520             {
521                 connections += e.value.oclAsType(SpringModel::Collection)
                        .getInstConnections(c,name);
522             };
523         }
524     };
525     return connections;
526 }
527
528
529 ////////////////////////////////////////////////////////////////
530 //                      getClassConnections                   //
531 ////////////////////////////////////////////////////////////////
532
533 query SpringModel::Component::getClassConnections() : Sequence(DedalModel
        ::ClassConnection) {
534     var connections : Sequence(DedalModel::ClassConnection);
535     self.features>forEach(f)
536     {
537         if(f.artefact.oclIsTypeOf(SpringModel::Reference))
538         {
539             if(f.artefact.oclAsType(SpringModel::Reference).ref.
                    oclIsKindOf(SpringModel::Component))
540             {
541                 var exists : Boolean;
542                 exists := false;
543                 connections>forEach(c)
544                 {
545                     if(c.clientClassElem.name.=(self._class.
                            getComponentClassName())
546                         and c.serverClassElem.name.=(f.artefact.oclAsType
                                (SpringModel::Reference).getName()))
```

```
547                         {
548                             exists := true;
549                         };
550                     };
551                     if(not exists)
552                     {
553                         connections += self.map toConfigConnection(f.name, f.
                                artefact.oclAsType(SpringModel::Reference));
554                     }
555                 }
556             }
557             else if(f.artefact.oclIsTypeOf(SpringModel::Component))
558             {
559                 var exists : Boolean;
560                 exists := false;
561                 connections>forEach(c)
562                 {
563                     if(c.clientClassElem.name.=(self._class.
                            getComponentClassName())
564                         and c.serverClassElem.name.=(f.artefact.oclAsType(
                                SpringModel::Reference).getName()))
565                     {
566                         exists := true;
567                     };
568                 };
569                 if(not exists)
570                 {
571                     connections += self.map toConfigConnection(f.name, f.
                            artefact.oclAsType(SpringModel::Component));
572                 }
573             }
574             else if(f.artefact.oclIsKindOf(SpringModel::Collection))
575             {
576                     connections += f.artefact.oclAsType(SpringModel::
                            Collection).getClassConnections(self, f.name);
577             }
578         };
579         return connections;
580 }
581
582 query SpringModel::_Collection::getClassConnections(c:SpringModel::
        Component, name:String) : Sequence(DedalModel::ClassConnection) {
583         var connections : Sequence(DedalModel::ClassConnection);
584         if(self.oclIsTypeOf(SpringModel::sSet))
585         {
586             (self.oclAsType(SpringModel::sSet)).artefacts>forEach(f2)
587             {
588                 if(f2.oclIsTypeOf(SpringModel::Reference))
```

```
589             {
590                 if(f2.oclAsType(SpringModel::Reference).ref.oclIsKindOf(
                        SpringModel::Component))
591                 {
592                     var exists : Boolean;
593                     exists := false;
594                     connections>forEach(con)
595                     {
596                         if(con.clientClassElem.name.=(c._class.
                                getComponentClassName())
597                          and con.serverClassElem.name.=(f2.oclAsType(
                                 SpringModel::Reference).getName()))
598                         {
599                             exists := true;
600                         };
601                     };
602                     if(not exists)
603                     {
604                         connections += c.map toConfigConnection(name,f2.
                                oclAsType(SpringModel::Reference));
605                     }
606                 }
607             }
608             else if(f2.oclIsTypeOf(SpringModel::Component))
609             {
610                 var exists : Boolean;
611                 exists := false;
612                 dedal.objectsOfType(DedalModel::Configuration)>forEach(
                        config)
613                 {
614                     config.configComponents>forEach(comp)
615                     {
616                         if(comp.name = f2.oclAsType(SpringModel::
                                Component)._class.getComponentClassName())
617                         {
618                             exists := true
619                         }
620                     };
621                     if(not exists)
622                     {
623                         config.configComponents += f2.oclAsType(
                                SpringModel::Component).map toCompClass();
624                     };
625                     exists := false;
626                 };
627                 connections>forEach(con)
628                 {
```

```
629                     if(con.clientClassElem.name.=(c._class.
                            getComponentClassName())
630                         and con.serverClassElem.name.=(f2.oclAsType(
                                SpringModel::Component)._class.
                                getComponentClassName()))
631                     {
632                         exists := true;
633                     };
634                 };
635                 if(not exists)
636                 {
637                     connections += c.map toConfigConnection(name,f2.
                            oclAsType(SpringModel::Component));
638                 }
639             }
640         }
641     }
642     else if(self.oclIsTypeOf(SpringModel::sList))
643     {
644         (self.oclAsType(SpringModel::sList)).artefacts>forEach(f2)
645         {
646             if(f2.oclIsTypeOf(SpringModel::Reference))
647             {
648                 if(f2.oclAsType(SpringModel::Reference).ref.oclIsKindOf(
                        SpringModel::Component))
649                 {
650                     var exists : Boolean;
651                     exists := false;
652                     connections>forEach(con)
653                     {
654                         if(con.clientClassElem.name.=(c._class.
                                getComponentClassName())
655                             and con.serverClassElem.name.=(f2.oclAsType(
                                    SpringModel::Reference).getName()))
656                         {
657                             exists := true;
658                         };
659                     };
660                     if(not exists)
661                     {
662                         connections += c.map toConfigConnection(name,f2.
                                oclAsType(SpringModel::Reference));
663                     }
664                 }
665             }
666             else if(f2.oclIsTypeOf(SpringModel::Component))
667             {
668                 var exists : Boolean;
```

```
669                 exists := false;
670                 dedal.objectsOfType(DedalModel::Configuration)>forEach(
                        config)
671                 {
672                     config.configComponents>forEach(comp)
673                     {
674                         if(comp.name = f2.oclAsType(SpringModel::
                            Component)._class.getComponentClassName())
675                         {
676                             exists := true
677                         }
678                     };
679                     if(not exists)
680                     {
681                         config.configComponents += f2.oclAsType(
                            SpringModel::Component).map toCompClass();
682                     };
683                     exists := false;
684                 };
685                 connections>forEach(con)
686                 {
687                     if(con.clientClassElem.name.=(c._class.
                        getComponentClassName())
688                      and con.serverClassElem.name.=(f2.oclAsType(
                            SpringModel::Component)._class.
                            getComponentClassName()))
689                     {
690                         exists := true;
691                     };
692                 };
693                 if(not exists)
694                 {
695                     connections += c.map toConfigConnection(name,f2.
                        oclAsType(SpringModel::Component));
696                 }
697             }
698         }
699     }
700     else if(self.oclIsTypeOf(SpringModel::Map))
701     {
702         (self.oclAsType(SpringModel::Map)).entries>forEach(e)
703         {
704             if(e.key.key.oclIsTypeOf(SpringModel::Reference))
705             {
706                 if(e.key.key.oclAsType(SpringModel::Reference).ref.
                    oclIsKindOf(SpringModel::Component))
707                 {
708                     var exists : Boolean;
```

```
709                     exists := false;
710                     connections>forEach(con)
711                     {
712                         if(con.clientClassElem.name.=(c._class.
                                getComponentClassName())
713                             and con.serverClassElem.name.=(e.key.key.
                                    oclAsType(SpringModel::Reference).getName
                                    ()))
714                         {
715                             exists := true;
716                         };
717                     };
718                     if(not exists)
719                     {
720                         connections += c.map toConfigConnection(name, e.
                            key.key.oclAsType(SpringModel::Reference));
721                     }
722                 }
723             }
724             else if(e.key.key.oclIsTypeOf(SpringModel::Component))
725             {
726                 var exists : Boolean;
727                 exists := false;
728                 dedal.objectsOfType(DedalModel::Configuration)>forEach(
                        config)
729                 {
730                     config.configComponents>forEach(comp)
731                     {
732                         if(comp.name = e.key.key.oclAsType(SpringModel::
                                Component)._class.getComponentClassName())
733                         {
734                             exists := true
735                         }
736                     };
737                     if(not exists)
738                     {
739                         config.configComponents += e.key.key.oclAsType(
                                SpringModel::Component).map toCompClass();
740                     };
741                     exists := false;
742                 };
743                 connections>forEach(con)
744                 {
745                     if(con.clientClassElem.name.=(c._class.
                            getComponentClassName())
746                         and con.serverClassElem.name.=(e.key.key.
                                oclAsType(SpringModel::Component)._class.
                                getComponentClassName()))
```

```
747                     {
748                         exists := true;
749                     };
750                 };
751                 if(not exists)
752                 {
753                     connections += c.map toConfigConnection(name, e.key.
