A formal approach to automate the evolution management in component-based software development processes

Abderrahman Mokni

THÈSE

Pour obtenir le grade de

Docteur

Délivré par **l'Université de Montpellier**

Préparée au sein de l'école doctorale I2S (Information Structures et Systèmes)
Et de l'unité de recherche LGI2P de l'Ecole des Mines d'Alès

Spécialité : **Informatique**

Présentée par **Abderrahman MOKNI**

Une Approche Formelle pour automatiser la Gestion de l'Evolution dans les processus de Développement à Base de Composants

Soutenue le 14 décembre 2015 devant le jury composé de

M. Yves LEDRU, Professeur, Université Joseph-Fourier	Rapporteur
M. Salah SADOU, MCF HDR, Université Bretagne Sud	Rapporteur
M. Antoine BEUGNARD, Professeur, Télécom Bretagne	Examinateur
M. Philippe COLLET, Professeur, Université Nice	Examinateur
M. David DELAHAYE, Professeur, Université Montpellier	Président
Mme. Marianne HUCHARD, Professeur, Université Montpellier	Directeur de Thèse
Mme. Christelle URTADO, Maître assistant, Ecole des Mines d'Alès	Encadrant
M. Sylvain VAUTTIER, Maître assistant, Ecole des Mines d'Alès	Encadrant

Managing software evolution is a complex task. Indeed, throughout their whole lifecycle, software systems are subject to changes to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. If not evolved, software systems degrade, become obsolete or inadequate and be replaced. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully hence leading to software degradation and phase-out.

This thesis proposes an approach to improve the evolution management activity in component-based software development processes. The solution adopts a Model-Driven Engineering (MDE) approach. It is based on Dedal, an Architecture Description Language (ADL) that explicitly separates software architecture descriptions into three abstraction levels: specification, configuration and assembly. These abstraction levels respectively correspond to the three major steps of component-based development (design, implementation and deployment) and trace architectural decisions all along development. Dedal hence efficiently supports evolution management: It enables to determine the level of change, analyze its impact and plan its execution in order to prevent architecture inconsistencies (erosion, drift, etc.).

Rigorous evolution management requires the formalization, on the one hand, of intra-level relations linking components within models corresponding to different architecture abstraction levels and on the other hand, of the formalization of inter-level relations linking models describing the same architecture at different abstraction levels. These relations enable the definition of the consistency and coherence properties that prove necessary for architecture correctness analysis. The evolution process therefore consists of three steps: First, change is initiated on an architecture description at a given abstraction level; then, the consistency of the impacted description is checked out and restored by triggering additional changes; finally, the global coherence of the architecture definitions is verified and restored by propagating changes to other abstraction levels.

Relations and properties are expressed in B, a set-theoretic and first-order logic language. They are applied on B formal ADL the meta-model of which is mapped to Dedal's and helps automatic model transformations. This integration enables to implement a development environment that combines the benefits of both MDE and formal approaches: Software architecture design using Dedal tools (graphical modeler) and architecture analysis and evolution management using B tools (animator, model-checker, solver).

In particular, we propose to use a B solver to automatically calculate evolution plans according to our approach. The solver explores a set of defined evolution rules that describe the change operations that can apply on architecture definitions. It automatically searches for a sequence of operations that both changes the architecture as requested and preserves architecture consistency and coherence properties. The feasibility of the evolution management approach is demonstrated through the experimentation of three evolution scenarios, each addressing a change at different abstraction level. The experimentation relies on an implementation of a search-based software engineering approach mixing software engineering and optimization and integrates our own solver with specific heuristics that significantly improve calculation time.

Remerciements

Tout d'abord, je remercie Monsieur Yannick Vimont, directeur du Laboratoire de Génie Informatique et d'Ingénierie de Production (LGI2P) de l'École des Mines d'Alès pour m'avoir accueilli et permis de réaliser cette thèse dans les bonnes conditions.

Je voudrais par ailleurs remercier l'ensemble des membres du jury. Je remercie particulièrement Messieurs Yves Ledru et Salah Sadou pour avoir accepté de rapporter ma thèse. Je remercie également Messieurs Antoine Beugnard, David Delahaye et Philippe Collet d'avoir accepté d'examiner mon travail.

Je tiens à remercier tout particulièrement et à témoigner ma reconnaissance envers mon équipe encadrante pour leurs grandes qualités professionnelles et humaines : Madame Marianne Huchard pour avoir dirigé ce travail de thèse, Madame Christelle Urtado pour ses conseils judicieux et sa pédagogie. Elle a su toujours me motiver et me remonter le moral pendant les périodes difficiles de la thèse. Sylvain Vauttier avec qui j'ai eu beaucoup de plaisir à travailler ; ses échanges, ses discussions et ses conseils étaient très enrichissants.

Je remercie l'ensemble du personnel technique et administratif du LGI2P. Toujours disponibles et aimables. Merci aux enseignants-chercheurs, doctorants, post-doctorants et ingénieurs qui m'ont toujours bien accueilli et furent souvent d'une aide précieuse. Je tiens particulièrement à remercier mes collègues et amis : Stéphane Billaud, Sami Dalhoumi, Balzo Nastov, Pierre-Antoine Jean, Sébastien Harispe et Abdelhak Immousaten avec qui j'ai partagé beaucoup de beaux moments pendant les trois années de thèse.

Pour finir, je remercie ma famille qui m'a toujours accompagné et encouragé (malgré la distance), mes parents, pour leur éducation et leur amour, à jamais dans mon cœur.

Le 12 décembre 2015.

Contents

1 Introduction **1**
- 1.1 Context of Component-Based Software Engineering . . . 1
- 1.2 Problematic of architecture evolution in CBSD processes . . . 2
- 1.3 Thesis proposal and contributions . . . 3
- 1.4 Dissertation outline . . . 4

2 Context of component-based software engineering and software architectures **7**
- 2.1 Component-Based Software Engineering . . . 8
  - 2.1.1 Component-based software life-cycle . . . 8
  - 2.1.2 Discussion . . . 12
- 2.2 Software architectures . . . 13
  - 2.2.1 Basic concepts . . . 13
  - 2.2.2 Architecture modeling . . . 15
  - 2.2.3 Architecture evolution . . . 17
  - 2.2.4 Architecture analysis . . . 19
- 2.3 The Dedal architecture model . . . 21
  - 2.3.1 The Dedal architecture specification level . . . 22
  - 2.3.2 The Dedal architecture configuration level . . . 23
  - 2.3.3 The Dedal architecture assembly level . . . 24
- 2.4 Conclusion . . . 25

3 Software architecture evolution and formal modeling languages: A state of the art **27**
- 3.1 Study of existing evolution approaches . . . 28
  - 3.1.1 C2-SADEL . . . 29
  - 3.1.2 Dynamic Wright . . . 29
  - 3.1.3 Darwin . . . 30
  - 3.1.4 ArchWare . . . 31
  - 3.1.5 Dynamic reconfiguration with Plastik . . . 32
  - 3.1.6 Approach of Hansen *et al.* . . . 32
  - 3.1.7 SAEV . . . 33
  - 3.1.8 SAEM . . . 35
  - 3.1.9 Approach of Barnes *et al.* . . . 36
  - 3.1.10 Approach of Tibermacine *et al.* . . . 37
  - 3.1.11 Synthesis and comparison . . . 38
- 3.2 Formal modeling languages . . . 42

3.2.1 Overview of formal modeling languages . . . 43
3.2.2 Synthesis and comparison . . . 48
3.2.3 Discussion . . . 51
3.3 Conclusion . . . 52

4 A type theory for three-level software architectures 53
4.1 Overview of the three-level type theory . . . 54
4.1.1 Goals of the type theory . . . 54
4.1.2 Structure of the type theory . . . 55
4.2 Formalization of the Dedal architectural model . . . 55
4.2.1 Formalization of the common architectural concepts . . . 57
4.2.2 Formalization of Dedal architectural concepts . . . 60
4.2.3 An illustrative example: B formal specifications of the Home Automation Software . . . 62
4.3 Intra-level rules . . . 63
4.3.1 Intra-level component rules . . . 63
4.3.2 Intra-level architecture consistency . . . 71
4.4 Inter-level rules . . . 75
4.4.1 Inter-level component rules . . . 75
4.4.2 Inter-level architecture coherence . . . 78
4.5 Conclusion . . . 83

5 A formal approach to three-level software architecture evolution 85
5.1 The evolution management model . . . 86
5.1.1 The architectural model (Group 1) . . . 86
5.1.2 The architectural change (Group 2) . . . 90
5.1.3 The Evolution Manager System (Group 3) . . . 91
5.1.4 Synthesis . . . 95
5.2 Evolution management approach . . . 95
5.2.1 The overall approach . . . 96
5.2.2 The search algorithm . . . 97
5.3 Conclusion . . . 101

6 Implementation and experimentation: The DedalStudio tool suite 103
6.1 Overview of *DedalStudio* . . . 104
6.2 Implementation . . . 105
6.2.1 Architecture modeling (*DedalModeler*) . . . 105
6.2.2 Formal models generation (*FormalDedalGenerator*) . . . 108
6.2.3 Architecture analysis and evolution (*DedalManager*) . . . 110
6.2.4 Validating architecture evolution (*DedalChangeParser*) . . . 112
6.3 Experimentation and evaluation . . . 112
6.3.1 Experimentation . . . 112
6.3.2 Performance evaluation . . . 118
6.4 Conclusion . . . 121

7 Conclusion and future work directions 123
7.1 Contributions . . . 123

7.1.1 Conceptual contributions 124
7.1.2 Technical contributions 124
7.1.3 Applicative contributions 125
7.2 Limitations and future work directions 126
7.2.1 Conceptual perspectives 126
7.2.2 Technical perspectives 127
7.2.3 Applicative perspectives 128
7.2.4 Threats to validity and experimental perspectives 128

A Formal Dedal specifications (Home Automation Software Example) 129
A.1 Basic_concepts Machine 129
A.2 Arch_concepts Machine 136
A.3 Arch_specification Machine 141
A.4 Arch_configuration Machine 146
A.5 Arch_assembly Machine 153

B EvolutionManager machine 161

C Résumé en français 181

Bibliography 183

List of Figures

2.1 The waterfall process model . . . 9
2.2 Component-Based Development Process . . . 11
2.3 Component interfaces (adapted from [Sommerville, 2010]) . . . 14
2.4 Staged process model for evolution [Bennett and Rajlich, 2000] . . . 18
2.5 Reuse development process [Zhang, 2010] . . . 21
2.6 The Dedal architecture levels . . . 23

3.1 Evolution process in C2-SADEL [Oreizy and Taylor, 1998] . . . 30
3.2 Unanticipated changes management in Darwin [Kramer and Magee, 1990] . . . 31
3.3 The architecture of Plastik [Joolia et al., 2005] . . . 32
3.4 Runtime architecture model [Hansen and Ingstrup, 2010] . . . 33
3.5 SAEV meta-model [Oussalah et al., 2005] . . . 34
3.6 Architecture abstraction levels supported by SAEV [Sadou et al., 2005] . . . 34
3.7 Structure of Evolution Contracts [Tibermacine et al., 2006] . . . 37

4.1 Relations kinds in Dedal . . . 55
4.2 Modularization of Dedal formalization . . . 56
4.3 The architecture descriptions of HAS . . . 62
4.4 Example of component substitutability . . . 64
4.5 Example of name inconsistency . . . 72
4.6 Interface inconsistency . . . 73
4.7 Examples of interaction inconsistencies . . . 74
4.8 Inter-level component relations . . . 75
4.9 Example illustrating two realization cases . . . 76
4.10 Relations between architecture abstraction levels . . . 79

5.1 The evolution management model (ecore) . . . 87
5.2 EMS workflow . . . 92
5.3 Multi-level evolution management process . . . 96

6.1 The architecture of DedalStudio . . . 104
6.2 The Dedal meta-model (ecore) . . . 106
6.3 The interface of *DedalModeler* . . . 107
6.4 Configuration diagram editor . . . 107
6.5 Embedded textual editor . . . 108
6.6 The translation approach . . . 109
6.7 Example of mapping between meta-classes and B . . . 109
6.8 Evolution plan view . . . 111
6.9 HAS specification . . . 113

6.10 HAS configuration 113
6.11 HAS assembly 114
6.12 Evolving the HAS specification 114
6.13 Evolving the HAS configuration 114
6.14 Change propagation to the HAS assembly 115
6.15 Evolving the HAS configuration 116
6.16 Change propagation to the HAS specification 116
6.17 Change propagation to the HAS assembly 116
6.18 Evolving the HAS assembly 117
6.19 Change propagation to the HAS configuration 117

List of Tables

3.1 Classification of existing evolution approaches 40
3.2 Comparison of formal modeling languages 50

4.1 B specifications related to interfaces 58
4.2 B Specification of the underlying architecture concepts 59
4.3 B model of the specification level 60
4.4 B specification of the configuration level 61
4.5 B specification of the assembly level 62
4.6 An extract of the B model of the HAS at specification level 63

5.1 Summary of model manipulation operations 89
5.2 The *EvolutionManager* machine 92
5.3 The schema of an evolution rule 94
5.4 Evolution rule at configuration level 94

6.1 Test parameters for ProB 119
6.2 Test parameters for the evolution-solver algorithm 120
6.3 Performance evaluation 121

Chapter 1

Introduction

This introductory chapter briefly presents the context of this thesis, the addressed problematic, our proposal and the organization of the dissertation.

Contents

1.1 Context of Component-Based Software Engineering 1
1.2 Problematic of architecture evolution in CBSD processes 2
1.3 Thesis proposal and contributions 3
1.4 Dissertation outline 4

1.1 Context of Component-Based Software Engineering

Software engineering has greatly evolved since its early days due to the increasing complexity of software systems and the rising need to produce and maintain software cost-effectively. Component-Based Software Engineering (CBSE), a sub-discipline of software engineering, is one of the most promoted development trends that addresses these requirements. Emerged in late 1990s, CBSE promotes an approach to software development centered on effective component reuse [Sommerville, 2010]. It provides methods, models and guidelines to help and improve Component-Based Software Development (CBSD). The principle of CBSD is to build software systems by assembling pre-developed and decoupled components stored in software repositories (often referred to as *Off-The-Shelf* (OTS) components) instead of building all parts from scratch. This approach has the benefit to significantly decrease development cost and time-to-market without giving up quality [Crnkovic, 2002].

Central to CBSD are software architectures. They describe the high-level structure of the software system and expose the dimensions along which it is expected to evolve [Garlan, 2000]. An architecture model lists the constituent elements of the system and the way they

are connected together. It captures the design decisions of the software and enables to reason about its evolution. As the complexity of software systems increases, the focus of software development reorients from code to architectures [Shaw and Garlan, 1996]. Evolving software architectures avoids to delve into low-level source code which is harder to understand and modify [Georgas et al., 2006].

While CBSE has greatly evolved since its emergence, some issues such as reuse, complexity and maintenance [Crnkovic, 2003] are still challenging today. In particular, managing architecture evolution [Breivold et al., 2012] in CBSD processes is a serious issue.

1.2 Problematic of architecture evolution in CBSD processes

Managing architecture evolution in CBSD processes is non trivial. Indeed, throughout their whole lifecycle, software systems are subject to several changes to correct bugs, improve performance and quality, or be adapted to their host environment. While unavoidable, software changes may alter the system's architecture leading to several inconsistencies. If not analyzed and handled carefully, architecture inconsistencies engender the loss of evolvability of the software and lead to its degradation and phase-out. A famous problem is software architecture erosion [de Silva and Balasubramaniam, 2012, Perry and Wolf, 1992]. It arises when modifications of the software implementation violate the design principles captured by its architecture.

While a lot of work has been dedicated to architectural modeling and software evolution in general, there still is a lack of foundations and techniques to manage architecture evolution in CBSD processes and in particular to tackle architecture inconsistencies notably erosion. Indeed, existing approaches to architecture evolution hardly support the whole life-cycle of component-based software (*i.e.,* design, implementation and deployment). Changes are usually applied at architectural level and then mapped on one-way to implementation (source code) and runtime. The problem of such approaches is that they do not guarantee the traceability of design decisions since the links between design, implementation and deployment are not elucidated. This limits to determine the impact of software changes on architectural decisions across the whole development stages. Notably, dynamic changes (*i.e.,* at runtime) are not fully dealt with since changes are propagated only on one-way (from design to runtime) but not on the opposite way (from runtime to design), hence increasing the risk of erosion and its consequences (*i.e.,* loss of evolvability, degradation and phase-out).

1.3 Thesis proposal and contributions

To increase confidence in reuse-centered, component-based software systems, software architectures must support change at any step of CBSD (*i.e.,* specification, implementation and deployment). All architecture descriptions must remain consistent after changes and whatever part of the architectural description changes affect, their effects have to be propagated to the other parts.

This thesis proposes an approach to improve the evolution management activity in component-based software development processes. The approach is based on Dedal [Zhang et al., 2010, 2012], an Architecture Description Language (ADL) that explicitly separates software architecture descriptions into three abstraction levels: specification, configuration and assembly. These abstraction levels respectively correspond to three major steps of component-based development (design, implementation and deployment) and keep the traceability of architectural decisions across the whole development process.

The Dedal three architecture abstraction levels present a convenient support for managing architecture evolution in CBSD. We are confronted to the following research questions in order to establish the evolution management approach:

- What kind of relations should be made explicit between the three architecture abstraction levels to trace the design decisions all along CBSD?
- What architecture properties should be defined to avoid architecture inconsistencies and erosion in particular?
- How to enable architectural change and control its impact on the three architecture abstraction levels?
- How to enable automated architecture analysis and evolution?

This thesis answers these questions through two main contributions.

The first contribution consists in a type theory for Dedal that formally defines its semantics and the relations between the three architecture abstraction levels. On the one hand, we define intra-level architecture consistency properties to ensure the well-formedness of architecture definitions at any abstraction level. On the other hand, we define inter-level architecture coherence properties to guarantee the traceability of architectural decisions across the three architecture abstraction levels. To enable automated architecture analysis, the type theory is formalized in B [Abrial, 1996], a set-theoretic and first-order predicate language.

The second contribution consists in an evolution management model and a formal approach to deal with architectural change in CBSD processes. The approach enables to determine the level of change, analyze its impact and plan its execution in order to preserve architecture consistency and coherence properties. Evolution plans are constituted of sequences of architecture change operations calculated using a search-based strategy solver. Instead of using general-purpose solvers, we propose our own solver with specific heuristics to automatically calculate evolution plans according to our approach. This could significantly improve calculation time, especially when the complexity of architecture models increases.

Finally, we undertake an experimentation to demonstrate the feasibility of our approach. It consists in the experimentation of three evolution scenarios, each addressing a change at a different abstraction level. For that purpose, we have implemented *DedalStudio*, a CASE (Computer-Aided Software Engineering) tool that supports Dedal architecture modeling and evolution management. The tool integrates the specific solver to generate evolution plans. An evaluation of its performance is also presented.

1.4 Dissertation outline

The dissertation is organized as follows:

- Chapter 2 introduces the context of this thesis, namely the CBSE and software architecture fields and their issues. It then presents the Dedal ADL.
- Chapter 3 establishes two state-of-the-art studies related to this thesis. The first study surveys a number of existing approaches to architecture evolution and highlights their limits. The second study surveys five formal modeling languages including B and compares them in terms of expressiveness and tool support. This study argues our choice for the B modeling language to support our approach.
- Chapter 4 introduces the type theory underlying the Dedal architectural model and defines the rules to check architecture consistency and coherence between architecture definitions at all abstraction levels.
- Chapter 5 presents the evolution management model, its associated approach to automatically handle architectural change throughout the whole component-based software lifecycle and the evolution-solving algorithm enhanced with heuristics.
- Chapter 6 presents the implementation of *DedalStudio* tool suite supporting our approach, demonstrates its feasibility on three evolution scenarios and gives an evaluation of the proposed solver.

- Chapter 7 summarizes our contributions, discusses the limitations of our approach and proposes future work directions.

Chapter 2

Context of component-based software engineering and software architectures

As mentioned in the introductory chapter, this thesis contributes to the field of Component-Based Software Engineering (CBSE) and more precisely addresses the problematic of architecture evolution in reuse-intensive, component-based development processes. Thus, this chapter introduces in more details the context of CBSE and software architectures. Section 2.1 gives the principles of CBSE, compares the component-based software development (CBSD) process with traditional software development processes and agile methods. Section 2.2 introduces the concepts and definitions related to software architectures and finally Section 2.3 presents Dedal, an architecture model tailored for CBSD processes.

Contents

2.1 Component-Based Software Engineering 8
2.1.1 Component-based software life-cycle 8
2.1.2 Discussion 12
2.2 Software architectures 13
2.2.1 Basic concepts 13
2.2.2 Architecture modeling 15
2.2.3 Architecture evolution 17
2.2.4 Architecture analysis 19
2.3 The Dedal architecture model 21
2.3.1 The Dedal architecture specification level 22
2.3.2 The Dedal architecture configuration level 23
2.3.3 The Dedal architecture assembly level 24

2.4 Conclusion 25

2.1 Component-Based Software Engineering

Component-Based Software Engineering is the sub-discipline of software engineering that concerns the development of software systems making considerable use of components. It provides methods, models and guidelines for the developers of component-based systems [Pree, 1997]. CBSE emerged in the late 1990's as an approach centered on effective software reuse [Sommerville, 2010]. The principle is to produce software systems from already existing components instead of developing all parts from scratch. The main motivations of such approach is to reduce development cost and time to market, meet rapidly emerging customer demands and increase software reliability and maintainability [Szyperski, 2002]. CBSE has quickly grown up and becomes recognized as an important sub-discipline of software engineering. There are several reasons behind this fast emergence. First, software systems are becoming increasingly complex and provide more functionality. By using components, it is possible to produce more functionalities with the same investment of time and money [Pree, 1997]. Second, traditional approaches failed to support reuse. For instance, single object classes are too detailed and specific whereas components are more abstract and can be considered to be standalone service providers [Sommerville, 2010]. Third, systems need to be constantly updated and maintained to respond to new requirements. This need requires a support for easy additions. Heineman and Councill summarize the three major goals of CBSE [Heineman and Councill, 2001]:

- To support the development of components as reusable entities;
- To provide support for the development of systems as assemblies of components;
- To facilitate the maintenance and upgrading of systems by customizing and replacing their components.

Although these goals are very promising, it has been shown that achieving them in practice is very hard and the process of improving reuse has been long and laborious [Pree, 1997]. In particular, evolving component-based systems in practice is not that easy, especially after the deployment phase. A deep understanding of the component-based software life-cycle is hence required to better highlight the issues of CBSE.

2.1.1 Component-based software life-cycle

This section outlines traditional software development processes and agile methods then presents the component-based development process.

2.1.1.1 Traditional software development processes

The waterfall Process model. Most of traditional software development approaches adhere to the waterfall process model proposed by Royce in 1970 [Royce, 1987]. The principal stages of the waterfall model (*cf.* Figure 2.1) map to the following fundamental development activities [Sommerville, 2010]:

- **Requirements definition**: This step consists in identifying the system's functionalities, constraints and goals by consulting system users. A more detailed and possibly formal specification of the system is then defined.
- **System and software design**: This step consists in describing a conceptual and technical solution to realize the software. An overall system architecture is established.
- **Implementation and unit testing**: During this step, a set of program units are realized according to the software design. Unit testing ensures that each unit meets each specification.
- **Integration and system testing**: This step consists in integrating all the individual program units into a whole system, performing complete tests to ensure that system requirements are met, and delivering the product to the customer.
- **Operation and maintenance** This step is the post-delivery phase. It involves fixing bugs which were not yet discovered in earlier stages, improving the implementation of system units and including new services as new requirements might need to be considered.

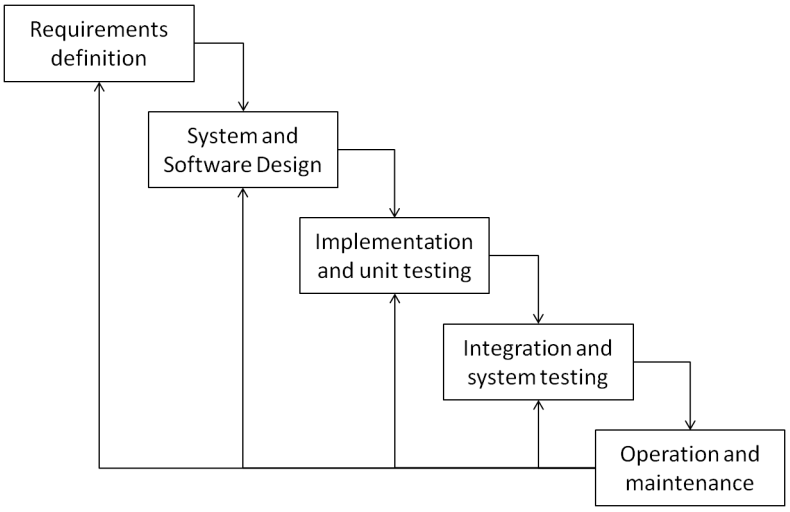


FIGURE 2.1: The waterfall process model

The waterfall model proposes a clear separation between the different stages of software development. Its main advantage is that phases do not overlap and documentation is produced at

each stage [Sommerville, 2010]. The waterfall model is more suitable for small projects where requirements are very well understood. However, such partitioning of software development makes it difficult to respond to changing customer requirements. Indeed, commitments must be made at an early stage in the process while results are produced very late [Sommerville, 2010].

2.1.1.2 Agile methods

Agile methods emerged in 1990s to cope with the disadvantages of traditional process models. They support rapid software development and are more amenable to requirement change. Agile methods were primarily designed to support the development of business applications and quickly deliver working software to customer in an iterative manner [Sommerville, 2010]. In 2001, a group of practitioners established a consensus -called the manifesto of agile software- that sets the values and principles of agile methods [1]. The manifesto advocates these four values:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Agile methods are iterative and focus on incremental development. They actively involve customer in the development process for feedback and facilitate requirements change even after software delivery [Sommerville, 2010].

The main disadvantage of such methods is that they depend too much on human personalities (team developers, customers). Developers are not always willing to support the pressure of such intense processes and customers are not always willing to spend the time necessary to give useful feedback [Sommerville, 2010]. Moreover, prioritizing changes may become a difficult task when the system involves many stakeholders since each stakeholder sees priorities differently [Sommerville, 2010]. Agile methods are more suitable for small and medium-sized systems with no associated risks than large, complex and critical systems [Sommerville, 2010].

2.1.1.3 Component-based software development process

The main idea of CBSD is building systems from pre-existing components. This entails two consequences on the software development process [Crnkovic et al., 2006] (*cf.* Figure 2.2).

[1] http://www.agilealliance.org/the-alliance/the-agile-manifesto/

First, software development (so called development by reuse) is separated from component development (so called development for reuse). Therefore, components are supposed to be already developed when the software development process starts. Second, a component identification step [Sommerville, 2010] is included in the development process. In the remainder, we detail the principal stages of CBSD.

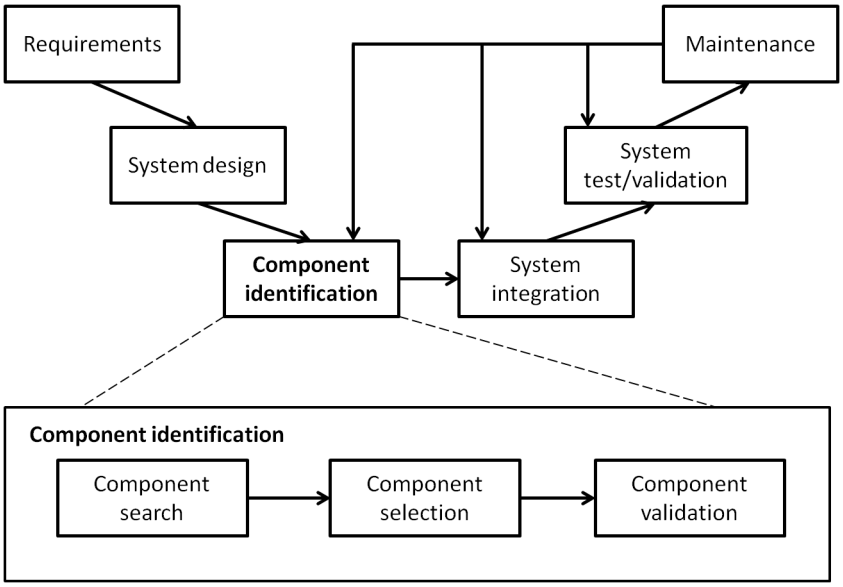


FIGURE 2.2: Component-Based Development Process

Requirements. In component-based approach, requirements can be defined as abstract component types describing the funtionalities of the system. Defining requirements in a component-based approach must consider that, if possible, resulting specification has to be fulfilled by available software components. If not, either new components have to be developed or requirements have to be negotiated and modified to use existing components [Crnkovic et al., 2006].

System design. The design phase consists in defining a complete system architecture with refined component types to be fulfilled with existing software components. Similar to the requirement phase, the design phase is strongly related to the availability of components.

Component identification. Component identification step is specific to CBSD. It replaces the implementation phase in traditional development processes. Component identification consists in three sub activities:

- *component search*: This activity consists in browsing component repositories (such as *Off-The-Shelf (OTS)* components) and identify the set of candidate components that are the most qualified to fulfill system design.

- *component selection*: This activity consists in deciding which component composition provides the best coverage of requirements [Sommerville, 2010]. Selection might be more complex than search since finding a perfect matching between components and requirements is often unrealistic.
- *component validation*: Once components are selected, the architect has to test and validate if they behave as advertised. This activity corresponds to unit testing in traditional development processes and can be omitted if the component provider is trustworthy.

System integration. This step consists in downloading component from repositories, deploying them into a component container and assembling them into an executable system architecture.

System validation. Like in traditional development processes, this step is performed to ensure that all system requirements are met.

Maintenance. In a component-based approach, maintenance is achieved by checking the availability of new components or new versions of available components. This step maps back to the component identification step to search for updates.

2.1.2 Discussion

CBSE presents a reuse-based approach to software development. While its associated development process (CBSD) provides numerous benefits compared to the traditional approaches (*e.g.,* reducing complexity and time to market, separating concerns, improving software quality, *etc.*), CBSD also presents some issues. First, reuse is not such easy since finding trusty software components that perfectly match with system requirements is not always possible. Revising and adapting requirements and/or developing new software components that fit to the system's specification is often necessary. Second, managing software evolution in CBSD processes is a complex task. Indeed changes may affect software at any step of its life-cycle (*e.g.,* new requirements, new software component versions, component failure, *etc.*). The impact of change might take high extents compromising thus the whole system, if not analyzed and handled carefully. We address this issue in chapter 5.

Crucial to Component-Based Software Engineering are software architectures. Next section introduces the definitions, concepts and activities related to software architectures.

2.2 Software architectures

This section introduces the basic concepts of software architectures as well as their related activities; architecture modeling, architecture evolution and architecture analysis.

2.2.1 Basic concepts

Software architectures are the blueprint of software system's construction and evolution [Taylor et al., 2009]. They describe the high-level structure of the software system and expose the dimensions along which it is expected to evolve [Garlan, 2000]. A software architecture captures the principal design decisions made about the system such as its structure, functional behavior, interaction and non-functional properties. Perry and Wolf [Perry and Wolf, 1992] define three types of architectural elements that may be consolidated into two major architectural concepts; components and connectors:

- Processing elements are components that process the data.
- Data elements are components that contains data to be processed.
- Connecting elements are mediators that hold connections between the different components.

In the remainder, we define and detail the concepts of components and connectors.

2.2.1.1 Components

Several definitions about software components exist in the literature. We give two widely cited definitions that say the essential about what a component is. The first definition was given by Taylor *et al.* [Taylor et al., 2009] as follows:

> "*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.*"

In other words, a component is a unit of computation that encapsulates data, provides services to process it and may require services from other components for its execution. Depending on the architecture, a component may be as simple as an operation or as complex as an entire system. The second definition is the one of Clemens Szyperski [Szyperski, 2002] who defined a software component as follows:

> "*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.*"

Roughly said, a component is a "black box" where code and implementation details are entirely hidden and data is accessed only via interfaces. Components hence adhere to the software engineering principles of encapsulation, abstraction and modularity. They are decoupled entities developed for reuse. A component comprises three parts: a set of interfaces, an implementation and a specification.

Interfaces. An interface is the communication point that manages the interaction of component with its environment, *i.e.,* the other components [Szyperski, 2002]. Components are made abstract thanks to their interfaces that hide the implementation details and ease reuse. Interfaces can be either provided or required (*cf.* Figure 2.3):

- A provided interface exposes the set of services provided by the component to other components. The communication point enables to receive service invocations from other components.
- A required interface defines the services required by the component from other components. The communication point enables to send service requests to other components.

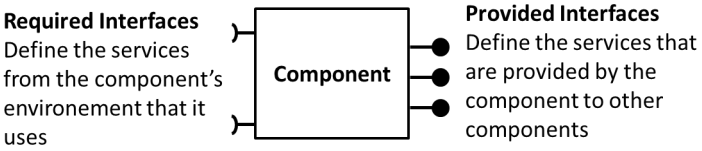


FIGURE 2.3: Component interfaces (adapted from [Sommerville, 2010])

Implementation. On the contrary, an implementation refers to the concrete and internal definition of a component (white box), *i.e.,* its source code. Once the component is developed, its implementation is hidden and decoupled thanks to interfaces. Components can then be (re)used to build architectures without any required knowledge about their inner implementation.

Specification. The component specification refers to the definition (type) of its interfaces [Crnkovic, 2002]. At the beginning, an interface specification was limited to its syntactical definition, *i.e.,* a set of operation signatures where each signature includes a name and a set of parameters. Several IDLs (Interface Definition Languages) were then proposed to specify component

interfaces. Later, Meyer introduced the notion of contract [Meyer, 1992] that extends the syntactical definition with behavior (by adding pre- and post-conditions on the interface operations). The contract concept was in turn enriched to include additional characteristics such as synchronization (the order of operations) and quality of service [Beugnard et al., 1999, 2010].

2.2.1.2 Connectors

Another fundamental aspect of software systems is managing interaction between its building blocks, *i.e.,* components. While software components implements the functionalities of the system, software connectors bind components together and act as mediators between them [Taylor et al., 2009]. This separates the computation concern handled by components from the interaction concern managed by connectors, hence strengthening reuse. Interaction may even become more challenging than selecting appropriate components. This is often the case when systems are built from large numbers of complex components distributed across multiple hosts and updated over long time periods [Taylor et al., 2009]. A software connector can provide multiple services in the architecture. Mehta *et al.* identify eight types of software connectors depending on the services they realize [Mehta et al., 2000]: procedure call, event, data access, linkage, stream, arbitrator, adapter and distributor.

Connectors can be explicitly represented as specific components with two communication points; the provided connector end and the required connector end (*e.g.,* C2SADEL [Medvidovic et al., 1999], Wright [Allen and Garlan, 1997]). Each connector end is the counterpart of the connected component interface. Connectors can also be implicitly represented by a simple link binding two opposite interfaces of two different components (*e.g.,* Darwin [Magee et al., 1995]).

2.2.2 Architecture modeling

Architecture modeling is the activity of documenting one or several aspects of a system's architecture using a particular notation. An architectural model is hence an artifact that captures some or all of the design decisions comprised in the software architecture [Taylor et al., 2009]. Architecture models are sometimes referred to as architecture descriptions. An architectural modeling notation is a language or means used to model software architectures. In the remainder, we focus on architecture modeling notations and namely, Architecture Description Languages.

2.2.2.1 Level of formalism of architectural modeling notations

Available notations range from the informal to the highly formal. The choice of a modeling architectural notation depends on the stakeholders to which models are addressed. Taylor *et al.* [Taylor et al., 2009] classify architectural modeling notations into three categories according to their level of formalism:

- Informal models: A model is said to be informal when it does not have a formally defined syntax. Usually, informal models are captured in boxes-and-lines diagrams and are intended for non-technical stakeholders such as managers and system customer.
- Semi-formal models: A semi-formal model has a formally defined syntax and is mostly used for both technical and non-technical system stakeholders. It tries to strike a balance between precision and formalism on one hand, and expressiveness and understandability on the other. A widely spread example of semi-formal languages is UML [UML, 2013]. Unlike informal boxes-and-lines notations, UML is standardized and its notation obey to a common specification.
- A formal model has formally defined semantics in addition to a formally defined syntax. Formal models are typically intended for the system's technical stakeholders. They are used to address the most critical aspects of a system and are amenable to automated analysis [Taylor et al., 2009].

2.2.2.2 Architecture Description Languages

An Architecture Description Language (ADL) is a language dedicated to architecture modeling. It provides features and specific constructs to express the basic concepts of a software architecture *i.e.,* components and their connections. The definition of an ADL has long been ambiguous. Consensus on this definition was established by Medvidovic *et al.* as they surveyed a large number of ADLs and identified their common properties [Medvidovic and Taylor, 2000]. The following definition can hence be given to an ADL [Medvidovic, 2006]:

> "*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.*"

An ADL must hence be able to express components and their interfaces, connectors (implicitly or explicitly) and configurations (*i.e.,* the topology of the architecture). First-generation ADLs

were domain-specific. Examples include C2-SADEL [Medvidovic et al., 1999] for the design of concurrent systems and Wright [Allen and Garlan, 1997] and Darwin [Magee and Kramer, 1996] for the design and analysis of distributed architectures. Later, attempts to unify ADLs and make them general-purpose were made. ACME [Garlan et al., 1997] was designed for such purpose. It consists in a common interchange description language that offers annotation facilities to support architecture descriptions in other languages. Another relevant example is xADL 2.0 [Dashofy et al., 2001]. It was designed to support various types of systems. The particularity of xADL 2.0 resides in its extensibility since it is XML-based. It offers then an easy way for architects to adapt its use to any kind of architectures.

ADLs can also be used to perform architecture analysis and support architecture evolution. Chapter 3 surveys a number of approaches that rely on ADLs to support such activities.

2.2.3 Architecture evolution

As software ages, its architecture must evolve to tackle all the changes that arise during the software life-cycle. Architecture evolution is considered as one of the most challenging tasks of component-based software engineering. To better identify the motivations and issues of architecture evolution, this section gives a quick overview of software evolution in general and architecture-centric evolution in particular.