                        key.oclAsType(SpringModel::Component));
754                 }
755             }
756             else if(e.key.key.oclIsKindOf(SpringModel::Collection))
757             {
758                 connections += e.key.key.oclAsType(SpringModel::
                    Collection).getClassConnections(c,name);
759             };
760             if(e.value.oclIsTypeOf(SpringModel::Reference))
761             {
762                 if(e.value.oclAsType(SpringModel::Reference).ref.
                    oclIsKindOf(SpringModel::Component))
763                 {
764                     var exists : Boolean;
765                     exists := false;
766                     connections>forEach(con)
767                     {
768                         if(con.clientClassElem.name.=(c._class.
                            getComponentClassName())
769                          and con.serverClassElem.name.=(e.value.
                             oclAsType(SpringModel::Reference).getName
                             ()))
770                         {
771                             exists := true;
772                         };
773                     };
774                     if(not exists)
775                     {
776                         connections += c.map toConfigConnection(name, e.
                            value.oclAsType(SpringModel::Reference));
777                     }
778                 }
779             }
780             else if(e.value.oclIsTypeOf(SpringModel::Component))
781             {
782                 var exists : Boolean;
783                 exists := false;
784                 dedal.objectsOfType(DedalModel::Configuration)>forEach(
                    config)
785                 {
786                     config.configComponents>forEach(comp)
```

```
787                         {
788                             if(comp.name = e.value.oclAsType(SpringModel::
                                    Component)._class.getComponentClassName())
789                             {
790                                 exists := true
791                             }
792                         };
793                         if(not exists)
794                         {
795                             config.configComponents += e.value.oclAsType(
                                    SpringModel::Component).map toCompClass();
796                         };
797                         exists := false;
798                     };
799                     connections>forEach(con)
800                     {
801                         if(con.clientClassElem.name.=(c._class.
                                getComponentClassName())
802                             and con.serverClassElem.name.=(e.value.oclAsType(
                                    SpringModel::Component)._class.
                                    getComponentClassName()))
803                         {
804                             exists := true;
805                         };
806                     };
807                     if(not exists)
808                     {
809                         connections += c.map toConfigConnection(name, e.value
                                .oclAsType(SpringModel::Component));
810                     }
811                 }
812                 else if(e.value.oclIsKindOf(SpringModel::Collection))
813                 {
814                     connections += e.value.oclAsType(SpringModel::Collection)
                            .getClassConnections(c,name);
815                 };
816             }
817         };
818         return connections;
819 }
820
821 query SpringModel::Reference::getName() : String {
822     if(self.ref.oclIsTypeOf(SpringModel::Reference))
823     {
824         return self.ref.oclAsType(SpringModel::Reference).getName();
825     }
826     else if(self.ref.oclIsTypeOf(SpringModel::Component))
827     {
```

```
828             return self.ref.oclAsType(SpringModel::Component)._class.
                    getComponentClassName()
829         };
830         return null
831     }
```

Appendix C

Re-documentation algorithm

Algorithm 1 Main re-documentation algorithm

Require:

1: $\forall\, s \in \{specification.componentRoles, specification.roleConnections\},\ s = \emptyset$

2: $\forall\, s \in \{classes, assembly.componentInstances, assembly.instanceConnections, configuration.componentClasses, configuration.configConnections\},\ s \neq \emptyset$

3: $\forall\, compInst \in assembly.componentInstances,\ compInst.compInterfaces = \emptyset$

4: $\forall\, compCl \in configuration.componentClasses,\ (compCl.compInterfaces = \emptyset)$
$\wedge\ (compCl.attributes = \emptyset)$

5: $\forall\, compInst \in assembly.componentInstances,$
$(configuration.componentClasses = \bigcup_i compInst_i.instantiates)$

6: $\forall\, ac \in assembly.assemblyConnections,\ (\exists\, cc \in configuration.configConnections \mid$
$ac.clientElem.instantitates = cc.clientElem$
$\wedge\ ac.serverElem.instantitates = cc.serverElem)$

Ensure:

7: $\forall\, compInstance \in assembly.compInstances($
$(\forall\, instanceConnection \in instanceConnections,$
$instanceConnection.serverInterface \preceq instanceConnection.clientInterface)) \bigwedge$
$(\forall\, interface \in compInstance.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, class \in classes($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name)))))))$

8: $\forall\, compClass \in configuration.compClasses($
$(\forall\, interface \in compClass.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, class \in classes($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name))))))$
$\bigwedge(\forall\, attribute \in compClass.attributes(\exists\, class \in classes($
$\exists\, attr \in class.attributes, attribute.type = attr.type \bigwedge$
$attribute.name = attr.name)))$

9: $\forall\, compClass \in configuration.compClasses,\ (compClass.realizes \neq \emptyset)$
$\wedge(\forall\, role \in compClass.realizes,\ (compClass \preceq role)$

10: $\forall\, role \in specification.specComponents($
$(\forall\, interface \in role.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name)))))$
$\bigwedge\ (\exists\, c \in classes \mid \forall\, interface \in role.componentInterfaces,$
$interface.signatures \subseteq c.methods))$

11: **procedure** MAPARCHITECTURELEVELS($assembly : Assembly$,
$configuration : Configuration$,
$specification : Specification$,
$classes : Set(Class)$)
12: MAPCOMPONENTINSTANCES($classes, assembly.assmComponents$)
13: MAPASSEMBLYCONNECTIONS($classes, assembly.assemblyConnections$)
14: MAPCOMPONENTCLASSES($classes, assembly.assmComponents$,
$configuration.configComponents$)
15: MAPCONFIGCONNECTIONS($classes, configuration.configConnections$
$assembly.assemblyConnections$)
16: $specification.componentRoles \leftarrow$ BUILDCOMPONENTROLES(
$classes, configuration.componentClasses$,
$configuration.configConnections$)
17: $specification.specConnections \leftarrow$ MAPSPECCONNECTIONS(
$configuration.configConnections$)
18: **end procedure**

Algorithm 2 Re-Documenting **Assembly**

Ensure:

$\forall\, compInstance \in compInstances($
$\quad (\forall\, instanceConnection \in instanceConnections,$
$\quad\; instanceConnection.serverInterface \preceq instanceConnection.clientInterface)) \bigwedge$
$\quad (\forall\, interface \in compInstance.componentInterfaces($
$\quad\; \forall\, signature \in interface.interfaceType.signatures($
$\quad\quad \exists\, class \in classes($
$\quad\quad\; \exists\, method \in class.methods, signature.type = method.type \bigwedge$
$\quad\quad\; signature.name = method.name \bigwedge$
$\quad\quad\; (\forall\, parameter \in signature.parameters($
$\quad\quad\quad \exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$\quad\quad\quad parameter.name = p.name)))))))$

procedure MAPCOMPONENTINSTANCES($classes : Set(Class)$,
$\qquad\qquad compInstances : Set(CompInstance)$)
  for all $compInstance \in compInstances$ **do**
    $class \leftarrow classes.getByName(compInst.instantiates.name)$
    $compInstance.interfaces \leftarrow$ MAPINTERFACES($class$, $DIRECTION.PROVIDED$)
    for all $attribute \in class.attributes$ **do**
      if $\neg attribute.type.isPrimitive()$ **then**
        $compInstance.interfaces \leftarrow compInstance.interfaces \bigcup$ MAPINTERFACES($attribute.type$, $DIRECTION.REQUIRED$)
      end if
    end for
  end for
end procedure

Ensure:

$\forall\, instanceConnection \in instanceConnections,$
$\quad instanceConnection.serverInterface \preceq instanceConnection.clientInterface$

procedure MAPASSEMBLYCONNECTIONS($classes : Set(Class)$,
$\qquad\qquad instanceConnections : Set(InstConnection)$)
  for all $instanceConnection \in instanceConnections$ **do**
    MAPINSTANCECONNECTION($classes$, $instanceConnection$)
  end for
end procedure

Ensure:

1: $instanceConnection.serverInterface \preceq instanceConnection.clientInterface$

1: **procedure** MAPINSTANCECONNECTION($classes : Set(Class)$, $instanceConnection : InstanceConnection$)
2: $server \leftarrow instanceConnection.server$
3: $client \leftarrow instanceConnection.client$
4: $injectedAttributeType \leftarrow instanceConnection.injectedAttribute.type$
5: $class \leftarrow classes.getByName(injectedAttributeType.name)$
6: $requiredInterfaceType \leftarrow$ MAPINTERFACETYPE($class$)
7: $instanceConnection.serverInterface \leftarrow server.interfaces.getByKind($ $requiredInterfaceType)$ // it may also be a subtype
8: $classConnection.clientInterface \leftarrow client.interfaces.getByType($ $requiredInterfaceType)$
9: **end procedure**

Algorithm 3 Re-Documenting **Configuration**

Ensure:

1: $\forall\, compClass \in compClasses((\forall\, interface \in compClass.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, class \in classes($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name))))))$
$\bigwedge(\forall\, attribute \in compClass.attributes(\exists\, class \in classes($
$\exists\, attr \in class.attributes, attribute.type = attr.type \bigwedge$
$attribute.name = attr.name)))$

2: **procedure** MAPCOMPONENTCLASSES($classes : Set(Class)$, $compInstances : Set(CompInstance)$, $compClasses : Set(ComponentClass)$)
3: $class \leftarrow classes.getByName(compInst.instantiates.name)$
4: **for all** $compInstance \in compInstances$ **do**
5: $compClass \leftarrow compInstance.instantiates$
6: **if** $compClass.interfaces = \emptyset$ **then**
7: **for all** $interface \in compInstance.interfaces$ **do**
8: $newInterface \leftarrow copy(interface)$
9: $compClass.interfaces \leftarrow compClass.interfaces \bigcup newInterface$
10: $interface.instantiates\ getsnewInterface$
11: **end for**
12: **for all** $attribute \in class.attributes$ **do**
13: $compClass.attributes \leftarrow compClass.attributes \bigcup$ MAPATTRIBUTE($attribute$)
14: **end for**
15: **end if**
16: **end for**
17: **end procedure**

Ensure:
result.$type = attribute.type$
result.$name = attribute.name$

function MAPATTRIBUTE($attribute : Attribute$)
    $result : DedalAttribute$
    $result.type \leftarrow attribute.type$
    $result.name \leftarrow attribute.name$
    return $result$
end function

Ensure:
$\forall\, instanceConnection \in instConnecions($
    $\exists\, classConnection \in classConnections,$
    $instanceConnection.client.instantiates.contains(classConnection.client)$
    $\bigwedge instanceConnection.server.instantiates.contains(classConnection.sever)$
    $\bigwedge instanceConnection.clientInterface.type \preceq$
      $classConnection.clientInterface.type$
    $\bigwedge instanceConnection.serverInterface.type \preceq$
      $classConnection.serverInterface.type$
    $\bigwedge classConnection.serverInterface.type \preceq classConnection.clientInterface.type)$

procedure MAPCONFIGCONNECTIONS($classes : Set(Class)$,
                $configConnections : Set(ClassConnection)$
                $instConnections : Set(InstanceConnection)$)
    for all $instConnection \in instConnections$ **do**
        $classConnection \leftarrow configConnections.findByAttributeName($
        $instConnection.injectedAttribute)$
        $classConnection.clientInterface \leftarrow classConnection.client.findInterface($
          $instConnection.clientInterface)$
        $classConnection.serverInterface \leftarrow classConnection.client.findInterface($
          $serverConnection.clientInterface)$
    end for
end procedure

Algorithm 4 Re-Documenting **Specification**

Require:

$(classes \neq \emptyset) \wedge\ (compClasses \neq \emptyset) \wedge\ (clConnections \neq \emptyset)$

Ensure:

$\forall\, compClass \in compClasses,\ (compClass.realizes \neq \emptyset)$
$\wedge(\forall\, role \in compClass.realizes,\ (compClass \preceq role)$
$\forall\, role \in compClass.realizes((\forall\, interface \in role.componentInterfaces($
$\forall\, signature \in interface.interfaceType.signatures($
$\exists\, method \in class.methods, signature.type = method.type \bigwedge$
$signature.name = method.name \bigwedge$
$(\forall\, parameter \in signature.parameters($
$\exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$parameter.name = p.name)))))$
$\bigwedge\ (\exists\, c \in classes \mid \forall\, interface \in role.componentInterfaces,$
$interface.signatures \subseteq c.methods))$

procedure BUILDCOMPONENTROLES($classes : Set(Class)$,
$compClasses : Set(ComponentClass)$,
$clConnections : Set(ClassConnection)$)
  for all $compClass \in compClasses$ **do**
    $initialContract : Pair(Set(InterfaceType), Set(InterfaceType))$
    $initialContract \leftarrow computeContract(compClass, clConnections)$
    $compClass.realizes \leftarrow$ MAPCOMPONENTROLES($initialContract$,
      $classes.getByName(compClass.name)$)
  end for
end procedure

Ensure:

1: $\forall\, role \in$ **result**$((\forall\, interface \in role.componentInterfaces($
$\quad \forall\, signature \in interface.interfaceType.signatures($
$\quad\quad \exists\, method \in class.methods, signature.type = method.type \bigwedge$
$\quad\quad\quad signature.name = method.name \bigwedge$
$\quad\quad\quad (\forall\, parameter \in signature.parameters($
$\quad\quad\quad\quad \exists\, p \in method.parameters, parameter.type = p.type \bigwedge$
$\quad\quad\quad\quad\quad parameter.name = p.name)))))$

2: $\forall\, reqInterfaceType \in contract.first(\exists\, role \in$ **result**,
$\quad role.getAllRequiredTypes().contains(reqInterfaceType))$

3: $\forall\, provInterfaceType \in contract.second(\exists\, role \in$ **result**,
$\quad role.getAllProvidedTypes().contains(provInterfaceType))$

4: **function** MAPCOMPONENTROLES(
$\quad\quad contract : Pair(Set(InterfaceType), Set(InterfaceType)),$
$\quad\quad class : Class)) : Set(ComponentRole)$
5: **if** $\neg class.$**hasSuperType**() **then**
6: **return** $\{$MAPCOMPONENTROLE$(class)\}$
7: **end if**
8: $result : Set(ComponentRole)$
9: **for all** $superType \in class.superTypes$ **do**
10: $result \leftarrow result \bigcup$ MAPCOMPONENTROLES(
$\quad$SPLITCONTRACT$(superType, contract), superType)$
11: **end for**
12: **if** SATISFYCONTRACT$(roles,\ contract)$ **then**
13: **return** $result$
14: **else**
15: **return** $\{$MAPCOMPONENTROLE$(class)\}$
16: **end if**
17: **return** $result$
18: **end function**

Ensure:

$\forall interface \in \textbf{result}.componentInterfaces($
$\quad \forall signature \in interface.interfaceType.signatures($
$\quad\quad \exists method \in class.methods, signature.type = method.type \bigwedge$
$\quad\quad\quad signature.name = method.name \bigwedge$
$\quad\quad\quad (\forall parameter \in signature.parameters($
$\quad\quad\quad\quad \exists p \in method.parameters, parameter.type = p.type \bigwedge$
$\quad\quad\quad\quad\quad parameter.name = p.name))))$

procedure MAPCOMPONENTROLE($class : Class$) : $ComponentRole$
  $result : ComponentRole$
  $result.interfaces \leftarrow$
    MAPINTERFACES($class, DIRECTION.PROVIDED$)
  for all $attribute \in class.attributes$ **do**
    if $\neg attribute.type$.ISPRIMITIVE() **then**
      $result.interfaces \leftarrow$
        MAPINTERFACE($attribute.type, DIRECTION.REQUIRED$)
    end if
  end for
end procedure

Ensure:

$\forall interfaceType \in \textbf{result}.first(\exists interface \in compClass,$
$\quad interfaceType = interface.interfaceType)$
$\forall interfaceType \in \textbf{result}.second((\exists interface \in compClass,$
$\quad interfaceType = interface.interfaceType) \bigwedge$
$\quad (\exists classConnection \in classConnections,$
$\quad\quad classConnection.serverInterface.interfaceType = interfaceType))$

function COMPUTECONTRACT($compClass : ComponentClass,$
$\quad\quad\quad\quad\quad\quad classConnections : Set(ClassConnection)$
$\quad\quad\quad\quad\quad\quad): Pair(Set(InterfaceType), Set(InterfaceType))$
  $requiredInterfaceTypes \leftarrow componentClass.interfaces.getAllRequiredTypes()$
  for all $classConnection \in classConnections$ **do**
    if $classConnection.serverInterface \in compClass.interfaces$ **then**
      $connectedProvidedInterfaceTypes \leftarrow$
        $connectedProvidedInterfaceTypes$
          $\bigcup classConnection.serverInterface.interfacetype$
    end if
  end for
  return $(requiredInterfaceTypes,$
    $connectedProvidedInterfacetypes)$
end function

Ensure:

1: **result** $\subseteq initialContract$
2: $\forall interfaceType \in$ **result**$.first(\exists interface \in$MAPINTERFACES$(class, null)$,
$interfaceType = interface.interfaceType)$
3: $\forall interfaceType \in$ **result**$.second(\exists interface \in$MAPINTERFACES$(class, null)$,
$interfaceType = interface.interfaceType)$

4: **function** SPLITCONTRACT($class : Class$,
$initialContract : Pair(Set(InterfaceType)$,
$Set(InterfaceType))$
5: $compRole : ComponentRole$
6: $compRole \leftarrow$ MAPCOMPONENTROLE$(class)$
7: $requiredInterfaceTypes \leftarrow compRole.interfaces.getAllRequiredTypes()$
8: $providedInterfaceTypes \leftarrow compRole.interfaces.getAllProvidedTypes()$
9: **return** $((initialContract.first \bigcap requiredInterfaceTypes)$,
$(initialContract.second \bigcap providedInterfaceTypes))$
10: **end function**

Ensure:

1: **result** $= true \Rightarrow (contract.first \subseteq requiredInterfaces)$
$\wedge (contract.second \subseteq providedInterfaces)$

2: **function** SATISFYCONTRACT($compRoles : Set(ComponentRoles)$,
$contract : Pair(Set(InterfaceType)$,
$Set(InterfaceType))$:$Boolean$
3: $requiredInterfaces : Set(Interface)$
4: $providedInterfaces : Set(Interface)$
5: **for all** $compRole \in compRoles$ **do**
6: $requiredInterfaces \leftarrow requiredInterfaces$
$\bigcup compRole.interfaces.$**resolveAll**$(interface : Interface|$
$interface.direction = DIRECTION.REQUIRED)$
7: $providedInterfaces \leftarrow providedInterfaces$
$\bigcup compRole.interfaces.$**resolveAll**$(interface : Interface|$
$interface.direction = DIRECTION.PROVIDED)$
8: **end for**
9: **return** $(contract.first \subseteq requiredInterfaces)$