2.2.3.1 Software evolution

Since the early years of software engineering, there was an awareness that the most costly and difficult part of software development is its maintenance phase. In a study conducted by Lientz and Swanson *et al.* in the 70's [Lientz et al., 1978], it has been proven that maintenance costs represent about 60% of the overall costs of software production. Maintenance as shown in the waterfall development process (*cf.* Figure 2.1) is the final phase of the life-cycle of a software system that comes after its deployment to fix bugs and bring some adjustments. This classical view has long governed in practice and is still widely used in industry today [Mens and Demeyer, 2008]. It also became a part of IEEE 1219 Standard for software maintenance [IEEE1219-1998, 1998] which 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.*"

This process model is too strict and not flexible enough to deal with evolution. Indeed, requirements continue to change during the entire software life-cycle. It is hence unrealistic

to assume that the requirements are all known and fixed before starting software design. Moreover, knowledge gained during the later phases may need to be fed back to the earlier phases [Mens and Demeyer, 2008]. The limitation of this process model was noticed a long time ago and a particular interest to software evolution was started by Manny Lehman when he stated the famous "*Laws of software evolution*" [Lehman, 1984]. Lehman 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 particularity of Lehman's definition is that it deals with the evolution of systems not just the evolution of code. Later research on software evolution resorted to evolving software at the architectural level rather than source code.

Several evolutionary process models were proposed to tackle the limitations of the waterfall process model. Among the most famous models is the staged process model of Bennett and Rajlich [Bennett and Rajlich, 2000] presented in Figure 2.4.

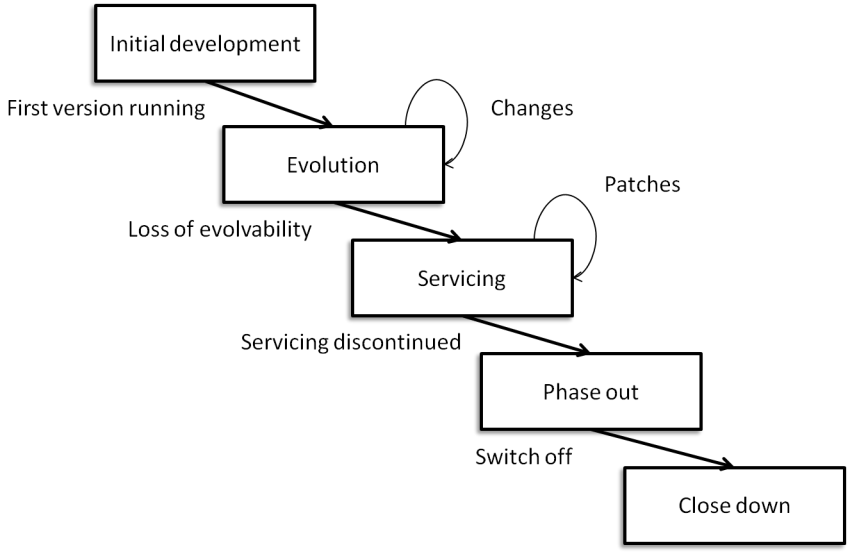


FIGURE 2.4: Staged process model for evolution [Bennett and Rajlich, 2000]

The Bennett and Rajlich model has the particularity to explicit the inevitable problem of software aging [Parnas, 1994]. After the initial development, the software is subject to several changes that lead to its degradation. When there is a loss of evolvability, servicing stage starts to keep software running up by applying small patches (minor changes) [Mens and Demeyer, 2008]. When the servicing stage becomes too hard and costly, the software enters the phase out stage and then is closed down.

2.2.3.2 Architecture-centric evolution

One of the major assets of software architecture is undoubtedly its support for evolution. As stated in the most common definitions, a software architecture is also the backbone of software evolution. It exposes the dimensions along which the system is expected to evolve [Garlan, 2000]. Reasoning about evolution at the architectural level enables a better understanding of change and allows a better estimation of modifications costs. Evolving software architectures avoids to delve into low-level source code which is harder to understand and modify [Georgas et al., 2006]. Indeed, evolution is resorted out to manipulating the constituent parts of the software and their connections as for instance replacing, deleting or adding components.

Nevertheless, architecture evolution presents several issues. Software architectures may be altered due to changes leading to several inconsistencies (or mismatches). Architecture mismatches engender loss of evolvability and lead to software degradation (*cf.* Figure 2.4). Avoiding architectural mismatches is one of the major challenges of CBSE. As argued by Garlan *et al.*, the difficulty of reuse comes essentially from architectural mismatches that arise due to several changes that affect software [Garlan et al., 1995, 2009]. Next section discusses in more details architecture mismatches and chapter 5 presents our architecture evolution model to deal with such inconsistencies.

2.2.4 Architecture analysis

Architecture analysis can be defined as the activity of discovering important system properties captured by the system's architecture models [Taylor et al., 2009]. It relies on rigorous formal models and helps the early detection of inappropriate or incorrect design decisions. Architecture analysis presents an efficient way to anticipate architecture inconsistencies that may arise during software evolution.

Architecture inconsistencies. Taylor *et al.* [Taylor et al., 2009] define consistency as an internal property intended to ensure that different elements of an architectural model do not contradict one another. They cite five examples of inconsistencies that may occur in an architectural model:

- **Name inconsistencies** concern the names of components or their constituent elements such as exported services. Inconsistency may occur when multiple elements have the same name and the wrong one is accessed or when attempting to access a non-existent element.

- **Interface inconsistencies** can result from name inconsistencies when the name of a required service does not match with another component's provided one. Interface inconsistencies concern also type mismatches between a required and a provided interface. Notably, when the parameter types of the required service or their return types do not match with the provided one according to the type system followed by the architectural model.
- **Behavioral inconsistencies** can occur between two components that satisfy name and interface consistencies but have services which behaviors do not match.
- **Interaction inconsistencies** occur when the interaction protocol between two components is violated. An example is the order of accessing their services.
- **Refinement inconsistencies** concern multiple levels of abstraction of a software system description and occur when architectural design decisions are omitted, changed or violated in a description according to a more abstract one.

Apart from the inconsistencies identified by Taylor *et al.*, there are two famous architecture mismatches highly cited in the literature which are *erosion* and *drift* introduced by Perry and Wolf in 1992 [Perry and Wolf, 1992]. These architecture problems have gained a lot of interest several years ago and are now recognized as major issues of architecture-centric evolution [de Silva and Balasubramaniam, 2012]. *Erosion* can be defined as the violation of design decisions described at a higher abstraction level by its lower abstraction level whereas *drift* can be defined as the introduction of new design decision at a lower abstraction level that are not included or implied by its higher abstraction level. Given these definitions, erosion and drift might be considered as refinement inconsistencies. Some other definitions refer to these terms in a more generalized way such as architectural degeneration [Hochstein and Lindvall, 2005] or architectural decay [Riaz et al., 2009]. De Silva *et al.* [de Silva and Balasubramaniam, 2012] proposed the following definition of architecture erosion:

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

This definition is interesting because it highlights the relation between two architecture levels, *i.e.,* the intended architecture and the implemented architecture. Taylor *et al.* use the terms prescriptive architecture and descriptive architecture [Taylor et al., 2009] to respectively denote these levels. Few work however has been dedicated to expliciting these two levels and studying their relationship. To better control erosion in CBSE processes, the intended architecture and the implemented architecture must be explicitly described. Additionally, the runtime architecture level must be taken into account since erosion can be engendered by dynamic changes.

The problem of erosion can also be seen in the opposite way, *i.e.,* the intended architecture diverges from its implemented architecture. This often happens when new requirements are included in the system's specification but some of them are not implemented. Zhang, in her thesis [Zhang, 2010], gives the name *pendency* to this problem and defines it as the introduction of new design decisions into a higher architecture level that are not implemented by its lower architecture level. Next section discusses the abstraction levels that must be elucidated in CBSD process to foster reuse and better control architecture inconsistencies that may arise during architecture-centric evolution.

2.3 The Dedal architecture model

Zhang proposes a new vision of CBSD [Zhang, 2010] that explicits the architecture descriptions produced at each development step (*cf.* Figure 2.5).

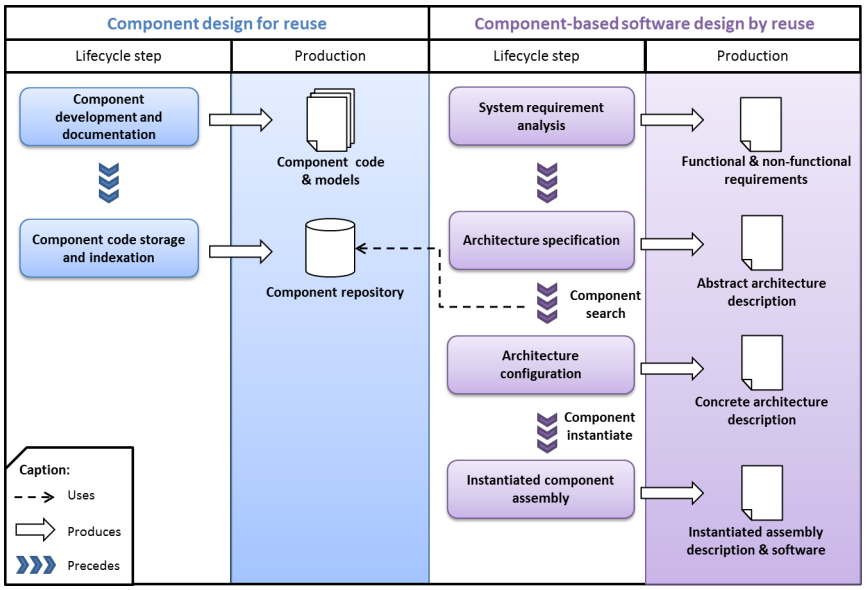


FIGURE 2.5: Reuse development process [Zhang, 2010]

The proposed process focuses on three development steps: specification (or design), implementation and deployment. After a classical requirement analysis, architects design an *architecture specification* that defines which services should be supplied by components and how components should be connected to meet requirements. The architecture specification corresponds to the architecture *as intended*. The next step consists in creating an *architecture configuration* that implements the specification. During this step architects select suitable concrete components that match those specified in the previous step and compose them to realize a complete architecture. The architecture configuration corresponds to the architecture *as implemented*. The final step consists in instantiating and deploying the architecture configuration. The

corresponding model is called the *architecture assembly* and represents the architecture *as deployed*.

Dedal [Zhang et al., 2010, 2012] is an ADL and architecture model that supports such process. It explicitly separates software architecture definitions into three abstraction levels: specification, configuration and assembly. To illustrate the concepts of Dedal, we introduce the Home Automation Software (HAS) example that manages comfort scenarios. Here, it automatically controls the building's lighting in function of the time. For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the desired scenario. In the remainder, we present the Dedal three architecture levels. Figure 2.6 is recalled to illustrate each architecture abstraction level corresponding to the HAS example.

2.3.1 The Dedal architecture specification level

The architecture specification level corresponds to the design step of the CBSD process. It defines the functional requirements of the software system and gives an abstract view of its constituent elements. The design decisions to be taken at this level consist in identifying the types of components which will be (re)used to perform the required functionalities. These abstract component types are called *component roles*.

Component roles. Specifying an architecture consists on defining component roles that represent the expected fucntionalities to be provided by available components. A component role only declares the functionalities (as a set of interface specifications). This abstract definition allows a wider set of components to match the specification and be later selected to implement the architecture. Hence, component roles are used as a guide to help the search for concrete existing component in the next step.

Figure 2.6-a shows an example of architecture specification. It is constituted of the *Home-Orchestrator* component role that manages the building's lighting using both *Light* and *Luminosity* component roles, in function of time by calling the *Time* component role.

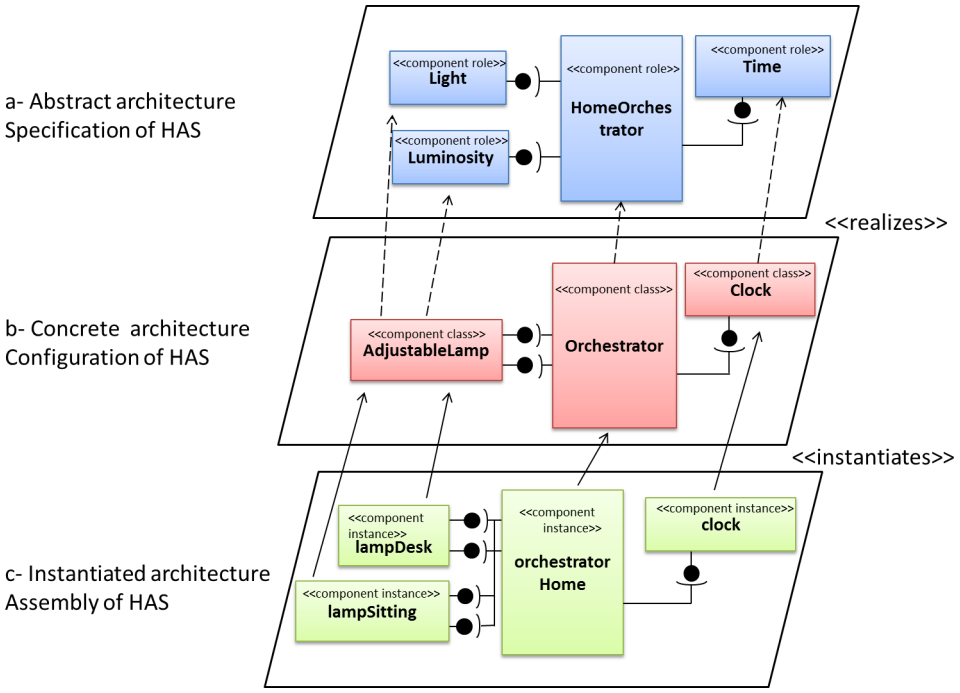


FIGURE 2.6: The Dedal architecture levels

2.3.2 The Dedal architecture configuration level

The configuration architecture level corresponds to the implementation step of the CBSD process. It describes a concrete implementation of the software system. An architecture configuration is defined by the set of components selected during the identification process and that best match with the abstract component types (*i.e., component roles*) defined in the architecture specification. These artifacts are called *component classes* and their associated types are called *concrete component types*.

Component classes. Component classes correspond to existing software components stored in repositories (such as OTS components). In Dedal, component classes can either be primitive or composite. A primitive component class encapsulates executable code. A composite component class encapsulates an inner architecture configuration (*i.e.,* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to the unconnected interfaces of its inner components. Component classes can also have attributes that enable parametrization.

Concrete component types. A component type gives an abstract representation of a set of component classes. It defines the set of interfaces that a class must hold to be an implementation of this type. Component types are used to classify component classes and

build indexes on the content of component repositories. To search for component classes that can be used to implement an architecture specification, component roles are matched with component types (using classification based on specialization and substitution in a manner similar to Arévalo *et al.* [Aboud et al., 2009, Arévalo et al., 2008]).

Figure 2.6-b shows an example of architecture configuration. It represents an implementation of HAS conforming to the architecture specification shown in Figure 2.6-a. The *HomeOrchestrator* and *Time* component roles are respectively realized by the *Orchestrator* and *Clock* component classes whereas the *Light* and *Luminosity* coomponent roles are both realized by the *AdjustableLamp* component class. The relations between component classes and component roles are studied in depth in Chapter 4.

2.3.3 The Dedal architecture assembly level

The architecture assembly level corresponds to the deployment step of the CBSD process. It consists in an assembly of instantiated concrete component selected during the implementation step. These artifacts are called *component instances*. The architecture assembly describes software at runtime and holds information about its internal state. It lists the instances of the component classes that compose the deployed architecture at runtime and their assembly constraints (such as the maximum number of connected instances).

Component instances. Component instances document how the component classes from an architecture configuration are instantiated in the deployed software. Each component instance has an initial and a current state defined by a list of valued attributes.

Figure 2.6-c shows an example of architecture assembly. It represents a possible instantiation of HAS. This assembly enables to manage the lighting in two rooms, desk and sitting room, thanks to the two instances of the *AdjusatbleLamp* component class: *lampDesk* and *lampSitting*.

Discussion. Dedal introduces three architecture abstraction descriptions that document the three majors steps of CBSD process. It helps hence a better understanding of the component-based development activity and provides more flexibility to software design by reuse. It provides an XML-based textual syntax (the ADL) that enables to specify software architectures in three separate architecture descriptions (*i.e.,* specification, configuration and assembly). In this work, we focus on studying the relations between the Dedal architecture levels (*e.g.,* the *realizes* relation between component roles and component classes) and formally define their semantics to enable architecture analysis and evolution. Therefore, the Dedal ADL syntax is out of the scope of this work and the interested reader may find it in Zhang's thesis [Zhang, 2010].

2.4 Conclusion

This chapter introduced the context of Component-Based Software Engineering and Software Architectures and highlighted the issues of both fields. We draw attention to two main issues. First, reuse is still difficult because fulfilling requirements with existing software components often needs revision and requirement adaptation. Second, managing architecture evolution in CBSD processes is a complex task because of the several inconsistencies that may alter the architecture (*e.g.,* interface inconsistencies, interaction inconsistencies, erosion, etc.). Changes may intervene at any step of CBSD, *i.e.,* requirements, implementation or deployment. Therefore, determining and handling the impact of change on the traceability of architecture decisions is also challenging.

This chapter also presented Dedal, a novel architecture model tailored for CBSD processes. The particularity of Dedal is that it explicitly separates architecture definitions into three abstraction levels (*i.e.,* specification, configuration and assembly) that correspond to there major steps of CBSD process (*i.e.,* design, implementation and deployment). Therefore, we opt for Dedal to support our approach to architecture evolution management.

Chapter 3

Software architecture evolution and formal modeling languages: A state of the art

The previous chapter introduced the context of component-based software development and the concepts related to software architectures as they present the blueprint of software system construction and evolution. This chapter addresses the issues of managing software architecture evolution. For that, we conduct two studies. In the first study (Section 3.1), we survey a number of existing approaches based on architecture-centric evolution and classify them according to a number of criteria we consider crucial to handle evolution in reuse-centered, component-based processes. This study aims to highlight the advantages and limits of existing approaches and to help us set the basis for an alternative approach. In the second study (Section 3.2), we survey a number of formal modeling languages and compare them in terms of expressiveness and tool support. This study aims to evaluate the suitability of formal modeling languages to support architecture evolution analysis. This chapter concludes on the choices considered to address the issues of component-based architecture evolution management.

Contents

3.1 Study of existing evolution approaches 28

3.1.1 C2-SADEL 29

3.1.2 Dynamic Wright 29

3.1.3 Darwin 30

3.1.4 ArchWare 31

3.1.5 Dynamic reconfiguration with Plastik 32

3.1.6 Approach of Hansen *et al.* 32

3.1.7 SAEV 33

3.1.8 SAEM . . . 35
3.1.9 Approach of Barnes *et al.* . . . 36
3.1.10 Approach of Tibermacine *et al.* . . . 37
3.1.11 Synthesis and comparison . . . 38
3.2 Formal modeling languages . . . 42
3.2.1 Overview of formal modeling languages . . . 43
3.2.2 Synthesis and comparison . . . 48
3.2.3 Discussion . . . 51
3.3 Conclusion . . . 52

3.1 Study of existing evolution approaches

In this state-of-the-art, we survey a number of approaches addressing the evolution of software architectures. Since the literature about architecture evolution is very abundant, the survey covers the most relevant approaches proposed during the two last decades and that are related to this thesis. In this survey, we try to answer the following questions in order to evaluate existing approaches:

- Is the approach general-purpose or domain-specific?
- What does the approach use as a support to model architectures and manage software evolution?
- What architecture abstraction levels does the approach take into account? And does the approach support multi-level evolution (*i.e.,* top-down evolution and bottom-up evolution).
- What kinds of change operations are possible with the approach? Is there any mechanism to specialize and substitute components?
- What kind of analysis does the approach support? And what kind of formalism is used to perform architecture analysis?
- Is there any tool support for the proposed approach?

In the remainder, we present existing evolution approaches and in Section 3.1.11 we give their comparison based on the previous questions.

3.1.1 C2-SADEL

Medvidovic *et al.* [Medvidovic et al., 1999] propose an approach to evolve software architectures based on the C2-style [Taylor et al., 1995]. C2 is a component and message-based architecture style designed to specify GUI (Graphical User Interface) and distributed applications. The approach relies on C2-SADEL (Software Architecture Description and Evolution Language), an ADL for modeling architectures in the C2-style provided with subtyping mechanisms to enable architecture evolution. C2-SADEL separates between component types and component instances. Architectures hence can be explicitly modeled at two abstraction levels: a description that contains component types and their connections and another description that contains instances of component types and connectors. However, there is no clear distinction however between abstract component types and concrete ones. C2-SADEL also provides a sub language for describing changes called AML (Architecture Modification Language). AML allows five change operations: addition (*addComponent*), deletion (*removeComponent*), connection (*weld*), disconnection (*unweld*) and replacement (*upgrade*).

Component subtyping is in the heart of the evolution approach proposed by Medvidovic *et al.*. The subtyping mechanism is built upon a type theory [Medvidovic et al., 1998] inspired from Object-Oriented subtyping rules. The theory proposes multiple rules for subtyping components. Depending on the architect's goal, it is possible to specialize one or multiple aspects of components (*i.e.,* name, interfaces, behavior or implementation). The type theory also enables architecture analysis like type-checking and checking interoperability between two components. The evolution process depicted in Figure 3.1 enables evolution on one direction. Changes are firstly applied on architectural models. Then, they are reified to synchronize the implementation with architectural models and finally implemented. While the evolution process is complete, it only supports forward evolution. Dynamic changes cannot be captured and propagated to architecture models through a reverse evolution process. Moreover, the approach applies only to C2-style architectures that imposes strict communication rules between components and requires a specific expertise from the architect (*e.g.,* event-based and asynchronous programming, concurrency handling) [Oreizy et al., 1998].

3.1.2 Dynamic Wright

Dynamic Wright [Allen et al., 1998] is an extension of the Wright [Allen and Garlan, 1997] ADL that supports architecture evolution. Wright is basically designed to model and analyze the dynamic behavior of distributed architectures. It provides only one representation to model architectures. An architecture description in Wright (called *Style*) lists component and connector types as well as their instances. Reuse is not favored since component types are not defined as independent entities. Wright supports architecture analysis thanks to CSP

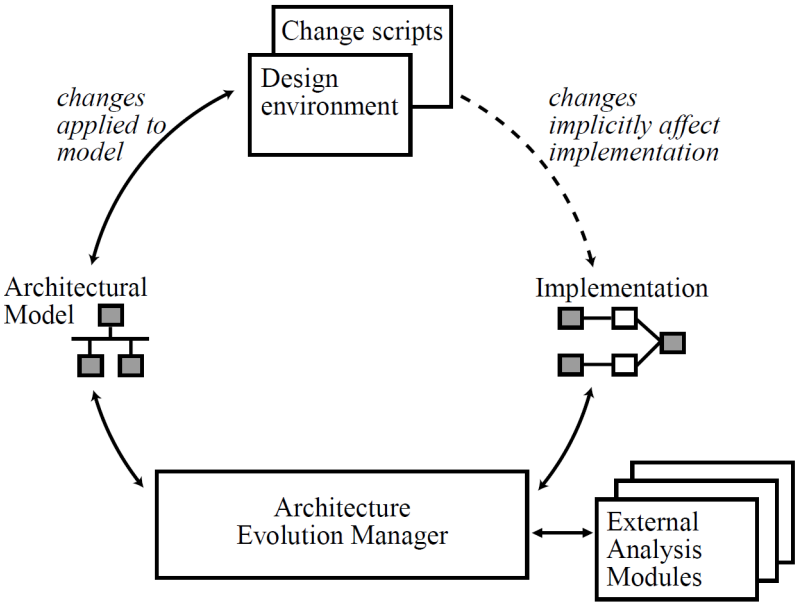


FIGURE 3.1: Evolution process in C2-SADEL [Oreizy and Taylor, 1998]

(Communicating Sequence Processes) [Hoare, 1978], its formal ground. The evolution in Dynamic Wright is managed by *control events*. A *control event* consists in a condition under which the dynamic change is allowed. Control events are grouped into a *configuration program* that specifies how reconfiguration actions (*new*, *del*, *attach* and *detach*) will be triggered. The approach based on Dynamic Wright addresses only one level of change (runtime). Moreover, the ADL is domain-specific (distributed architectures) and is based on the CSP notation which makes its use limited to experienced users.

3.1.3 Darwin

Darwin [Magee et al., 1995] is an ADL designed for the specification of the structure of distributed systems. It provides a hierarchical decomposition scheme to define components. Hence, an architecture description in Darwin consists in a root composite component including a set of component instances and their connections. Basic semantics in Darwin are defined with π-calculus [Milner et al., 1992]. They consist in services (*provide* and *require*), bindings (*bind*) and primitive components defined as compositions of services. Darwin hence enables analysis and guarantees the correctness of connections.

Darwin provides two mechanisms to deal with anticipated dynamic architecture evolution [Magee and Kramer, 1996]: lazy instantiation and direct dynamic instantiation. Lazy instantiation consists in instantiating a component only when one of its services is invoked. However, involved components and the way they are attached must have been specified at design-time. Direct dynamic instantiation allows dynamic structures to evolve in arbitrary ways. Darwin also deals with unanticipated evolution. This process is controlled through a reconfiguration

manager that makes decisions regarding changes. Required changes are specified in a script using a change language providing addition (*create*), deletion (*remove*) connection (*link*) and disconnection (*unlink*) directives [Kramer and Magee, 1990]. When a change affects one or several components, all impacted entities (either directly or through impact propagation) are temporarily disabled (passive state) until change is processed (*cf.* Figure 3.2). Darwin provides an approach to capture and control the dynamic change and its impact. However, it only covers a single abstraction level (the configuration) and does not support change propagation from runtime to the design level.

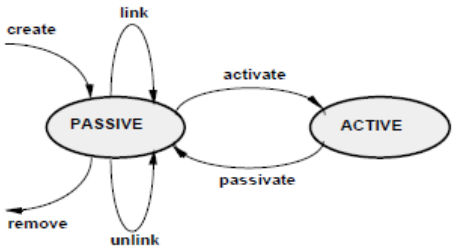


FIGURE 3.2: Unanticipated changes management in Darwin [Kramer and Magee, 1990]

3.1.4 ArchWare

ArchWare is an European project [Oquendo et al., 2004] that provides a set of architecture-centric languages and tools for the engineering of evolvable software systems. π-ADL [Oquendo, 2004a] is the language proposed to model and evolve static, dynamic and mobile architectures. It is founded on the higher order logic π-calculus [Milner et al., 1992] – from hence its name. Component and connectors can be explicitly modeled as independent entities. The architecture structure is composed of instances of these entities and their links. Only the assembly level is hence covered by π-ADL. ArchWare also includes π-ARL [Oquendo, 2004b], (Architecture Refinement Language) that enables the stepwise refinement of architectures expressed in π-ADL. The refinement in π-ARL enables a forward evolution of the original architecture description to a more concrete one. The language provides refinement actions to add and delete architectural elements to/from the original architecture. Dynamic evolution in Archware is managed through two dedicated components: the "choreographer" and the "evolver". The "choreographer" is in charge of applying user-requested changes like the creation/deletion of new component or connector instances (*new*/*del*) and attachment/detachment (attach/detach). The "evolver" manages unanticipated changes using a virtual machine in a manner similar to Darwin. This component is called whenever an evolution must be done. The "evolver" responds to the change by loading the necessary elements. The latter should have been previously described in a file that will be loaded by the virtual machine. Like Darwin, Archware covers both static and dynamic evolution and enables verification and architecture analysis since it is founded on π-calculus. However, there is neither a clear distinction between

the abstraction levels related to the component-based development process nor a support for reverse evolution.

3.1.5 Dynamic reconfiguration with Plastik

Plastik [Joolia et al., 2005] is an approach that addresses dynamic architecture reconfiguration at both design-time and runtime (*cf.* Figure 3.3). It explicitly separates the ADL level from the runtime level. The ADL level consists of a style description where component types, connector types and constraints (invariant) over them are expressed and, a system description that lists the instances of component and connector types as well as their attachments. The runtime level consists of the runtime engine that supports the execution of the system and manages its reconfiguration.

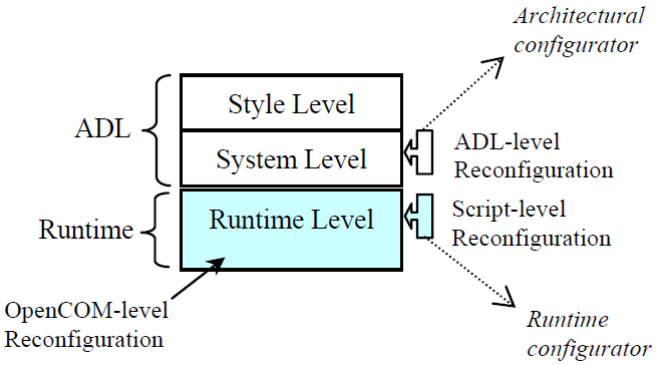


FIGURE 3.3: The architecture of Plastik [Joolia et al., 2005]

The ADL level descriptions are expressed using an extension of ACME [Garlan et al., 1997], a general-purpose and extensible ADL providing common architectural constructs (*i.e.,* components, ports, connectors, roles, attachments and representations). The extension (called Armani [Monroe et al., 1998]) is based on first-order predicate logic to support analysis (*e.g.,* topology constraints) and provides specific constructs to support programmed (anticipated) and ad-hoc (unanticipated) changes. The mapping of the ADL level to the runtime level is realized using the OpenCOM component model [Coulson et al., 2004] that supports runtime reconfiguration. Like all other surveyed approaches, Plastik supports only ADL-to-runtime evolution and the used ADL, ACME/Armani, covers only a single architecture level that corresponds to the configuration architecture.

3.1.6 Approach of Hansen *et al.*

Hansen and Ingstrup propose a way to model and analyze runtime architectural change [Hansen and Ingstrup, 2010, Ingstrup and Hansen, 2009]. The approach is based on a runtime architecture model (*cf.* Figure 3.4) that closely maps to the OSGi [OSGi, 2015] platform to facilitate

implementation and on Alloy [Jackson, 2002] as a relational first-order logic modeling language to formalize the static and dynamic concepts (operations) of the architecture model.

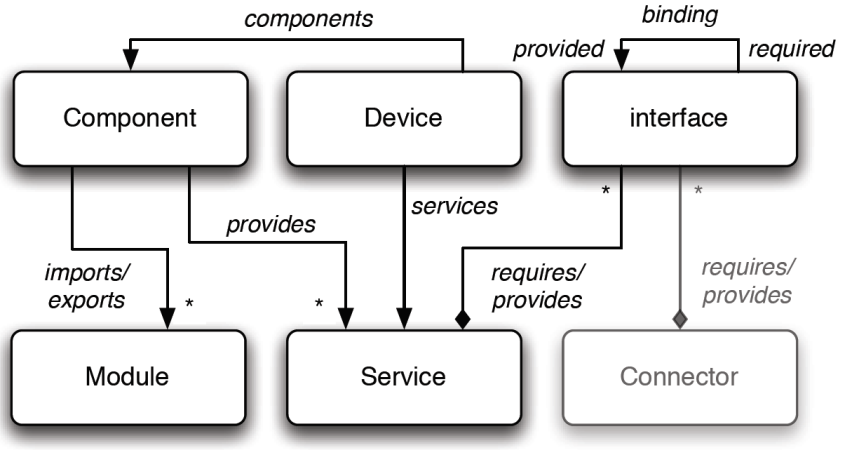


FIGURE 3.4: Runtime architecture model [Hansen and Ingstrup, 2010]

The choice of Alloy is motivated by its support for object-oriented modeling and its accompanying analyzer that enables automated verification. The objective is to ensure a reliable way to apply architectural changes without violating some predefined properties. For this purpose, the authors model the reconfiguration planning as a predicate satisfaction problem with pre- and post-conditions. Then, they run the Alloy analyzer to find sequences of the model instances satisfying the problem where the first instance satisfies the pre-conditions and the last instance satisfies the post-conditions. This work focuses only on one level of change which is runtime. Moreover, the adopted architecture model is dependent on OSGi implementation.

3.1.7 SAEV

SAEV (Software Architecture EVolution Model) [Oussalah et al., 2005, Sadou et al., 2005] is a generic model that enables the specification and management of software architecture evolution. The model, as shown in Figure 3.5, associates a set of evolution strategies to each architectural element.

An evolution strategy consists of a set of evolution rules applicable to the architectural model with respect to its invariants. Evolution rules are specified using the ECA formalism (Event-Condition-Action) where the event represents the invocation message, the condition represents the constraints to be satisfied to execute the rule and the action corresponds to an elementary evolution operation (*e.g.,* addition, modification, deletion). The whole process is managed by the evolution manager. SAEV is intended to be generic and independent from any ADL. For this purpose, the model is based on three abstraction levels *meta-level*, *architectural level* and *application level* related by an instantiation relation as illustrated in Figure 3.6.

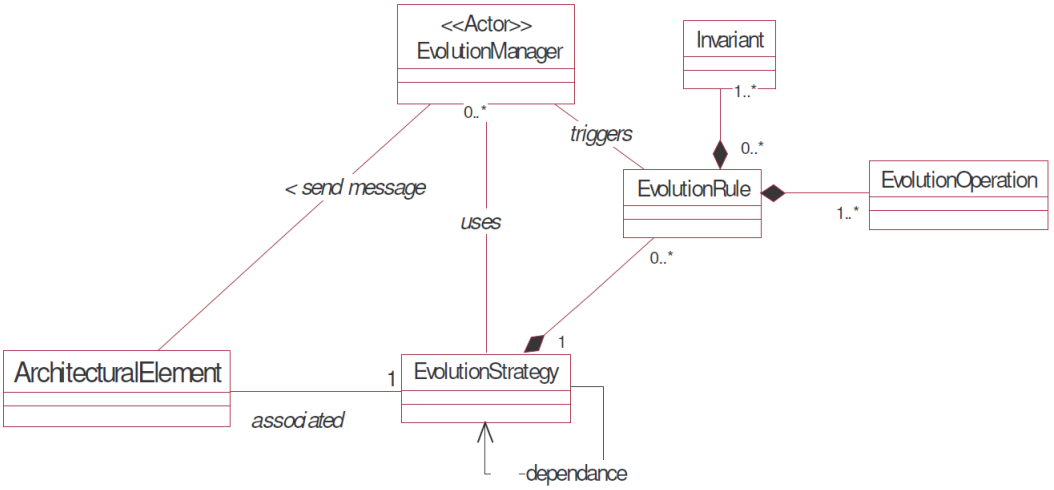


FIGURE 3.5: SAEV meta-model [Oussalah et al., 2005]

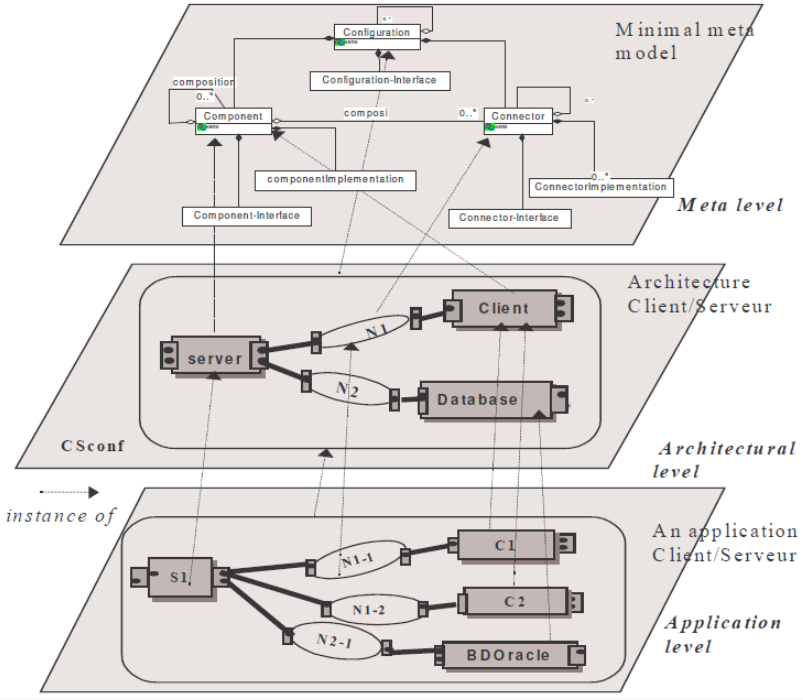


FIGURE 3.6: Architecture abstraction levels supported by SAEV [Sadou et al., 2005]

The *meta-level* gathers the generic architectural concepts (*i.e.,* components, connectors, configurations) to be used/extended by any ADL to model architectures. The *architectural level* represents the architecture descriptions expressed by instantiating the concepts of the meta-level. Finally, the application level represents the assemblies that instantiate the descriptions of the architectural level. This way, SAEV manages static evolution at both architectural and application levels by applying the same mechanisms. The evolution process includes a specification phase and an execution phase. The specification phase consists in defining the evolution strategies and the invariants to be preserved by the architectural elements targeted

by the change. The execution phase starts when the user selects the element to evolve and the associated change operation. The evolution manager intercepts the event and searches for the adequate evolution strategy to use and then selects the associated evolution rules. Rules are then triggered by forward chaining to handle the impact of change. The evolution is validated or canceled by the user after the verification of invariants. In the second case, the user has to choose another strategy.

SAEV proposes a generic approach to deal with architecture evolution. Unlike the other approaches, it addresses two abstraction levels (*i.e.,* the architectural and application levels) that correspond to configuration and assembly levels according to Dedal. The approach however does not support dynamic change and still lacks tool support and formal concepts and mechanisms to automate the evolution process. Moreover, change is propagated only in one direction since the evolution of each abstraction level is handled by its top level.

3.1.8 SAEM

SAEM (Style-based Architecture Evolution Model) [Goaer et al., 2008, Tamzalit et al., 2006] is a model that addresses the specification and evolution of domain-specific architectures. It adopts the same architecture levels as SAEV (*i.e.,* meta-level, architectural level and application level) to be generic and independent from any ADL. Unlike the other approaches, SAEM considers that architectural evolutions must be treated as first-class entities that can be specified independently and classified for reuse by a particular family of systems. For that, it introduces the notion of *evolution style* that refers to such entities. SAEM defines the vocabulary of evolution styles using UML 2.0 [UML, 2013] (notably OCL to specify the semantics). It proposes four relations to manipulate them: instantiation, specialization, composition and use. Evolution styles can hence be instantiated several times by an architecture (at the application level), specialized to more refined/specific styles, composed to form complex styles and used directly by other styles. The evolution process in SAEM contains four phases: invocation of the evolution by the user or by another evolution style, search for the required evolution style (user may be requested if several possibilities are presented), execution and validation. An evolution invocation implies its instantiation. Change is propagated during execution in several ways thanks to composition, specialization and use relations. Validation ensures that invariants related to an architectural element are respected like in SAEV. SAEM focuses more on specifying and reusing evolution styles rather than o the way evolution is managed. It addresses only the application level (which corresponds to Dedal's assembly level) even if evolution is specified and managed at the top level (architectural level). Hence, changes are captured at only one level (the application) and only top-down propagation is considered. Moreover, like SAEV, evolution management lacks tool support and rigorous mechanisms.