$\wedge (contract.second \subseteq providedInterfaces)$
10: **end function**

Ensure:

$\forall\, classConnection \in classConnecions(\exists\, roleConnection \in \textbf{result},$
$\quad classConnection.client.realizes.contains(roleConnection.client)$
$\quad \bigwedge classConnection.server.realizes.contains(roleConnection.sever)$
$\quad \bigwedge classConnection.clientInterface.type \preceq roleConnection.clientInterface.type$
$\quad \bigwedge classConnection.serverInterface.type \preceq$
$\quad\quad roleConnection.serverInterface.type$
$\quad \bigwedge roleConnection.serverInterface.type \preceq roleConnection.clientInterface.type)$

function MAPSPECCONNECTIONS($classConnections : Set(ClassConnection)$)
  $result : Set(RoleConnection)$
  for all $classConnection \in classConnections$ **do**
    $result \leftarrow result \bigcup$ MAPROLECONNECTION($classConnection$)
  end for
  return $result$
end function

Ensure:

$classConnection.client.realizes.contains(\textbf{result}.client)$
$classConnection.server.realizes.contains(\textbf{result}.sever)$
$classConnection.clientInterface.type \preceq \textbf{result}.clientInterface.type$
$classConnection.serverInterface.type \preceq \textbf{result}.serverInterface.type$
$\textbf{result}.serverInterface.type \preceq \textbf{result}.clientInterface.type$

function MAPROLECONNECTION(
$\quad\quad\quad\quad\quad\quad\quad\quad classConnection : ClassConnection$)
  $result : RoleConnection$
  $serverInterfaceType \leftarrow classConnection.serverInterface.type$
  $clientInterfaceType \leftarrow classConnection.clientInterface.type$
  $serverRoles \leftarrow classConnection.server.realizes$
  $clientRoles \leftarrow classConnection.client.realizes$
  $serverRole \leftarrow serverRoles.findRoleByInterfaceType(serverInterfaceType))$
  $clientRole \leftarrow clientRoles.findRoleByInterfaceType(clientInterfaceType))$
  $serverInterface \leftarrow serverRole.interfaces.getByKind(serverInterfaceType)$
  $clientInterface \leftarrow clientRole.interfaces.getByType(clientInterfaceType)$
  $result.server \leftarrow serverRole$
  $result.serverInterface \leftarrow serverInterface$
  $result.client \leftarrow clientRole$
  $result.clientInterface \leftarrow clientInterface$
  return $result$
end function

Algorithm 5 Mapping interfaces

Ensure:

$\forall\, interface \in \textbf{result}(\forall\, signature \in interface.interfaceType.signatures($
$\exists\, method \in class.methods, signature.type = method.type$
$\bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name))))$

function MAPINTERFACES($class : Class,\ direction : DIRECTION$)
  $result : Set(Interface)$
  $result \leftarrow \{\text{MAPINTERFACE}(class,\ direction)\}$
  for all $super \in class.superTypes$ **do**
    $result \leftarrow result \bigcup \text{MAPINTERFACES}(super,\ direction)$
  end for
  return $result$
end function

Ensure:

$\forall\, signature \in \textbf{result}.interfaceType.signatures(\exists\, method \in class.methods,$
$signature.type = method.type \bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name)))$

function MAPINTERFACE($class : Class,\ direction : DIRECTION$)
  $result : Interface$
  $result.direction \leftarrow direction$
  $result.name \leftarrow "I" + class.name$
  $result.interfaceType \leftarrow \text{MAPINTERFACETYPE}(class)$
  return $result$
end function

Ensure:

$\forall\, signature \in \textbf{result}.signatures(\exists\, method \in class.methods,$
$signature.type = method.type \bigwedge signature.name = method.name$
$\bigwedge(\forall\, parameter \in signature.parameters(\exists\, p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name)))$

function MAPINTERFACETYPE($class : Class$)
  $result : InterfaceType$
  $result.name \leftarrow "I" + class.name + "\_type"$
  for all $m \in class.methods$ **do**
    $result.signatures \leftarrow result.signatures \bigcup \text{MAPSIGNATURE}(m)$
  end for
  return $result$
end function

Ensure:
result.$type = method.type$
result.$name = method.name$
$\forall parameter \in$ **result**.$parameters(\exists p \in method.parameters,$
$parameter.type = p.type \bigwedge parameter.name = p.name)$

function MAPSIGNATURE($method : Method$)
    $result : Signature$
    $result.type \leftarrow method.type$
    $result.name \leftarrow method.name$
    for all $p \in method.parameters$ **do**
        $result.parameters \leftarrow result.parameters \bigcup$ MAPPARAMETER(p)
    end for
    return $result$
end function

Ensure:
result.$type = parameter.type$
result.$name = parameter.name$

function MAPPARAMETER($parameter : Parameter$) : $DedalParameter$
    $result : DedalParameter$
    $result.type \leftarrow parameter.type$
    $result.name \leftarrow parameter.name$
    return $result$
end function

Appendix D

Papers and tools

D.1 Released tools

item	url
GitHub repositories	
DedalStudio	https://github.com/DedalArmy/DedalStudio
Dedal	https://github.com/DedalArmy/Dedal
SpringDSL	https://github.com/DedalArmy/SpringDSL
Re-Documentation module	https://github.com/DedalArmy/Redoc
DedalModelComparator	https://github.com/DedalArmy/DedalModelComparator
DiffAnalyzer	https://github.com/DedalArmy/DiffAnalyzer
Maven repository	
http://www.dev.lgi2p.mines-ales.fr/ariane/mvn	
Eclipse plugin sites	
Dedal	http://www.dev.lgi2p.mines-ales.fr/ariane/p2/dedal/2019-06/
SpringDSL	http://www.dev.lgi2p.mines-ales.fr/ariane/p2/springdsl/2019-06/
Re-Documentation	http://www.dev.lgi2p.mines-ales.fr/ariane/p2/redoc/2019-06/

TABLE D.1: Tool release websites

D.2 Published papers

Le Borgne, A., Delahaye, D., Huchard, M., Urtado, C., Vauttier, S. Substitutability-Based Version Propagation to Manage the Evolution of Three-Level Component-Based Architectures. In SEKE: Software Engineering and Knowledge Engineering, Jul 2017, Pittsburgh, United States. pp.18-23

Le Borgne, A., Delahaye, D., Huchard, M., Urtado, C., Vauttier, S. Recovering Three-Level Architectures from the Code of Open-Source Java Spring Projects. In SEKE: Software Engineering and Knowledge Engineering, Jul 2018, San Francisco, United States. pp.199-202

Quentin Perez, Alexandre Le Borgne, Christelle Urtado, Sylvain Vauttier. An Empirical Study about Software Architecture Configuration Practices with the Java Spring Framework. In SEKE: Software Engineering and Knowledge Engineering, Jul 2019, Lisbon, Portugal. pp.465-468

Appendix E

Résumé en français

Tout au long de son cycle de vie, un logiciel peut connaître de nombreux changements affectant potentiellement sa conformité avec sa documentation originelle. De plus, bien qu'une documentation à jour, conservant les décisions de conception prises pendant le cycle de développement, soit reconnue comme une aide importante pour maîtriser les évolutions, la documentation des logiciels est souvent obsolète. Les modèles d'architectures sont l'une des pièces majeures de la documentation. Assurer leur cohérence avec les autres modèles d'un logiciel (incluant son code) pendant les processus d'évolution (co-évolution) est un atout majeur pour la qualité logicielle. En effet, la compréhension des architectures logicielles est hautement valorisable en termes de capacités de réutilisation, d'évolution et de maintenance. Pourtant les modèles d'architectures sont rarement explicitement disponibles et de nombreux travaux de recherche visent à les retrouver à partir du code source. Cependant, la plupart des approches existantes n'effectuent pas un strict processus de rétro-documentation afin de re-documenter les architectures "comme elles sont implémentées" mais appliquent des étapes de ré-ingénierie en regroupant des éléments de code dans de nouveaux composants. Ainsi, cette thèse propose un processus de re-documentation des architectures telles qu'elles ont été conçues et implémentées, afin de fournir un support d'analyse des décisions architecturales effectives. Cette re-documentation se fait par l'analyse du code orienté objet et les descripteurs de déploiement de projets. Le processus re-documente les projets dans le langage de description d'architecture Dedal, qui est spécialement conçu pour contrôler et guider l'évolution des logiciels. Un autre aspect très important de la documentation des logiciels est le suivi de leurs différentes versions. Dans de nombreuses approches et gestionnaires de version actuels, comme GitHub, les fichiers sont versionnés de manière agnostique. S'il est possible de garder une trace de l'historique des versions de n'importe quel fichier, aucune information ne peut être fournie sur la sémantique des changements réalisés. En particulier, lors du versionnement d'éléments logiciels, il n'est fourni aucun diagnostic de retro-compatibilité avec les versions précédentes. Cette thèse propose donc un mécanisme de versionnement d'architectures logicielles basé sur le métamodèle et les propriétés formelles de l'ADL Dedal. Il permet d'analyser automatiquement les versions en termes de substituabilité, de gérer la propagation de version et d'incrémenter automatiquement les numéros de versions en tenant compte de l'impact des changements. En proposant

cette approche formelle, cette thèse vise à prévenir le manque de contrôle des décisions architecturale (dérive / érosion). Cette thèse s'appuie sur une étude empirique, pour valider notre approche nommée ARIANE, en appliquant les processus de re-documentation et de versionnement à de nombreuses versions d'un projet industriel extrait de GitHub.

Bibliography

[Abo+09] Nour Alhouda Aboud, Gabriela Arévalo, Jean-Rémy Falleri, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, Sylvain Vauttier, and Gabriela Ar. "Automated architectural component classification using concept lattices". In: *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE. 2009, pp. 21–30. ISBN: 9781424449859.

[Abo+19] Nour Aboud, Gabriela Areévalo, Olivier Bendavid, Jean-Rémy Falleri, Nicolas Haderer, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. "Building Hierarchical Component Directories". In: *Journal of Object Technology* 18.1 (Mar. 2019), 2:1–37. ISSN: 1660-1769.

[Abr96] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ADG98] Robert Allen, Remi Douence, and David Garlan. "Specifying and analyzing dynamic software architectures". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 1998, pp. 21–37.

[ADO14] Abdelkrim Amirat, Afrah Djeddar, and Mourad Oussalah. "Evolving and Versioning Software Architectures Using ATL Transformations". In: *The International Arab Conference on Information Technology (ACIT'2014)*. 2014.

[AG97] Robert Allen and David Garlan. "A formal basis for architectural connection". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.3 (1997), pp. 213–249.

[Als+16] Z. Alshara, A. D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi. "Materializing Architecture Recovered from Object-Oriented Source Code in Component-Based Languages". In: *10th ECSA Proc.* Vol. 9839. LNCS. Copenhagen, Denmark: Springer, 2016, pp. 309–325. ISBN: 978-3-319-48992-6-23.

[Alt+08] Kerstin Altmanninger, Kerstin Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. "AMOR – Towards Adaptable Model Versioning". In: *1st International Workshop on Model CoEvolution and Consistency Management in conjunction with MODELS 08* (2008), pp. 55–60.

[Alt+09] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. "Why model versioning research is

needed!? an experience report". In: *Proceedings of the MoDSE-MCCM Workshop@ MoDELS*. Vol. 9. 2009.

[AP03] Marcus Alanen and Ivan Porres. "Difference and union of models". In: *International Conference on the Unified Modeling Language*. Springer. 2003, pp. 2–17.

[Aré+07] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. "Precalculating component interface compatibility using FCA". In: *Proceedings of the 5^{th} international conference on Concept Lattices and their Applications*. Ed. by Jean Diatta, Peter Eklund, and Michel Liquière. Montpellier, France: CEUR Workshop Proceedings Vol. 331, 2007, pp. 241–252.

[Aré+09] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. "Formal concept analysis-based service classification to dynamically build efficient software component directories". In: *International journal of general systems* 38.4 (2009), pp. 427–453.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. "A survey on model versioning approaches". In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304. ISSN: 1744092.

[BB09] Jaroslav Bauml and Premek Brada. "Automated versioning in OSGi: A mechanism for component software consistency guarantee". In: *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2009, pp. 428–435.

[BB11] Jaroslav Bauml and Premek Brada. "Reconstruction of type information from java bytecode for component compatibility". In: *Electronic Notes in Theoretical Computer Science* 264.4 (2011), pp. 3–18.

[BCL12] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. "A systematic review of software architecture evolution research". In: *Information and Software Technology* 54.1 (2012), pp. 16 –40. ISSN: 0950-5849.

[Beu+99] Antoine Beugnard, J-M Jézéquel, Noël Plouzeau, and Damien Watkins. "Making components contract aware". In: *Computer* 32.7 (1999), pp. 38–45.

[Bey01] Derek Beyer. *C# Com+ Programming*. M & T Books, 2001.

[BHP06] Tomas Bures, Petr Hnětynka, and František Plášil. "Sofa 2.0: Balancing advanced features in a hierarchical component model". In: *Software Engineering Research, Management and Applications. 4^{th} International Conference on*. IEEE. 2006, pp. 40–48.

[BJP10] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. "Contract aware components, 10 years after". In: *arXiv preprint arXiv:1010.2822* (2010).

[BM88] David Beech and Brom Mahbod. "Generalized version control in an object-oriented database". In: *Proceedings. Fourth International Conference on Data Engineering*. IEEE. 1988, pp. 14–22.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. " O'Reilly Media, Inc.", 2006.

[BR00a] Keith H Bennett and Václav T Rajlich. "Software maintenance and evolution: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 73–87.

[BR00b] David J Brown and Karl Runge. "Library Interface Versioning in Solaris and Linux." In: *Annual Linux Showcase & Conference*. 2000.

[Bra01a] Premysl Brada. "Towards automated component compatibility assessment". In: *Workshop on Component-Oriented Programming (WCOP'2001)*. Citeseer. 2001.

[Bra01b] Přemysl Brada. "Component Revision Identification Based on IDL/ADL Component Specification". In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria: ACM, 2001, pp. 297–298. ISBN: 1-58113-390-1.

[Bra03] Premysl Brada. "Specification-based component substitutability and revision identification". In: *Univerzita Karlova, Matematicko-fyzikální fakulta* (2003).

[Bra99] Přemysl Brada. "Component change and version identification in SOFA". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1725 (1999), pp. 360–368. ISSN: 16113349.

[Bro+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. "An Introduction to Model Versioning". In: *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*. SFM'12. Bertinoro, Italy: Springer-Verlag, 2012, pp. 336–398. ISBN: 978-3-642-30981-6.

[BV06] Premysl Brada and Lukas Valenta. "Practical verification of component substitutability using subtype relation". In: *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*. IEEE. 2006, pp. 38–45.

[CC90] Elliot J. Chikofsky and James H Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE software* 7.1 (1990), pp. 13–17.

[CCL05] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. "Component-based development process and component lifecycle". In: *Journal of Computing and Information Technology* 13.4 (2005), pp. 321–327.

[CCL12] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. "A Solution for Concurrent Versioning of Metamodels and Models." In: *Journal of Object Technology* 11.3 (2012), pp. 1–32.

[CDP09] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. "Managing dependent changes in coupled evolution". In: *Proceedings of the International Conference on Theory and Practice of Model Transformations*. Springer. 2009, pp. 35–51.

[CH01] WT Council and GT Heineman. "Component-based Software Engineering Putting the Pieces Together". In: *Addison Weysley* (2001).

[Cha+08] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. "Extraction of component-based architecture from object-oriented systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5292 LNCS (2008), pp. 322–325. ISSN: 03029743.

[CK88] Hong-Tai Chou and Won Kim. "Versions and change notification in an object-oriented database system". In: *Proceedings of the 25th ACM/IEEE design automation conference*. IEEE Computer Society Press. 1988, pp. 275–281.