3.1.9 Approach of Barnes *et al.*

Barnes *et al.* [Barnes et al., 2014] propose an approach to specify and plan architecture evolution for specific classes of systems. The approach is centered on the concepts of *evolution styles* and *evolution paths*. An *evolution path* represents an evolution trace leading from an initial architecture to a desired target architecture. The notion of evolution style is wider than the one introduced by SAEM. An evolution style defines the set of *path constraints* (*e.g.,* invariants, ordering) that identify permissible evolution paths and provides adequate operators to apply transformations (that represent transitions in an evolution path) as well as a set of evaluation functions used to compare evolution paths according to quality metrics. Thereby, an evolution style denotes a family of evolution paths sharing common properties and satisfying a common set of constraints. Evolution operators are specified using combinations of elementary change operations (*e.g.,* component addition, attachment, replacement, *etc.*).

The approach is intended to be language-neutral and formalism-independent. Indeed, there is no proposed formalism to support architecture modeling. It is presumed that architectures are modeled using an ADL or UML (with OCL for defining constraints and QVT [OMG, 2011] for transformations). Specifying constraints and operators depends on the chosen formalism. While this language in-dependency may be advantageous, it is not guaranteed that the chosen formalism will be able to cover all the features of the approach. Incidentally, Barnes *et al.* conducted two case studies, one using ACME [Garlan and Schmerl, 2009] and one using UML [Barnes et al., 2014], which cover only some features of the approach. The approach also admits multiple architecture views such as those proposed by Clements *et al.* [Garlan et al., 2010] (module view, component and connector view and allocation view) to capture properties related to different facets of the software architecture. However, there is no explicit distinction between the three main steps of the software lifecycle nor a way to capture and control the impact of change since evolution is fully planned and analyzed beforehand.

Technically, the approach is still not mature. Path constraints can be formally specified using the *path constraint language*, a specific extension of LTL (Linear Temporal Logic) proposed by Barnes *et al.*. While the computability of the language was proved, there is still no existing model checker to support the automated analysis of path constraints. Moreover, the practicality of such a language is not guaranteed since the language has not been widely used yet. Barnes *et al.* also propose a solution to automate evolution planning [Barnes et al., 2013]. This solution relies on PDDL [McDermott, 1998] –the Planning Domain Definition Language– to specify the evolution problem. Since there is no proposed translation from the ADL level to PDDL, this solution requires an expertise from the architect that he does not necessary have.

3.1.10 Approach of Tibermacine *et al.*

Tibermacine *et al.* [Tibermacine et al., 2005, 2006] propose an approach to assist the architecture evolution activity at component-based software design and implementation steps. The approach aims to preserve architecture decisions related to quality attributes that may be weakened during software evolution thus leading to its degradation. To do so, the authors introduced the concept of *evolution contract* described by the meta-model shown in Figure 3.7.

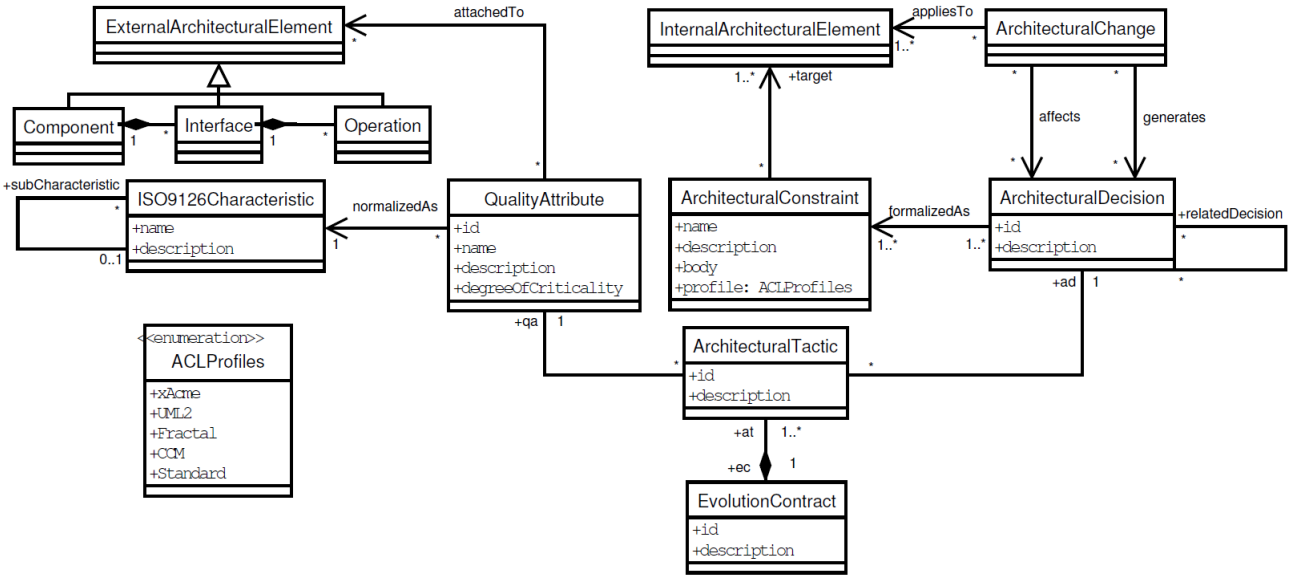


FIGURE 3.7: Structure of Evolution Contracts [Tibermacine et al., 2006]

An *evolution contract* encompasses a set of *architectural tactics* that link architectural decisions to quality attributes (for instance those defined by the ISO/IEC 9126 [ISO/IEC, 2001]). The interest of such linkage is twofold. First, it enables to warn the developer about the impact of architectural changes (that directly affect architectural decisions) on quality attributes (*e.g.,* potential loss, deterioration). Second, it helps architecture comprehension by highlighting the motivation of some architectural decisions.

Technically, the structure of evolution contracts is specified using XML [1] representations that capture architectural decisions and their rationale. To document and check architectural decisions, the authors proposed ACL (Architecture Constraint Language) that enables to express structural constraints on architecture models at both design and implementation levels. The language is based on CCL (Core Constraint Language) a slightly modified version of OCL and a set of MOF [2] meta-models. On the one hand, CCL enables to express constraints on a particular instance of the UML component meta-model (which is not possible using OCL). Additionally, it enables to constrain evolution by defining conditions relating two versions of architectural artifacts (the old version is designated using the *@old* operator). On the other hand, the

[1] http://www.w3.org/XML/
[2] http://www.omg.org/mof/

MOF meta-models represent architectural abstractions related to ADLs (for the design step) and component technologies (for the implementation step). The interest of meta-models is to make ACL as generic as possible by allowing developers to use their preferred ADL and/or component technology. CCL combined with one of these meta-models form an ACL profile. The approach also guarantees the traceability of architecture decisions throughout the design and implementation steps of CBSD. For this purpose, the authors defined a pivot meta-model called ArchMM that captures the generic architectural abstractions and make it possible to automatically translate ACL constraints from one profile to another. The approach is tooled with AURES (ArchitectURe Evolution aSsistant) with which evolution contracts can be specified and evaluated. Given a version of an architecture description (using one of the supported ADLs, for instance xADL [Dashofy et al., 2001]), AURES evaluates its associated evolution contract and notifies the maintainer if any architectural decisions do not hold anymore. The user hence has to either reconfigure the architecture again or modify the evolution contract.

Unlike other approaches, the work of Tibermacine *et al.* highlights the importance of maintaining architectural decisions while evolving component-based software architectures. It covers both design (specification level) and implementation (configuration level) steps of CBSD processes and traces architectural decisions at both steps. Runtime changes however are not taken into account since the assembly level is not explicitly defined. Moreover, the accompanying tool is only a notifier of the violation of evolution contracts but it does not yet include any mechanism to handle the changes initiated by the architect.

3.1.11 Synthesis and comparison

Most of the approaches dealing with architecture evolution adopt an ADL to model architectures and propose a mapping between the ADL and a runtime framework in order to implement the change and enable dynamic evolution. C2-SADEL, Darwin, ArchWare, Plastik fall into this category. C2-SADEL models architectures in the C2 style and provides multiple component subtyping mechanisms to favor reuse and enable architecture evolution. Its tool support is Dradel, an environment that enables the mapping between architectural description and the implementation by translating them into Java code. The tool supports static evolution by applying changes on architectural descriptions first and then implementing them. The architecture analysis however is limited since no powerful analysis techniques were integrated. Darwin and ArchWare (which provides π-ADL as an ADL) focus on modeling dynamic structures. They both rely on π-calculus to define the semantics of architecture constructs and guarantee a reliable interaction between components. They compile architecture descriptions into code. ArchWare also proposes π-ARL an architecture refinement language to evolve architecture descriptions by stepwise refinement. Plastik was also proposed to deal with dynamic reconfigurations. It relies on Armani, an extension of the ACME ADL to express invariants and

reconfigurations properties. Compared to the previous approaches, Plastik has the advantage to map its ADL to OpenCOM, a runtime component model dedicated to component-based programming and proposing built-in reconfiguration operations. The main shortcoming of these approaches is that they do not consider changes as first-class elements and focus more on how to implement architecture evolution rather than specify, analyze and propagate it. Moreover, adopted ADLs hardly cover the entire CBSD process. The specification level (necessary to guide reuse) and assembly level (that describes the software at runtime) are often missing. To some extent, C2-SADEL is the only ADL that favors the reuse of OTS components. It provides the "*upgrade*" function to replace components by their new versions. C2-SADEL however is domain-specific and like all the other approaches, it does not guarantee the coherence between architectural descriptions and implementations since evolution is only processed top-down.

The second category of evolution approaches showed a particular interest to specifying architecture evolution as first-class entities, independently from any ADL. First insights were given by SAEV that enables to specify evolution rules or use predefined ones to handle architecture evolution at both configuration and assembly levels (so called architectural and application levels). SAEM then introduced the notion of *evolution styles*, first-class entities that can be specified and classified for reuse to evolve a particular family of systems. Evolution styles include evolution operations that can be specialized, composed and instantiated to deal with change. SAEM adopts the same architecture abstraction levels as SAEV and both support only top-down evolution ignoring the specification level necessary for reuse-based processes. Moreover, evolution analysis is limited to checking invariant related to architectural artifacts. Barnes *et al.* adopt a wider definition of evolution styles than the one of Tamazalit, Le Goaer and others and introduce the concept of evolution paths as a way to analyze and plan the evolution of domain-specific software systems. Like the other approaches, it does not cover the whole component-based development process (mainly the specification level needed to foster and guide reuse is ignored). The approach of Tibermacine *et al.* shows a particular interest to analyzing the impact of architecture evolution at specification and configuration levels using evolution contracts. While it is able to detect inconsistencies between the two architecture levels, the approach lacks support to manage architecture evolution in general.

Table 3.1 classifies existing approaches in terms of architecture activities (*i.e.,* modeling, analysis and evolution) and tool support, and answers the questions asked in the beginning of this section 3.1.

Approach	Architecture modeling					Architecture analysis		Architecture evolution			Tool support
	Domain	ADL	CBSD support			Formalism	Analysis	Change operations	Substitutability mechanism	Propagation	
			specification	implementation	deployment						
C2-SAEDL	C2-style architectures	C2-SADEL	✓	✓	-	Z (first-order-logic)	type-checking, consistency-checking, C2-topology checking	predefined with AML (addComponent, removeComponent, weld, unweld, upgrade)	multiple subtyping	top-down	Dradel
Dynamic Wright	distributed applications	Dynamic Wright	-	✓	-	CSP	behavior and interaction	predefined (new, del, attach, detach)	-	-	-
Darwin	distributed applications	Darwin	-	✓	✓	π-calculus	binding, interaction	predefined (create, remove, link, unlink)	-	top-down	Regis
ArchWare	static, dynamic and mobile architectures	π-ADL, π-ARL	-	✓	✓	π-calculus	type-checking, interaction checking	predefined (new, del, attach, detach)	-	top-down	ArchWare toolset
Plastik	general-purpose	ACME/ Armani	-	✓	-	first-order logic	invariants	predefined	-	top-down	OpenCOM
Hansen *et al.*	OSGi architectures	OSGi model	-	-	✓	Alloy (first-order logic)	invariants, predefined properties	specified by the architect using Alloy	-	-	-
SAEV	general-purpose	ADL-independent	-	✓	✓	ECA, first-order logic	invariants	specified by the architect using ECA rules	-	top-down	-
SAEM	style-based architectures	ADL-independent	-	✓	✓	UML, OCL	invariants	specified by the architect for reuse (using UML/OCL)	-	top-down	-
Barnes *et al.*	style-based architectures	ADL-independent	-	✓	✓	path constraint language (LTL)	invariants, paths	specified by the architect using QVT for instance	-	-	-
Tibermacine *et al.*	general-purpose	ADL-independent	✓	✓	-	Architecture Constraint Language (ACL)	architectural decisions	-	-	-	AURES

TABLE 3.1: Classification of existing evolution approaches

Discussion. With respect to the state of the art, our study shows that existing approaches to architecture evolution lack support for component-based software development. We highlight the limits of the surveyed approaches as well as our requirements according to the classification shown in Table 3.1.

Architecture modeling. Existing approaches fall into two categories. The first category (C2-SAEDL, Dynamic Wright, Darwin, ArchWare, Plastik and Hansen *et al.*) relies on an ADL to model architectures while a second category is intended to be ADL-independent (SAEV, SAEM, Barnes *et al.* and Tibermacine *et al.*). The latter provide flexibility to the architect to use his preferred ADL and/or modeling language. All things considered, both categories hardly support three-level architecture modeling in conformance with CBSD. In particular, the specification level, we judge crucial to design for reuse is usually ignored.

Requirement 1: To support architecture modeling in CBSD processes, we need a general-purpose modeling language that explicitly represents the three architecture abstraction levels of component-based software. Dedal presented in Chapter 2 corresponds to this requirement.

Architecture analysis. Almost all approaches support some static architecture analysis. Domain-Specific ADLs like C2-SAEDL, Wright, Darwin and π-ADL rely on rigorous formalisms to verify specific architecture properties. For instance, Darwin and π-ADL focus on behavioral and interaction checking using languages tailored for such purpose (mainly CSP or π-calculus). The other approaches (*e.g.,* SAEV and SAEM) enable invariant checking (*e.g.,* properties specified by the architect). However, none of the surveyed approaches addresses all the architecture inconsistencies as identified in Chapter 2. In particular, refinement inconsistencies that arise between architecture abstraction levels (*e.g.,* erosion) are hardly addressed. To some extent, Tibermacine *et al.* focus on maintaining architectural decisions at specification and configuration levels but they do not address the assembly level.

Requirement 2: To support rigorous architecture analysis, we need to ensure that architecture descriptions are consistent at all abstraction levels and all descriptions are coherent with each other (*i.e.,* traceability of design decisions is preserved at all abstraction levels). To respond to this requirement, consistency and coherence properties have to be specified and verified using a formal modeling language with rich expressiveness and that supports powerful automated analysis. In Section 3.2, we survey five modeling languages to choose the one we judge most suitable to our problem.

Architecture evolution. Almost all approaches enable architecture-centric evolution (except the approach of Tibermacine *et al.* that focuses on analyzing the impact of architecture

evolution on architectural decisions). Some approaches (C2-SADEL, Wright, Darwin, Archware and Plastik) propose a set of predefined elementary operations to enable architecture change. However, these approaches hardly treat changes as first class entities and focus more on implementing change than specifying and analyzing it. The other approaches (Hansen *et al.*, SAEV, SAEM and Barnes *et al.*) consider software changes as first-class entities and let the architect specifying change operations by himself. However, component substitutability is hardly provided by existing approaches (except for C2-SAEDL that provides multiple component subtyping mechanism). This feature enables to replace software components while preserving architecture consistency. Moreover, none of the surveyed approaches supports multi-directional (top-down and bottom-up) evolution. In most of cases changes are propagated top-down but not bottom-up, hence increasing the risk of architecture inconsistencies like erosion. There is no proposed process to analyze and handle the impact of change at all architecture levels. In most cases, changes are analyzed at architectural level and then committed and applied on implementation.

Requirement 3: To enable architecture evolution in CBSD processes, we need to establish a formal approach to capture the change at any abstraction level, analyze and handle its impact locally (*i.e.,* at the same abstraction level) and propagate it to the other abstraction levels. We address this requirement in Chapter 5.

3.2 Formal modeling languages

Formal modeling is a way to bring software systems design to a high abstraction level. It intervenes at the very early stages of software development to give a formal specification of the system's requirements. Formal modeling requires a certain mathematical knowledge from the specifier. It helps a deep understanding of the system by encouraging him to be rigorous, precise and abstract. Resulting models hence constitute unambiguous descriptions to be used by software analysis, verification and validation. Several languages and methods were proposed to aid formal modeling. Formal languages provide abstractions to represent concepts, properties over them and sometimes behavior. however, they differ in expressiveness, underlying semantics and purpose. Some languages focus more on descriptions and how to make formal modeling more accessible whereas others focus more on automated analysis and neglect expressiveness. A good formal language must present a compromise between both aspects. In the following, we survey five formal modeling languages among the most prominent ones. These languages are B, Z, OCL, Alloy and VDM. All of them share many similarities and have been widely used and applied in several projects. We finish by a comparison of these languages and a discussion that justifies the choice of the B language.

3.2.1 Overview of formal modeling languages

3.2.1.1 The B method

B [Abrial, 1996, Cansell and Méry, 2003] is a formal modeling language and a proof-based development method for software systems. The principle of such method is to start from a very abstract model of the system and then gradually refine it. Initially designed by Jean-Raymond Abrial in 1985 as to specify critical systems, B was rapidly adopted by industry and used in several case studies such as the METEOR project [Behm et al., 1999] for controlling train traffic and the PCI protocol [Cansell et al., 2002]. B is also widely used and studied in academia, mainly as a formal modeling language for verification, validation and model-checking.

B basics. B is based on Zermelo-Fraenkel (ZF) set theory and first order logic language. The B notation is very similar to mathematical language and includes all standard logical connectors (e.g. $\wedge, \vee, \Rightarrow$), set-theoretic operations (e.g. $\in, \subset, \cup$), closure and specific relations like injective ($\rightarrowtail$), surjective ($\twoheadrightarrow$) and bijective ($\rightarrowtail\!\!\!\!\twoheadrightarrow$) functions. B also supports sequences, booleans (BOOL), integers (INTEGER) and naturals (NAT) as basic types.

B specifications are composed of abstract machines similar to modules. They are defined independently and can be reused as modules and refined to obtain more concrete models. An abstract machine is divided into a declarative part and a dynamic part. The declarative part contains the declaration of sets (*SETS*), constants (*CONSTANTS*), variables (*VARIABLES*) which represent the state of the machine and invariant properties related to variables (*INVARIANT*). Optionally, it is also possible to set definitions (*DEFINITIONS*) in the spirit of macros. Definitions are useful to define extensive sets and parametrized predicates and can be reused by invariants and operations. The dynamic part contains the initialization (*INITIALISATION*) of the machine as well as operations (*OPERATIONS*) over the state (variables) of the machine. The behavior of operations is explicitly defined in B using various constructs such as preconditions (*PRE P THEN S END*), bounded choice (*CHOICE S1 OR S2*) or non-determinism (*ANY v WHERE P THEN S END*). Post-conditions are expressed by substitutions that state the new assignments of the involved variables. Output variables may also be defined as values returned by operations.

Tool support for B. B tools focus mainly on theorem proving and code generation like Atelier-B [ClearSy, 2001] and BToolkit [Bto, 1997]. Atelier-B enables type checking and generates proof obligations (PO) related to a model. It is also able to prove automatically a number of generated PO. For the unproven ones, Atelier-B provides an interactive assistant for experienced users to prove them manually. Recently, Bware [Delahaye et al., 2014], a front

end platform was developed to automatically prove as much PO generated by Atelier-B as possible. BWare gives the input to several provers and merge their results to obtain a maximum number of proven POs.

Another kind of powerful tool for B is ProB [Leuschel and Butler, 2008]. ProB is a model checker and animator for B models. It automatically generates counterexamples for given assertions by exploring exhaustively the model (using state space exploration techniques). It also simulates the execution of operations on a given subset of the model and generates traces leading to some desired state. ProB also enables constraint-solving by reduction to SAT using a front-end engine called KodKod [Torlak and Jackson, 2007]. This feature does not yet cover all the abstractions of B but it is being improved in the recent releases. An Api is also provided for developers to integrate the features of ProB in their tools.

3.2.1.2 The Z language

Z (Zed) [Spivey, 1992] is a formal specification language and a mathematical notation widely used in education and research. Firstly developed at Oxford University in the 1980's, Z was then standardized by ISO in 2002. The language has been successfully applied on several industrial projects like the CICS system conducted by IBM and Oxford university and the security verification of the Mondex electronic purse developed by NatWest Bank.

Z basics. Z is also based on ZF-set theory and first order logic. It shares the same roots as B since Jean-Raymond Abrial, the designer of B is also one of the contributors to Z. The main difference between both languages is that Z is more abstract and was basically designed for modeling whereas B is emphasized on refining models into code. Like B, the Z notation is close to the mathematical language. It provides a number of useful constructs like comprehensive and extensive sets, power set, operations on sets, closure, relations and several kinds of functions.

A Z specification is a collection of schemas. A schema has a name and is divided into a signature part and a predicate part. The signature part lists the attributes of the schema and the predicate part lists all the constraints and properties over these attributes. A schema can also represent an operation. In contrast to B, there is no distinction between pre and post conditions in an operation definition. However, pre and post states are differentiated using the prime symbol (*e.g.,* s and s').

Tool support for Z. Z was mainly designed to perform proof on models, the reason why theorem provers are the most promoted tools for Z. Examples include Z/Eves [Saaltink et al., 1999] that can perform automatic proof in many steps. Complex theorems however need

experienced user assistance. Z/Eves can also perform domain checks (to ensure that partial functions are not applied outside their domains) and evaluate preconditions. Z is also supported by animation tools such as Jaza [Utting, 2005]. Jaza can simulate the execution of operations and check invariant properties on a given state. The animator is also able to perform constraint-solving but restricted to very small domains. Unlike B and Alloy, Z is not well supported by automatic verification tools especially model-checkers [Smith and Wildman, 2005]. This is due to the high abstract nature of the Z language making its handling challenging. Nevertheless, continuous attempts to build a model checker for Z are undertaken [Derrick et al., 2011].

3.2.1.3 OCL

OCL [OCL, 2012], the Object Constraint Language was developed at IBM in 1995. It was rapidly adopted as a formal specification language within UML and became an OMG standard. OCL was firstly designed in the hope to be simpler than the classical formal languages such as Z and VDM. Hence, OCL was basically not emphasized to support strong model analysis such as proof or model-checking. Later, several attempts were undertaken by the Precise UML Group to enhance OCL and make it comparable to the other formal languages [Bruel et al., 2000].

OCL basics. OCL is also based on first order logic. OCL specifications are defined as annotations of UML diagrams. Hence, unlike the other studied formal languages, OCL is dependent on graphical descriptions. The main syntactic unit in OCL is an expression. Expressions are used to define invariants, pre-conditions or post-conditions. OCL includes a standard library of primitive types like basic types (Integer, Boolean, String, *etc.*), collection types (Set, Bag, Sequence) and enumerations. Class types in OCL are related to a class diagram. Indeed, each defined class in a class diagram is a type in an OCL expression. However relations are not included in the OCL type library. As a consequence, expressing constraints is sometimes verbose compared to other formal languages. OCL enables predicates and functions to be defined recursively. While this brings useful expressiveness such as enabling multi-step navigation, it makes the logical interpretation of predicates hard. This is the reason why constraint-solving can not be applied on OCL [Jackson, 2006].

Tool support for OCL. OCL is supported by several standalone testing tools like USE [Gogolla et al., 2007]. USE (UML-based Specification Environment) is an environment for the specification of Information Systems. It proposes a textual syntax of UML class diagrams that includes constraints expressed with OCL. USE also enables model animation and the creation and

manipulation of snapshots (*i.e.,* states of a running system). Constraints can be automatically checked over each snapshot. The environment provides a graphical UML view and an evaluation view to query detailed information about a system's state.

HOL-OCL [Brucker and Wolff, 2008] is another tool that aims to provide interactive proof for OCL. The tool is based on translating OCL to HOL (High Order Logic) to benefit from the Isabelle theorem-prover. As being a part of UML, OCL has been also the cornerstone of several works integrating UML with other formal methods notably B, Z and Alloy.

3.2.1.4 Alloy

Alloy [Jackson, 2006] is a modeling language designed by Daniel Jackson at the Massachusetts Institute of Technology. The first version of Alloy was developed in 1997. Since, the language has been greatly improved through several iterations and now includes many features such as signatures, subtyping and polymorphism. Alloy has now become a strong competitor to its anterior formal modeling languages.

Alloy basics. Alloy was inspired by Z in its semantics and by object modeling languages in its notation. Like B and Z, Alloy is state-based and it is based on first-order relational logic. The particularity of Alloy is that all data structures (even sets and scalars) are unified within the notion of relation. This makes Alloy usage more generic since its structures can be used for various purposes. For instance, the relational composition operator "." can be used in a manner similar to object-oriented programming (*e.g.,* accessing signature fields, applying functions). However, Alloy is less expressive than the other languages and is strictly first order whereas B, Z and VDM support higher order structures such as set of sets (*e.g.,* the domain of domain of a relation that maps two functions) [Jackson, 2006].

An Alloy module is mainly constituted of *signature* declarations and *facts*. A *signature* declaration includes a set of immutable units called *atoms* and ,optionally, *fields* that relate between *signatures*. Signatures can be subtyped using the *extends* operator. *Facts* are used to express constraints on signatures that represent invariant properties. Alloy also enables to declare named and parametrized constraints using *predicates*. *Predicates* may be invoked and reused in *facts*. Being generic, Alloy does not provide built-in structures to model operations. Hence, unlike B, there is no standard way to express the dynamic behavior of operations. One way is to use predicates to represent operations. Pre- and post-states have to be defined explicitly using a specific signature (*e.g., Time*). This operation representation is similar to operation schemas in Z.

Tool support for Alloy. Alloy was developed step by step with its accompanying tool the Alloy analyzer, a SAT-based constraint-solver. Given constraints of a model, the Alloy analyzer finds structures satisfying them. It performs two opposite checking ways, either as a model finder by exhaustively exploring the model and generating sample structures or as a model checker by generating counterexamples. Alloy analyzer is similar to ProB in the sense that both tools perform model checking and work as constraint solvers using reduction to SAT technique (namely KodKod). One of the advantages of ProB compared to the Alloy analyzer is that ProB is also an animator and enables checking on a given restricted subset (instance) of the model.

3.2.1.5 VDM

VDM [Jones, 1990], the Vienna Development Method was developed in the early 1970's at IBM Vienna Laboratory. The latest version of the language, VDM-SL was then standardized by ISO in 1990's. VDM was firstly developed to model the semantics of programming languages. It was later adopted as a language for modeling software systems in general. Like B, VDM is a language and also a method for refining specifications into code. Another variant of VDM called VDM^{++} was developed to support object-oriented features and concurrency. The language has been widely applied in several industrial projects such as the development of electronic trading systems and secure smart cards.

VDM basics. A VDM specification is a description of a state machine. It is mainly composed of a set of states (*state*) and a collection of operations (*operations*). Unlike the other approaches, states are explicitly defined in VDM. They consist in mutable structures with two separate values denoting pre- and post-states. A specification also contains type declarations to be used by states. Unlike states, types are immutable and can be simple (token types) or composite (record types). Each declaration may be followed by an invariant (*inv*). Operations and their pre- (*pre*) and post-conditions (*post*) are explicitly defined in VDM. Additionally, VDM introduces the notion of *frame condition* to specify the state variables that may be accessed (read) or modified (written) by an operation. Using *frame condition* simplifies the operation definition since there is no need to reassign state variables left unchanged by the operation like in Alloy and Z.

Tool support for VDM. VDM is supported by VDMTools, a software development tool suite based on formal specifications. VDMTools supports the analysis models written in both the standard version VDM-SL and the object-oriented one VDM^{++}. The toolkit includes syntax

and type checker, theorem prover and code generator. Like Z, VDM does not support model-checking. One of the proposed solutions was to translate a subset of VDM-SL to Alloy in order to enable automated analysis [Lausdahl, 2013].

3.2.2 Synthesis and comparison

Almost all surveyed formal modeling languages – B, Z, Alloy, VDM and at some extent OCL – follow the same approach for modeling software systems. They are all state-based in the sense that specifications can be seen as state machines manipulated using a collection of operations. All of them are based on first-order logic to express predicates. Comparing these languages is a little bit challenging since there is no best language but one can be more convenient than the others in a particular context. In this comparison, we focus on two main aspects: expressiveness and analysis support.

Expressiveness. This aspect denotes how much the language is expressive. The more the language is expressive, the easier modeling becomes. Expressiveness is crucial if models have to be shared between multiple users (For instance, descriptions may be validated by a user different from the specifier) and hence must be as much clear and understandable as possible. Moreover, expressiveness is important when it comes to translate another language (*e.g.,* ADLs, semi-formal models) to the formal one. Indeed, higher expressive languages are privileged to find a mapping for the most constructs of the source language and cover as much features as possible. Expressiveness depends on the structures provided by the language and the notation afforded to write models. To compare the expressiveness of the formal modeling languages, we propose the following criteria:

- *States*: How states are represented? Are they explicit? Implicit?
- *Operations*: Are operations and their behavior (preconditions, actions, post-conditions) explicitly represented?
- *Invariants*: Are invariants explicit? Grouped into a single clause or declared separately anywhere in the specification?
- *Types*: What are the types supported by the language?
- *Notation*: What kind of notation is supported by the language (mathematical, object-oriented, relational, *etc.*)?

Analysis support. Model analysis [Jackson and Rinard, 2000] is crucial to check that the model is consistent and satisfies the desired properties. It may address one or several aspects of the model related to a verification purpose (*e.g.,* syntax, types, correctness, *etc.*) and/or a validation purpose (*e.g.,* required properties, expected behavior, *etc.*). There are different ways and techniques to perform model analysis. The use of any technique depends firstly on the semantics of the language and then on the availability of the tool supporting it. To compare languages according to analysis, we focus on the following techniques:

- *Type-checking*: Type-checking is compulsory when languages are typed. It verifies that types are used correctly and in conformance with the typing rules of the language. Type-checking is generally performed with syntax-checking and precedes any other verification activity like proof.
- *Animation*: Animation is a technique to test models by simulation. Given an input (subset of the model), the user can see the model at different states and observe its behavior. Animation is efficient for design-time validation and test-driven modeling.
- *Proof*: Theorem proving is the process that attempts to find a mathematical proof for a property of the specification. It is mostly used for the development of critical systems to ensure that resulting programs are error-free and do not present risks. While theorem proving guarantees the correctness of models, this process can hardly be fully automated because it is an undecidable problem for expressive logic like the first order logic.
- *Model-checking*: Model-checking is a technique that checks automatically if a desired property holds on a finite model [Clarke and Wing, 1996]. The check is performed by exhaustive exploration of the model searching for a counter-example breaking the property. In contrast to theorem proving, the model-checking process is guaranteed to terminate on finite models. However, the main inconvenient of such technique is the *state explosion problem* that occurs in large scale systems with an enormous number of states.
- *Constraint-solving*: Constraint-solving is an analysis based on boolean satisfiability (SAT) to find a solution for a propositional formula. In other words, given a constraint expressed with propositional variables, the aim is to find a boolean assignment of those variables so that the constraint returns true. Since modeling languages are usually more expressive than propositional logic, this technique is applied by translating constraints to propositional logic (also called reduction to SAT) to benefit from SAT solvers.

Table 3.2 summarizes the characteristics of the surveyed modeling languages according to expressiveness and analysis support.

Language	Expressiveness					Analysis support				
	States	**Operations**	**Invariants**	**Types**	**Notation**	**Type checking**	**Animation**	**Proof**	**Model-checking**	**Constraint-solving**
B	implicit, represented by variables, no explicit differentiation between pre and post states	built-in, explicit preconditions and substitutions	grouped in the *INVARIANT* clause	arithmetic (integers, naturals), strings, booleans, sets, relations, functions, sequences, predicates, variables typed in the *INVARIANT* clause	mathematical, first order logic, support higher order structures	yes	yes	yes	yes	yes
Z	implicit, pre and post states are differentiated	operation schema, implicit preconditions	implicit, declared with predicates and operations	arithmetic (integers, naturals), chars, booleans, sets, relations, functions, sequences, predicates	mathematical, first order logic, support higher order structures	yes	yes	yes	no	restricted
OCL	implicit	built-in, explicit pre- and post-conditions, implicit actions	explicit, declared with contexts	primitive types (integers, real numbers, ...), collection types (sets, bags, ...), enumerations, class types	first-order logic, object-oriented, dependent on UML, support higher order structures and recursion	yes	yes	no	no	no
Alloy	not provided, should be explicitly represented	not provided, should be represented using predicates	facts	relations, functions, predicates, integers, sequences	strictly first order, relational, close to object-oriented notation	yes	yes	no	yes	yes
VDM	built-in structures (*State*), differentiated pre- and post-states	built-in, explicit pre ad post conditions	explicit (*inv*)	arithmetic (naturals, integers and real numbers), chars, booleans, sets, relations, functions, sequences, predicates	mathematical, first-order logic, support higher order structures	yes	yes	yes	no	no

TABLE 3.2: Comparison of formal modeling languages

B, Z and VDM are quite similar in term of expressiveness since they were basically designed for theorem-proving. All of them enable to express properties in a similar way and support almost the same types (In addition, VDM supports real numbers). However there are some subtle differences between them. Z is more abstract while VDM and B are more low level and intended to be refined into code. Both (*i.e.,* VDM and B) adopt a similar structure that reflects the notion of state machines. They explicitly separate between the declarative part and the dynamic part (operations) and unlike Z they separate between pre-conditions and post-conditions. B has the particularity to modify variables by assignments like in programming languages while in VDM and Z, pre and post states must be explicit. OCL and Alloy are different and were designed for different purposes. OCL was basically developed to enforce UML diagrams with constraints that cannot be expressed using graphical notations. It has an object-oriented notation and heavily relies on navigation. Hence predicate expressions are sometimes verbose compared to the mathematical notation adopted by the other languages. Alloy is less expressive than all the other languages since it is intended to be more generic and simple. The reason is also that Alloy was mainly designed to support automated lightweight analysis which requires simple semantics.
Regarding analysis support, all the surveyed languages are typed and hence support type-checking. Except for Alloy, all languages are provided with animators. The reason is that the Alloy analyzer aims to perform more powerful analysis than animators and does not restrict the model to an executable subset. Theorem-proving is only supported by Z, B and VDM which were basically designed for software correctness. B and VDM provide also a refinement mechanism and enables code generation. Model-checking and constraint solving is only supported by B (through the ProB tool) and Alloy (through the Alloy analyzer). At some extent, Jaza, the animator of Z can do some constraint-solving on small domains.

3.2.3 Discussion

B seems to be the best compromise between expressiveness and analysis support. It has a clear structuring and it enables to model explicitly both static and dynamic aspects of software systems. Moreover, it supports all kinds of analysis. We believe that the use of B and its accompanying ProB tool provide a good support for formalizing and analyzing software architecture evolution. Alloy is also a good candidate. However, regardless its expressiveness, it presents another shortcoming. As witnessed by Torlak *et al.* [Torlak and Jackson, 2007], Alloy lacks support of partial instances. Partial instances are explicit representations of instances included in the specification of the model. Montaghami *et al.* [Montaghami and Rayside, 2012] argued that this feature enables a number of capabilities such as test-driven development, regression testing, modeling by example, and combined modeling and meta-modeling. Authors

also proposed a syntax extension of Alloy to support partial instance definition but, as far as we know, this feature is not yet integrated in the last version of Alloy [3].

3.3 Conclusion

The state of the art study showed that evolution management in reuse-centered, component-based development processes still lacks both foundations and techniques. Foundations relate to explicit the semantics of the three steps of CBSD (*i.e.,* specification, implementation and deployment) as well as the rules that govern their relationships (*i.e.,* how to get from specification to implementation to deployment and back). For that, we need an ADL that explicitly models architectures conforming to CBSD process. For that, we opt for Dedal introduced in Chapter 2. Then, we need a formal modeling language to set the semantics of Dedal and enable automatic evolution management in a reliable way (*i.e.,* preventing architecture inconsistencies). For that, we opt for the B modeling language as our study showed that it happens to be the best candidate.

[3] http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf

Chapter 4

A type theory for three-level software architectures

Dedal introduces a novel approach for component-based software engineering by representing explicitly three architecture descriptions corresponding to the specification, implementation and deployment steps of CBSD (*cf.* chapter 2). The semantics of these descriptions as well as their relationships, need to be elucidated and formally defined to enable component reuse and support architecture analysis and evolution throughout the whole component-based software lifecycle. This chapter introduces a type theory for Dedal that formally defines its semantics and the rules that govern their relationships. Section 4.1 gives an overview of the proposed type theory and its goals. The formalization of Dedal using the B modeling language is presented in Section 4.2. Then, Section 4.3 and Section 4.4 respectively explain the intra-level and inter-level rules.

Contents

4.1 Overview of the three-level type theory 54
4.1.1 Goals of the type theory 54
4.1.2 Structure of the type theory 55
4.2 Formalization of the Dedal architectural model 55
4.2.1 Formalization of the common architectural concepts 57
4.2.2 Formalization of Dedal architectural concepts 60
4.2.3 An illustrative example: B formal specifications of the Home Automation Software 62
4.3 Intra-level rules 63
4.3.1 Intra-level component rules 63
4.3.2 Intra-level architecture consistency 71
4.4 Inter-level rules 75

4.4.1 Inter-level component rules 75
4.4.2 Inter-level architecture coherence 78
4.5 Conclusion 83

4.1 Overview of the three-level type theory

The three level type theory [Mokni et al., 2014] defines the semantics of the Dedal architecture model and the rules that govern the relationships between Dedal artifacts. It consists of a set of mathematical definitions and rules built upon the set theory and first-order predicate logic. In the remainder, we set the goals, hypothesis and structure of the type theory.