[CKS05] A. Christl, R. Koschke, and M. A. Storey. "Equipping the reflexion method with automated clustering". In: *12th WCRE Proc.* Pittsburgh, USA, 2005, pp. 10–98. ISBN: 0-7695-2474-5.

[CL02] Ivica Crnkovic and Magnus Peter Henrik Larsson. *Building reliable component-based software systems*. Artech House, 2002.

[Cre94] Régis Bernard Joseph Crelier. "Separate compilation and module extension". PhD thesis. ETH Zurich, 1994.

[CW98] Reidar Conradi and Bernhard Westfechtel. "Version models for software configuration management". In: *ACM Computing Surveys* 30.2 (1998), pp. 232–282.

[CY07] Feng Chen and Hongji Yang. "Model oriented evolutionary redocumentation". In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. Vol. 1. IEEE. 2007, pp. 543–548.

[Dem+16] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E Lopez-Herrejon, and Alexander Egyed. "Co-evolution of metamodels and models through consistent change propagation". In: *Journal of Systems and Software* 111 (2016), pp. 281–297.

[Dev99] P Devanbu. "The ultimate reuse nightmare: Honey, i got the wrong dll". In: *the 5th Symposium on Sofware Reuseability*. 1999, pp. 178–180.

[DHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages". In: *ACM Transactions on Software Engineering and Methodology* 14.2 (Apr. 2005), pp. 199–245. ISSN: 1049-331X.

[DM01] Lei Ding and Nenad Medvidovic. "Focus: A light-weight, incremental approach to software architecture recovery and evolution". In: *Proceedings - Working IEEE/IFIP Conference on Software Architecture, WICSA 2001* (2001), pp. 191–200.

[DM08] Serge Demeyer and Tom Mens. *Software Evolution*. Springer, 2008.

[DP09] S. Ducasse and D. Pollet. "Software architecture reconstruction: A process-oriented taxonomy". In: *IEEE TSE* 35.4 (2009), pp. 573–591. ISSN: 00985589.

[DSB12] Lakshitha De Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012), pp. 132–151.

[EC95] Jacky Estublier and Rubby Casallas. "Three dimensional versioning". In: *Software Configuration Management* (1995), pp. 118–135.

[EET11] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. "A formal resolution strategy for operation-based conflicts in model versioning using graph modifications". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2011, pp. 202–216.

[Eix+98] W Eixelsberger, M Ogris, H Gall, and B Bellay. "Software architecture recovery of a program family". In: *International Conference on Software Engineering* (1998), pp. 508–511.

[EJS03] Susan Eisenbach, Vladimir Jurisic, and Chris Sadler. "Managing the evolution of. net programs". In: *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer. 2003, pp. 185–198.

[EKS03] T. Eisenbarth, R. Koschke, and D. Simon. "Locating features in source code". In: *IEEE TSE* 29.3 (2003), pp. 210–224. ISSN: 00985589.

[Eng97] Robert Englander. *Developing JAVA beans*. " O'Reilly Media, Inc.", 1997.

[Est+05] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. "Impact of software engineering research on the practice of software configuration management". In: *ACM Transactions on Software Engineering and Methodology* 14.4 (2005), pp. 383–430.

[GAO09] David Garlan, Robert Allen, and John Ockerbloom. "Architectural mismatch: Why reuse is still so hard". In: *IEEE software* 26.4 (2009), pp. 66–69.

[Gar00] David Garlan. "Software architecture: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. Citeseer. 2000, pp. 91–101.

[Gar+09] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. "Managing Model Adaptation by Precise Detection of Metamodel Changes". In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F Paige, Alan Hartman, and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 34–49. ISBN: 978-3-642-02674-4.

[GDT06] John C Georgas, Eric M Dashofy, and Richard N Taylor. "Architecture-centric development: a different approach to software engineering". In: *XRDS: Crossroads, The ACM Magazine for Students* 12.4 (2006), pp. 6–6.

[Gin+87] Robert A Gingell, Meng Lee, Xuong T Dang, and Mary S Weeks. "Shared libraries in SunOS". In: *AUUGN* 8.5 (1987), p. 112.

[GK00] M. Gogolla and R. Kollmann. "Re-documentation of Java with UML class diagrams". In: *Proc. 7th Reengineering Forum, Reengineering Week 2000*. Zurich, Switzerland, 2000.

[GMW97] David Garlan, Robert Monroe, and David Wile. "Acme : An Architecture Description Interchange Language 1 Introduction". In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON)* (1997), p. 7.

[Hat04] A. Hatch. "Software Architecture Visualisation". PhD thesis. Durham University, 2004, 173 pp.

[Hay+68] Seymour Hayden, Ernst Zermelo, Abraham Adolf Fraenkel, and John F Kennison. *Zermelo-Fraenkel set theory*. CE Merrill, 1968.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. "COPE - Automating Coupled Evolution of Metamodels and Models". In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by Sophia Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 52–76. ISBN: 978-3-642-03013-0.

[Her09] Markus Herrmannsdoerfer. "Operation-based versioning of metamodels with COPE". In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM 2009*. 2009, pp. 49–54. ISBN: 9781424437146.

[HHT01] Jochen Hartmann, Shihong Huang, and Scott Tilley. "Documenting software systems with views II: an integrated approach based on XML". In: *Proceedings of the 19th annual international conference on Computer documentation*. ACM. 2001, pp. 237–246.

[HMY06] G. Huang, H. Mei, and F. Q. Yang. "Runtime recovery and manipulation of software architecture of component-based systems". In: *Automated Software Engineering* 13.2 (2006), pp. 257–281. ISSN: 0928-8910.

[Hoa78] Charles Antony Richard Hoare. "Communicating sequential processes". In: *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[HP04] Petr Hnětynka and František Plášil. "Distributed versioning model for MOF". In: *Proceedings of the winter international synposium on Information and communication technologies*. Trinity College Dublin. 2004, pp. 1–6.

[HP06] K Hussey and M Paternostro. "Advanced features of EMF". In: *Tutorial at EclipseCon* (2006), p. 218.

[Iee] "IEEE Standard for Software Maintenance". In: *IEEE Std 1219-1998* (1998), pp. 1–56.

[Jar+14] Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. "Github projects. quality analysis of open-source software". In: *International Conference on Social Informatics*. Springer. 2014, pp. 80–94.

[Java] *Controlling Access to Members of a Class*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html.

[Javb] *JavaBeans documentation*. URL: https://www.oracle.com/technetwork/java/javase/documentation/index-141852.html.

[Joh+04] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, et al. "The Spring framework – Reference documentation". In: *Interface* 21 (2004), p. 27.

[Kal+16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. ISSN: 1573-7616.

[Kat90] Randy H Katz. "Toward a unified framework for version modeling in engineering databases". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 375–409.

[KCB86] Randy H Katz, Ellis Chang, and Rajiv Bhateja. *Version modeling concepts for computer-aided design databases*. Vol. 15. 2. ACM, 1986.

[Keb+12] Selim Kebir, Abdelhak Djamel Seriai, Sylvain Chardigny, and Allaoua Chaoui. "Quality-centric approach for software component identification from object-oriented code". In: *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012* (2012), pp. 181–190.

[Kha+01] Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, N Medvidovic, Richard N R N Taylor, and Nenad Medvivovic. "xADL: Enabling Architecture-Centric Tool Integration with XML". In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. 2001, pp. 3–6. ISBN: 0-7695-0981-9.

[KHS09] M. Koegel, J. Helming, and S. Seyboth. "Operation-based conflict detection and resolution". In: *2009 ICSE Workshop on Comparison and Versioning of Software Models*. 2009, pp. 43–48.

[Kno+06] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. "Static evaluation of software architectures". In: *10th CSMR Proc.* Bari, Italy: IEEE, 2006, pp. 279–294. ISBN: 0-7695-2536-9.

[Kos02] R Koschke. "Atomic Architectural Component Recovery for Program Understanding and Evolution". In: *International Conference on Software Maintenance (ICSM'02)*. 2002, pp. 2–5. ISBN: 0769518192.

[Lam92] Patrick Lambrix. *Aspects of version management of composite objects*. Linköping University, Department of Computer and Information Science, 1992.

[Lan86] Gordon Landis. "Design Evolution and History in an Object-Oriented CAD/-CAM Database." In: *COMPCON*. 1986, pp. 297–305.

[Le +17] A. Le Borgne, D. Delahaye, M. Huchard, C. Urtado, and S. Vauttier. "Substitutability-Based Version Propagation to Manage the Evolution of Three-Level Component-Based Architectures". In: *29th SEKE Proc.* Pittsburgh, USA, 2017, pp. 18–23.

[Le +18] A. Le Borgne, D. Delahaye, M. Huchard, C. Urtado, and S. Vauttier. "Recovering Three-Level Architectures from the Code of Open-Source Java Spring Projects". In: ***to appear*** *30th SEKE Proc.* San-Francisco, USA, 2018.