4.1.1 Goals of the type theory

The three-level type theory aims to:

- enable component substitutability which is central to architecture evolution.
- formally document the traceability of structural design decisions throughout the whole CBSD lifecycle.
- foster component reuse by elucidating the semantics of the linkage between the specification level and implementation level.
- enable architecture analysis and evolution and deal with architecture inconsistencies, notably the erosion problem discussed in Chapter 2, Section 2.2.4.

The type theory assumes that:

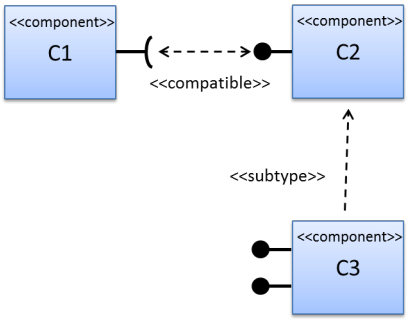
- reused components are trustworthy and behave correctly once assembled into software.
- adaptability between components is implicitly managed by connections.

The type theory established in this thesis addresses the syntactic level of software component specification and the structural aspect of software architectures. These aspects present the basis that may be extended later to support other component specification levels (*e.g.,* behavior, synchronization and quality of service) [Beugnard et al., 1999].

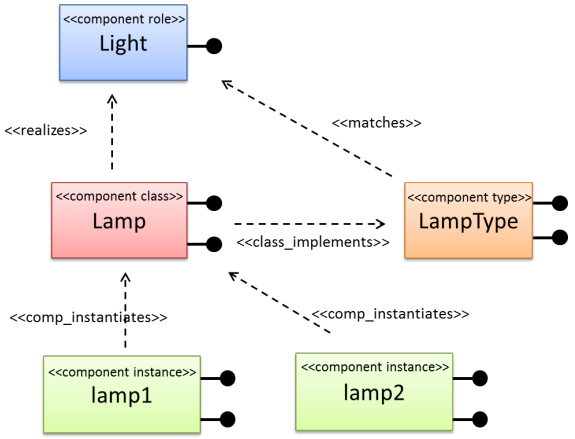
4.1.2 Structure of the type theory

To establish the type theory, we start by formalizing the underlying concepts of the Dedal three-level architectural model using the B formal modeling language as argued in Chapter 3. The set of resulting models, named *FormalDedal* are presented in Section 4.2.

On the basis of the Dedal formalization, we define the semantics related to the relations between the Dedal artifacts. In Dedal, there are two kinds of relations (*cf.* Figure 4.1) that adhere to the type theory:



(A) Example of intra-level relations



(B) Example of inter-level relations

FIGURE 4.1: Relations kinds in Dedal

- **Intra-level relations**: concern the relations between the architecture elements at the same architecture level. They include component substitutability rules and compatibility rules and intra-level architecture consistency rules. The rules for intra-level relations are discussed through Section 4.3.
- **Inter-level relations**: concern the relations between the architecture elements of different architecture levels. They include the realization relation between component roles and component classes, the instantiation relation between component classes and component instances as well as the coherence relations between each two adjacent abstraction levels. The rules for inter-level relations are discussed through Section 4.4.

4.2 Formalization of the Dedal architectural model

The formal Dedal comprises two kinds of concepts:

- The common architectural concepts (*i.e.,* architecture, component, interface, *etc.*). These concepts define the generic structure of architecture descriptions (*e.g.,* an architecture is composed of components and their connections).

- The Dedal architectural concepts related to the component-based development steps (*i.e.,* the architecture specification, the architecture configuration and the architecture assembly). These concepts are a specialization of the common concepts (*e.g.,* component roles and concrete component types inherit from the concept of component) and have specific relations between them (*e.g.,* a component class realizes a component role and a component instance instantiates a component class).

Using the modularity of B, we divide the formalization of Dedal into five B machines as depicted in Figure 4.2:

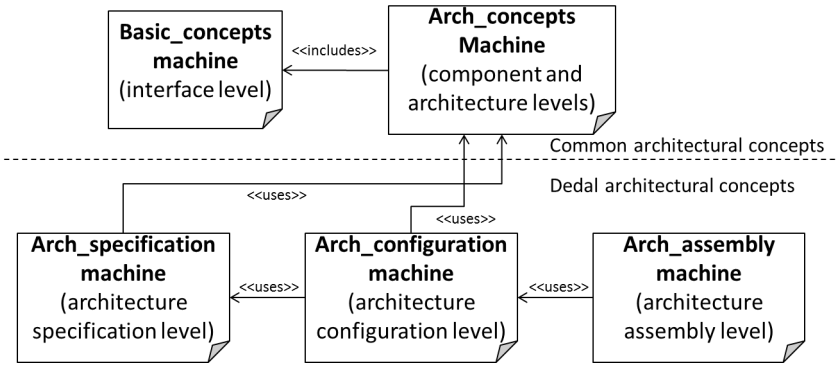


FIGURE 4.2: Modularization of Dedal formalization

- *Basic_concepts* machine includes the formalization of the interface concept part (*i.e.,* interface directions, interface types, signatures and parameters).

- *Arch_concepts* machine includes the formalization of the common architecture concepts (*i.e.,* architecture, components and connections). It includes the *Basic_concepts* machine as it reuses the definitions related to the interface concept.

- *Arch_specification* machine includes the formalization of the concepts related to the architecture specification level (*i.e.,* specification and component roles). It includes the *Arch_concepts* machine as it specializes the common architectural concepts.

- *Arch_configuration* machine includes the formalization of the concepts related to the architecture configuration level (*i.e.,* configuration, component types, component classes, composite component classes). It uses the *Arch_specification* machine to define the relations between the specification and configuration levels.

- *Arch_assembly* machine includes the formalization of the concepts related to the architecture assembly level (*i.e.,* assembly and component instances). It uses the *Arch_configuration* machine to define the relations between the configuration and assembly levels.

The motivation behind organizing the formalization in such way is twofold: (1) As separated into an independent B model, the common architectural concepts may be reused, extended and specialized by other models. (2) The formalization must explicit the three architecture descriptions of component-based software in accordance with Dedal. This enables to perform verification and validation at each development step separately.

Remark. Intuitively, one could think that the B machines corresponding to the three architecture abstraction levels should be linked through a refinement relation. However, this is not the case for two reasons. The first is that semantically, the assembly level can not be considered as a refinement of the configuration level since both levels represent two different aspects of the software (*i.e.,* design-time and runtime). The second is that the operational semantics (*i.e.,* the set of operations discussed in Chapter 5) of the three architecture levels is different and technically cannot be specified using refinement. Indeed, B imposes that a refinement machine should reuse the same operations as the refined machine whereas two operations at two different abstraction levels do not necessarily manipulate the same variables. Still, the refinement mechanism opens an interesting perspective toward obtaining a concrete architecture configuration from an abstract architecture specification by successive refinements.

4.2.1 Formalization of the common architectural concepts

This section presents the formalization related to common architectural concepts, namely components, interfaces, connections and architectures.

The *Basic_concepts* machine. The *Basic_concepts* machine (*cf.* Table 4.1) represents a formal model of the component concept at the interface granularity level. It proposes a syntactic definition of the interface concept. Structurally, each interface has a direction (*int_direction*) (PROVIDED or REQUIRED) and a type (*int_type*). Each interface type is constituted of a non-empty set of signatures (*int_signatures*) and each signature has a name (*sig_name*), a return type (*sig_return*) and eventually a parameter list (*parameters*). Finally, each parameter has a name (*param_name*) and a type (*param_type*).

MACHINE *Basic_concepts*
SETS
$PARAM\_NAMES$; $PARAMETERS$; $INTERFACES$;
$TYPES$; $INTERFACE\_TYPES$;
$SIGNATURES$; SIG_NAMES;
$DIRECTION = \{PROVIDED, REQUIRED\}$
VARIABLES
$parameter, interface, type, subtype,$
$param\_name, param\_type, int\_direction,$
$int\_type, interfaceType,$
$signature, sig\_name, parameters, sig\_return, int\_signatures$
INVARIANT
/* Definition of type */
$type \subseteq TYPES \wedge$
/*A parameter has a name and a type */
$parameter \subseteq PARAMETERS \wedge$
$param\_name \in parameter \rightarrow PARAM\_NAMES \wedge$
$param\_type \in parameter \rightarrow type \wedge$
/*A signature has a name, a list of parameters and a return type */
$signature \subseteq SIGNATURES \wedge$
$sig\_name \in signature \rightarrow SIG\_NAMES \wedge$
$parameters \in signature \rightarrow \mathcal{P}(parameter) \wedge$
$sig\_return \in signature \rightarrow type \wedge$
/* An interface type has a name and a list of signatures */
$interfaceType \subseteq INTERFACE\_TYPES \wedge$
$int\_signatures \in interfaceType \rightarrow \mathcal{P}1(signature) \wedge$
/* An interface has a type and a direction (provided or required) */
$interface \subseteq INTERFACES \wedge$
$int\_type \in interface \rightarrow interfaceType \wedge$
$int\_direction \in interface \rightarrow DIRECTION$

Specific B notations:
$\leftrightarrow$: relation $\rightarrow$: total function $\mathcal{P}$(`<set>`): powerset of `<set>`
$\mathcal{P}_1$(`<set>`): powerset of `<set>` minus the empty set (*i.e.*, $\mathcal{P}$(`<set>`) $- \varnothing$)

TABLE 4.1: B specifications related to interfaces

The *Arch_concepts* machine. The *Arch_concepts* machine (*cf.* Table 4.2) represents a formal model of component-based architectures from a structure viewpoint and independently from any architectural style. It reuses the formalization of the interface concept defined in the *Basic_concepts* machine. Structurally, an architecture is composed of a set of components (*arch_components*) and a set of connections (*arch_connections*). A component has a name (*comp_name*) and a set of interfaces (*comp_interfaces*). The *comp_interfaces* function is injective since each set of interfaces belongs to one and only one component. A connection relates a client element (*client*) to a server element (*server*). A client (resp. server) element is defined as a component's required interface (resp. provided interface).

MACHINE *Arch_concepts* **INCLUDES** *Basic_concepts* **SETS** $ARCHITECTURES; COMPONENTS; COMP\_NAMES$ **VARIABLES** $component, comp\_name, connection, comp\_interfaces, client, server$ $architecture, arch\_components, arch\_connections$ **INVARIANT** $component \subseteq COMPONENTS \wedge$ $comp\_name \in component \rightarrow COMP\_NAMES \wedge$ $comp\_interfaces \in component \rightarrowtail \mathcal{P}(interface) \wedge$ $client \in component \leftrightarrow interface \wedge$ $server \in component \leftrightarrow interface \wedge$ $connection \in client \leftrightarrow server \wedge$ $architecture \subseteq ARCHITECTURES \wedge$ $arch\_components \in architecture \rightarrow \mathcal{P}(component) \wedge$ $arch\_connections \in architecture \rightarrow \mathcal{P}(connection)$ $arch\_clients \in architecture \rightarrow \mathcal{P}(client)$ $arch\_servers \in architecture \rightarrow \mathcal{P}(server)$
Specific B notations: $\rightarrowtail$: injective function

TABLE 4.2: B Specification of the underlying architecture concepts

Using the *DEFINITION* clause of B machine, we add the following macros:

- *providedInterfaces* and *requiredInterfaces* respectively denote the set of provided interfaces and the set of required interfaces of a component (*comp*). Both definitions are deduced from the *comp_interfaces* relation:

$$
\begin{aligned}
&providedInterfaces(comp) == \{int \mid int \in interface \wedge \\
&\qquad int \in comp\_interfaces(comp) \wedge \\
&\qquad int\_direction(int) = PROVIDED\};
\end{aligned}
$$

$$
\begin{aligned}
&requiredInterfaces(comp) == \{int \mid int \in interface \wedge \\
&\qquad int \in comp\_interfaces(comp) \\
&\qquad \wedge\, int\_direction(int) = REQUIRED\}
\end{aligned}
$$

- It will be practical to get the component element and/or the interface element from the *client* and *server* relations. We define the following relations for such purpose:

$$
\begin{aligned}
&clientInterfaceElem = \{cl, int \mid cl \in client \wedge int \in interface \wedge \\
&\qquad \exists(comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp, rint) \wedge int = rint)\}
\end{aligned}
$$

The $clientInterfaceElem$ enables to get the interface element (int) of a given client (cl). This relation is calculated by identifying the required interface ($rint$) related to the given cl client. Similarly, the $clientComponentElem$ enables to get the component element (c) as follows:

$$
\begin{aligned}
&clientComponentElem = \{cl, c \mid cl \in client \wedge c \in component \wedge \\
&\qquad \exists(comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp, rint) \wedge c = comp)\}
\end{aligned}
$$

In the same way, $serverInterfaceElem$ and $serverComponentElem$ enable to respectively get the interface element and component element of a given server (se):

$$
\begin{aligned}
&serverInterfaceElem = \{se, int \mid se \in server \wedge int \in interface \wedge \\
&\quad \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp, pint) \wedge int = pint)\} \\
&serverComponentElem(se) = \{se, c \mid se \in server \wedge c \in component \wedge \\
&\quad \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp, pint) \wedge c = comp)\}
\end{aligned}
$$

The formalization of the common architectural concepts serves as a basis to define the intra-level rules that apply to any architecture abstraction level (see Section 4.3).

4.2.2 Formalization of Dedal architectural concepts

This section presents the formalization related to the Dedal architecture model, namely the architecture specification, configuration and assembly levels.

The *Arch_specification* machine. The *Arch_specification* machine (*cf.* Table 4.3) represents a formal model of the architecture specification concepts. It specializes the definitions of $COMPONENT$ to define the concept of component role ($COMP\_ROLES$). The inheritance feature is not explicitly provided by the B modeling language. Therefore, we represent inheritance with an inclusion property relating the subtype to its supertype. For instance, the statement "*a component role is a component*" is formalized as $COMP\_ROLES \subseteq COMPONENTS$. This way, all the relations that apply to components also apply to component roles without being obliged to redefine them. For instance, the $comp\_interfaces$ relation is also valid to use for any element of the set *compRole*.

MACHINE
$Arch\_specification$ **USES** $Arch\_concepts$
CONSTANTS
$COMP\_ROLES$
PROPERTIES
$COMP\_ROLES \subseteq COMPONENTS$
VARIABLES
$specification, spec\_components, spec\_connections, compRole, spec\_clients, spec\_servers$
INVARIANT
$compRole \subseteq COMP\_ROLES \wedge$
$specification \subseteq ARCHITECTURES \wedge$
$spec\_components \in specification \rightarrow \mathcal{P}(compRole) \wedge$
$spec\_connections \in specification \rightarrow \mathcal{P}(connection) \wedge$
$spec\_clients \in specification \rightarrow \mathcal{P}(client) \wedge$
$spec\_servers \in specification \rightarrow \mathcal{P}(server)$

TABLE 4.3: B model of the specification level

The *Arch_configuration* machine. The *Arch_configuration* machine (*cf.* Table 4.4) represents the formal model of the architecture configuration level. The *Arch_concepts* machine is used again to define concrete component types ($COMP\_TYPES$) as a specialization of $COMPONENTS$. The *Arch_configuration* machine also introduces the concepts of component attribute (*class_attributes*), composite component class (*compositeComp*) and delegated interface ($delegatedInterface$). A composite component class is a specialization of a component class. This specialization preserves the encapsulation principle since the type description of a composite component exposes only its delegated interfaces. Its composition is defined as an inner architecture configuration (*composite_uses*). The *Arch_configuration* machine also uses *Arch_specification* to check the inter-level relations between both levels.

MACHINE $Arch\_configuration$
USES $Arch\_concepts, Arch\_specification$
SETS
$COMP\_CLASS; CLASS\_NAME; ATTRIBUTES; ATT\_NAMES;$
$CONFIGURATIONS$
CONSTANTS
$COMP\_TYPES$
PROPERTIES
/* Component types are also a specialization of components distinct from component roles */
$COMP\_TYPES \subseteq COMPONENTS \wedge$
$COMP\_TYPES \cap COMP\_ROLES = \varnothing$
VARIABLES
$configuraton, config\_components, config\_connections,$
$compType, compClass, class\_name, class\_attributes,$
$compositeComp, delegatedInterface, delegation,$
$attribute, attribute\_name, attribute\_type, composite\_uses$
INVARIANT
$compType \subseteq COMP\_TYPES \wedge$
/* A component class has a name and a set of attributes */
$compClass \subseteq COMP\_CLASS \wedge$
$class\_name \in compClass \rightarrow CLASS\_NAME \wedge$
$class\_attributes \in compClass \rightarrow \mathcal{P}(attribute) \wedge$
/* An attribute has a name and a type */
$attribute \subseteq ATTRIBUTES \wedge$
$attribute\_name \in attribute \rightarrow ATT\_NAMES \wedge$
$attribute\_type \in attribute \rightarrow TYPES \wedge$
/* A composite component is a component class
$compositeComp \subseteq compClass \wedge$
/* A delegation is a mapping between a delegated interface and its corresponding one */
$delegatedInterface \subset interface \wedge$
$delegation \in delegatedInterface \overset{inj.}{\rightarrow} interface \wedge$
/* A configuration is a set of component classes */
$configuration \subseteq CONFIGURATIONS \wedge$
$config\_components \in configuration \rightarrow \mathcal{P}_1(compClass) \wedge$
$config\_connections \in configuration \rightarrow \mathcal{P}(connection) \wedge$
/* The composition of a composite component is described using a configuration */
$composite\_uses \in compositeComp \rightarrow configuration$

Specific B notations:
$\overset{inj.}{\rightarrow}$: injective function $\mathcal{P}$(`<set>`): powerset of `<set>`
$\mathcal{P}_1$(`<set>`): powerset of `<set>` without the empty set (*i.e.,* $\mathcal{P}$(`<set>`) $- \varnothing$)

TABLE 4.4: B specification of the configuration level

The *Arch_assembly* machine. The *Arch_assembly* machine (*cf.* Table 4.5) represents the formal model of the architecture assembly level. It introduces the concept of component instance (*compInstance*), its initial state (*initial_state*) and its current state (*current_state*).

MACHINE *Arch_assembly*
USES *Arch_configuration*
SETS
$COMP\_INSTANCES; INSTANCE\_NAME; ASSEMBLIES;$
$ATTRIBUTES\_VALUES$
VARIABLES
$compInstance, compInstance\_name, initial\_state,$
$current\_state, attribute\_value,$
$assm, assm\_components$
INVARIANT
$compInstance \subseteq COMP\_INSTANCES \wedge$
$compInstance\_name \in compInstance \rightarrow INSTANCE\_NAME \wedge$
$attribute\_value \in attribute \rightarrow \quad ATTRIBUTES\_VALUES \wedge$
$initial\_state \in compInstance \rightarrow \mathcal{P}(attribute\_value) \wedge$
$current\_state \in compInstance \rightarrow \mathcal{P}(attribute\_value) \wedge$
$assm \subseteq ASSEMBLIES \wedge$
$assm\_components \in assm \rightarrow \mathcal{P}_1(compInstance)$

TABLE 4.5: B specification of the assembly level

The formalization of Dedal architectural concepts serves as a basis to define the inter-level rules that govern the relations between two adjacent abstraction architecture levels (*cf.* Section 4.4).

4.2.3 An illustrative example: B formal specifications of the Home Automation Software

We consider the Home Automation Software example illustrated in Figure 4.3 and which recalls the example presented in Chapter 2, Section 2.3.

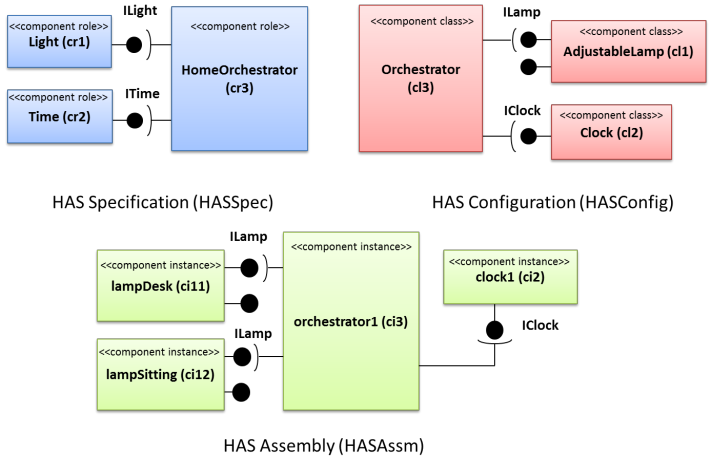


FIGURE 4.3: The architecture descriptions of HAS

The software enables to manage the light of the building in function of time through an orchestrator. As shown in the specification, the architecture involves three component roles. The *HomeOrchestrator* component role checks the time using the *Time* component role and turns on/off the light using the *Light* component role according to time information. The architecture configuration shows a concrete implementation of the HAS. The assembly architecture

shows a possible instantiation enabling to control the light in two locations (sitting room and desk) at different times.

For the sake of simplicity, we list just an extract of the *Arch_specification* machine corresponding to the HAS specification (*cf.* Table 4.6). The full B specifications corresponding to the HAS example are shown in Appendix A.

MACHINE
$Arch\_specification$ **USES** $Arch\_concepts$
$CONSTANTS$
$COMP\_ROLES$
PROPERTIES
$COMP\_ROLES \subseteq COMPONENTS \wedge$
$COMP\_ROLES = \{cr1, cr2, cr3\}$
VARIABLES
$specification, spec\_components, spec\_connections, compRole, spec\_clients, spec\_servers$
INVARIANT
...
INITIALISATION
$compRole := \{cr1, cr2, cr3\}||$
$specification := \{HASSpec\}||$
$spec\_components := \{HASSpec \mapsto \{cr1, cr2, cr3\}\}||$
$spec\_connections := \{HASSpec \mapsto \{$
$((cr3, rintILight) \mapsto (cr1, pintILight)),$
$((cr3, rintITime) \mapsto (cr2, pintITime))\}||$
$spec\_clients := \{(HASSpec \mapsto \{(cr3, rintILight), (cr3, rintITime)\}||$
$spec\_servers := \{(HASSpec \mapsto \{(cr1, pintILight), (cr2, pintITime))\}\}$

TABLE 4.6: An extract of the B model of the HAS at specification level

4.3 Intra-level rules

Intra-level rules are crucial to enable architecture analysis at a given abstraction level. They provide a subtyping mechanism for software components that enables to check their substitutability and compatibility. Moreover, intra-level rules include general-purpose architecture consistency properties that we judge crucial to check by any architecture definition apart from its specific properties like the architectural style. This section explains these rules.

4.3.1 Intra-level component rules

Intra-level component rules govern the relations between components. They include substitutability and compatibility rules. The definitions of these rules depend on the syntactic specification of components and provide a basis to involve other aspects such as behavior. We draw inspiration from the Object-Oriented type theories [Abadi and Cardelli, 1996] to define intra-level component rules, namely subtyping and method specialization.

4.3.1.1 Substitutability rules

Substitutability determines whether a component can be substituted for another in a given architecture without altering its consistency. Specifying substitutability is non trivial since components are complex entities and include different specification levels (syntactic, behavior, synchronization, non-functional properties, *etc.*). Substitutability thus depends on the architectural aspect that we would like to preserve. As discussed in Section 4.1, this work focus on the syntactic level since it presents the basis. To ease the explanation of substitutability rules, we illustrate them along considering the example shown in Figure 4.4.

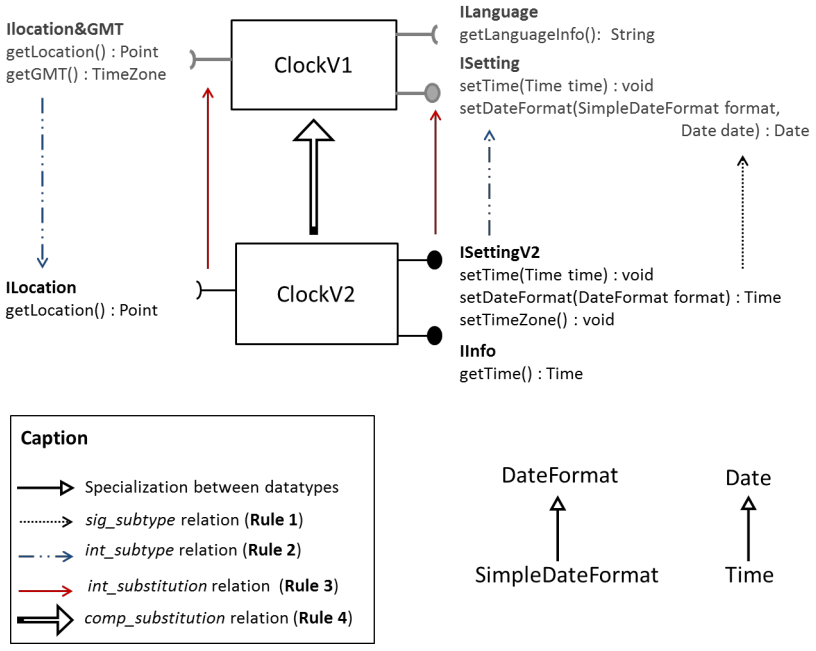


FIGURE 4.4: Example of component substitutability

The example shows that deciding about the subtitutability of two components (*comp_substitution*, *cf.* Rule 4) requires to check the substitutability between their interfaces (*int_substitution*) which in turn depends on their directions and the relations held between their types (*cf.* Rule 3). Interface subtyping (*int_subtype*, *cf.* Rule 2) in turn, depends on signature specialization (*sig_subtype* and *param_subtype*, *cf.* Rule 1). In the remainder, we detail all these rules.

Signature specialization. Signature specialization is analogous to method subtyping in Object-Oriented Programming (OOP). A specialized signature must have contravariant specialization of parameter types and covariant specialization of return type as it must require less information and provide richer results when invoked. To define signature specialization, we first consider parameter list specialization.

Rule 1. *Parameter specialization*: a signature sig_{sub} is parameter subtype of a signature sig_{sup} iff there exists an injection inj between the parameters of sig_{sup} and the parameters of sig_{sub} and for each parameter p of sig_{sup}, $inj(p)$ has the same name as p and the type of $inj(p)$ is same as or subtype of p's type.

$$
\begin{array}{l}
\forall(sig_{sup}, sig_{sub}). \\
(sig_{sup} \in signature \wedge sig_{sub} \in signature \wedge sig_{sup} \neq sig_{sub} \\
\Rightarrow (\\
\quad (sig_{sup}, sig_{sub}) \in param\_subtype \\
\quad \Leftrightarrow \\
\quad \exists inj.(inj \in parameters(sig_{sup}) \overset{inj.}{\rightarrow} parameters(sig_{sub}) \wedge \\
\qquad \forall p.(p \in parameter \wedge \\
\qquad\qquad p \in parameters(sig_{sup}) \\
\qquad \Rightarrow \\
\qquad param\_name(p) = param\_name(inj(p)) \wedge \\
\qquad param\_type(inj(p)) \\
\qquad\qquad \in subtype[\{param\_type(p)\}])) \\
\qquad) \\
\quad)
\end{array}
$$

We note that the injective function ensures that the subtype has at least the same elements as the supertype. The parameter list specialization enables to add more parameters.

Signature specialization: a signature *sig_sub* specializes a signature *sig_sup* if and only if they have the same name and *sig_sup* is parameter subtype of *sig_sub* (contravariant parameter specialization) and the return type of *sig_sub* is same or subtype of the return type of *sig_sup* (covariant return type specialization).

$$
\begin{array}{l}
\forall(sig_{sup}, sig_{sub}). \\
(sig_{sup} \in signature \wedge sig_{sub} \in signature \wedge sig_{sup} \neq sig_{sub} \\
\Rightarrow \\
\quad (sig_{sup}, sig_{sub}) \in sig\_subtype \\
\quad \Leftrightarrow (\\
\qquad sig\_name(sig_{sup}) = sig\_name(sig_{sub}) \wedge \\
\qquad (sig_{sub}, sig_{sup}) \in param\_subtype \wedge \\
\qquad sig\_return(sig_{sub}) \in subtype[\{sig\_return(sig_{sup})\}]) \\
\qquad) \\
\quad)
\end{array}
$$

Unlike OOP, our definition of signature specialization tolerates that subtype signature has less parameters than its supertype. Hence, even when invoked with extra parameters, the subtype operation still returns the expected result. This flexibility is allowed in components to enhance their interoperability. Adaptability issues are managed by connectors (*cf.* Chapter 2).

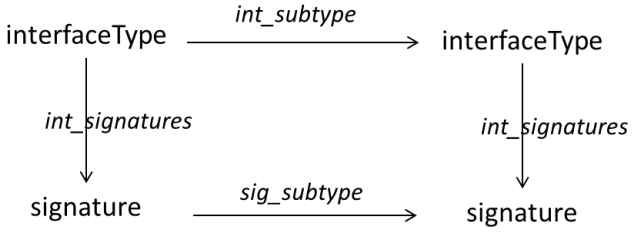
To illustrate signature specialization, we consider the case of *setDateFormat* as described in the example shown in Figure 4.4. The supertype signature (belonging to *ISetting*) requires two parameters, the *format* of type *SimpleDateFormat* and the *date* of type *Date* and it returns an object of type *Date* including the date conforming to the given *format* parameter. In accordance with rule 1, the subtype signature (belonging to *ISettingV2*) requires less parameters than the supertype since the Date parameter is no more required and the type of *format* is generalized to *DateFormat*. In the counter part, it provides a richer result since the return type, *Time* is a subtype of *Date*.

On the basis of signature specialization, we define interface subtyping.

Interface subtyping. Interface subtyping deals with the interface type description independently from its direction. Indeed, an interface type can be assigned to several interfaces with same or opposite directions. Interface subtyping relies on signature specialization.

Rule 2. *Interface subtyping.* An interface type $intTypeSub$ is a subtype of an interface type $intTypeSup$ iff there exists an injection inj between the signature set of $intTypeSup$ and the signature set of $intTypeSub$ such that for each signature sig of $intTypeSup$, $inj(sig)$ specializes sig.

Interface subtyping involves the following relations:



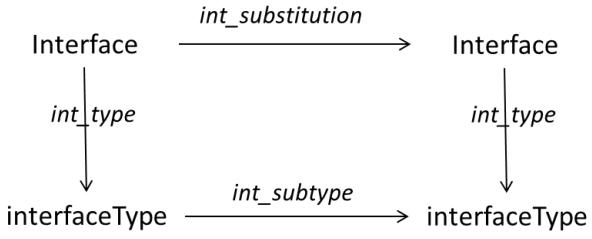
$$
\begin{array}{l}
int\_subtype \in interfaceType \leftrightarrow interfaceType \land \\
\forall\,(intTypeSup, intTypeSub). \\
(intTypeSup \in interfaceType \land intTypeSub \in interfaceType \land \\
intTypeSup \neq intTypeSub \\
\quad \Rightarrow \\
\quad ((intTypeSup, intTypeSub) \in int\_subtype \\
\quad\quad \Leftrightarrow \\
\quad\quad \exists\, inj. \\
\quad (inj \in int\_signatures(intTypeSup) \overset{inj.}{\rightarrow} int\_signatures(intTypeSub) \land \\
\quad\quad \forall(sig). \\
\quad\quad (sig \in signature \land sig \in int\_signatures(intTypeSup) \\
\quad\quad\quad \Rightarrow \\
\quad\quad inj(sig) \in sig\_subtype[\{sig\}])) \\
\quad) \\
)
\end{array}
$$

Similar to signature specialization, interface subtyping enables to have more signatures in the subtype (ensured by the injection condition). As an example of interface subtyping, we consider the relation between the interface types *ISetting* and *ISettingV2* (*cf.* Figure 4.4). *ISettingV2* is a subtype of *ISetting* since it has the same (or specialized) signatures as *ISetting* (*setTime* and *setDateFormat*) plus one more signature to set the time zone (*setTimeZone*).

Interface substitutability. Interface substitutability determines whether an interface can replace another while holding all the connections with the other interfaces correct.

Rule 3. Interface substitutability depends on the interface type and direction. When both interfaces are provided, substitutability is covariant with interface subtyping (*i.e.,* a provided interface int_{sup} is substituted for a provided interface int_{sub} iff the type of int_{sub} is a subtype of int_{sup}'s type). In the second case where the two interfaces are required, substitutability is contravariant with interface subtyping (*i.e.,* a required interface int_{sup} is substituted for a required interface int_{sub} iff the type of int_{sup} is a subtype of int_{sub}'s type).

The interface substitutability rule involves the following relations:



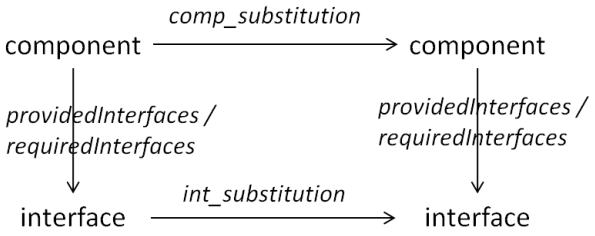
$$
\begin{array}{l}
int\_substitution \in interface \leftrightarrow interface \wedge \\
\forall (int_{sup}, int_{sub}). \\
(int_{sup} \in interface \wedge int_{sub} \in interface \wedge \\
int_{sup} \neq int_{sub} \\
\quad \Rightarrow \\
\quad ((int_{sup}, int_{sub}) \in int\_substitution \\
\quad \Leftrightarrow \\
\quad ((int\_type(int_{sub}) \in \\
\quad\quad int\_subtype[\{int\_type(int_{sup})\}]) \wedge \\
\quad\quad int\_direction(int_{sup}) = PROVIDED \wedge \\
\quad\quad int\_direction(int_{sub}) = PROVIDED) \\
\quad\quad\quad \vee \\
\quad ((int\_type(int_{sup}) \in \\
\quad\quad int\_subtype[\{int\_type(int_{sub})\}]) \wedge \\
\quad\quad int\_direction(int_{sup}) = REQUIRED \wedge \\
\quad\quad int\_direction(int_{sub}) = REQUIRED) \\
))
\end{array}
$$

Again, we illustrate interface substitutability with the example shown in Figure 4.4. *ISetting* can be substituted for *ISettingV2* since they have the same direction (both are provided interfaces) and *ISettingV2*'s type is subtype of *ISetting*'s. In a contravariant way, *ILocation&GMT* can be substituted for *ILocation* since they are both required and *ILocation&GMT*'s type is subtype of *ILocation*'s type.

Component substitutability. We consider that two components are substitutable if the subtype component exposes at least the same provided interfaces and at most the same required interfaces as the supertype component.

Rule 4. A component C_{sup} can be substituted for a component C_{sub} iff there exists an injection inj_1 between the set of provided interfaces of C_{sup} and the set of provided interfaces of C_{sub} such that int can be substituted for $inj_1(int)$, int being a provided interface of C_{sup}, and there exists an injection inj_2 between the set of required interfaces of C_{sub} and the set of required interfaces of C_{sup} such as $inj_2(int)$ can be substituted for int, int being a required interface of C_{sub}.

The component substitutability rule involves the following relations:



$$
\begin{array}{l}
comp\_substitution \in component \leftrightarrow component \land \\
\forall (C_{sup}, C_{sub}). \\
\quad (C_{sup} \in component \land C_{sub} \in component \land C_{sup} \neq C_{sub} \\
\qquad \Rightarrow \\
\quad (C_{sub} \in comp\_substitution[\{C_{sup}\}] \\
\qquad \Leftrightarrow \\
\quad \exists (inj_1, inj_2). \\
(inj_1 \neq \varnothing \lor inj_2 \neq \varnothing \land \\
inj_1 \in providedInterfaces(C_{sup}) \overset{inj.}{\rightarrow} providedInterfaces(C_{sub}) \land \\
\quad \forall (int). \\
\quad (int \in interface \land int \in providedInterfaces(C_{sup}) \\
\qquad \Rightarrow \\
\quad inj_1(int) \in int\_substitution[\{int\}]) \land
\end{array}
$$

$$
\begin{array}{l}
inj_2 \in requiredInterfaces(C_{sub}) \overset{inj.}{\rightarrow} requiredInterfaces(C_{sup}) \land \\
\quad \forall (int). \\
(int \in interface \land int \in requiredInterfaces(C_{sub}) \land \\
\qquad \Rightarrow \\
\quad int \in int\_substitution[\{inj_2(int)\}])))
\end{array}
$$

We note that two injections are required so that substitutability holds. The first injection (inj_1) is covariant with the substitutability rule and it ensures that the subtype component has "at least" the same provided interfaces as the supertype component. the second injection (inj_2) is in contravariance with the substitutability rule and it ensures that the subtype has "at most" the same required interfaces as the supertype component. At least one injection must be different from the empty set. This condition avoids the case where component with only provided interfaces can substitutes for a component with only required interfaces.

We retake the example introduced in Figure 4.4. *ClockV2* substitutes for *ClockV1* since it provides the *ISettingV2* interface which substitutes for *ISetting* and provides an extra one (*IInfo*). On the other side, *ClockV2* requires less interfaces than *ClockV1* (*ILanguage* is no more required) and the *ILocation* required interface substitutes for *ILocation&GMT*.

Discussion. Substitutability rules present a mechanism to replace software components while holding compatibility with other components correct. The principle that is enforced is that a subtype should provide at least the same services as its supertype and require the same or less services. This way, substitution avoids the risk of unavailability errors such as invoking non provided methods or type errors such as using the wrong parameter types when calling some method.

4.3.1.2 Compatibility rules

Compatibility is important to decide whether two entities can be connected together without compromising the functioning of the system. From a syntactic point of view, verifying compatibility amounts to check if a component can invoke the services of another one without unavailability or type error risks.

Interface compatibility. Interfaces present the connection points of components. Interface compatibility is crucial to determine whether two components can interact together from a syntactic point of view. A provided interface should declare the same, a specialization of and possibly extra signatures than the required interface to ensure that all the required functionalities can be supplied. Like the interface substitutability rule, the interface compatibility rule requires that the two interfaces are of the same type. However, in the former the two directions must be equal whereas in the latter, the two directions must be opposite.

Rule 5. *Interface compatibility.*

A provided interface int_1 and a required interface int_2 are compatible iff the type of int_1 is a subtype of int_2's.

$$
\begin{array}{l}
int\_compatible \in interface \leftrightarrow interface \wedge \\
\forall (int_1, int_2). \\
(int_1 \in interface \wedge int_2 \in interface \wedge \\
int\_direction(int_1) \neq int\_direction(int_2) \\
\Rightarrow (\\
\quad (int_1, int_2) \in compatible \\
\quad \Leftrightarrow (\\
\qquad (int\_direction(int_1) = PROVIDED \wedge \\
\qquad int\_direction(int_2) = REQUIRED \wedge \\
\qquad (int\_type(int_1) \in int\_subtype[\{int\_type(int_2)\}]) \\
\qquad) \\
\quad) \\
))
\end{array}
$$

Component compatibility. Component compatibility relies on interface compatibility. Two components can interact if and only if they have at least two compatible (connectable) interfaces.

Rule 6. A component c_1 is compatible with a component c_2 if and only if they have at least two compatible interfaces. Formally:

$$
\begin{array}{l}
comp\_compatible \in component \leftrightarrow component \wedge \\
\forall (c_1, c_2). \\
(c_1 \in component \wedge c_2 \in component \wedge c_1 \neq c_2 \\
\quad \Rightarrow \\
((c_1, c_2) \in comp\_compatible \\
\quad \Leftrightarrow \\
\exists (int_1, int_2). \\
(int_1 \in interface \wedge int_1 \in comp\_interface(c_1) \wedge \\
int_2 \in interface \wedge int_2 \in comp\_interface(c_2) \wedge \\
\quad (int_1, int_2) \in compatible)))
\end{array}
$$

Intra-level component rules provide formal basis to check substitutability and compatibility between software components. To enable architecture analysis at any abstraction level, we need general-purpose rules that concern the whole software architecture. These rules are discussed in next section.

4.3.2 Intra-level architecture consistency

Chapter 2 enumerates five categories of architecture inconsistencies proposed by Taylor et al. [Taylor et al., 2009]: name, interface, behavioral, interaction and refinement inconsistencies.

Name and interface inconsistencies are related to the syntactic aspect of the architectural model. Detecting them is processed using syntactic and typing rules. Behavioral inconsistencies concern rather the behavior of the operations exposed by components. Verifying them requires the use of other rules involving the pre- and post conditions of operations. Interaction inconsistencies concern both the structural and behavioral aspects. Indeed, from a structural point of view a correct interaction requires that all required services are supplied and the architecture graph is connected. From a behavioral point of view, a correct interaction is ensured by a correct synchronization between component services invocation (*i.e.,* the flow of in/out calls). The behavioral verification comes usually after the syntactic one. Indeed, it makes no sense for instance to check the behavioral consistency between two components before ensuring that there is no type mismatches between their interfaces. Refinement inconsistencies concern the relations between different architecture abstraction levels. This kind of inconsistencies can be dealt with inter-level coherence rules presented in Section 4.4.2.

At this stage of work, we are confined to the syntactic and structural part of software architectures. We define architecture consistency as an ensemble of properties that must be checked within each abstraction level. These properties include name, interface and interaction inconsistencies. Therefore, we adapt their definition to the syntactic and structural aspect of software architectures and in accordance with the intra-level component rules.

Definition 1. *Name consistency*: This property ensures that each component belonging to the architecture holds a unique name and hence avoids conflicts when selecting and accessing components. It avoids component name inconsistencies. Formally, let *arch* be an architecture description at any abstraction level.

$name_{consistency} =$

$$\forall(c_1, c_2).(c_1 \in component \wedge c_2 \in component \wedge \{c_1, c_2\} \in arch\_components(arch) \\ \Rightarrow comp\_name(c_1) \neq comp\_name(c_2)$$

Figure 4.5 shows an example of inconsistency that can be detected using the *Name consistency* rule. There are two components named *C2* which may produce a conflict when *C1* invokes a service of one of them.

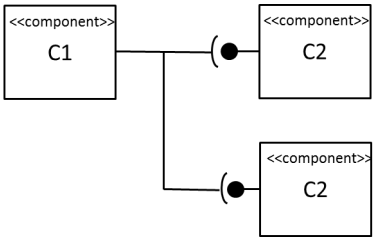


FIGURE 4.5: Example of name inconsistency

Definition 2. *Interface consistency*: There is interface consistency when all architecture connections are hold between a client interface and a server interface and satisfy compatibility between both sides (*i.e.,* a required interface is always connected to a compatible provided one). Ensuring this property avoids interface inconsistencies since no connection between two incompatible interfaces is permitted. Formally,

$interface_{consistency} =$

$$\forall(cl, se).(cl \in client \wedge se \in server \\ \Rightarrow \\ ((cl, se) \in connection \\ \Rightarrow \\ (serverInterfaceElem(se), clientInterfaceElem(cl)) \in int\_compatible)))$$

$interface_{consistency}$ uses the $clientInterfaceElem$ and $serverInterfaceElem$ functions defined in Section 4.2 as well as the interface compatibility Rule 5.

Figure 4.6 shows an example of interface inconsistency engendered by the incompatibility between the interfaces *IClockPos* and *IPosition*. Indeed, *IPosition* does not provide the *getLocation()* method required by *IClockPos*.

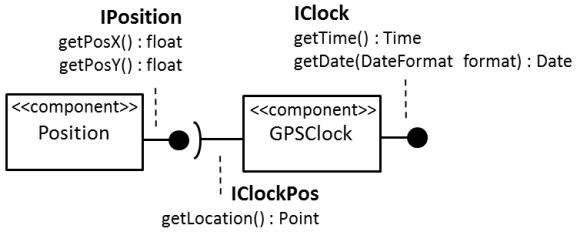


FIGURE 4.6: Interface inconsistency

Definition 3. *Interaction consistency*: This property ensures that (1) the architecture realizes its functional objectives (*i.e.,* components are able to soundly cooperate through their connected interfaces) and (2) the architecture definition is composed of a connected graph, so that no part of the architecture is isolated. Accordingly, we divide the Interaction consistency definition in two parts:

- All the architecture required interfaces are connected to compatible provided ones. Formally, let *arch* be an architecture description at any abstraction level,

$arch_{completness} =$

$\forall\, cl.(cl \in client \wedge cl \in arch\_clients(arch)$

$\Rightarrow$

$\exists\, conn.(conn \in connection \wedge conn \in arch\_connections(arch) \wedge cl \in dom(conn)$

- There is a path between every pair of components. A path can be defined as a closure of links where a link is a direct mapping between a client and a server. Unlike connections, links can exist between two clients or two servers and independently from orientation (*i.e.,* the server and client order does not matter). Formally, the link and a path definitions can be deduced from the connection definition as follows:

$link = connection \cup connection^{-1}$

To avoid cycles (*i.e.,* link with the same edge), identity relations are removed from the *link* relation closure:

$path = \mathbf{closure}(link) - \mathbf{id}(link)$

The graph connectivity property can then be defined as follows:

$graph_{connectivity} =$

$\forall(c_1, c_2, arch).$
$(c_1 \in component \wedge c_2 \in component \wedge arch \in architecture \wedge c_1 \neq c_2 \wedge \{c_1, c_2\} \subseteq$

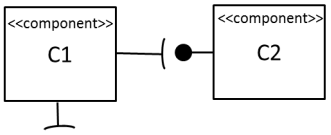
$$arch\_components(arch)$$
$$\Rightarrow$$
$$\exists\, p_1.(p_1 \in path \wedge c_1 = \mathbf{dom}(\mathbf{dom}(p_1)) \wedge c_2 = \mathbf{dom}(\mathbf{ran}(p_1))))$$

We note that p_1 is a relation between two links. In turn, a link is a relation between two server or client elements. To get the component element of each, **dom** and **ran** operators are accordingly applied where **dom** gives the domain of a relation and **ran** gives its image.

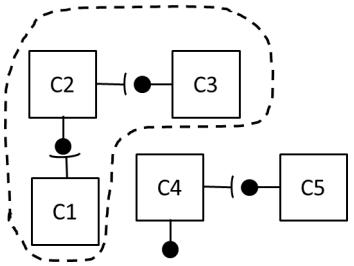
Finally, interaction consistency could be formally defined as follows:

$$interaction_{consistency} = arch_{completness} \wedge graph_{connectivity}$$

Figure 4.7 illustrates two examples of interaction inconsistencies that can be detected using *Interaction consistency* rule.



(A) incomplete architecture



(B) unconnected graph

FIGURE 4.7: Examples of interaction inconsistencies

The first case (*cf.* Figure 4.7a) is due to the required interface of *C1* left unconnected. The second case (*cf.* Figure 4.7b) is due to an unconnected graph (for instance, there is no path linking *C5* to *C2*).

Definition 4. Intra-level architecture consistency ensures the well-formedness of an architecture description at any abstraction level from a structural viewpoint. Architecture consistency is hence the conjunction of all the above properties:

$$architecture_{consistency} = name_{consistency} \wedge interface_{consistency} \wedge interaction_{consistency}$$

Intra-level rules are crucial to ensure the structural consistency of architecture descriptions at any abstraction level. These rules however are not sufficient to check that architecture definitions are coherent with each other at all abstraction levels, namely the specification, configuration and assembly levels related to CBSD process. Next section addresses the inter-level rules defined for such purpose.

4.4 Inter-level rules

Inter-level rules govern the relations between components and architecture definitions of different abstraction levels. Specifying inter-level rules is a crucial step to ensure coherence between architecture levels from specification to runtime. In order to go from an architecture specification to an architecture configuration, the architect must select suitable concrete component classes that realize the specified roles. The implementation can then be instantiated and deployed in multiple contexts. Inter-level rules set the basis to guide such reuse process and ensure the traceability of structural design decisions throughout the whole component-based software lifecycle. This section presents the inter-level rules.

4.4.1 Inter-level component rules

Inter-level component rules concern the relations between component of different abstraction levels as depicted in Figure 4.8 which recalls Figure 4.1 (right-hand side).

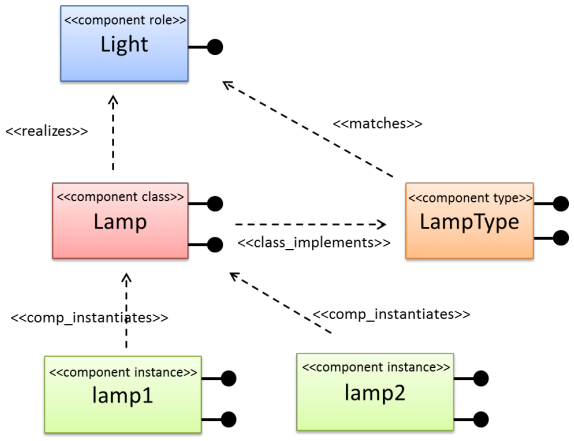


FIGURE 4.8: Inter-level component relations

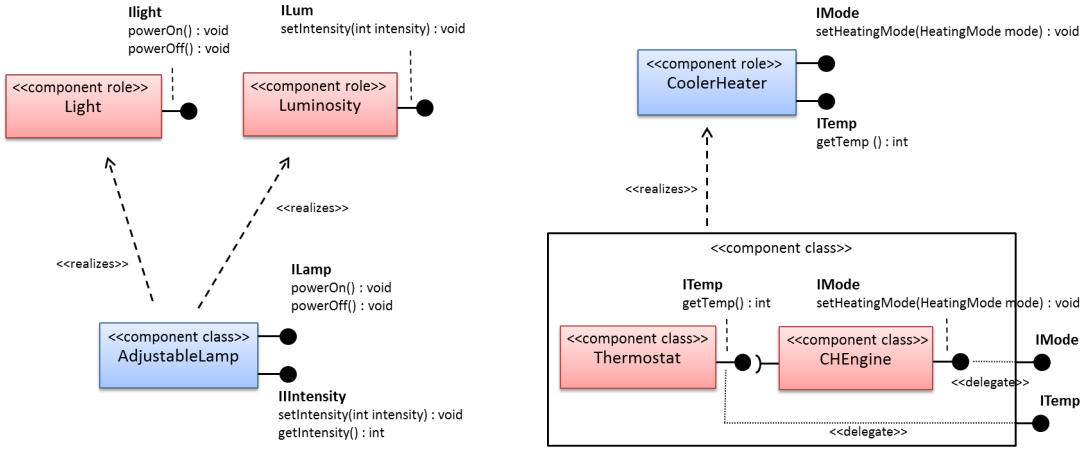
They include two main relations. The first is the realization relation between component roles and component classes (for instance the *Lamp* component class *realizes* the *Light* component role). This relation is conditioned by the matching between concrete component type, implemented by the component class, and the component role. The second relation holds between component instances and component classes (for instance, the *Lamp* component class has two component instances, *lamp1* and *lamp2*). In the remainder, we elucidate the semantics of these relations.

4.4.1.1 Relation between component classes and component roles

Component roles hold abstract and partial descriptions of software components. They specify exactly the functional requirements of the intended software. Component classes on the other

side, are reusable entities made available in component repositories to fit the largest possible category of component-based software. Finding a perfect match for a component role is thus mostly not obvious. A one-to-one mapping between a component role and a component class is not preferred in this case. To afford more flexibility for realizing component roles, a many-to many mapping is more promising. Indeed, a component class may be efficient to realize many component roles and a component role may be realized through the composition of many component classes. Figure 4.9 illustrates two possible realization cases according to the many-to-many matching relation:

$$realizes \in compClass \leftrightarrow compRole$$



(A) Multiple realizations (many to one) (B) Realization by composition (one to many)

FIGURE 4.9: Example illustrating two realization cases

The syntactic description of a component class is hold by the notion of concrete component type while the implementation details are hold by the component class itself. A concrete component type may hence be implemented in several ways by several component classes:

$$class\_implements \in compClass \rightarrow compType$$

Respecting the principle of component-based development, the implementation details must not be considered while selecting component classes stored in repositories. The concrete component type is checked instead to see whether the match holds.

Rule 7. A *cl* component class realizes a *cr* component role iff the *ct* concrete component type implemented by *cl* matches with *cr*. Formally:

$$
\begin{array}{l}
realizes \in compClass \leftrightarrow compRole \wedge \\
\forall(cl, cr).(cl \in compClass \wedge cr \in compRole \\
\Rightarrow \\
\quad ((cl, cr) \in realizes \\
\quad\quad \Leftrightarrow \\
\quad \exists\, ct.(ct \in compType \wedge (ct, cr) \in matches \wedge \\
\quad (cl, ct) \in class\_implements)) \\
)
\end{array}
$$

The syntactic matching between a component type and a component role is proceeded by comparing the interfaces of both. We consider that a concrete component type matches with a given component role if the substitutability rule between both holds.

Rule 8. A *ct* component type matches with a component role *cr* iff *ct* is subtype of *cr*:

$$
\begin{array}{l}
matches \in compType \leftrightarrow compRole \wedge \\
\forall(ct, cr).(ct \in compType \wedge cr \in compRole \\
\Rightarrow \\
((ct, cr) \in matches \\
\quad \Leftrightarrow \\
(ct \in comp\_substitution[\{cr\}]) \\
))
\end{array}
$$

Rule 8 calls the component substitutability rule (*cf.* Rule 4) to check that the concrete component type is subtype of the component role. As component roles are abstract component types, subtyping enables to refine them into concrete component types. This way, subtyping relation preserves matching, *i.e.,* when a component class is substituted in the configuration, the matching with the realized component roles is preserved.

To illustrate component realization rule, we retake the example shown in Figure 4.9a. The *AdjustableLamp* component class realizes the *Light* component role since the *ILamp* interface substitutes for the *ILight* interface. The same condition holds between *AdjusatbleLamp* and *Luminosity* since the *IIntensity* interface substitutes for the *ILum* interface.

4.4.1.2 Relation between component instances and component classes

The relation between component classes and component instances is analogous to the relation between classes and objects in object-oriented paradigm. A component instance is related to exactly one component class and a component class may have zero to many component instances. This relation can then be formalized by a total function mapping the set of component classes to the one of component instances:

$$comp\_instantiates \in compInstance \rightarrow compClass$$

Some architectures may require a limited number of component instances per component class. For instance, in client-server architectures, the number of connected client components may be restricted due to the limited capacity of the server component. This relation can be formalized as a total function mapping a component class to a natural integer:

$$nb\_instances \in compClass \rightarrow \mathbf{NAT}$$

Component instances may also be parametrized according to the attributes of the component class. The values of these attributes represent the state of the component instance.

$$instance\_state \in compInstance \leftrightarrow \mathcal{P}(attributeValue)$$
$$attributeValue \in attribute \rightarrow value$$

To map the component instance state to its corresponding component class attributes, the *instance_state* relation is enforced by the following predicate:

$$\forall(ci, state).(ci \in compInstance \wedge state \in \mathcal{P}(attribute\_value) \wedge (ci, state) \in instance\_state$$
$$\Rightarrow$$
$$\forall\, att.(att \in attribute \wedge att \in dom(state) \Rightarrow$$
$$\exists\, cl.(cl \in compClass \wedge (ci, cl) \in comp\_instanciates \wedge att \in class\_attributes(cl))))$$

Inter-level component rules give formal basis to the relations hold between the different Dedal components (*i.e.,* component roles, component classes and component instances). On the basis of these relations, we define the inter-level architecture coherence.

4.4.2 Inter-level architecture coherence

Architecture coherence is defined between two adjacent abstraction levels. It is used to check the conformity between two architecture descriptions at two adjacent abstraction levels. Coherence is important to determine whether design decisions are maintained across all the software development steps. Verifying architecture coherence helps to detect architectural inconsistencies such as erosion and prevents software from degradation. Inter-level coherence definition may be confusing in the sense that it seems to match with the notion of refinement consistency given by Taylor *et al.* (*cf.* Chapter 2). Indeed, both definitions address the preservation of design decisions while moving from an abstraction level to another. However, we consider that refinement is complementary to inter-level architecture coherence. Indeed, each of the three architecture descriptions addressed by the coherence definition may itself be

the result of several architecture refinements. Inter-level coherence is more specific than the refinement notion since it addresses only the relations between the three main architecture descriptions of component-based software.

Inter-level coherence rules are depicted in Figure 4.10.

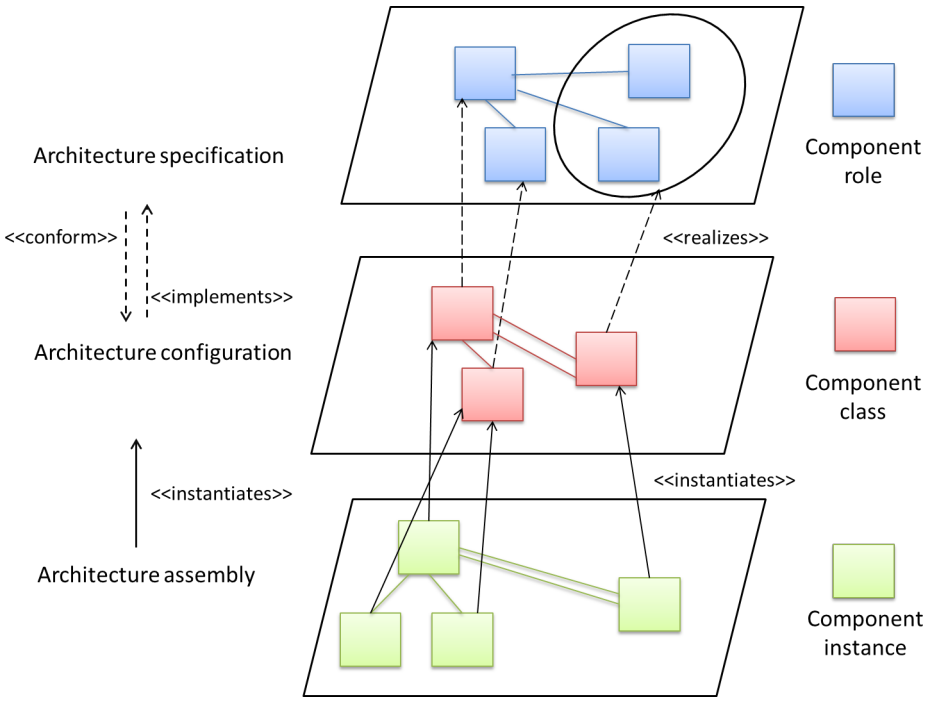


FIGURE 4.10: Relations between architecture abstraction levels

Inter-level coherence rules include the relation between specification and configuration levels and the relation between the configuration and assembly levels. In the remainder, we elucidate the semantics of these relations.

4.4.2.1 Relation between specification level and configuration level

As discussed in Chapter 2, one of the issues of CBSD is the difficulty to find suitable software components that fulfill requirements. The reason is that the link between the two steps is not elucidated. An architecture specification is a description that captures functional software requirements within abstract component types (*i.e.*, component roles). These abstract descriptions provide a guide to search for suitable component classes needed to implement the software (using the Rule 7). To ensure that design decisions are preserved in both specification and configuration architectures, coherence between them has to be checked. Two conditions must be met to satisfy this coherence:

- all component roles from the specification are realized by component classes in the configuration. This results in a many-to-many relation as several component roles may be

realized by a single component class and a composition of component classes (composite component class) may be needed to realize a single role.

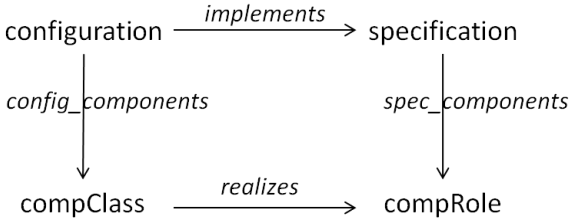
Formally:

$$
\begin{array}{l}
implements \in configuration \leftrightarrow specification \wedge \\
\forall (Conf, Spec).(Conf \in configuration \wedge Spec \in specification \\
\quad \Rightarrow \\
\qquad (Conf, Spec) \in implements \\
\qquad\quad \Leftrightarrow \\
\quad \forall\, cr.(cr \in compRole \wedge cr \in spec\_components(Spec) \Rightarrow \\
\qquad \exists\, cl.(cl \in compClass \wedge cl \in config\_components(Conf) \wedge \\
\qquad\quad (cl, cr) \in realizes)))
\end{array}
$$

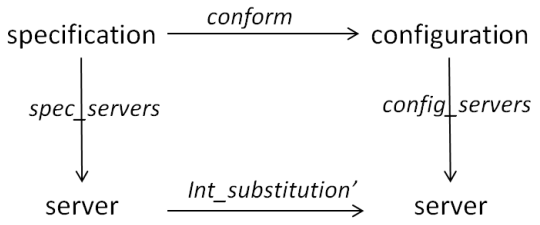
- each connected provided (*server*) interface in the configuration is included in the specification. This prevents having a configuration that includes an extra set of functionalities that are not specified at the higher level (increasing hence the risk of software erosion). Formally:

$$
\begin{array}{l}
conform \in specification \leftrightarrow configuration \wedge \\
\forall (Conf, Spec).(Conf \in configuration \wedge Spec \in specification \\
\quad \Rightarrow \\
\qquad (Spec, Conf) \in conform \\
\qquad\quad \Leftrightarrow \\
\quad \forall\, se_1.(se_1 \in server \wedge se_1 \in config\_servers(Conf) \Rightarrow \\
\qquad \exists\, se_2.(se_2 \in server \wedge se_2 \in spec\_servers(Spec) \wedge \\
\qquad\quad (serverInterfaceElem(se_1), serverInterfaceElem(se_2)) \in int\_substitution)))
\end{array}
$$

The coherence between the specification and configuration levels involves the following relations



and,



where $(se_1, se_2) \in int\_substitution' \Leftrightarrow ((serverInterfaceElem(se_1), serverInterfaceElem(se_2)) \in int\_substitution)$

4.4.2.2 Relation between configuration level and assembly level

An architecture configuration may be instantiated in several ways and deployed in multiple platforms. An architecture assembly is a representation of the software at runtime (after its deployment). Elucidating the link between both levels is crucial to keep track of software change and avoid architecture inconsistencies. Coherence between assembly and configuration is satisfied if every component class (*resp.* connected provided interface) of the configuration is instantiated at least once in the assembly and, every component instance (*resp.* connected provided interface instance) of the assembly is instance of a component class (*resp.* connected provided interface) of the configuration. Formally:

An architecture assembly Asm instantiates an architecture configuration $Conf$ iff:

$$
\begin{array}{l}
instantiates \in assembly \rightarrow configuration \\
\forall (Asm, Conf).(Asm \in assembly \wedge Conf \in configuration \\
\Rightarrow \\
((Asm, Conf) \in instantiates \\
\quad \Leftrightarrow
\end{array}
$$

- every component class cl of $Conf$ is instantiated at least once by a component instance ci in Asm and every component instance ci in Asm is an instance of a component class in $Conf$:

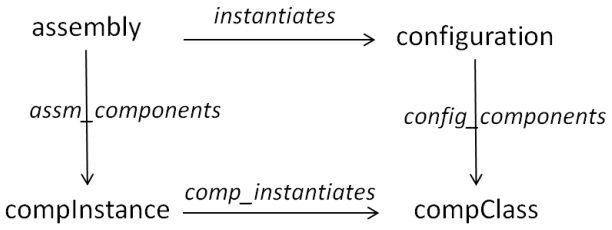
$$
\begin{array}{l}
\forall\, cl.(cl \in compClass \wedge cl \in config\_components(Conf) \\
\quad \Rightarrow \\
\quad \exists\, ci.(ci \in compInstance \wedge ci \in assm\_components(Asm) \wedge \\
\quad (ci, cl) \in comp\_instantiates) \wedge \\
\forall\, ci.(ci \in compInstance \wedge ci \in assm\_components(Asm) \\
\quad \Rightarrow \\
\quad \exists\, cl.(cl \in compClass \wedge cl \in config\_components(Conf) \wedge \\
(ci, cl) \in comp\_instantiates)))
\end{array}
$$

- and, it exists a surjective function (f_s) between the set of connected assembly servers ($assm\_servers(Asm)$) and the set of connected configuration servers ($config\_servers(Conf)$) such that for each se_1 connected server instance, $f_s(se_1)$'s interface has the same type as se_1's interface and the se_1's component instance element ($serverInstElem(se_1)$) is an instance of $f_s(se_1)$'s component class element ($serverClassElem(f_s(se_1))$):

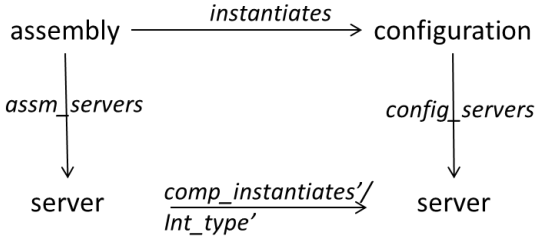
$$
\begin{array}{l}
\wedge\, \exists\, f_s.(f_s \in assm\_servers(Asm) \overset{surj.}{\rightarrow} config\_servers(Conf) \wedge \\
\quad \forall\, se_1.(se_1 \in server \wedge se_1 \in assm\_servers(Asm) \Rightarrow \\
\quad\quad (int\_type(serverInterfaceElem(f_s(se_1))) = int\_type(serverInterfaceElem(se_1))) \wedge \\
\quad\quad (serverInstElem(se_1), serverClassElem(f_s(se_1))) \in comp\_instantiates) \\
))
\end{array}
$$

The surjection condition imposes the one-to-many relation between a configuration servers and assembly servers. This condition also applies to the relation between component classes and component instances an that we express in a different (but equivalent) manner in the first point.

To summarize, this rule involves the following relations:



and,



where $(se_1, se_2) \in comp\_instantiates' \Leftrightarrow ((serverInstElem(se_1), serverClassElem(se_2)) \in comp\_instantiates)$ and $int\_type' = int\_type \circ serverInterfaceElem$

Global coherence. There is a global coherence when all architecture descriptions at all abstraction levels are coherent with each other. Let $specConfigCoherence$ be the coherence condition between the specification and configuration level and $configAssemblyCoherence$ be the coherence condition between the configuration and assembly levels. The global coherence is defined as the conjunction of the these two conditions:

$Global\_Coherence = specConfigCoherence \wedge configAssemblyCoherence$

4.5 Conclusion

This chapter presented the formal grounds of Dedal as well as a type theory to support the structural analysis of software architecture descriptions at the three main steps of component-based development (*i.e.,* specification, implementation and deployment). The formalization of Dedal using the B modeling language offers a way to automate the analysis activity in a transparent manner. The three-level type theory sets the basic rules crucial to enable the analysis of Dedal architecture descriptions. While only the syntactic level is taken into account, the type theory provides the basis to reason about more challenging aspects out of the scope of this thesis, notably the behavioral one. The structural analysis addresses two main architecture properties characterizing the component-based development process: the consistency of architecture descriptions at any abstraction level and the coherence between all those descriptions. The next step is to enable an automatic evolution management of the three architecture descriptions guaranteeing the consistency and coherence properties.

Chapter 5

A formal approach to three-level software architecture evolution

Managing software evolution in component-based software development processes is a complex task. Indeed, software architectures must support change at any step of component-based development to meet new user needs, improve software quality, or cope with component failure. As discussed in Chapter 3, existing approaches to architecture-centric evolution hardly deal with this issue. They hardly cover the three main steps of CBSD, necessary to keep the traceability of design decisions. This shortcoming often leads to architecture erosion which in turn leads to software degradation. Dedal (*cf.* Chapter 2) presents a convenient way to model software architectures at the three main steps of the component-based software lifecycle (*i.e.*, design, implementation and runtime). In the previous chapter, we defined the semantics of Dedal as well as the links between its three architecture levels. While this contribution provides a formal ground to support architecture analysis, it still lacks foundations and mechanisms to support evolution management throughout the whole component-based software lifecycle. This chapter discusses such proposal. Section 5.1 presents the proposed evolution management model based on Dedal and Section 5.2 presents the approach undertaken to automate the evolution process.

Contents

5.1 The evolution management model 86
5.1.1 The architectural model (Group 1) 86
5.1.2 The architectural change (Group 2) 90
5.1.3 The Evolution Manager System (Group 3) 91
5.1.4 Synthesis 95
5.2 Evolution management approach 95
5.2.1 The overall approach 96

5.2.2 The search algorithm 97
5.3 Conclusion 101

5.1 The evolution management model

Architecture evolution management is a complex task. Indeed, many criteria should be taken into account while evolving software architectures. For instance, the initial level of change and its impact on the other abstraction levels, the properties to be preserved when applying change or the available operations used to apply the change. To make all these criteria explicit, we propose an evolution management model [Mokni et al., 2015] (*cf.* Figure 5.1) that encompasses all the elements involved in the evolution management activity. The interest of the evolution management model is twofold. First, it simplifies the evolution management task by separating the evolution management concern from the architectural modeling concern. Second, it helps understanding and analyzing the architectural change by representing changes as first class entities at the same level as the architectural model.

The evolution management model consists of three main parts:

- **The architectural model** (meta-classes of group 1 in Figure 5.1) is the target of change. Since the objective is to manage evolution throughout all the stages of component-based development, the target is architectural models derived from Dedal (*cf.* Dedal meta-model in Figure 6.2).
- **The architectural change** (meta-classes of group 2 in Figure 5.1) gathers all the necessary information about the change such as its origin and abstraction level.
- **The evolution manager** (meta-classes of group 3 in Figure 5.1) is the system that captures the change request and tries to find a solution to evolve the architectural model with respect of all the pre-defined conditions.

Each part of the evolution management model is detailed through the following sections.

5.1.1 The architectural model (Group 1)

The architectural model is an abstract representation of the software system. From an evolution point of view, the architectural model provides a support to reason about the change and to simulate it before effectively applying it on the software system. The objective of this thesis is to manage evolution throughout all the stages of component-based development. Therefore, we consider three-level architectural models derived from Dedal since it supports well

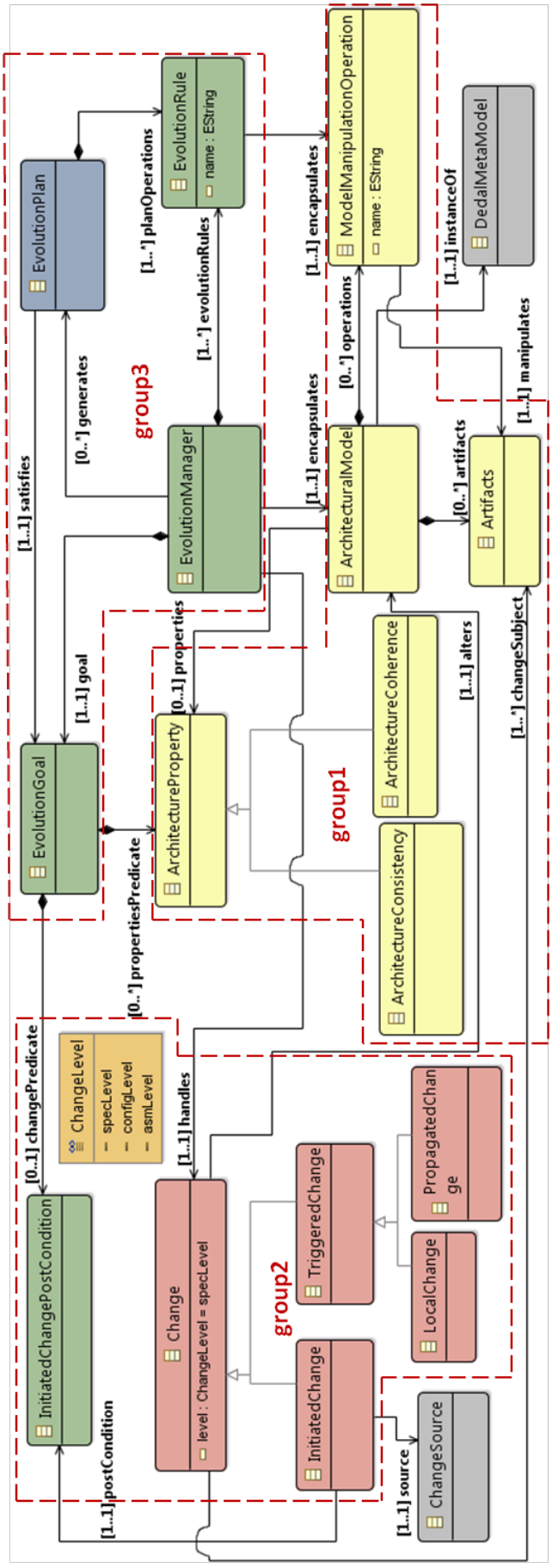


FIGURE 5.1: The evolution management model (ecore)

the three main steps of component-based development. Consequently, the target architectural models are constituted of three architecture descriptions conform to Dedal (*i.e.,* architecture specification, architecture configuration and architecture assembly).

5.1.1.1 Artifacts subject to change

Artifacts are the elements of architectures that are subject to change. At specification level, artifacts are component roles and connections between them. At configuration level, artifacts are component classes, server interfaces (provided interfaces to be connected) and connections. And, at assembly level, artifacts are component instances, server interface instances and bindings. As they represent dependencies, client interfaces are not dissociated from component artifacts and must be connected in all cases (at the difference of server interfaces that represent functionalities and can be left unconnected).

5.1.1.2 Architecture properties

Architecture properties refer in general to all the properties that must be verified during the architectural analysis activity to guarantee that the system works correctly and respects all its requirements. These properties also have to be maintained when evolving the software system.

Architecture properties could be general-purpose like intra-level architecture consistency rules defined in chapter 4 or specific properties related for instance to an architecture style (*e.g.,* pipe and filter style) [Garlan and Shaw, 1994]. In this thesis, we consider two main properties that we judge crucial to control multi-level architecture evolution: intra-level architecture consistency and inter-level architecture coherence.

5.1.1.3 Model manipulation operations

Model manipulation operations are elementary operations that manipulate the artifacts of the architectural model (*e.g.,* addition, deletion and replacement). A model manipulation operation is composed of four parts:

- a signature that defines the operation name and states its arguments,
- pre-conditions related to the architectural model (*e.g.,* a precondition checks if substitutability between two components holds),
- actions that update a set of variables related to the architectural model (*e.g.,* the set of components belonging to an architecture),

- and post-conditions that describe the desired effect of actions after their execution. Post-conditions are not explicitly defined in the operation but they are deduced from the operation actions.

Table 5.1 summarizes the model manipulation operations according to the type of the operation, abstraction levels supporting the operation, arguments, pre-conditions and post-conditions. Operations are specified using the B modeling language and we use the same notational guidelines defined in Chapter 4 to represent architectural elements.

Operation type	**Level**	**Arguments**	**Pre-condition**	**Post-condition**
component addition	specification, configuration	arch, newComp	$newComp \notin arch\_components(arch)$	$newComp \in arch\_components(arch)$
component deletion	specification, configuration and assembly	arch, comp	$comp \in arch\_components(arch)$	$comp \notin arch\_components(arch)$
component substitution	specification, configuration and assembly	arch, oldComp, newComp	$oldComp \in arch\_components(arch) \land newComp \notin arch\_components(arch) \land (oldComp, newComp) \in comp\_substitution$	$oldComp \notin arch\_components(arch) \land newComp \in arch\_components(arch)$
adding connection	specification, configuration and assembly	arch, cl, se	$cl \in arch\_clients(arch) \land se \in arch\_servers(arch) \land (cl \mapsto se) \notin arch\_connections(arch) \land (interfaceElem(se), interfaceElem(cl)) \in int\_compatible$	$(cl \mapsto se) \in arch\_connections(arch)$
removing connection	specification, configuration and assembly	arch, cl, se	$cl \in arch\_clients(arch) \land se \in arch\_servers(arch) \land (cl \mapsto se) \in arch\_connections(arch)$	$(cl \mapsto se) \notin arch\_connections(arch)$
server addition	configuration and assembly	arch, se	$se \in server \land se \notin arch\_servers(arch)$	$se \in arch\_servers(arch)$
server deletion	configuration and assembly	arch, se	$se \in server \land se \in arch\_servers(arch) \land \forall cl.(cl \in client \Rightarrow (cl \mapsto se) \notin arch\_connections(arch)$	$se \notin arch\_servers(arch)$
component instance deployment	assembly	assm, newInstance, compClass	$newInstance \in instantiates(compClass) \land newInstance \notin assm\_components(assm) \land nbInstance(compClass) \leq maxInstance(compClass)$	$newInstance \in assm\_components(assm)$

TABLE 5.1: Summary of model manipulation operations

Component addition is the same for all abstraction levels except for the assembly which is component instance deployment instead. At assembly level, component instance deployment takes an extra argument which is the component class to be instantiated and its precondition is enforced with the maximum number of allowed instances. Component deletion, substitution, connection and disconnection operations follow the same schema at all abstraction levels. Component substitution imposes that the new component is a subtype of the replaced one according to the substitutability rule (*cf.* Chapter 4, *Rule* 4). Adding a connection requires that the server interface and the client interface elements are compatible and are not already connected. We note that server addition and deletion are not supported by the specification level since its definition (*i.e.,* specification architecture) imposes that all interfaces must be

connected to hold interaction consistency. Indeed, at specification level, what is described is exactly what is required whereas at configuration level, what is described is what is reused to realize the requirements. This entails that at most all the proposed implemented functionalities will be reused and not necessary all of them. Server addition and deletion operations are thus used for this purpose.