[Leh79] Meir M Lehman. "On understanding laws, evolution, and conservation in the large-program life cycle". In: *Journal of Systems and Software* 1 (1979), pp. 213–221.

[Lev99] John R Levine. *Linkers & loaders*. Vol. 1. Morgan Kaufmann, 1999.

[Low05] J Lowy. "Programming .NET components (ed.)" In: *OReilly Media Inc* (2005).

[LST78] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. "Characteristics of application software maintenance". In: *Communications of the ACM* 21.6 (1978), pp. 466–471.

[LW94] Barbara H Liskov and Jeannette M Wing. "A behavioral notion of subtyping". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994), pp. 1811–1841.

[Mag+95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. "Specifying distributed software architectures". In: *Software Engineering—* (1995).

[MB17] Sander Mak and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. " O'Reilly Media, Inc.", 2017.

[ME04] Stephen McCamant and Michael D. Ernst. "Early Identification of Incompatibilities in Multi-component Upgrades". In: *ECOOP 2004 – Object-Oriented Programming*. Ed. by Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 440–464. ISBN: 978-3-540-24851-4.

[Mey92] Bertrand Meyer. "Applying'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[MH97] Vlada Matena and Mark Hapner. "Enterprise JavaBeans TM". In: *Sun Microsystems* (1997).

[MJ06] Nenad Medvidovic and Vladimir Jakobac. "Using software evolution to Focus architectural recovery". In: *Automated Software Engineering* 13.2 (2006), pp. 225–256. ISSN: 09288910.

[MK01] Nabor C. Mendonça and Jeff Kramer. "An approach for recovering distributed system architectures". In: *Automated Software Engineering* 8.3-4 (2001), pp. 311–354. ISSN: 09288910.

[MMP00] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. "Towards a taxonomy of software connectors". In: *Proceedings of the 22nd international conference on Software engineering*. ACM. 2000, pp. 178–187.

[Mok15] Abderrahman Mokni. "A formal approach to automate the evolution management in component-based software development processes. (Une approche formelle pour automatiser la gestion de l'{é}volution dans les processus de d{é}veloppement {à} base de composants)". PhD thesis. University of Montpellier, France, 2015.

[Mok+15] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Yulin Zhang. "An evolution management model for multi-level component-based software architectures". In: 2015.

[Mok+16a] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, and H. Y. Zhang. "A formal approach for managing component-based architecture evolution". In: *SCP* 127 (2016), pp. 24–49.

[Mok+16b] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. "A three-level versioning model for component-based software architectures". In: *Proceedings of the* 11^{th} *International Conference on Software Engineering Advances*. Roma, Italy, 2016, pp. 178 –183.

[Moo01] Leon Moonen. "Generating robust parsers using island grammars". In: *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE. 2001, pp. 13–22.

[Mor96] Tom Morse. "CVS". In: *Linux Journal* 1996.21es (1996), p. 3.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I". In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 10902651.

[MRT98] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. "A type theory for software architectures". In: *Tech. Rep. UCI-ICS-98-14* (1998).

[MRT99] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. "A language and environment for architecture-based software development and evolution". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE. 1999, pp. 44–53.

[Nar+09] Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. "Automatic Domain Model Migration to Manage Metamodel Evolution". In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 706–711. ISBN: 978-3-642-04425-0.

[NER01] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. "Making inconsistency respectable in software development". In: *Journal of Systems and Software* 58.2 (2001), pp. 171–180.

[OMG+02] Object Management Group OMG et al. *Common Object Request Broker Architecture: Core Specification*. 2002.

[OMW05] Hamilton Oliveira, Leonardo Murta, and Cláudia Werner. "Odyssey-VCS: A Flexible Version Control System for UML Model Elements". In: *Proceedings of the 12th International Workshop on Software Configuration Management*. SCM '05. Lisbon, Portugal: ACM, 2005, pp. 1–16. ISBN: 1-59593-310-7.

[Oqu+04] Flavio Oquendo, Brian Warboys, Ron Morrison, and Régis Dindeleux. "ARCHWARE : Architecting Evolvable Software". In: *Engineering* (2004), pp. 1–16.

[OTC93] Chabane Oussalah, Guilaine Talens, and MF Colinas. "Concepts and methods for version modeling". In: *Proceedings of EURO-DAC 93 and EURO-VHDL 93-European Design Automation Conference*. IEEE. 1993, pp. 332–337.

[PBJ98] František Plášil, Dušan Bálek, and Radovan Janec. "SOFA / DCUP : Architecture for Component Trading and Dynamic Updating". In: *Proc. Fourth Int'l Conf. Configurable Distributed Systems (ICCDS '98)* (1998), pp. 43–52.

[Per+19] Quentin Perez, Alexandre Le Borgne, Christelle Urtado, and Sylvain Vauttier. "An Empirical Study about Software Architecture Configuration Practices with the Java Spring Framework". In: 2019.

[PMR16] Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. "Evolving models in Model-Driven Engineering: State-of-the-art and future challenges". In: *Journal of Systems and Software* 111 (2016), pp. 272 –280. ISSN: 0164-1212.

[PR04] I. Pashov and M. Riebisch. "Using feature modeling for program comprehension and software architecture recovery". In: *Proc. 11th IEEE Int'l Conf. and Wkshp on the Engineering of Computer-Based Systems* (2004), pp. 406–417.

[Pra01] Steven Pratschner. "Simplifying deployment and solving DLL Hell with the .NET framework". In: *MSDN Magazine* (2001).

[Pre97] Wolfgang Pree. "Component-based software development-a new paradigm in software engineering?" In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. IEEE. 1997, pp. 523–524.

[PW92] Dewayne E Perry and Alexander L Wolf. "Foundations for the study of software architecture". In: *ACM SIGSOFT Software engineering notes* 17.4 (1992), pp. 40–52.

[Raj00] Václav Rajlich. "Incremental redocumentation using the web". In: *IEEE Software* 17.5 (2000), pp. 102–106.

[Raj97] Václav Rajlich. "Incremental redocumentation with hypertext". In: *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*. IEEE. 1997, pp. 68–72.

[RE12] Alexander Reder and Alexander Egyed. "Incremental consistency checking for complex design rules and larger model changes". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2012, pp. 202–218.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[Rog97] D Rogers. *Inside COM: Microsoft's Component Object Model*. 1997.

[Ros+04] Roshanak Roshandel, André van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. "Mae—a system model and environment for managing architectural evolution". In: *ACM Transactions on Software Engineering and Methodology* 13.2 (2004), pp. 240–276.

[Ros+10] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, and Fiona A C Polack. "Model Migration with Epsilon Flock". In: *Theory and Practice of Model Transformations*. Ed. by Laurence Tratt and Martin Gogolla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 184–198. ISBN: 978-3-642-13688-7.

[Ros+14] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, Fiona A C Polack, and Simon Poulding. "Epsilon Flock: a model migration language". In: *Software & Systems Modeling* 13.2 (2014), pp. 735–755. ISSN: 1619-1374.

[Roy87] Winston W Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press. 1987, pp. 328–338.

[RSB04] Ed Roman, Rima Patel Sriganesh, and Gerald Brose. *Mastering enterprise javabeans*. John Wiley & Sons, 2004.

[SAO05] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. "A study of the documentation essential to software maintenance". In: *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM. 2005, pp. 68–75.

[Sar03] K. Sartipi. "Software architecture recovery based on pattern matching". In: *ICSM Proc.* Amsterdam, The Netherlands: IEEE, 2003, pp. 293–296. ISBN: 0-7695-1905-9.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[Sha+01] Bill Shannon et al. "Java 2 platform enterprise edition specification, v1. 3". In: *Sun Microsystems* 901 (2001).

[Sha+17] A. Shatnawi, A. D. Seriai, H. Sahraoui, and Z. Alshara. "Reverse engineering reusable software components from object-oriented APIs". In: *JSS* 131 (2017), pp. 442–460. ISSN: 01641212.

[SIS03] Shahida Sulaiman, Norbik Bashah Idris, and Shamsul Sahibuddin. "Re-documenting, visualizing and understanding software system using DocLike Viewer". In: *Tenth Asia-Pacific Software Engineering Conference, 2003.* IEEE. 2003, pp. 154–163.

[SM07] A. Sutton and J. I. Maletic. "Recovering UML class models from C++: A detailed explanation". In: *IST* 49.3 (2007), pp. 212–229. ISSN: 09505849.

[SO01] C. Stoermer and L. O'Brien. "MAP–Mining Architectures for Product Line Evaluations". In: *IEEE/IFIP WICSA Proc.* Amsterdam, The Netherlands, 2001, pp. 35–44. ISBN: 0769513603.

[Som11] Ian Sommerville. "Software engineering 9th Edition". In: *ISBN-10137035152* (2011).

[SS13] Anas Shatnawi and Abdelhak Djamel Seriai. "Mining reusable software components from object-oriented source code of a set of similar software". In: *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013* (2013), pp. 193–200.

[Stu05] Alexander Stuckenholz. "Component evolution and versioning state of the art". In: *ACM SIGSOFT Software Engineering Notes* 30.1 (2005), p. 7.

[SWB03] Perdita Stevens, Jon Whittle, and Grady Booch. *UML 2003–The Unified Modeling Language, Modeling Languages and Applications: 6th International Conference San Francisco, CA, USA, October 20-24, 2003, Proceedings*. Vol. 2863. Springer, 2003.

[Tae+13] Gabriele Taentzer, Florian Mantz, Thorsten Arendt, and Yngve Lamo. "Customizable Model Migration Schemes for Meta-model Evolutions with Multiplicity Changes". In: *Model-Driven Engineering Languages and Systems*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 254–270. ISBN: 978-3-642-41533-3.

[Tay+96] Richard N Taylor, Nenad Medvidovic, Kenneth M Anderson, E James Whitehead Jr, Jason E Robbins, Kari A Nies, Peyman Oreizy, and Deborah L Dubrow. "A Component- and Message- Architectural Style for GUI Software". In: *IEEE Transactions on Software Engineering* 22.6 (1996), pp. 390–406.

[TH10] Linus Torvalds and Junio Hamano. "Git: Fast version control system". In: *URL http://git-scm. com* (2010). last visited: 03.05.2017.

[TH99] John B. Tran and Richard C. Holt. "Forward and reverse repair of software architecture". In: *1999 Conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)* (1999), pp. 12–21.

[TMD10] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. "Software architecture: foundations, theory, and practice". In: (2010).

[TO93] Guilaine Talens and Chabane Oussalah. "Versions of simple and composite objects". In: *In Proc. 19th VLDB*. Citeseer. 1993.

[UO96] Christelle Urtado and Chabane Oussalah. "Propagation de versions dans les objets complexes." In: *INFORSID*. 1996, pp. 331–349.

[UO98] Christelle Urtado and Chabane Oussalah. "Complex entity versioning at two granularity levels". In: *Information systems* 23.3-4 (1998), pp. 197–216.

[VDK99] Arie Van Deursen and Tobias Kuipers. "Building documentation generators". In: *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE. 1999, pp. 40–49.

[Yan+04] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. "DiscoTect: a system for discovering architectures from running systems". In: *26th ICSE Proc.* Edinburgh, UK, 2004, pp. 470–479. ISBN: 0-7695-2163-0.

[Zdo87] Stanley B Zdonik. "Version management in an object-oriented database". In: *Advanced Programming Environments*. Springer. 1987, pp. 405–422.

[Zha10] Huaxi Yulin Zhang. "Multi-dimensional architecture description language for forward and reverse evolution of component-based software". PhD thesis. Montpellier 2, 2010.

[Zha+12a] Huaxi Zhang, Christelle Urtado, Sylvain Vauttier, Lei Zhang, Marianne Huchard, and Bernard Coulette. "Dedal-CDL: Modeling First-class Architectural Changes in Dedal". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 272–276.

[Zha+12b] Huaxi Yulin Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. "A three-level component model in component based software development". In: *Proceedings of ACM SIGPLAN Notices*. Vol. 48. 3. ACM. 2012, pp. 70–79.

[ZUS10] H. Y. Zhang, C. Urtado, and S.Vauttier. "Architecture-centric component-based development needs a three-level ADL". In: *4th ECSA Proc.* Vol. 6285. LNCS. Copenhagen, Denmark: Springer, 2010, pp. 295–310.

[ZUV10] Huaxi Yulin Zhang, Christelle Urtado, and Sylvain Vauttier. "Architecture-centric development and evolution processes for component-based software". In: *Proc. of 22nd SEKE Conf., Redwood City, USA (July 2010)*. 2010, p. 25.

ARIANE : Re-documentation automatique pour améliorer la compréhension et l'évolution d'architectures logicielles

Tout au long de son cycle de vie, un logiciel peut connaître de nombreux changements affectant potentiellement sa conformité avec sa documentation originelle. De plus, bien qu'une documentation à jour, conservant les décisions de conception prises pendant le cycle de développement, soit reconnue comme une aide importante pour maîtriser les évolutions, la documentation des logiciels est souvent obsolète. Les modèles d'architectures sont l'une des pièces majeures de la documentation. Assurer leur cohérence avec les autres modèles d'un logiciel (incluant son code) pendant les processus d'évolution (co-évolution) est un atout majeur pour la qualité logicielle. En effet, la compréhension des architectures logicielles est hautement valorisable en termes de capacités de réutilisation, d'évolution et de maintenance. Pourtant les modèles d'architectures sont rarement explicitement disponibles et de nombreux travaux de recherche visent à les retrouver à partir du code source. Cependant, la plupart des approches existantes n'effectuent pas un strict processus de rétro-documentation afin de re-documenter les architectures "comme elles sont implémentées" mais appliquent des étapes de ré-ingénierie en regroupant des éléments de code dans de nouveaux composants. Ainsi, cette thèse propose un processus de re-documentation des architectures telles qu'elles ont été conçues et implémentées, afin de fournir un support d'analyse des décisions architecturales effectives. Cette re-documentation se fait par l'analyse du code orienté objet et les descripteurs de déploiement de projets. Le processus re-documente les projets dans le langage de description d'architecture Dedal, qui est spécialement conçu pour contrôler et guider l'évolution des logiciels. Un autre aspect très important de la documentation des logiciels est le suivi de leurs différentes versions. Dans de nombreuses approches et gestionnaires de version actuels, comme GitHub, les fichiers sont versionnés de manière agnostique. S'il est possible de garder une trace de l'historique des versions de n'importe quel fichier, aucune information ne peut être fournie sur la sémantique des changements réalisés. En particulier, lors du versionnement d'éléments logiciels, il n'est fourni aucun diagnostic de retro-compatibilité avec les versions précédentes. Cette thèse propose donc un mécanisme de versionnement d'architectures logicielles basé sur le métamodèle et les propriétés formelles de l'ADL Dedal. Il permet d'analyser automatiquement les versions en termes de substituabilité, de gérer la propagation de version et d'incrémenter automatiquement les numéros de versions en tenant compte de l'impact des changements. En proposant cette approche formelle, cette thèse vise à prévenir le manque de contrôle des décisions architecturale (dérive / érosion). Cette thèse s'appuie sur une étude empirique, pour valider notre approche nommée ARIANE, en appliquant les processus de re-documentation et de versionnement à de nombreuses versions d'un projet industriel extrait de GitHub.

ARIANE: Automated Re-Documentation to Improve software Architecture uNderstanding and Evolution

All along its life-cycle, a software may be subject to numerous changes that may affect its coherence with its original documentation. Moreover, despite the general agreement that up-to-date documentation is a great help to record design decisions all along the software life-cycle, software documentation is often outdated. Architecture models are one of the major documentation pieces. Ensuring coherence between them and other models of the software (including code) during software evolution (co-evolution) is a strong asset to software quality. Additionally, understanding a software architecture is highly valuable in terms of reuse, evolution and maintenance capabilities. For that reason, re-documenting software becomes essential for easing the understanding of software architectures. However architectures are rarely available and many research works aim at automatically recovering software architectures from code. Yet, most of the existing re-documenting approaches do not perform a strict reverse-documenting process to re-document architectures "as they are implemented" and perform re-engineering by clustering code into new components. Thus, this thesis proposes a framework for re-documentating architectures as they have been designed and implemented to provide a support for analyzing architectural decisions. This re-documentation is performed from the analysis of both object-oriented code and project deployment descriptors. The re-documentation process targets the Dedal architecture language which is especially tailored for managing and driving software evolution. Another highly important aspect of software documentation relates to the way concepts are versioned. Indeed, in many approaches and actual version control systems such as GitHub, files are versioned in an agnostic manner. This way of versioning keeps track of any file history. However, no information can be provided on the nature of the new version, and especially regarding software backward-compatibility with previous versions. This thesis thus proposes a formal way to version software architectures, based on the use of the Dedal architecture description language which provides a set of formal properties. It enables to automatically analyze versions in terms of substitutability, version propagation and proposes an automatic way for incrementing version tags so that their semantics corrrespond to actual evolution impact. By proposing such a formal approach, this thesis intends to prevent software drift and erosion. This thesis also proposes an empirical study, to validate our approach named ARIANE, based on both re-documenting and versioning processes on numerous versions on an enterprise project taken from GitHub.