The complete formalization related to model manipulation operations at specification, configuration and assembly levels can be respectively found in Appendix A, Section A.3, Section A.4 and Section A.5.

5.1.2 The architectural change (Group 2)

Architectural changes characterize all the modifications that alter the software architecture. They meet new requirements to keep software up-to-date or arise due to an environmental change (*e.g.,* lack of resources, or faults). Software architectures are subject to change at any abstraction level of component-based development. The impact of change may affect the other abstraction levels. All of these facts about software change make its handling a non trivial task.

5.1.2.1 Change origin

There are two types of **change origin**: *initiated change* and *triggered change*. Initiated change has an external source. It may be originated from user action or from the execution environment. Triggered change is internal and is induced by the evolution manager to reestablish the architecture consistency at the same abstraction level (*local change*) and/or preserve coherence between all architecture descriptions at other abstraction levels (*propagated change*). In multi-level evolution, propagated change can be *bottom-up* when it is propagated to the higher abstraction levels, *top-down* when it is propagated to the lower abstraction levels or *mixed* when it is propagated to both lower and higher levels.

5.1.2.2 Change level

Change may be initiated at any step of the component-based software lifecycle.

Change at specification level. Change at specification level is usually a response to a new software requirement. For instance, the architect may need to add new functionalities to the system by adding new roles to the architecture specification. Enabling change at architecture specification level makes CBSD process similar to agile methods (*cf.* Chapter 2) where software

increments can be specified and implemented easily and rapidly in the development process. A specification may also be modified due to a change on its lower configuration level (through change propagation) to be adapted to the software implementation.

Change at configuration level. Change is usually initiated at configuration level due to the need to upgrade to new versions of component classes or to adapt the implementation to a specific operating system before deployment. An implementation may also be impacted by change propagation either from the specification level, in response to new requirements, or from the assembly level, in response to a dynamic change of the system.

Change at assembly level. Several kinds of change may occur at runtime. For instance, dynamic software change may be needed due to a change in the execution context (*e.g.,* lack of resources, mobility).Dealing with unanticipated changes is one of the most important issues in software evolution. Indeed, some software systems have to be self-adaptive to keep providing their functions despite environmental changes.

5.1.2.3 Change subject

It designates the artifacts subject to change. This information is useful to identify the elements that have to be manipulated during the evolution process.

5.1.3 The Evolution Manager System (Group 3)

The evolution manager is the system in charge of handling the architectural change throughout the whole software lifecycle. It captures software change, controls its impact on the local abstraction level to preserve architecture consistency and to propagate it to the other abstraction levels to keep all architecture descriptions coherent. The state of the architecture model is analyzed and evolved through a formal evolution manager model. The latter contains all the definitions of *architecture properties* and the *evolution rules* required by the Evolution Manager System (EMS) to generate *evolution plans* satisfying a given *evolution goal*. The generation process is performed using a solver. Figure 5.2 describes the workflow of the EMS.

5.1.3.1 The evolution manager machine

Chapter 4, Section 4.2 presents the B formal models of Dedal. These models however, are not sufficient to support evolution management and to enable the analysis of architectural change

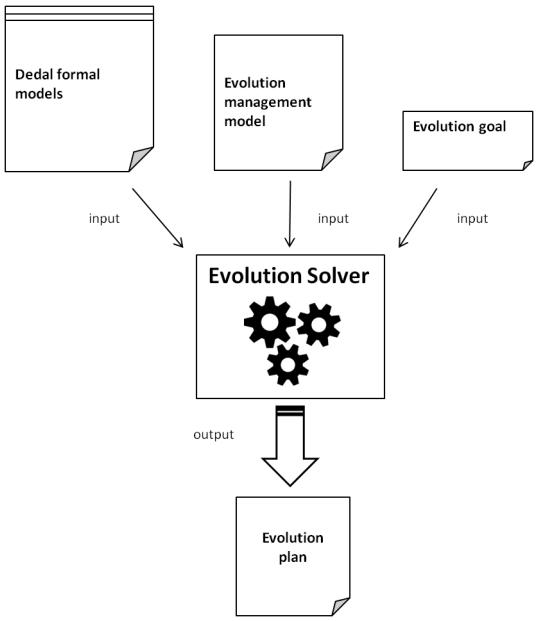


FIGURE 5.2: EMS workflow

across all the three abstraction levels. Therefore, we propose a B model called *EvolutionManager* that includes the concepts required for evolution management. Table 5.2 presents the schema of the evolution manager machine. The full evolution manager machine specification can be found in Appendix B.

MACHINE
EvolutionManager
INCLUDES
Arch_specification, *Arch_configuration*, *Arch_assembly*
SETS
/*Enumerated set to designate the level of change*/
$CHANGE\_LEVEL = \{eLevel, specLevel, configLevel, asmLevel\}$
VARIABLES
/*Variable to designate the level of change*/
changeLevel,
/*Variables used to store the history of manipulated artifacts (*i.e.*, added and deleted artifacts)*/
/*Boolean variables used to initialize the included machines*/
DEFINITIONS
/*Macros for the preconditions of the model manipulation operation*/
/*Predicates defining consistency and coherence properties*/
spec_consistency == specification consistency predicate
config_consistency == configuration consistency predicate
assm_consistency == assembly consistency predicate
specConfigCoherence == coherence between specification and configuration
configAssemblyCoherence == coherence between configuration and assembly
INITIALISATION
/* *changeLevel* **is initialized to *eLevel*,**
Initialization of boolean variables,
history sets are initialized to empty set */
OPERATIONS
/*Initialization operations*/
/*Architecture model setter*/
mng_setTargetArchitectures(*spec*, *config*, *assm*) = ...
/*Change level setter*/
setChangeLevel(*newChangeLevel*) = ...
/*Evolution rules*/
END

TABLE 5.2: The *EvolutionManager* machine

The evolution manager machine includes the following features:

- The *arch_specification*, *arch_configuration* and *arch_assembly* machines corresponding respectively to the architecture specification, architecture configuration and architecture assembly (*INCLUDES* clause in Table 5.2).
- The level of change (*changeLevel*). This variable is typed with the enumerated set *CHANGE_LEVEL*. It indicates the current level of change during the evolution process. Its value is initialized to *eLevel* and set to the current level of change through the *setChangeLevel* operation.
- Macros related to the architectural model such as the pre-conditions of the model manipulation operations (*cf.* Appendix B for details).
- Architecture properties that have to be preserved before and after the evolution process (*cf.* Section 5.1.1.2).
- Initialization operations. These operations are used to calculate the compatibility and substitutability between interfaces and components, the matching between component roles and component classes and the set of all possible connections at each abstraction level. Initialization is crucial to enable the evaluation of the operation pre-conditions during the evolution process. For instance, replacing a component by another requires to check if substitutability between them holds.
- Evolution rules. (They are defined in the next section).

5.1.3.2 Evolution rules

Evolution rules are specific operations that encapsulate model manipulation operations. There is an evolution rule for each model manipulation operation and it takes the same arguments as the embedded operation. Evolution rules manage and control access to model operations using pre-conditions related to the level of change and the history of manipulated artifacts. These pre-conditions act as a primary filter to model manipulation operations. For instance, the change level pre-condition restricts the access of the EMS to operations related only to the current level of change. History about manipulated artifacts is used to avoid redundant unnecessary transitions that may decrease the efficiency of the EMS. Particularly, deleting and adding the same artifact several times is unworthy during the evolution process. History consists of two sets: one for added artifacts and the other for deleted ones. Whenever an artifact is added (respectively deleted), the corresponding set is updated. It then becomes possible to inhibit the execution of its inverse operation. Evolution rules also inform the EMS about the artifacts that have to be manipulated after the last executed change operation.

This information is used as a heuristic to increase the efficiency of the EMS as discussed in the remainder (*cf.* Section 5.2.2.3).

Table 5.3 presents the schema of an evolution rule.

$output \leftarrow$ **evolutionRuleName**($targetArchitecture$, $artifacts$) =
PRE
$initialization = true \wedge$
$changeLevel = currentChangeLevel \wedge$
$artifacts \notin addedArtifacts \cup deletedArtifacts \wedge$
$manipulationOperationPrecondition$
THEN
/* execute manipulationOperationName(targetArchitecture, artifacts), update the sets of added artifacts and deleted artifacts and, set the value of output parameters */
END

TABLE 5.3: The schema of an evolution rule

The pre-condition of an evolution rule (**PRE**) stipulates that:

- the model should be initialized (*i.e.,* all relations between artifacts are calculated),
- the evolution is applicable at the current level of change,
- artifacts concerned by change have not been manipulated before (*i.e.,* added or deleted) and,
- the pre-condition of the embedded manipulation operation is satisfied.

When satisfied, this precondition enables to execute the embedded manipulation operation. Additionally, the sets of added and/or deleted artifacts are updated as well as the output parameters (output) to be analyzed by the EMS.

Table 5.4 presents an example of evolution rule applicable at configuration level.

$changeArtifact \leftarrow$ **config_addClass**($config$, $newClass$) =
PRE
$initialisation =$ **TRUE** $\wedge$
$changeLevel = configLevel \wedge$
$classAdditionPrecondition \wedge$
$newClass \notin (deletedClasses \cup addedClasses) \wedge$
$selectedConfig = config$
THEN
addClass($config$, $newClass$) ||
$addedClasses := addedClasses \cup \{newClass\}$ ||
$changeArtifact := class\_implements(newClass)$
END;

TABLE 5.4: Evolution rule at configuration level

The rule adds a new component class to a target configuration ($config$). The output parameter is set to the type of the added component class ($class\_implements(newClass)$). This information indicates to the EMS that the next preferred operations are those involving the elements of the new added component class (*e.g.,* connecting some of its interfaces). The full list of evolution rules is detailed in Appendix B.

5.1.3.3 Evolution goal

The evolution goal defines all the conditions that must be satisfied by the architectural model after the change. When dealing with local change, the evolution goal is set to the consistency property related to the level of change and the post-condition of the initiated change. When dealing with propagated change, it is set to coherence with adjacent level in addition to consistency at the current change level.

5.1.4 Synthesis

In this section, we defined and formalized the concepts underlying evolution management in multi-level software architectures. The identified concepts relate three orthogonal dimensions: the architectural model, the change that may affect it and the system in charge of handling this change. The formalization results in an evolution manager model including the architecture formal models as well as a set of evolution rules that control the change on these models. It is henceforth possible to put all these concepts into an automated process that manages the evolution of component-based software architectures.

5.2 Evolution management approach

In the previous section, we identified and defined the concepts related to multi-level architecture evolution management. In this section, we present our approach to deal with architecture evolution in reuse-centered, component-based development processes. This approach has the following objectives:

- To capture architectural change at any of the three main stages of component-based software lifecycle (*i.e.,* design, implementation and runtime).
- To control the impact of change where it is initiated by reestablishing the architecture consistency if altered.
- To propagate the change to the other abstraction levels in order to restore the global coherence of the architecture descriptions if altered.

In the remainder, we give an overview of the evolution approach and then we get into the details of the evolution-solving process.

5.2.1 The overall approach

Evolution management starts when a change is requested somewhere in the architectural model. For instance, a component class addition is requested in the configuration architecture. The evolution manager captures change and initiates it, by running adequate evolution rules on the corresponding formal model. It then runs a resolution algorithm that attempts to find a sequence of evolution rules leading to a consistent state of the architecture. The same algorithm is applied to the other levels given the adequate evolution goal to restore global coherence if it is necessary. If a solution is found, the evolution manager generates the corresponding evolution plan that can then be committed by the user. In the other case (*i.e.,* failure), the evolution manager rejects the change request. Figure 5.3 shows the activity diagram corresponding to the multi-level evolution management process.

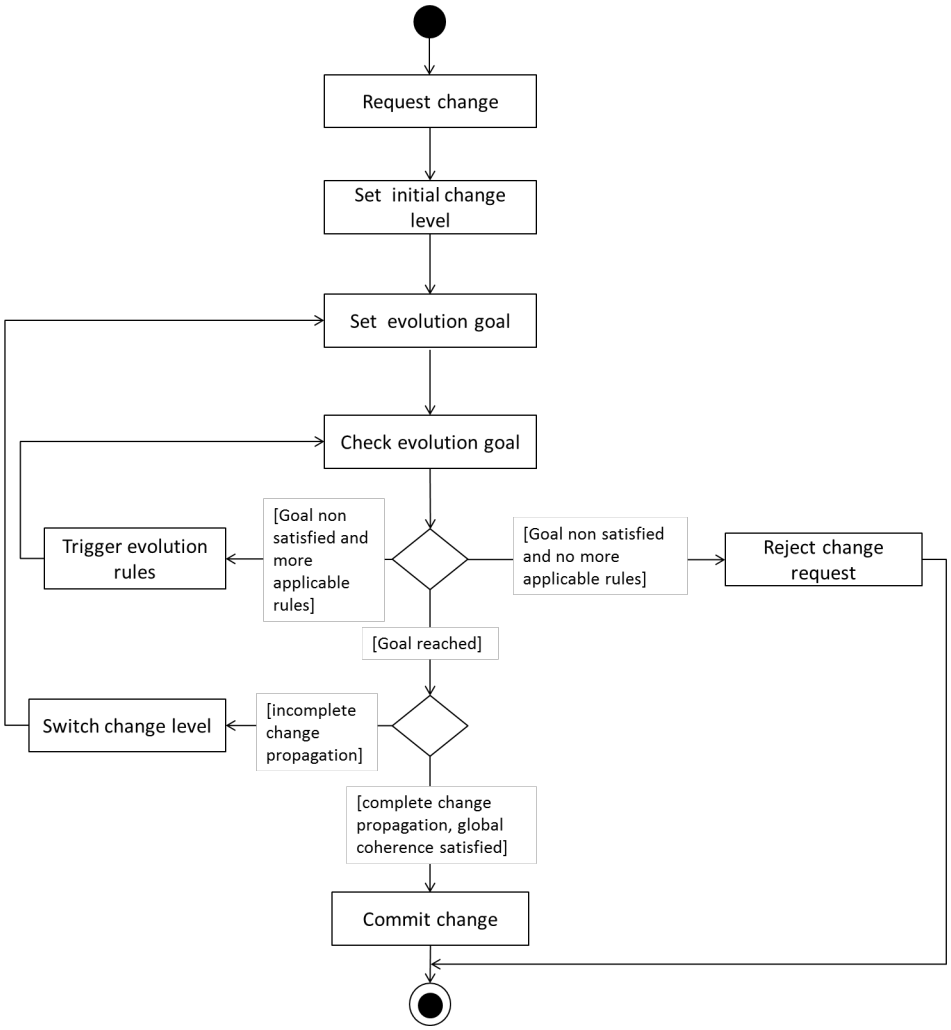


FIGURE 5.3: Multi-level evolution management process

5.2.2 The search algorithm

The problem of evolution management can be seen as follows: considering an initiated change on the architectural model we would like to find a sequence of evolution rules that reaches the evolution goal. The latter is defined in terms of consistency and coherence properties and the post-condition of the requested change.

5.2.2.1 Findings

The above problem can be categorized as a classical planning problem in the area of Artificial Intelligence [Russell and Norvig, 2010]. In classical planning, the initial state, goal state and a number of actions with pre-conditions and effects are provided. The plan is a sequence of actions that transform the initial state to the goal state when executed one after the other. Planning problems can be automatically solved using explicit propositional or relational representations of states and actions and algorithms operating on that representations. General planners with competitive performances are proposed to solve planning problems for many domains. The most prominent ones are those supporting the Planning Domain Definition Language (PDDL) [McDermott, 1998]. An alternative to general planners is to develop specialized heuristic solvers. The idea is to take advantage of specific information known about the problem to avoid exploring less promising paths and improve the performance of the search algorithm. We opt for the second alternative for the following reasons:

- Using heuristics usually shows better results than general approaches in resolving NP-Complete problems [Bonet and Geffner, 2001].
- In our case, implementation in our case is quite simple since the Dedal formal model is specified in B which can be manipulated using the ProB API [Leuschel and Butler, 2008].
- B supports practically the same constructs as PDDL except some notions like metric and durative actions [Fox and Long, 2003] that we do not need at this stage of work.
- This also make it possible to experimentally compare with other approaches (notably general ones) for evaluation and improvement.

5.2.2.2 Issues related to establishing the search algorithm

Three questions arise when reasoning about developing an efficient evolution solver:

- Q1- Which search strategy is more efficient in our case?

- Q2- How to avoid redundant transitions during the search process?
- Q3- What specific heuristics can be defined to improve the search strategy?

Answer to Q1: Forward search [Nau et al., 2004] is an intuitive strategy to resolve a planning problem. It starts from the initial state and applies iteratively and non deterministically an enabled operation until reaching the goal state or terminating after exploring all possibilities. However, this strategy does not guarantee a good performance to find the goal state especially when we are confronted to large state space. This is the reason why the use of effective heuristics is highly recommended. Therefore, Section 5.2.2.3 defines specific heuristics to improve the forward search.

Answer to Q2: Redundant transitions are forbidden using pre-conditions that inhibit the call of the same operation (with the same arguments) more than once. Concretely, addition and deletion operations (related to the same arguments) are mutually exclusive. Hence, once executed, an operation cannot be canceled. Backtracking is used instead in the case where the resulting transitions do not lead to the goal state and other directions should be explored.

5.2.2.3 Defining search heuristics

Answer to Q3: Forward search applies non deterministically an enabled operation to explore the state space. Using information about the change request and the last executed operation determinates which operation should be applied at the current state or at least which operations are more promising than others.

The artifact-oriented heuristic The idea of artifact-oriented heuristic is to prioritize the operations manipulating the artifacts that are more likely to satisfy the evolution goal (thereafter called the main artifacts). For instance, adding a new component usually entails several connection operations on that component to restore architecture consistency. Main artifacts are determined at each iteration of the search process by the output of the last executed evolution rule.

The operation-oriented heuristic The operation-oriented heuristic adopts an opposite point of view. It delays the use of operations that engender unsatisfied dependencies between the components of the architecture and hence more evolution operations to be found in order to reestablish architecture consistency. Addition operations are the most concerned ones. They are therefore ordered as the least priority operations while performing the search process.

5.2.2.4 Pseudo-code of the algorithm

Let a be the selected main artifact on the current state (s) of the model, e_i an enabled evolution rule.

We define $h1$ as the relation that checks whether the parameters of e_i include the artifact a or not: $\begin{cases} h1(e_i) = true \quad if \quad a \in param(e_i) \\ h1(e_i) = false \quad otherwise \end{cases}$

$h1$ is used to evaluate the enabled evolution rules at each new state of the architectural model and prioritize those who involve the main artifact.

We define $h2$ as the relation that checks whether e_i is a component addition operation or not: $\begin{cases} h2(e_i) = true \quad if \quad type(e_i) = \quad "componentaddition" \\ h2(e_i) = false \quad otherwise \end{cases}$

The search algorithm. Listing 5.1 describes the search algorithm of our customized solver. Lines 1–14 define and initialize the main variables of the algorithm. `Transitions` refers to the set of all evolution rules instances in the current state of the architecture model. The set of already explored transitions is stored in `visited`, in order to avoid cycles in the search process. The current sequence of executed transitions is stored in `pl`, to gradually collect the candidate evolution plan. The traversal of the search graph is handled by `stack`. At each step of the search process, the set of all the enabled transitions (*i.e.,* the evolution rule instances whose preconditions are verified) is pushed on the stack in order to explore them in the next steps. Transitions are pushed on the stack along with the current state of the architecture model and the current evolution plan. This enables to backtrack to previous nodes in the search graph and explore other paths when dead-ends are reached. The main artifact `a` is used in the evaluation of the artifact-oriented heuristics. The `initialMainArtifact` references the artifact modified by the initiated change. It is calculated from the post-conditions of the corresponding operations.

```
1  // initialisation step
2  s = initialState;
3  a = initialMainArtifact;
4  pl = null;
5  stack = null;
6  visited = ∅;
7  enabledTransitions = {e_i ∈ Transitions  where pre(e_i) == true};
8  priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
9  lowpriorTransitions = ∅;
10 enabledTransitions = enabledTransitions - priorTransitions;
11
12 // organizing stack
13 stack.push(s, pl, enabledTransitions);
14 stack.push(s, pl, priorTransitions);
15
```

```
16 // starting forward, DF search
17 while (stack ≠ ∅)
18  {
19  (s, pl, e_i) = stack.pop();
20  if ((s, e_i) ∉ visited)
21   {
22   visited = visited ∪ {(s, e_i)};
23   s = execute(e_i);
24   pl = pl+e_i;
25   if (goal == true) return pl;
26   a = output(e_i);
27   enabledTransitions = {e_i ∈ Transitions  where pre(e_i) == true};
28   priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
29   lowpriorTransitions = {e_i ∈ enabledTransitions where h2(e_i) == true};
30   enabledTransitions = enabledTransitions - (priorTransitions ∪ lowpriorTransitions);
31   stack.push(s, pl, lowpriorTransitions);
32   stack.push(s, pl, enabledTransitions);
33   stack.push(s, pl, priorTransitions);
34   }
35  }
36 return null; // no solution for this change request
```

Listing 5.1: Search algorithm of our specific solver

At each iteration of the search process (lines 17–33), the top of the `stack` is popped (line 19), setting a context consisting of an architecture model state (`s`), an evolution plan (`pl`) and an enabled transition (e_i). If the transition has already been visited from this state, another context is popped from the `stack` (this happens when a state can be reached by several paths of the search tree). If the transition has not been explored (line 20), it is listed as `visited` (line 22) and executed (line 23), updating the state of the architecture model. The last executed transition is appended to the evolution plan (line 24). If the `goal` is satisfied, an evolution plan has been found and it is returned (line 25). Otherwise, the set of the enabled transitions in the current state is calculated (line 27) as is the set of higher priority enabled transitions (line 28) based on the artifact-oriented heuristic (`h1`). This uses the main artifact defined as the output of the last executed transition (line 26). The set of lower priority enabled transitions is also calculated (line 29), based on the operation-oriented heuristic (`h2`). This enables to push on the `stack` the enabled transitions to be explored depending on the priority determined by our heuristics (lines 31–33). The use of a `stack` enables a Depth-First (DF) traversal of the graph: the next iteration of the search process will pop one of the currently enabled transitions, from the current architecture state, trying to extend the search path down to the `goal`. When a dead-end is reached (no transitions are enabled in the current state), the search process implicitly backtracks to a previous graph node by popping from the top of the `stack` a previously pushed context. This enables the complete traversal of the search graph (breadth search). The search process is iterated until the `goal` is reached or there is no more transition to explore (line 17). In this latter case, the requested change is rejected (line 36).

5.3 Conclusion

In this chapter, we presented an evolution management approach that deals with architectural change at the three main steps of component-based development. The proposed approach relies on Dedal formal models corresponding to the target software architecture. The Dedal formal models are used by the EMS to analyze the impact of change and to find an evolution plan that leads the architecture to a consistent evolved state. Moreover, the approach guarantees the coherence of the three software architecture descriptions by propagating the change to the impacted abstraction levels. The next step is to implement the evolution approach and demonstrate its feasibility.

Chapter 6

Implementation and experimentation: The DedalStudio tool suite

Chapters 4 and 5 present the foundations of our approach to analyze the structure of software architectures and manage architecture evolution in CBSD processes. To establish a proof-of-concept and demonstrate the feasability of our approach, we have implemented a CASE (Computer-Aided Software Engineering)) tool called *DedalStudio*. This chapter presents the architecture of *DedalStudio* (Section 6.1) and the techniques used to implement each of its parts (Section 6.2). An experimentation based on the Home Automation Software case study and a performance evaluation are then presented in Section 6.3.

Contents

6.1 Overview of *DedalStudio* 104

6.2 Implementation 105

6.2.1 Architecture modeling (*DedalModeler*) 105

6.2.2 Formal models generation (*FormalDedalGenerator*) 108

6.2.3 Architecture analysis and evolution (*DedalManager*) 110

6.2.4 Validating architecture evolution (*DedalChangeParser*) 112

6.3 Experimentation and evaluation 112

6.3.1 Experimentation 112

6.3.2 Performance evaluation 118

6.4 Conclusion 121

6.1 Overview of *DedalStudio*

Apart from the proof-of-concept purpose, the implementation is intended to assist architects in modeling, analyzing and evolving component-based software architectures. Hence, we emphasized on developing an extensible, user-friendly and efficient tool support, *DedalStudio*. We chose quite modern software technologies and an object-oriented language (JAVA) to develop our tool. *DedalStudio* is based on Eclipse [1] which is an open-source development platform for building, deploying and managing software. It provides extensibility features and allows developers to add functionalities via plugins. The current version of *DedalStudio* is composed of four parts -*DedalModeler*, *FormalDedalGenerator*, *DedalManager* and *DedalChangeParser*- as shown in Figure 6.1.

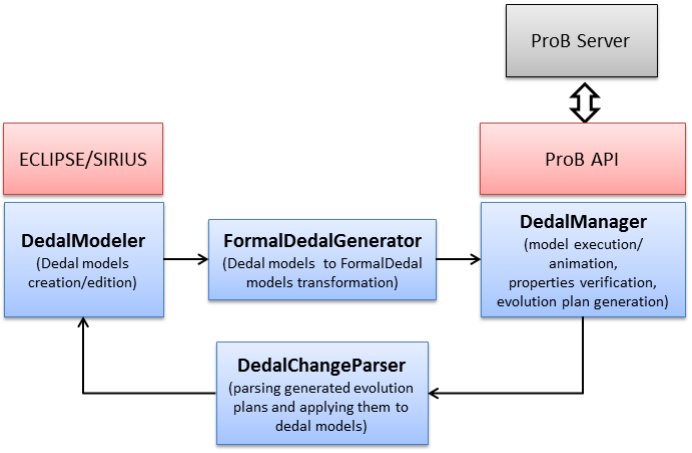


FIGURE 6.1: The architecture of DedalStudio

DedalModeler enables the creation of architecture definitions according to the Dedal architectural model. The diagram editor (*i.e., DedalModeler*) is based on SIRIUS [2], a generic platform that enables the creation of graphical modeling tools on top of EMF (Eclipse Modeling Framework) [3]. The *FormalDedalGenerator* generates Formal B models corresponding to Dedal diagrams. The *DedalManager* handles the evolution process and the generation of evolution plans. It implements a customized solver built upon the ProB API [4] that enables the animation and model-checking of B models. Finally, the *DedalChangeParser* parses the generated evolution plans and applies the adequate manipulation operations on Dedal models. All these tools, except for *DedalModeler* which is targeted to the architect, are fully automatic. Next section presents the techniques used to implement each of these parts and describes their functionalities in detail.

[1] https://eclipse.org/
[2] https://eclipse.org/sirius/
[3] https://eclipse.org/modeling/emf/
[4] http://stups.hhu.de/ProB/w/ProB_Java_API

6.2 Implementation

This section presents the implemented functionalities of *DedalStudio* and the techniques used to implement them.

6.2.1 Architecture modeling (*DedalModeler*)

Architects usually opt for graphical architecture modeling notation to design the structure of software systems. Such semi-formal models provide a balance between precision and formalism on one hand, and expressiveness and understandability on the other. *DedalStudio* follows this guideline. It provides a diagram editor (*DedalModeler*) with an attractive graphical syntax to model software architectures according to Dedal. *DedalModeler* is built upon EMF and SIRIUS. On the one side, EMF enables to define models using the Ecore meta-model and generates editors to create instances of these models. It also generates a JAVA API to create programs manipulating the instances of the defined models. On the other side, SIRIUS enables to design a graphical concrete syntax for the Ecore models. It generates rich graphical editors to create model instances. Figure 6.2 shows the Dedal meta-model (actually, an Ecore model) that defines the abstract syntax of the architecture modeling notation provided by *DedalModeler*.

The main view of *DedalModeler* (*cf.* Figure 6.3) enables to create any of the three Dedal architecture descriptions (*i.e.,* specification, configuration or assembly) and also component repositories representations.

When the architect adds a new architecture description, the tool redirects him to the adequate diagram editor (which can be Specification Diagram (SD), Configuration Diagram (CD), Assembly Diagram (AD) or Repository Diagram (RD)). Depending on the chosen architecture abstraction level, the displayed view provides the adequate constructs to use. For instance, Figure 6.4 shows the tool section corresponding to the creation of a configuration diagram.

The configuration diagram editor enables in addition to create composite component classes (as the Dedal meta-model includes the *CompositeComponentClass* meta-class). In the same way as for creating configurations, the architect is redirected to a new configuration diagram that maps to the added composite component class. Specification, configuration and repository diagram editors enable to define interface types and assign them to interfaces. Interface types are more suitable to be defined textually since they contain signature definitions. Therefore, an embedded textual editor is displayed when the user invokes an interface type assignment. This feature is enabled using xText [5], an eclipse-based framework for defining textual Domain-Specific Languages (DSLs). Like SIRIUS, it can generate editors based on Ecore models. Xtext

[5] http://www.eclipse.org/Xtext/

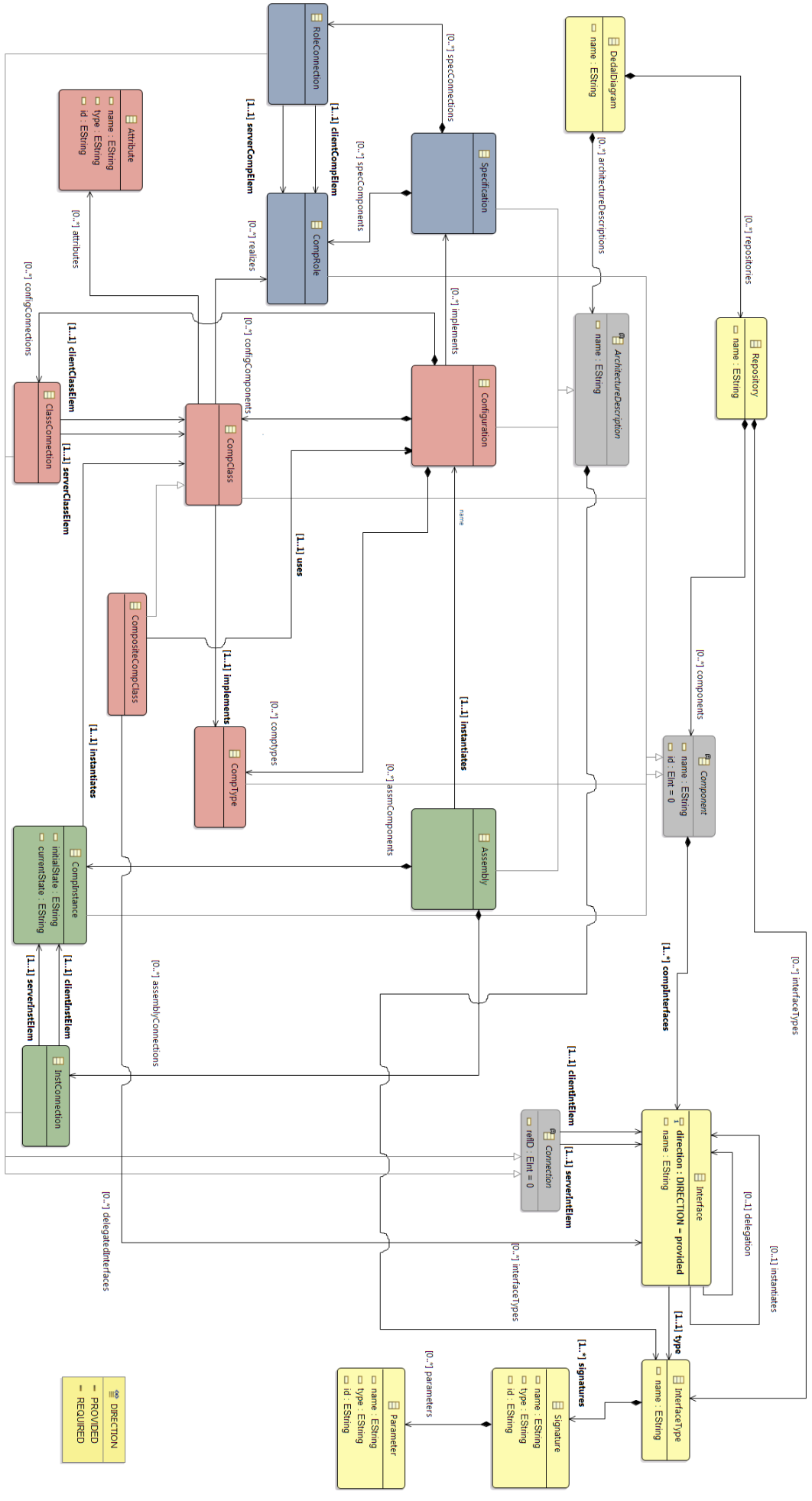


FIGURE 6.2: The Dedal meta-model (ecore)

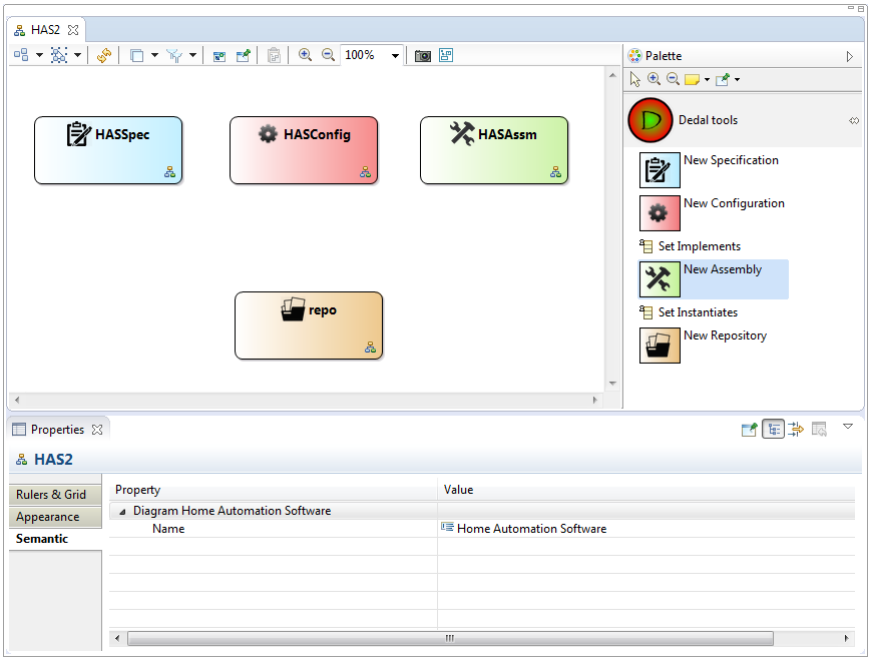


FIGURE 6.3: The interface of *DedalModeler*

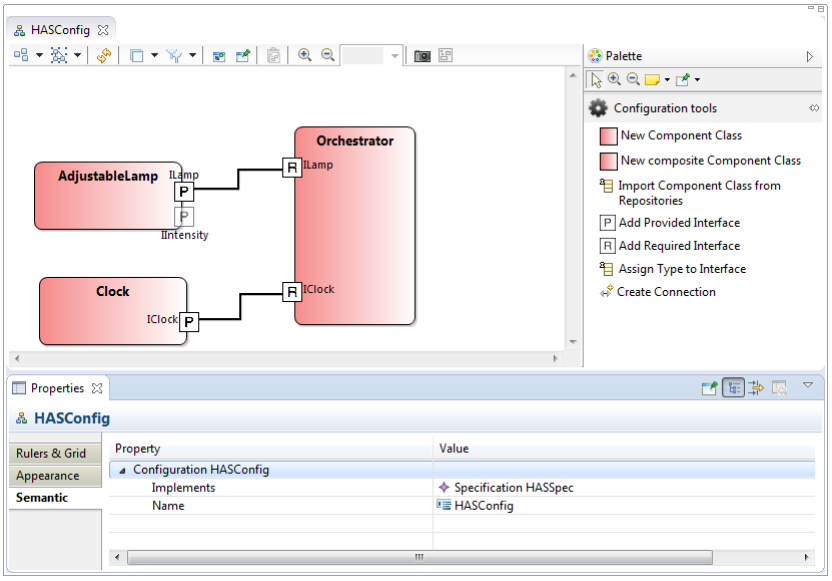


FIGURE 6.4: Configuration diagram editor

also generates parsers for DSLs. Figure 6.5 shows the embedded textual editor provided to define interface types according to the Dedal meta-model (*InterfaceType* meta-class).

DedalModeler also imposes some control on user operations. For instance, connections cannot be added between two interfaces that have the same direction or that belongs to the same component. These kinds of rules are implemented using a subset of OCL integrated in SIRIUS. More complex rules like consistency and coherence properties involve too many architectural elements at the same time and are too hard and even impossible to implement using such a

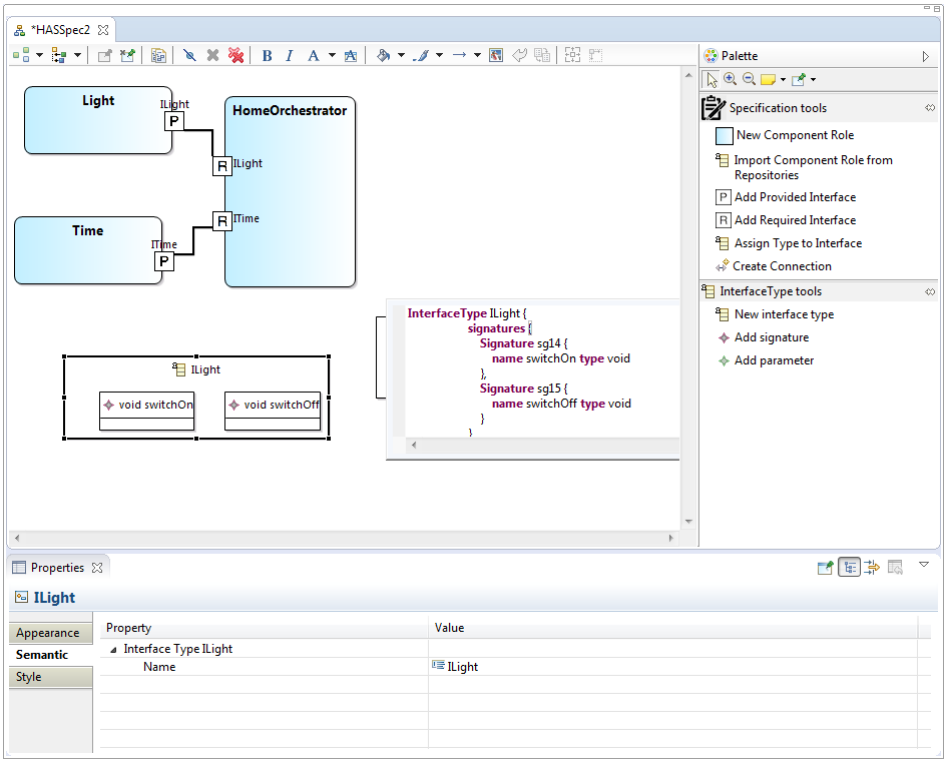


FIGURE 6.5: Embedded textual editor

feature. The generation of formal models tackles with this limit and enables more powerful architecture analysis.

6.2.2 Formal models generation (*FormalDedalGenerator*)

While semi-formal notations are relevant for architecture modeling, formal models are crucial for the verification and validation of software architectures. Combining between the two levels of formalism enables to leverage the benefits of both. The integration between semi-formal models (notably the UML standard) and formal models has been widely studied before and has shown a high efficiency to increase confidence in the design of software systems. Examples include integration between UML and Z [Dupuy et al., 2000], B [Idani, 2006, Snook and Butler, 2006] and Alloy [Anastasakis et al., 2010], SecureUML and B [Ledru et al., 2011] and, QVT and Alloy [Macedo et al., 2013]. Such transformations alleviate the burden of dealing with the verbose syntax of formal modeling languages that require mathematical knowledge and a lot of expertise. The *FormalDedalGenerator* is implemented for this purpose. It automatically generates formal Dedal models (*cf.* Chapter 4) from the architecture descriptions created using the *DedalModeler*. The resulting models are then used to perform analysis in a transparent way (*i.e.,* the transformation is hidden from the end-user). The transformation approach is described by Figure 6.6.

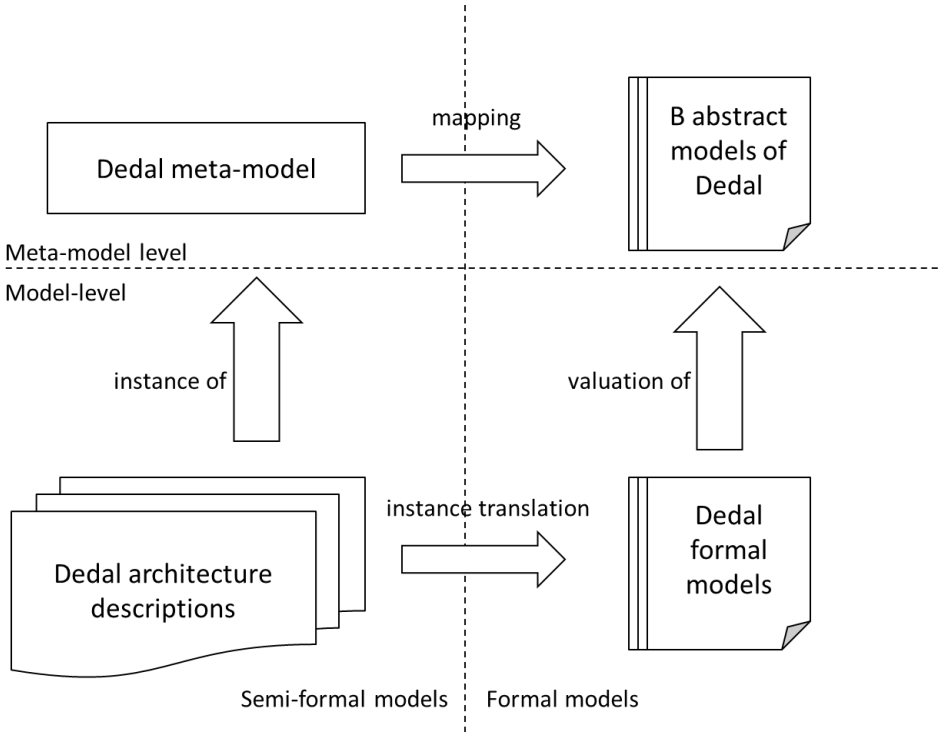


FIGURE 6.6: The translation approach

It follows a deep embedding technique [Svenningsson and Axelsson, 2013] where the elements of the source model are firstly mapped to those of the target model. Then, the instances of the source model are translated and assigned to the corresponding elements of the target model. Hence, the transformation approach consists of a static part (meta-model level) and a dynamic part (model level). The static part corresponds to a mapping between the Dedal meta-model (*cf.* Figure 6.2) and the formal Dedal models expressed using the B modeling language (*cf.* Chapter 4). A meta-class is usually mapped to a B variable typed by an abstract B set while an association relation corresponds to a B relation. For instance, Figure 6.7 presents the extract of the *arch_concept* machine that maps to the *Component* and *Interface* meta-classes and their *compInterfaces* association.

SETS
$COMPONENTS; INTERFACES$
VARIABLES
$component, interface, comp\_interfaces$
INVARIANT
$component \subseteq COMPONENTS \wedge$
$interface \subseteq INTERFACES \wedge$
$comp\_interfaces \in component \rightarrow \mathcal{P}_1(interface)$

FIGURE 6.7: Example of mapping between meta-classes and B

The dynamic part of the transformation approach represents the translation of the Dedal instance elements into B atoms. The resulting elements are assigned to the corresponding B

set and/or variable. This process is automated using Acceleo [6], a model-to-text transformation engine. It provides conditional ([if]...[/if]) and iterative structures ([for]...[/for]) and partially relies on OCL to query the Ecore models (the instances of the Dedal meta-model in our case). Listing 6.1 shows an example of a query that selects the component classes of the configuration model and assigns them to the *COMP_CLASS* set.

```
1  MACHINE
2      Arch_configuration
3  USES
4      Arch_specification
5  SETS
6      COMP_CLASS = {eClass
7  [if (self.repositories->notEmpty())]
8          [for (rep : Repository|self.repositories) before(',') separator(',')]
9                  [for (cl : CompClass|rep.eContents(CompClass)) separator(',')]
10                         ['s_cl'+cl.id/]
11                 [/for]
12         [/for]
13 [/if]
14
15 [for (cl : CompClass|self.eContents(Configuration).configComponents) before(',')
16     separator(',')]
17         ['cl'+cl.id/]
18 [/for]
19 };
```

LISTING 6.1: Acceleo query example

The query first fetches component classes in the set of *Repository* instances if it is not empty (lines 7-13). Then, these of configurations are retrieved and added to the *COMP_CLASS* set (lines 15-18).

Once the formal models are generated, it is possible to perform architecture analysis and automatically manage architecture evolution.

6.2.3 Architecture analysis and evolution (*DedalManager*)

Architecture analysis is crucial to ensure reliable evolution and preserve the consistency and coherence properties after any architectural change. This task becomes more complex when it comes to deal with multi-level architecture evolution. Indeed, the impact of change has to be analyzed and controlled both locally (*i.e.,* at a given abstraction level) and globally (for all the architecture abstraction levels). Architects hence need an evolution manager system that handles this task automatically. *DedalManager* is designed for such purpose. It conforms to the evolution management model and follows the evolution process proposed in Chapter 5. *DedalManager* is a front-end of ProB [Leuschel and Butler, 2008], a powerful animator and

[6] http://www.eclipse.org/acceleo/

model-checker for B models. The evolution manager is built upon the ProB API that provides three programmatic abstractions necessary to communicate with the ProB kernel- *Model*, *StateSpace* and *Trace*. The *Model* abstraction enables to retrieve static information about the B model such as sets, variables and invariants. The *StateSpace* abstraction enables to retrieve information about the states of the model and evaluate formulas (*i.e.,* predicates) as well as outgoing transitions. The *Trace* abstraction enables the animation of the model (*i.e.,* moving forward or backward by executing enabled operations). A trace corresponds to a path through the state space of the model and points to its current state. Using all these abstractions, the *DedalManager* inspects the *EvolutionManager* machine (presented in chapter 5) to analyze the generated formal Dedal models and control the evolution process. Evolution management starts when a change to the architecture model is requested (for instance, a component class addition is requested in the configuration). The *DedalManager* receives the request, identifies the change level and deduces the corresponding evolution goal (a set of conditions that must be verified by the sate of the model). It then invokes its solver that explores the search space to find a sequence of evolution rules leading to the evolution goal. If a solution is found, the *DedalManager* generates an evolution plan that can then be committed by the user. Otherwise (*i.e.,* in case of failure), the *DedalManager* rejects the change request. Figure 6.8 shows the view that displays the generated evolution plans.

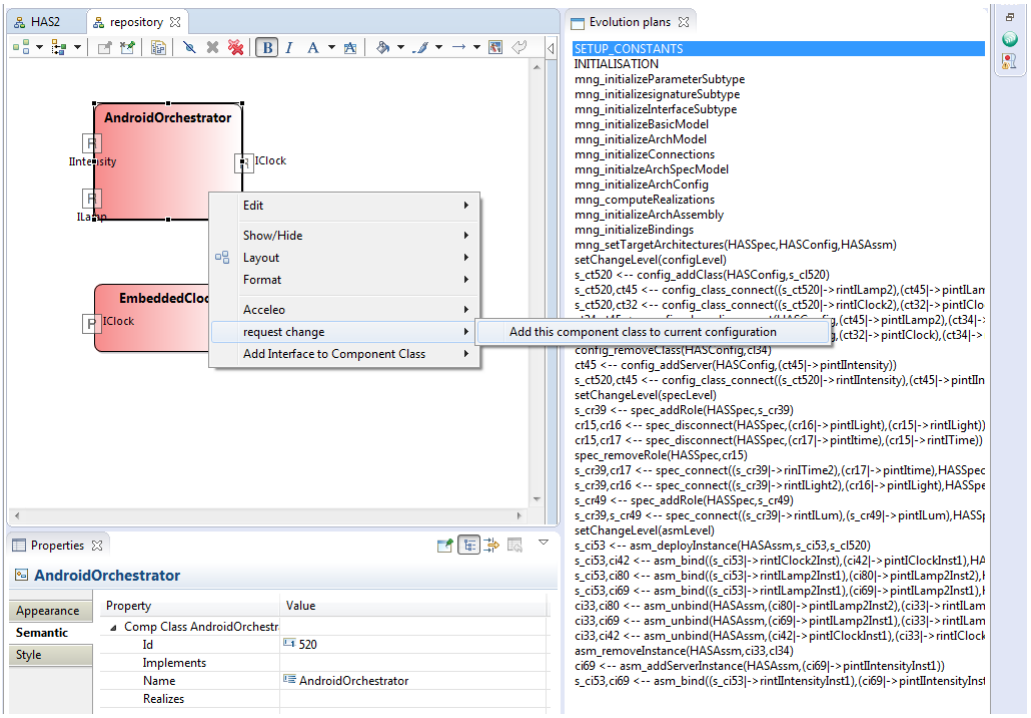


FIGURE 6.8: Evolution plan view

The *DedalManager* solver State space exploration is often confronted to an increasing calculation time especially when models become complex and the state space becomes larger.

One of the major assets of using the ProB Api is to customize state space exploration in order to prune the search space. Indeed, the ProB solver is general-purpose and uses classical search space strategies for model-checking, namely Depth-first, Breadth-first and mixed Depth-first/Breadth-first search. Aiming to provide a more efficient strategy to resolve architecture multi-level evolution, we have implemented a customized solver based on the search algorithm proposed in Chapter 5. The performance of the solver is evaluated in Section 6.3 and compared to the ProB one.

6.2.4 Validating architecture evolution (*DedalChangeParser*)

DedalManager is only a model analyzer and animator. It generates evolution plans to be committed by the architect but it does not apply any modifications on the architectural models. The *DedalChangeParser* is then in charge of this task. When the architect validates a change, it parses the generated evolution plans and applies the modification operations on the Ecore models using the EMF Api. Consequently, graphical models, which are synchronized with Ecore models, are in turn updated. The architect can hence visualize the evolved architecture descriptions.

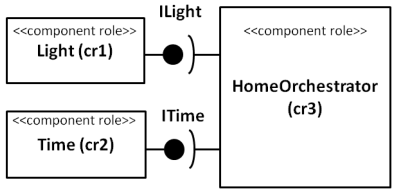
6.3 Experimentation and evaluation

This section firstly presents the case study used to demonstrate the feasibility of our approach through three evolution scenarios. Then, it evaluates the *DedalStudio* tool and more specifically the *DedalManager* solver and compares its performance with the ProB solver.

6.3.1 Experimentation

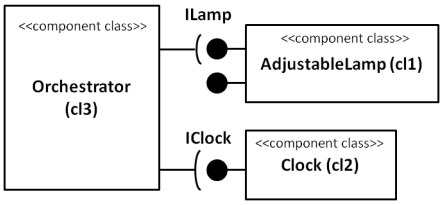
Case study description. The case study handles the evolution of the Home Automation Software introduced in Chapter 4. Three experiments are presented to assert the feasibility of our formal evolution approach. Each evolution scenario illustrates a change propagation issue that starts at a different abstraction level, in order to cover the three kinds of multi-level evolution: top-down, bottom-up and mixed. While the examples are artificial, we believe that they provide a realistic demonstration of how our approach can be applied in practice.

The initial state of HAS. Figure 6.9, Figure 6.10 and Figure 6.11 show respectively the initial architecture of Has at specification, configuration and assembly levels and their corresponding B model states. The meaning of the used notation is as follows:



specification := $\{HAS\_spec\}$
spec_components := $\{HAS\_spec \mapsto \{cr1, cr2, cr3\}\}$
comp_name := $\{cr1 \mapsto Light, cr2 \mapsto Time, cr3 \mapsto HomeOrchestrator\}$
spec_connections := $\{HAS\_spec \mapsto \{$
$((cr3, rintILight) \mapsto (cr1, pintILight)),$
$((cr3, rintITime) \mapsto (cr2, pintITime))\}$
spec_clients := $\{(HAS\_spec \mapsto \{(cr3, rintILight), (cr3, rintITime)\}$
spec_servers := $\{(HAS\_spec \mapsto \{(cr1, pintILight), (cr2, pintITime))\}\}$

FIGURE 6.9: HAS specification



configuration := $\{HAS\_config\}$
config_components := $\{HAS\_config \mapsto \{cl1, cl2, cl3\}\}$
class_name := $\{cl1 \mapsto AdjustableLamp, cl2 \mapsto Clock,$
$cl3 \mapsto Orchestrator\}$
class_implements := $\{cl1 \mapsto ct1, cl2 \mapsto ct2,$
$cl3 \mapsto ct3\}$
config_connections := $\{HAS\_config \mapsto \{$
$((ct3, rintIPower) \mapsto (ct1, pintILamp)),$
$((ct3, rintIClock) \mapsto (ct2, pintIClock))\}$
config_clients := $\{(HAS\_config \mapsto \{(ct3, rintILamp), (ct3, rintIClock)\}$
config_servers := $\{(HAS\_config \mapsto \{(ct1, pintILamp), (ct2, pintIClock))\}$

FIGURE 6.10: HAS configuration

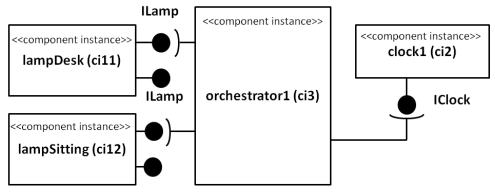
- *cr*, *cl*, *ct* and *ci* denote respectively a component role, a component class, a component type and a component instance.
- The *pint* and *rint* prefixes denote respectively a provided interface and a required interface.
- The *spec*, $config$ and *assm* prefixes denote respectively the specification, configuration and assembly architectures.
- The suffix *inst* denotes that the interface belongs to a component instance.

We note that an interface belongs to one and only one component (type, role, class or instance). Two components may have two interfaces of the same type. We often use the same name to denote two interfaces of the same type. Numeration is used to distinguish them. For example *ci*11 and *ci*12 receptively have $pintILampInst1$ and $pintILampInst2$ as interfaces of the same type.

We note that for the configuration level, connections, clients and servers are expressed using component types (*ct*). Indeed, the specification of a component class (in this case its interfaces) is hold by the component type it implements.

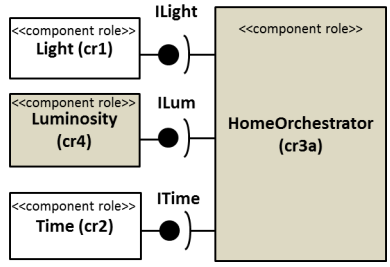
6.3.1.1 First experiment: requirement change

The first scenario addresses a requirement change. The initial HAS architecture enables to switch on / off the lights at specific hours. However, it does not enable any control on light intensity. To add this new functionality, the architect should modify the HAS specification. This corresponds to a top-down evolution since the change starts at the highest abstraction level and is then propagated to the lower abstraction levels.



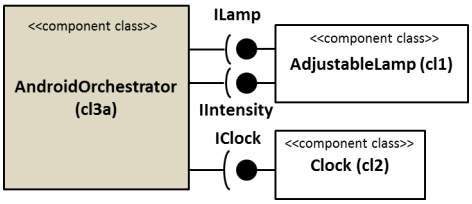
$$
\begin{array}{l}
\textbf{assembly} := \{HAS\_assembly\}|| \\
\textbf{assm\_components} := \{HAS\_assembly \mapsto \{ci11, ci12, ci2, ci3\}\} \\
\textbf{compInstance\_name} := \{ci11 \mapsto lampDesk, \\
\quad ci12 \mapsto lampSitting, ci2 \mapsto clock1 \\
\quad ci3 \mapsto orchestrator1\}|| \\
\textbf{assm\_connections} := \{HAS\_assembly \mapsto \{ \\
\quad ((ci3, rintILampInst) \mapsto (ci11, pintILampInst1)), \\
\quad ((ci3, rintILampInst) \mapsto (ci11, pintILampInst2)), \\
\quad ((ci3, rintIClock) \mapsto (ci2, pintIClockInst))\}\} \\
\textbf{comp\_instantiates} := \{ci11 \mapsto cl1, ci12 \mapsto cl1, \\
\quad ci2 \mapsto cl2, ci3 \mapsto cl3\} \\
\textbf{assm\_clients} := \{(HAS\_assembly \mapsto \{(ci3, rintILampInst), \\
\quad (cl3, rintIClockInst)\}|| \\
\textbf{assm\_servers} := \{(HAS\_assembly \mapsto \{(ci11, pintILampInst1), \\
\quad (ci12, pintILampInst2), (ci2, pintIClockInst))\}
\end{array}
$$

FIGURE 6.11: HAS assembly



$$
\begin{array}{l}
spec\_addRole(HAS\_spec, cr4) \\
spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1, pintILight)) \\
spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime)) \\
spec\_deleteRole(HAS\_spec, cr3) \\
spec\_addRole(HAS\_spec, cr3a) \\
spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1, pintILight)) \\
spec\_connect(HAS\_spec, (cr3a, rintILum), (cr4, pintILum)) \\
spec\_connect(HAS\_spec, (cr3a, rintITime), (cr2, pintITime))
\end{array}
$$

FIGURE 6.12: Evolving the HAS specification

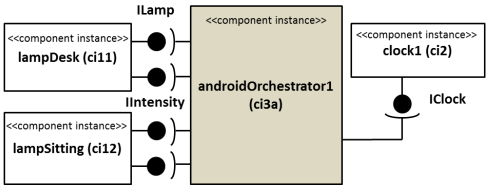


$$
\begin{array}{l}
config\_addServer(HAS\_config, (cl1, pintIIntensity)) \\
config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp)) \\
config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock)) \\
config\_deleteClass(HAS\_config, cl3) \\
config\_addClass(HAS\_config, cl3a) \\
config\_connect(HAS\_config, (cl3a, rintILamp2), (cl1, pintILamp)) \\
config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock)) \\
config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1, pintIIntensity))
\end{array}
$$

FIGURE 6.13: Evolving the HAS configuration

Evolving the HAS specification. The change is initiated by the addition of the *Luminosity* component role ($cr4$) providing the *ILum* interface to control the luminosity. This operation engenders an interaction inconsistency at the specification level since the $ILum$ server interface is not connected. To restore the consistency, an orchestrator with a new required interface (($cr3a$)) is added and connected to *Luminosity*, *Light* and *Time*. The old orchestrator ($cr3$) is disconnected and removed. Figure 6.12 shows the evolved HAS specification and its corresponding evolution plan.

Propagating the change to the HAS configuration. Adding new functionalities to the system's specification necessarily implies updating the system's implementation to include these new functionalities. This is ensured by reestablishing the coherence between the specification and configuration levels. In this case, the specified functionality (*i.e.,* control of the luminosity) can be realized by the *AdjustableLamp* component class via its *IIntensity* server interface. The addition of this server interface induces an interaction inconsistency in the configuration since no connection with an existing compatible client interface exists. To restore consistency, the *Orchestrator* ($cl3$) component class is removed and the *AndroidOrchestrator* ($cl3a$) component class is added instead (*cf.* the evolution plan in Figure 6.13).



$asm\_addServerInstance(HAS\_assembly, (ci11, pintIIntensityInst1))$
$asm\_addServerInstance(HAS\_assembly, (ci12, pintIIntensityInst2))$
$asm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$asm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci12, pintILampInst2))$
$asm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))$
$asm\_removeInstance(HAS\_assembly, ci3)$
$asm\_deployInstance(HAS\_assembly, ci3a)$
$asm\_bind(HAS\_assembly, (ci3a, rintILampInst), (ci11, pintILampInst1))$
$asm\_bind(HAS\_assembly, (ci3a, rintIIntensityInst), (ci11, pintIIntensityInst1))$
$asm\_bind(HAS\_assembly, (ci3a, rintIClock2Inst), (ci2, pintIClockInst))$
$asm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci12, pintILampInst2))$
$asm\_bind(HAS\_assembly, (ci3a, rintIIntensityInst), (ci12, pintIItensityInst2))$

FIGURE 6.14: Change propagation to the HAS assembly

Propagating the change to the HAS assembly. In the same way, changes are propagated to the HAS assembly in order to restore coherence with the evolved HAS configuration. The *IIntensity* server interfaces of both component instances $ci11$ and $ci12$ are added. The old component instance $ci3$ is removed and replaced with the *androidOrchestrator* one ($ci3a$) (*cf.* Figure 6.14).

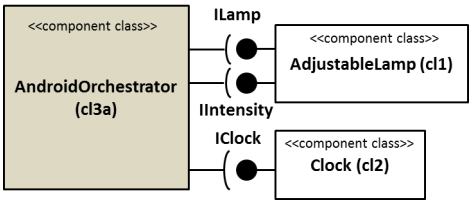
6.3.1.2 Second experiment: implementation change

The second scenario addresses an implementation change. The objective is to enable the control of the building through a mobile device (running Android OS for example). To adapt the current implementation to Android, the *Orchestrator* component class should be removed and replaced with an Android compatible one available in the component repository (*AndroidOrchestrator*). The change is initiated at the configuration level and then propagated to both specification and assembly levels.

Evolving the HAS configuration. The change is initiated by the addition of the *AndroidOrchestrator* ($cl3a$) component class to the HAS configuration. This entails the connection of all required interfaces of *AndroidOrchestrator* and the disconnection and removal of *Orchestrator* to restore the architecture consistency. The service *IIntensity* is added since it is required by the new orchestrator. We note that the orchestrator replacement cannot be done using component substitutability since the new orchestrator engenders more dependencies (*i.e.,* the *IIntensity* required interface) breaking hence the substitutability rule (*cf.* Chapter 4, *Rule* 4).

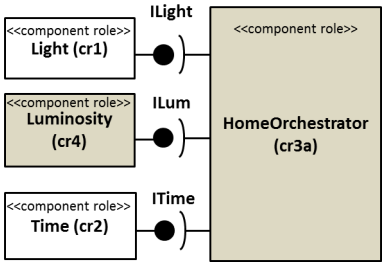
Figure 6.15 illustrates the evolved HAS configuration as well as an alternative evolution plan.

Propagating the change to the HAS specification. The evolved HAS configuration implements all the specified functionalities of the current HAS specification. However, the latter does not capture the new implemented functionality that enables the control of the



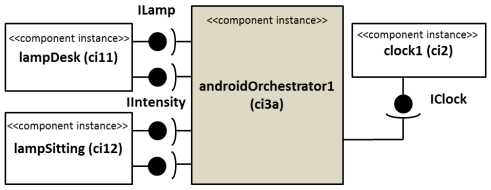
$config\_addClass(HAS\_config, cl3a)$
$config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp))$
$config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock))$
$config\_deleteClass(HAS\_config, cl3)$
$config\_connect(HAS\_config, (cl3a, rintILamp2), (cl1, pintILamp))$
$config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock))$
$config\_addServer(HAS\_config, (cl1, pintIIntensity))$
$config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1, pintIIntensity))$

FIGURE 6.15: Evolving the HAS configuration



$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1, pintILight))$
$spec\_connect(HAS\_spec, (cr3a, rintITime), (cr2, pintITime))$
$spec\_addRole(HAS\_spec, cr4)$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr4, pintIIntensity))$

FIGURE 6.16: Change propagation to the HAS specification



$asm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$asm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci12, pintILampInst2))$
$asm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))$
$asm\_removeInstance(HAS\_assembly, ci3)$
$asm\_deployInstance(HAS\_assembly, ci3a)$
$asm\_bind(HAS\_assembly, (ci3a, rintILampInst), (ci11, pintILampInst1))$
$asm\_addServerInstance(HAS\_assembly, (ci11, pintIIntensityInst1))$
$asm\_bind(HAS\_assembly, (ci3a, rintIIntensityInst), (ci11, pintIIntensityInst1))$
$asm\_bind(HAS\_assembly, (ci3a, rintIClock2Inst), (ci2, pintIClockInst))$
$asm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci12, pintILampInst2))$
$asm\_addServerInstance(HAS\_assembly, (ci12, pintIIntensityInst2))$
$asm\_bind(HAS\_assembly, (ci3a, rintIIntensityInst), (ci12, pintIItensityInst2))$

FIGURE 6.17: Change propagation to the HAS assembly

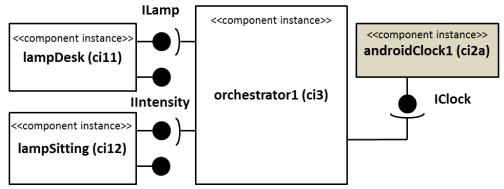
luminosity. Hence, the coherence between the two architecture descriptions does not hold anymore and the change must be propagated to the specification level to restore coherence.

This entails the addition of an orchestrator with a new required interface (*ILuminosity* ($cr3a$)) and the *Luminosity* component role ($cr4$) providing this interface. The evolved HAS specification and a potential evolution plan are shown in Figure 6.16.

Propagating the change to the HAS assembly. Th current HAS assembly is not a valid instantiation of the evolved HAS configuration. Therefore, the change must be propagated to the assembly level to restore the coherence between the configuration and assembly. An instance ($ci3a$) of the android orchestrator is then deployed and connected to the *lampSitting*, *deskSitting* and *clock* component instances. The ancient orchestrator instance ($ci3$) is removed. The resulted HAS Assembly and the corresponding evolution plan are shown in Figure 6.17.

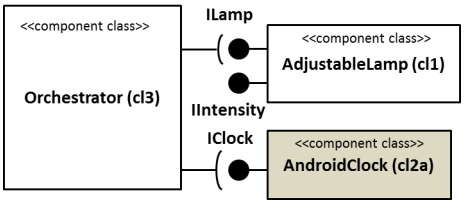
6.3.1.3 Third experiment: runtime change

The third scenario addresses a runtime change. It corresponds to a bottom-up evolution since the change is initiated at the lowest abstraction level. Because of a dry battery, the clock device



$asm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))$
$asm\_deleteServerInstance(HAS\_assembly, (ci2, pintIClockInst))$
$asm\_replaceInstance(HAS\_assembly, ci2, ci2a)$
$asm\_addServerInstance(HAS\_assembly, (ci2a, pintIClock2Inst))$
$asm\_bind(HAS\_assembly, (ci3, rintIClockInst), (ci2a, pintIClock2Inst))$

FIGURE 6.18: Evolving the HAS assembly



$config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock))$
$config\_deleteServer(HAS\_config, (cl2, pintIClock))$
$config\_replaceClass(HAS\_config, cl2, cl2a)$
$config\_addServer(HAS\_config, (cl2, pintIClock2))$
$config\_connect(HAS\_config, (cl3, rintIClock), (cl2a, pintIClock2))$

FIGURE 6.19: Change propagation to the HAS configuration

in the building is out of service. This environmental change induces the dysfunction of the *clock1* driver (*ci2*). The objective is to find a solution to dynamically repair the architecture in order to maintain the functionalities of the system.

Evolving the Has assembly. *clock1* (*ci2*) must be replaced by another component instance that provides the same services. An instance of the *AndroidClock* component class, *androidClock1* (*ci2a*), is thus chosen to replace *clock1*. The *clock1* component instance is disconnected first then replaced by *androidClock1*. This change does not alter the consistency of the assembly architecture and no further operations are required (*cf.* evolution plan in Figure 6.18).

Propagating the change to the Has configuration. Coherence with the configuration architecture has to be reestablished. Indeed, the evolved assembly architecture is not a valid instantiation of the current configuration architecture since the *ci2a* component instance is not recognized as an instance of the *cl2* component class. Change propagation induces the substitution of the *AndroidClock* component class (*cl2a*) for the *Clock* component class (*cl2*), which amounts to the evolution plan shown in Figure 6.19.

We note that this evolution scenario uses a strict evolution strategy that does not take into account component polymorphism (*i.e.,* a supertype component class can have component instances of its subtype component classes). This issue can be managed using versioning strategies where it is possible to decide to keep the old version of the clock component class since there no type error risk or to update it accordingly to the new clock component instance to keep an up-to-date architecture configuration.

Propagating the change to the Has specification. The component class substitution preserves the coherence between the specification and the configuration. Indeed, when a component class implements a given role, any component subclass, as a substitute, also implements the role. As a consequence, no change needs to be propagated to the specification.

6.3.1.4 Discussion

The three experiments illustrate the feasibility of our approach to evolve component-based software architectures. They show that change can be initiated from any abstraction level, controlled locally to preserve consistency and propagated to other levels to keep all architecture descriptions coherent with each other. The time for resolving evolution and the way plans are generated depend on the technique used by the solver. Next section evaluates evolution resolution time using the customized solver of our tool *DedalStudio* and the general-purpose solver of ProB.

6.3.2 Performance evaluation

This section evaluates the performance of two tools that automatically generate evolution plans: ProB and *DedalStudio*. The Has architecture descriptions are modeled using *DedalModeler* then their corresponding B formal models are generated using *FormalDedalGenerator*. The same resulting models are then used to test both ProB solver and *DedalManager* customized solver.

6.3.2.1 Test preliminaries for ProB

ProB [Leuschel and Butler, 2008] is basically used to ensure that a model is free from error states such as deadlocks or invariant violations by exploring its state space. Additionally, it can be used as a solver that attempts to find a state satisfying a given goal. It proposes one of the three search strategies to explore the state space: depth-first (DF), breadth-first (BF) or mixed depth-first/breadth-first (DF/BF). The most suitable strategy depends on the kind of checking the user wants to perform. In the case of searching for a goal state by exploring a large state space, depth first seems to be the most efficient strategy [Leuschel and Bendisposto, 2011].

ProB requires to manually set the search goal (GOAL) and the change level (*changeLevel*) in the input formal model. To correctly perform the test, we use three different instances of the *EvolutionManager* machine for each experiment. Each instance contains the adequate change level and goal definition (Table 6.1).

	Evolution Manager instance	**State**	**Change level**	**GOAL**
Exp 1	Instance 1	initial state of HAS	*specLevel*	$spec\_consistency \wedge cr4 \in spec\_components(HAS\_spec)$
	Instance 2	evolved specification	$configLevel$	$config\_consistency \wedge specConfigCoherence$
	Instance 3	evolved configuration	$assmLevel$	$assm\_consistency \wedge configAssemblyCoherence$
Exp 2	Instance 1	initial state of HAS	$configLevel$	$config\_consistency \wedge cl3a \in config\_components(HAS\_config)$
	Instance 2	evolved configuration	*specLevel*	$spec\_consistency \wedge specConfigCoherence$
	Instance 3	evolved configuration	$assmLevel$	$assm\_consistency \wedge configAssemblyCoherence$
Exp 3	Instance 1	initial state of HAS	$assmLevel$	$assm\_consistency \wedge ci2 \notin assm\_components(HAS\_assembly) \wedge ci2a \in assm\_components(HAS\_assembly)$
	Instance 2	evolved assembly	$configLevel$	$config\_consistency \wedge configAssemblyCoherence$
	Instance 3	evolved configuration	*specLevel*	$spec\_consistency \wedge specConfigCoherence$

TABLE 6.1: Test parameters for ProB

As stated in Chapter 5, Section 5.1.3.3, the goal corresponding to the initiated change is defined by its post-condition. For instance, in *Exp1* and *Exp2*, the initiated change respectively requires the addition of $cr4$ and $cl3a$. Hence, their respective post-conditions are $cr4 \in spec\_components(HAS\_spec)$ and $cl3a \in config\_components(HAS\_config)$. In the third experiment, the initiated change requires the replacement of the $ci2$ component instance by $ci2a$ component instance. Hence, the corresponding post-condition is as follows:
$ci2 \notin assm\_components(HAS\_assembly) \wedge ci2a \in assm\_components(HAS\_assembly)$

Additionally, we remove all the operations unnecessary for plan generation such as initialization operations and *setChangeLevel* (*cf. EvolutionManager* machine in chapter 5). Therefore, we make sure that the solver explores only the evolution rules and nothing else that may decrease its performances.

6.3.2.2 Test preliminaries for *DedalManager*

To evaluate the efficiency of the heuristics proposed in Chapter 5, we test two search strategies implemented by *DedalManager* solver: Simple Depth-First search (*S-DF*) and Heuristic Depth-First search (*H-DF*). Our prototype automatically initializes the model, sets the change level and restricts the search space to applicable evolution rules (unnecessary operations are filtered and removed from the search space). For each experiment, one instance of the *EvolutionManager* machine is sufficient to perform the test since the last state is stored in the trace after each evolution.

For each evolution scenario, initial change is requested via the diagram editors. Figure 6.8 shows an illustration of a change request at configuration level.

Table 6.2 shows the parameters that *DedalManager* uses to perform the search for each experiment.

	Run	State	Change level	Initial main artifact	GOAL
Exp 1	1st run	initial state of HAS	$specLevel$	$cr4$	$specification\_consistency \wedge cr4 \in spec\_components(HAS\_spec)$
	2nd run	evolved specification	$configLevel$	$\{cl1\} = realizes(cr4)$	$config\_consistency \wedge specConfigCoherence$
	3rd run	evolved configuration	$assmLevel$	$\{ci11, ci12\} = comp\_instantiates(cl1)$	$assembly\_consistency \wedge configAssemblyCoherence$
Exp 2	1st run	initial state of HAS	$configLevel$	$cl3a$	$config\_consistency \wedge cl3a \in config\_components(HAS\_config)$
	2nd run	evolved configuration	$specLevel$	$\{cr3a\} = realizes^{-1}(cl3a)$	$spec\_consistency \wedge specConfigCoherence$
	3rd run	evolved configuration	$assmLevel$	$\{ci3a\} = comp\_instantiates(cl3a)$	$assembly\_consistency \wedge configAssemblyCoherence$
Exp 3	1st run	initial state of HAS	$assmLevel$	$ci2$	$assembly\_consistency \wedge ci2 \notin assm\_components(HAS\_assembly) \wedge ci2a \in assm\_components(HAS\_assembly)$
	2nd run	evolved assembly	$configLevel$	$\{cl2\} = comp\_instantiates^{-1}(ci2)$	$config\_consistency \wedge configAssemblyCoherence$
	3rd run	evolved configuration	$specLevel$	$\{cr2\} = realizes^{-1}(cl2)$	$spec\_consistency \wedge specConfigCoherence$

TABLE 6.2: Test parameters for the evolution-solver algorithm

The change level, initial main artifact and evolution goal parameters are deduced from the change request. At the propagation step, the initial main artifact is identified thanks to the intra-level rules defined in Chapter 4. For instance, in the second experiment, the initial main artifact is selected from the set of component roles realized by the *AndroidOrchestrator* ($cr3a$) component class as follows: $\{cr3a\} = realizes\ (cl3a)$. By selecting $cr3a$, the solver will first try the evolution rules applicable on $cr3a$. In the case where many candidates are found, the current solver is not able to evaluate which component class is the best. Hence, an artifact is non deterministically chosen from the list. The initial main artifact identification process concerns only the *H-DF* strategy.

6.3.2.3 Performance evaluation

The performance of both solvers has been measured on the three experiments, in order to evaluate the influence of our proposed heuristics. Tests were run on a standard PC (2.5 GHZ Intel Core i5, 8 GB SDRAM) under Windows 7. Table 6.3 shows the average time in milliseconds of three runs for each evolution scenario, using all of the exposed strategies. Timeout is set to 3 minutes.

We note that the order and number of evolution rules may differ from a generated evolution plan to another (our algorithms are not deterministic as they make random choices when sets of equivalent elements are considered, such as a set of candidate main artifacts) but all generated plans are valid and lead to the same goal state.

		ProB solver			DedalManager solver	
	Change level	**DF (ms)**	**BF (ms)**	**DF/BF (ms)**	**S-DF (ms)**	**H-DF (ms)**
Exp 1	specLevel (initial)	2944	8774	4448	3260	2100
	configLevel	9100	26779	27177	3254	1393
	asmLevel	9140	TIME-OUT	30424	26738	1926
Exp 2	configLevel (initial)	9112	104629	12235	4712	2537
	specLevel	3006	8710	5135	8733	1896
	asmLevel	TIME-OUT	TIME-OUT	28177	TIME-OUT	1927
Exp 3	asmLevel (initial)	11589	72480	37090	4745	1184
	configLevel	78939	36187	85195	TIME-OUT	2351
	specLevel (not affected)	–	–	–	–	–

TABLE 6.3: Performance evaluation

6.3.2.4 Discussion

Results doubtlessly show the benefits of a custom solver that integrates specific heuristics. In all experiments, *H-DF* performs the best with a resolution time varying in $[1184, 2537]$ ms. In most cases, the depth-first strategy (*S-DF* and *DF*) comes after *H-DF* (*cf.* Table 6.3 Exp1(all levels), Exp2(*configLevel* and *specLevel*) and Exp3(*asmLevel*)). Some other cases however shows that it leads to TIME-OUT (*cf.* Table 6.3 Exp2(*asmLevel*) and Exp3(*configLevel*)). Indeed, exploring state space in depth is sometimes risky because the solver may waste a lot of time in exploring a "wrong" branch, especially when the state space is large. Breadth-first strategy is in most cases by far the worst (see for instance Exp2(*configLevel*) and Exp3(*asmLevel*) in Table 6.3). Our interpretation is that *BF* waste a lot of time in trying all outgoing transitions from a source state before exploring in depth one of them (and thus trying to find actually a suitable evolution plan), especially when the set of applicable transitions becomes large. The mixed *DF/BF* strategy has sometimes good surprising results (see for instance Exp2(*asmLevel*) in Table 6.3). This is due to the fact that this strategy randomly mix between the *DF* and *BF* which is sometimes more efficient than *DF*.

A more precise performance evaluation, based on a larger set of experiments and a theoretical study of the combinatorial complexity of the search space is needed. Performance is indeed an inherent limitation for search-based software engineering, as the resolution time of solvers generally grows exponentially depending on the size of the problem. Designing and integrating new heuristics to cut down calculation time is promising (we can for instance choose preferentially transitions that generate no or little incoherence in the architecture model).

6.4 Conclusion

This chapter presented an implementation and experimentation of our approach to manage software architecture evolution in CBSD processes. The implementation consists in *DedalStudio*, an eclipse-based tool suite that supports architecture modeling and evolution management based on Dedal. The tool was implemented with respect to the following criteria: extensible

(since it is based on Eclipse plugins), user-friendly (since it is based on SIRIUS) and powerful (since it integrates the ProB Api and uses search-based engineering techniques to automatically manage architecture evolution).

Experimentation is based on three evolution scenarios for the Home Automation Software. The generated evolution scenarios and resulting evolved architecture definitions were firstly presented as a proof-of-concept. Then, tests were performed using both *ProB* and *DedalStudio* to automatically generate evolution plans for each evolution scenario. The evaluation showed that our customized solver based on specific heuristics outperforms the ProB general-purpose solver.

Nonetheless, we assume that an evaluation on a larger set of experiments is required to determine the limits of our solver. Moreover, experimenting with real architecture descriptions might reveal other issues.

Chapter 7

Conclusion and future work directions

This chapter summarizes the contributions of the thesis, points out their main limitations and discusses future work directions.

Contents

7.1 Contributions 123
7.1.1 Conceptual contributions 124
7.1.2 Technical contributions 124
7.1.3 Applicative contributions 125
7.2 Limitations and future work directions 126
7.2.1 Conceptual perspectives 126
7.2.2 Technical perspectives 127
7.2.3 Applicative perspectives 128
7.2.4 Threats to validity and experimental perspectives 128

7.1 Contributions

This thesis contributes to Component-Based Software Engineering. In particular, it addresses the problematic of evolution in reuse-intensive, architecture-centric, component-based development processes. Two main issues are considered. First, dealing with evolution at any step of component-based software lifecyle namely, requirements changes, implementation changes and runtime changes. Second, controlling the impact of software changes all along its lifecycle to avoid architectural inconsistencies that compromise the functioning of the software system and lead to its degradation and phase-out.

Therefore, a formal approach to automate architecture evolution management in CBSD processes is proposed. The contributions of this thesis can be classified according to three concerns: conceptual, technical and applicative.

7.1.1 Conceptual contributions

From a theoretical point of view, this thesis proposes a type theory for the Dedal three-level architecture model tailored for CBSD. It consists of intra-level rules and inter-level rules. Intra-level rules concern the relations linking components within models corresponding to different architecture abstraction levels whereas inter-level rules concern the relations linking models describing the same architecture at different abstraction levels. It follows from these rules the definition of two main architecture properties: architecture consistency and architecture coherence. Architecture consistency includes general-purpose architecture properties regarding name, interface and interaction consistency that must be preserved to ensure the correctness of the software system. Architecture coherence encompasses the properties that must be preserved between two architecture definitions at two adjacent abstraction levels to ensure the traceability of architectural design decisions throughout the whole software development steps. Coherence notably enables to avoid the architectural erosion problem.

A second conceptual contribution is an evolution management model that considers architectural changes as first-class entities and separates the architecture modeling concern from the architecture evolution management concern. The proposed model supports the Dedal architectural model and highlights three abstraction levels of change in accordance with it. Besides, it classifies architectural changes into the ones initiated externally and the ones triggered by the evolution manager system to preserve architecture properties. Moreover, the evolution management model introduces the notions of *evolution rules*, *evolution goal* and *evolution plan*. An evolution rule is a specific operation that controls an elementary modification on the architectural model at a given abstraction level. An *evolution goal* denotes the condition that must be satisfied on the architectural model after some change. An *evolution plan* consists of a sequence of evolution rules that when executed, leads the architectural model to a state satisfying the evolution goal.

7.1.2 Technical contributions

A first technical contribution consists in the B formalization of Dedal concepts and its associated type theory. The resulting B abstract models (so called *FormalDedal*) include both generic architectural concepts and Dedal specific concepts. The generic concepts are included in the *Arch_concepts* machine and could be reused and/or specialized to formalize other ADLs. The specific Dedal concepts are included in the *Arch_specification*, *Arch_configuration* and

Arch_assembly machines that respectively correspond to the specification, configuration and assembly levels. *FormalDedal* hence presents a support for powerful architecture analysis such as proof, model-checking and constraint-solving.

A second technical contribution is the B formalization of the evolution management model. It results in the *EvolutionManager* model that includes evolution rules corresponding to each Dedal architecture abstraction level. The *EvolutionManager* model enables to analyze architectural change and is used to calculate evolution plans to restore the consistency and coherence properties of the altered architecture. An evolution management process is also proposed. It consists of three steps: First, change is initiated on an architecture description at a given abstraction level; then, the consistency of the impacted description is checked out and restored by triggering additional changes; finally, the global coherence of the architecture definitions is verified and restored by propagating changes to the other abstraction levels.

A third technical contribution consists in an evolution-solving algorithm that automatically generates evolution plans according to a given evolution goal. The algorithm performs a forward depth-first search on the state space of the architectural model. To improve calculation time, it is enhanced with two specific heuristics: *artifact-oriented* heuristic and *operation-oriented* heuristic. The *artifact-oriented* heuristic is used to prioritize the operations manipulating the artifacts that are more likely to satisfy the evolution goal. The *operation-oriented* heuristic delays the use of operations that engender unsatisfied dependencies between the components and hence risk to increase the search time.

7.1.3 Applicative contributions

A tool suite, named *DedalStudio* was implemented to demonstrate the feasibility of the evolution management approach. *DedalStudio* is an eclipse-based environment for architecture modeling and evolution management. Modern technologies (*e.g.,* EMF/SIRIUS, Acceleo and ProB Api) and approaches were used to implement *DedaLStudio*. In particular, *DedalStudio* presents an application of both Model-Driven engineering and search-based engineering.

Model-Driven engineering application. Adopting a Mde approach, *FormalDedal* is mapped with the Dedal meta-model to enable automatic model transformations. This integration enables to combine both the benefits of semi-formal modeling notations and formal notations: architecture modeling using the *DedalModeler* tool and architecture analysis and evolution management using *DedalManager* tool.

Search-based engineering application. An application of search-based engineering consists in the implementation of a specific solver to tackle a software engineering problem, *i.e.,*

planning the evolution of software architecture models. The interest of such approach is to optimize the planning calculation time using search-based algorithm enhanced with specific heuristics.

7.2 Limitations and future work directions

This section points out the limitations of our work and discusses future work directions. As for contribution, we classify perspectives into conceptual, technical and applicative. Additionally, we discuss the threats to validity and experimental perspectives.

7.2.1 Conceptual perspectives

A first perspective is to add support for multi-level architectures versioning. Software versioning is intrinsic to software evolution. It enables to keep history of previous software states (versions) and traces of the updates (changes) up to the latest available software versions. Versioning is necessary to backup to previous versions when a bug or faulty change decision is detected in the current version. Moreover, many variants of the software could coexist and evolve concurrently. In Dedal, versioning consists in studying the impact of evolving an architecture at a given abstraction level in all the related existing architecture definitions. For instance, evolving an architecture specification may imply the evolution of all its associated architecture configurations or evolving an architecture configuration may simply imply the addition of a new software variant in the spirit of a software product line [Pohl et al., 2005].

At this stage of work the proposed type theory addresses only the syntactic level of software components and the structural and functional properties of software architectures. This contribution sets the basis to address more challenging issues, namely the behavioral aspect and non-functional component properties such as performance, trustworthiness and quality. For instance, the rules between the specification and configuration levels could be extended to include constraints related to non-functional attributes. This would enable to trace both functional and non-functional design decisions. Related work includes the approach of Tibermacine *et al.* [Tibermacine et al., 2005] that addresses the traceability of non-functional properties in component-based software lifecycle.

Another perspective is to extend the evolution management model to enable customized and user-assisted evolution management. In particular, the following features could be added:

- Evolution rules are predefined and only control elementary model manipulation operations. An interesting perspective is to propose a language for architects to specify

customized evolution rules in a manner similar to Le Goaer [Le Goaer, 2009] and Sadou [Sadou, 2007]. The language would first enable to specify more complex manipulation operations as composition of elementary ones. The resulting operations could then be stored for reuse in repositories that will be made available for the Evolution Manger System. Second, specific evolution rules could be specified to take into account additional criteria related for instance to the cost of using an operation in terms of time and performance. This will enable to generate multiple candidate evolution plans and select the optimal one according to the chosen criteria in a manner similar to Barnes et al. [Barnes et al., 2014].

- Only general-purpose architecture properties are taken into account. Architect may need to specify additional properties related for instance to architecture styles [Garlan and Shaw, 1994].

7.2.2 Technical perspectives

Technical perspectives address mainly the B formalization of Dedal. A first technical limitation of using B formal models is the inability to add new entities. All atoms should have been predefined before in the *SETS* clause. This does not really matter for component classes since they are pre-existant (stored in repositories and available for reuse). However, to deal with unanticipated changes, component roles and component instances may need to be created further to change propagation. We are investigating on how to make evolution manager adding component instances and component roles to formal Dedal models during evolution without anticipation.

A second perspective is to establish a refinement process between the specification and configuration levels using the B method as support. The objective is to enable a more rigorous link between architecture specification and configuration. An architecture specification including partial and abstract component roles could be successively refined to obtain more concrete specification. The latter would then be used as support to obtain an implementation of the software. Model manipulation operations however cannot be refined using B. This limitation is fixed in Event-B [Abrial, 2010], an extended variant of B. Refining operations could be interesting to specify and specialize model manipulation operations as mentioned in conceptual perspectives.

Existing work in translating OCL to B [Ledang and Souquieres, 2002] can be integrated to ours in order to allow the architect to specify additional architecture properties as mentioned in theoretical perspectives.

Another perspective addresses the algorithm that calculates evolution plans. Existing heuristics can be improved. For instance, operation-oriented heuristic may prioritize transitions that generate no or little incoherence in the architecture model like replacement operations.

An alternative idea to minimize change propagation time is to start change propagation from an initial solution that would be a transformation of the plan generated for the adjacent architecture level. Further evolution rules are then triggered in need to reestablish consistency and coherence. Transformation of evolution plans can be calculated using the semantic links that exist between two adjacent levels. For instance, $asm\_replaceInstance(HAS\_assembly, ci2, ci2a)$ can be translated to its equivalent in configuration level as follows: $config\_replaceClass(HAS\_config, cl2, cl2a)$ where $cl2 = comp\_instantiates^{-1}(ci2)$ and $cl2a = comp\_instantiates^{-1}(ci2a)$.

7.2.3 Applicative perspectives

Applicative perspectives concern the *DedalStudio* tool suite. *DedalModeler* can be improved by enhancing user control and adding other features like specifying architecture properties. A real architecture implementation using existing component technologies [Crnković et al., 2011] can be generated from the architecture configuration. The languages and features mentioned in technical perspectives can also be added to *DedalStudio*.

7.2.4 Threats to validity and experimental perspectives

Threats to the validity of our approach lie in the example scenarios that we have considered for experimental validation (*cf.* Chapter 6), as experimenting with real architecture descriptions might reveal scalability issues, for example. Indeed, the resolution time of solvers generally grows exponentially depending on the size of the problems. This is a classical inherent limitation of search-based engineering. A real-world case study is hence crucial to validate our approach. A theoretical study of the combinatorial complexity of the search space is also needed to investigate on more efficient heuristics. Alternatively, other solving approaches (using for instance constraint-solving techniques) can be evaluated and compared to the one proposed in this thesis.

Appendix A

Formal Dedal specifications (Home Automation Software Example)

A.1 Basic_concepts Machine

MACHINE

Basic_concepts

SETS

PARAM_NAMES = {*none* ,*intensity* };

PARAMETERS = {*nonep* ,*s_p5* ,*p8* };

INTERFACES = { *rintILight2* ,*rintILum* ,*rinITime2* ,*pintILum* ,*rintILamp2* ,*rintIIntensity* ,*rintIClock2* ,*pintIClock2* , *rintILight* ,*rintITime* ,*pintILight* ,*pintItime* ,*rintILamp* ,*rintIClock* ,*pintIClock* ,*pintILamp2* ,*pintIIntensity* };

TYPES = {*void*, *int*, *Date*};

INTERFACE_TYPES = {*ILuminosity*,*ILight* ,*ITime* ,*ILamp* ,*IClock* ,*IIntensity* };

SIGNATURES= {*s_sg320*,*sg14*,*sg15*,*sg16*,*sg33*,*sg34*,*sg35*,*sg36*,*sg350*,*sg360*};

SIG_NAMES = {*switchOn*,*switchOff*,*getTime*,*setDate*,*setIntensity*,*getIntensity*};

DIRECTION = {*PROVIDED*, *REQUIRED*}

VARIABLES

parameter, *interface*, *type*, *subtype*, *parameter_name*, *parameter_type*,

int_direction, *int_type*, *interfaceType*, *int_subtype*,

int_substitution, *param_subtype*, *int_compatible*, *signature*, *sig_name*, *parameters*,

sig_return, *int_signatures*, *sig_subtype*

INVARIANT

$type \subseteq TYPES \wedge$

$subtype \in type \leftrightarrow type \wedge$

$parameter \subseteq PARAMETERS \wedge$

$parameter\_name \in parameter \rightarrow PARAM\_NAMES \wedge$

$parameter\_type \in parameter \rightarrow type \wedge$

$signature \subseteq SIGNATURES \wedge$

$sig\_name \in signature \rightarrow SIG\_NAMES \wedge$

$parameters \in signature \rightarrow \mathcal{P}\ (parameter) \wedge$

$\forall\ (p1, p2, sig).(sig \in signature \wedge p1 \in parameter \wedge p1 \in parameters(sig) \wedge p2 \in parameter \wedge p2 \in parameters(sig) \Rightarrow (p1 \neq p2 \Leftrightarrow parameter\_name(p1) \neq parameter\_name(p2))) \wedge$

$sig\_return \in signature \rightarrow type \wedge$

/*Parameter specialization rule */

$param\_subtype \in signature \leftrightarrow signature \wedge$

$\forall\ (sig1, sig2).(sig1 \in signature \wedge sig2 \in signature \wedge sig1 \neq sig2 \Rightarrow ((sig1, sig2) \in param\_subtype \Leftrightarrow$

$\exists\ inj.(inj \in parameters(sig1)\ æ\ parameters(sig2) \wedge \forall\ param.(param \in parameter \wedge param \in \mathbf{dom}(inj) \Rightarrow parameter\_name(param) = parameter\_name(inj(param)) \wedge parameter\_type(inj(param)) : (subtype[\{parameter\_type(param)\}] \cup \{parameter\_type(param)\}))))) \wedge$

/*Signature specialization rule */

$sig\_subtype \in signature \leftrightarrow signature \wedge$

$\forall\ (sig1, sig2).(sig1 \in signature \wedge sig2 \in signature \wedge sig1 \neq sig2 \Rightarrow ((sig1, sig2) \in sig\_subtype \Leftrightarrow (sig\_name(sig1) = sig\_name(sig2) \wedge (sig2, sig1) \in param\_subtype \wedge sig\_return(sig2) \in subtype[\{sig\_return(sig1)\}] \cup \{sig\_return(sig1)\}))) \wedge$

$interfaceType \subseteq INTERFACE\_TYPES \wedge$

$int\_signatures \in interfaceType \rightarrow \mathcal{P}\ (signature) \wedge$

/*Interface subtyping rule */

$int\_subtype \in interfaceType \leftrightarrow interfaceType \wedge$

$\forall\ (intTypeA, intTypeB).(intTypeA \in interfaceType \wedge intTypeB \in interfaceType \wedge intTypeA \neq intTypeB \Rightarrow$

$((intTypeA, intTypeB) \in int\_subtype \Leftrightarrow \exists\ inj.(inj \in int\_signatures(intTypeA)\ æ\ int\_signatures(intTypeB) \wedge \forall\ sig.(sig \in signature \wedge sig \in int\_signatures(intTypeA) \Rightarrow (sig, inj(sig)) \in \mathbf{closure}(sig\_subtype) \vee sig{=}inj(sig))))) \wedge$

$interface \subseteq INTERFACES \wedge$

$int\_type \in interface \rightarrow interfaceType \wedge$

$int\_direction \in interface \rightarrow DIRECTION \wedge$

/*Interface substitution rule */

$int\_substitution \in interface \leftrightarrow interface \wedge$

$\forall\ (int\_sup,\ int\_sub).(int\_sup \in interface \wedge int\_sub \in interface \wedge int\_sup \neq int\_sub \Rightarrow$

$((int\_sup,\ int\_sub) \in int\_substitution \Leftrightarrow ((int\_type(int\_sub) \in int\_subtype[\{int\_type(int\_sup)\}] \cup \{int\_type(int\_sup)\}) \wedge int\_direction(int\_sup) = PROVIDED \wedge int\_direction(int\_sub) = PROVIDED) \vee ((int\_type(int\_sup) \in int\_subtype[\{int\_type(int\_sub)\}] \cup \{int\_type(int\_sub)\}) \wedge int\_direction(int\_sup) = REQUIRED \wedge int\_direction(int\_sub) = REQUIRED))) \wedge$

/*Interface compatibility rule */

$int\_compatible \in interface \leftrightarrow interface \wedge$

$\forall\ (intA,\ intB).(intA \in interface \wedge intB \in interface \wedge int\_direction(intA) \neq int\_direction(intB) \Rightarrow$

$((intA,\ intB) \in int\_compatible \Leftrightarrow$

$((int\_direction(intA) = PROVIDED \wedge int\_direction(intB) = REQUIRED \wedge (int\_type(intA) \in int\_subtype[\{int\_type(intB)\}] \cup \{int\_type(intB)\})) \vee$

$(int\_direction(intA) = REQUIRED \wedge int\_direction(intB) = PROVIDED \wedge (int\_type(intB) \in int\_subtype[\{int\_type(intA)\}] \cup \{int\_type(intA)\}))))))$

DEFINITIONS

$compatible\_interfaces1 == \{int1,\ int2 \mid int1 \in interface \wedge int2 \in interface \wedge int\_direction(int1) = PROVIDED \wedge int\_direction(int2) = REQUIRED \wedge int\_type(int1) \in int\_subtype[\{int\_type(int2)\}] \cup \{int\_type(int2)\}\};$

$compatible\_interfaces2 == \{int1,\ int2 \mid int1 \in interface \wedge int2 \in interface \wedge int\_direction(int1) = REQUIRED \wedge int\_direction(int2) = PROVIDED \wedge int\_type(int2) \in int\_subtype[\{int\_type(int1)\}] \cup \{int\_type(int1)\}\};$

$all\_compatible\_interfaces == compatible\_interfaces1 \cup compatible\_interfaces2;$

$all\_int\_substitution == \{int1,\ int2 \mid int1 \in interface \wedge int2 \in interface \wedge int1 \neq int2 \wedge$

$(((int\_type(int2) \in int\_subtype[\{int\_type(int1)\}] \cup \{int\_type(int1)\}) \wedge int\_direction(int1) = PROVIDED \wedge int\_direction(int2) = PROVIDED) \vee ((int\_type(int1) \in int\_subtype[\{int\_type(int2)\}] \cup \{int\_type(int2)\}) \wedge int\_direction(int1) = REQUIRED \wedge int\_direction(int2) = REQUIRED))\};$

$interface\_subtype == \{intTypeA,\ intTypeB \mid intTypeA \in interfaceType \wedge intTypeB \in interfaceType \wedge intTypeA \neq intTypeB \wedge$

$\exists\ inj.(inj \in int\_signatures(intTypeA)\ æ\ int\_signatures(intTypeB) \wedge \forall\ sig.(sig \in signature \wedge sig \in int\_signatures(intTypeA) \Rightarrow ((sig,\ inj(sig)) \in sig\_subtype \vee sig{=}inj(sig))))\};$

$signature\_subtype == \{sig1, sig2 \mid sig1 \in signature \wedge sig2 \in signature \wedge sig1 \neq sig2 \wedge sig\_name(sig1) = sig\_name(sig2) \wedge (sig2, sig1) \in param\_subtype \wedge sig\_return(sig2) \in subtype[\{sig\_return(sig1)\}] \cup \{sig\_return(sig1)\}\}$;

$sig\_param\_subtype == \{sig1, sig2 \mid sig1 \in signature \wedge sig2 \in signature \wedge sig1 \neq sig2 \quad \wedge$

$\exists\ inj.(inj \in parameters(sig1)\ æ\ parameters(sig2) \wedge \forall\ param.(param \in parameter \wedge param \in \mathbf{dom}(inj) \Rightarrow parameter\_name(param) = parameter\_name(inj(param)) \wedge (parameter\_type(inj(param)) \in subtype[\{parameter\_type(param)\}] \vee parameter\_type(param) = parameter\_type(inj(param)))))\}$

INITIALISATION

$type := \{void, int, Date\} \parallel$

$subtype := \varnothing \parallel$

$parameter := \{s\_p5,p8\} \parallel$

$parameter\_name := \{s\_p5 \mapsto intensity,p8 \mapsto intensity\} \parallel$

$parameter\_type := \{s\_p5 \mapsto int, p8 \mapsto int\} \parallel$

$signature := \{s\_sg320,sg14,sg15,sg16,sg33,sg34,sg35,sg36,sg350,sg360\ \} \parallel$

$sig\_name := \{s\_sg320 \mapsto setIntensity$

$,sg14 \mapsto switchOn$

$,sg15 \mapsto switchOff$

$,sg16 \mapsto getTime$

$,sg33 \mapsto switchOn$

$,sg34 \mapsto switchOff$

$,sg35 \mapsto getTime$

$,sg36 \mapsto setDate$

$,sg350 \mapsto setIntensity$

$,sg360 \mapsto getIntensity\} \parallel$

$parameters := \{s\_sg320 \mapsto \{s\_p5\ \}$

$,sg14 \mapsto \varnothing$

$,sg15 \mapsto \varnothing$

$,sg16 \mapsto \varnothing$

$,sg33 \mapsto \varnothing$

$,sg34 \mapsto \varnothing$

$,sg35 \mapsto \varnothing$

$,sg36 \mapsto \varnothing$

$,sg350 \mapsto \{p8\}$

$,sg360 \mapsto \varnothing\ \}\ ||$

$sig\_return := \{s\_sg320 \mapsto void$

$,sg14 \mapsto void$

$,sg15 \mapsto void$

$,sg16 \mapsto int$

$,sg33 \mapsto void$

$,sg34 \mapsto void$

$,sg35 \mapsto int$

$,sg36 \mapsto void$

$,sg350 \mapsto void$

$,sg360 \mapsto int\}\ ||$

$interfaceType := \{ILuminosity,ILight,ITime,ILamp,IClock,IIntensity\}\ ||$

$int\_signatures := \{ILuminosity \mapsto \{s\_sg320\},$

$ILight \mapsto \{sg14,sg15\},$

$ITime \mapsto \{sg16\},$

$ILamp \mapsto \{sg33,sg34\},$

$IClock \mapsto \{sg35,sg36\},$

$IIntensity \mapsto \{sg350,sg360\ \}\}\ ||$

$int\_subtype := \varnothing\ ||$

$interface := \{rintILight2, rintILum, rinITime2, pintILum, rintILamp2, rintIIntensity, rintIClock2, pintIClock2, rintILight, rintITime, pintILight, pintItime, rintILamp, rintIClock, pintIClock, pintILamp2, pintIIntensity\}\ ||$

$int\_type := \{rintILight2 \mapsto ILight$

$,rintILum \mapsto ILuminosity$

$,rinITime2 \mapsto ITime$

$,pintILum \mapsto ILuminosity$

$,rintILamp2 \mapsto ILamp$

$,rintIIntensity \mapsto IIntensity$

$,rintIClock2 \mapsto IClock$

$,pintIClock2 \mapsto IClock$

$,rintILight \mapsto ILight$

$,rintITime \mapsto ITime$

$,pintILight \mapsto ILight$

$,pintItime \mapsto ITime$

$,rintILamp \mapsto ILamp$

$,rintIClock \mapsto IClock$

$,pintIClock \mapsto IClock$

$,pintILamp2 \mapsto ILamp$

$,pintIIntensity \mapsto IIntensity\} \; ||$

$int\_direction := \{rintILight2 \mapsto REQUIRED$

$,rintILum \mapsto REQUIRED$

$,rinITime2 \mapsto REQUIRED$

$,pintILum \mapsto PROVIDED$

$,rintILamp2 \mapsto REQUIRED$

$,rintIIntensity \mapsto REQUIRED$

$,rintIClock2 \mapsto REQUIRED$

$,pintIClock2 \mapsto PROVIDED$

$,rintILight \mapsto REQUIRED$

$,rintITime \mapsto REQUIRED$

$,pintILight \mapsto PROVIDED$

$,pintItime \mapsto PROVIDED$

$,rintILamp \mapsto REQUIRED$

$,rintIClock \mapsto REQUIRED$

$,pintIClock \mapsto PROVIDED$

$,pintILamp2 \mapsto PROVIDED$

$,pintIIntensity \mapsto PROVIDED\} \; ||$

param_subtype := ∅ ||

sig_subtype := ∅ ||

sig_equals := ∅ ||

int_substitution := ∅ ||

int_compatible := ∅

OPERATIONS

initializeParam_subtype =

BEGIN

param_subtype := *sig_param_subtype*

END;

initializeSigSubtype =

BEGIN

sig_subtype := *signature_subtype_hierarchy*

END;

initializeIntSubtype =

BEGIN

int_subtype := *interface_subtype_hierarchy*

END;

computeSubstitutabilityAndCompatibility =

BEGIN

int_compatible := *all_compatible_interfaces* ||

int_substitution := *all_int_substitution*

END

END

A.2 Arch_concepts Machine

MACHINE

Arch_concepts

INCLUDES

Basic_concepts

SETS

$ARCHITECTURES = \{eArch, HASSpec\};$

$COMPONENTS = \{eRole, cr3a, cr4, ct3a, ct2a, cr3, cr1, cr2, ct3, ct2, ct1\};$

$COMP\_NAMES = \{HomeOrchestrator2, Luminosity, AndroidOrchestratorType, EmbeddedClockType, HomeOrchestrator, Light, Time, OrchestratorType, ClockType, AdjustableLampType\}$

VARIABLES

architecture, *arch_components*, *arch_connections*, *component*, *comp_name*, *connection*, *comp_interfaces*, *client*, *server*, *comp_substitution*, *comp_compatible*, *init*

INVARIANT

$component \subseteq COMPONENTS \wedge$

$comp\_name \in component \rightarrow COMP\_NAMES \wedge$

$comp\_interfaces \in component \text{ æ } \mathcal{P}\ (interface) \wedge$

/* component substitutability*/

$comp\_substitution \in component \leftrightarrow component \wedge$

$\forall\ (C\_sup, C\_sub).(C\_sup \in component \wedge C\_sub \in component \wedge C\_sup \neq C\_sub \Rightarrow$

$(C\_sub \in role\_substitution[\{C\_sup\}] \quad \Leftrightarrow \exists\ (inj1, inj2).(inj1 \in comp\_providedInterfaces(C\_sup) \text{ æ } comp\_providedInterfaces(C\_sub) \wedge \forall\ (int).(int \in interface \wedge \quad int \in comp\_providedInterfaces(C\_sup) \Rightarrow inj1(int) \in int\_substitution[\{int\}]) \wedge inj2 \in comp\_requiredInterfaces(C\_sub) \text{ æ } comp\_requiredInterfaces(C\_sup) \wedge \forall\ (int).(int \in interface \wedge \quad int \in comp\_requiredInterfaces(C\_sub) \Rightarrow int \in int\_substitution[\{inj2(int)\}]) \wedge (inj1 \neq \varnothing \vee inj2 \neq \varnothing\ ))))$

/* component compatibility*/

$comp\_compatible \in component \leftrightarrow component \quad \wedge$

$\forall\ (C1, C2).(C1 \in component \wedge C2 \in component \wedge C1 \neq C2 \Rightarrow$

$((C1, C2) \in comp\_compatible \Leftrightarrow \exists\ (int1, int2).(int1 \in interface \wedge int1 \in comp\_interfaces(C1) \wedge int2 \in interface \wedge int2 \in comp\_interfaces(C2) \wedge (int1, int2) \in int\_compatible))) \wedge$

$client \in component \leftrightarrow interface \wedge$

$\forall\ (c1).(c1 \in client \Rightarrow \forall\ (CR, int).(CR \in component \wedge int \in interface \wedge CR \in \mathbf{dom}(\{c1\}) \wedge int \in \mathbf{ran}(\{c1\}) \Rightarrow int \in comp\_interfaces(CR) \wedge int\_direction(int) = REQUIRED)) \wedge$

$\forall\ c1.(c1 \in client \Rightarrow \exists\ (comp, int).(comp \in component \wedge int \in interface \wedge int \in comp\_requiredInterfaces(comp) \wedge c1 = (comp, int))) \wedge$

$server \in component \leftrightarrow interface \wedge$

$\forall\ s1.(s1 \in server \Rightarrow \forall\ (CR, int).(CR \in component \wedge int \in interface \wedge CR \in \mathbf{dom}(\{s1\}) \wedge int \in \mathbf{ran}(\{s1\}) \Rightarrow int \in comp\_interfaces(CR) \wedge int\_direction(int) = PROVIDED)) \wedge$

$\forall\ s1.(s1 \in server \Rightarrow \exists\ (comp, int).(comp \in component \wedge int \in interface \wedge int \in comp\_providedInterfaces(comp) \wedge s1 = (comp, int))) \wedge$

$connection \in client \leftrightarrow server \wedge$

$\forall\ (c1, s1).(c1 \in client \wedge s1 \in server \Rightarrow ((c1, s1) \in connection \Rightarrow \exists\ (C1, C2, int1, int2).(C1 \in component \wedge C2 \in component \wedge C1 \neq C2 \wedge int1 \in interface \wedge int2 \in interface \wedge (C1,int1)=c1 \wedge (C2, int2)=s1 \wedge (int1, int2) \in int\_compatible))) \wedge$

$architecture \subseteq ARCHITECTURES \wedge$

$arch\_components \in architecture \rightarrow \mathcal{P}\ (component) \wedge$

$arch\_connections \in architecture \rightarrow \mathcal{P}\ (connection) \wedge$

$init \in \mathbf{BOOL}$

DEFINITIONS

$comp\_providedInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = PROVIDED\};$

$comp\_requiredInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = REQUIRED\};$

$comp\_elem == \{cl, comp \mid cl \in component \times interface \wedge comp \in component \wedge \exists\ int.(int \in interface \wedge cl = (comp, int))\};$

$interface\_elem == \{cl, int \mid cl \in component \times interface \wedge int \in interface \wedge \exists\ comp.(comp \in component \wedge cl = (comp, int))\};$

$all\_comp\_compatible == \{c1, c2 \mid c1 \in component \wedge c2 \in component \wedge c1 \neq c2 \wedge \exists\ (int1, int2).(int1 \in interface \wedge int1 \in comp\_interfaces(c1) \wedge int2 \in interface \wedge int2 \in comp\_interfaces(c2) \wedge (int1, int2) \in int\_compatible)\};$

$all\_comp\_substitution == \{c1, c2 \mid c1 \in component \wedge c2 \in component \wedge c1 \neq c2 \wedge \exists\ (inj1, inj2).(inj1 \in comp\_providedInterfaces(c1)\ æ\ comp\_providedInterfaces(c2) \wedge (inj1 \neq \varnothing \vee inj2 \neq \varnothing\ ) \wedge \forall\ (int).(int \in interface \wedge \quad int \in comp\_providedInterfaces(c1) \Rightarrow inj1(int) \in int\_substitution[\{int\}]) \wedge inj2 \in comp\_requiredInterfaces(c2)\ æ\ comp\_requiredInterfaces(c1) \wedge \forall\ (int).(int \in interface \wedge \quad int \in comp\_requiredInterfaces(c2) \Rightarrow inj2(int) \in int\_substitution[\{int\}]))\};$

$all\_clients == \{comp, int \mid comp \in component \wedge int \in interface \wedge int\_direction(int) = REQUIRED \wedge int \in comp\_interfaces(comp)\};$

$all\_servers == \{comp, int \mid comp \in component \wedge int \in interface \wedge int\_direction(int) = PROVIDED \wedge int \in comp\_interfaces(comp)\}$;

$all\_connections == \{cl, se \mid cl \in client \wedge se \in server \wedge \exists (c1, c2, int1, int2).(c1 \in component \wedge c2 \in component \wedge c1 \neq c2 \wedge int1 \in interface \wedge int2 \in interface \wedge (c1 \mapsto int1)=cl \wedge (c2 \mapsto int2)=se \wedge (int1, int2) \in int\_compatible)\}$

INITIALISATION

$component := \{cr3a, cr4, ct3a, ct2a, cr3, cr1, cr2, ct3, ct2, ct1\} \parallel$

$comp\_name := \{(cr3a \mapsto HomeOrchestrator2)$

$,(cr4 \mapsto Luminosity)$

$,(ct3a \mapsto AndroidOrchestratorType)$

$,(ct2a \mapsto EmbeddedClockType)$

$,(cr3 \mapsto HomeOrchestrator)$

$,(cr1 \mapsto Light)$

$,(cr2 \mapsto Time)$

$,(ct3 \mapsto OrchestratorType)$

$,(ct2 \mapsto ClockType)$

$,(ct1 \mapsto AdjustableLampType)\ \} \parallel$

$comp\_interfaces := \{cr3a \mapsto \{rintILight2, rintILum, rinITime2\}$

$,cr4 \mapsto \{pintILum\}$

$,ct3a \mapsto \{rintILamp2, rintIIntensity, rintIClock2\}$

$,ct2a \mapsto \{pintIClock2\}$

$,cr3 \mapsto \{rintILight, rintITime\}$

$,cr1 \mapsto \{pintILight\}$

$,cr2 \mapsto \{pintItime\}$

$,ct3 \mapsto \{rintILamp,rintIClock\}$

$,ct2 \mapsto \{pintIClock\}$

$,ct1 \mapsto \{pintILamp2,pintIIntensity\}\} \parallel$

$client := \{(cr3 \mapsto rintILight)$

$,(cr3 \mapsto rintITime)$

$,(ct3 \mapsto rintIClock)$

$,(ct3 \mapsto rintILamp)\}$ ||

$server := \{(cr1 \mapsto pintILight)$

$,(cr2 \mapsto pintItime)$

$,(ct2 \mapsto pintIClock)$

$,(ct1 \mapsto pintILamp2)\}$ ||

$connection := \{((cr3 \mapsto rintILight) \mapsto (cr1 \mapsto pintILight))$

$,((cr3 \mapsto rintITime) \mapsto (cr2 \mapsto pintItime))$

$,((ct3 \mapsto rintIClock) \mapsto (ct2 \mapsto pintIClock))$

$,((ct3 \mapsto rintILamp) \mapsto (ct1 \mapsto pintILamp2))\}$ ||

$architecture := \{HASSpec\}$ ||

$arch\_components := \{HASSpec \mapsto \{cr3,cr1,cr2\}\}$ ||

$arch\_connections := \varnothing$ ||

$comp\_substitution := \varnothing$ ||

$comp\_compatible := \varnothing$ ||

$init :=$ **FALSE**

OPERATIONS

/*Initialization operations*/

initializeArchModel =

BEGIN

$client := all\_clients$ ||

$server := all\_servers$ ||

$comp\_compatible := all\_comp\_compatible$ ||

$comp\_substitution := all\_comp\_substitution$

END;

initializeConnections =

BEGIN

$connection := all\_connections$

END;

initializeParameterSubtype =

```
    BEGIN
        initializeParam_subtype
    END;
    initializesignatureSubtype =
    BEGIN
        initializeSigSubtype
    END;
    initializeInterfaceSubtype =
    BEGIN
        initializeIntSubtype
    END;
    initializeBasicModel    =
    BEGIN
        computeSubstitutabilityAndCompatibility ||
        init := TRUE
    END
END
```

A.3 Arch_specification Machine

MACHINE

Arch_specification

USES

Arch_concepts

CONSTANTS

COMP_ROLES

PROPERTIES

$COMP\_ROLES \subseteq COMPONENTS \wedge$

$COMP\_ROLES = \{eRole, cr3a, cr4, cr3, cr1, cr2\}$

VARIABLES

specification, *spec_components*, *spec_connections*, *compRole*, *spec_clients*, *spec_servers*, *role_substitution*, *role_compatible*

INVARIANT

$compRole \subseteq COMP\_ROLES \wedge$

$specification \subseteq ARCHITECTURES \wedge$

$spec\_connections \in specification \rightarrow \mathcal{P}\ (connection) \wedge$

$spec\_components \in specification \rightarrow \mathcal{P}\ (compRole) \wedge$

$spec\_clients \in specification \rightarrow \mathcal{P}\ (client) \wedge$

$\forall\ (spec, cl).(spec \in specification \wedge cl \in client \Rightarrow$

$(cl \in spec\_clients(spec) \Leftrightarrow \exists\ comp.(comp \in compRole \wedge comp \in \mathbf{dom}(\{cl\}) \wedge comp \in spec\_components(spec)))) \wedge$

$spec\_servers \in specification \rightarrow \mathcal{P}\ (server) \wedge$

$\forall\ (spec, se).(spec \in specification \wedge se \in server \Rightarrow$

$(se \in spec\_servers(spec) \Leftrightarrow \exists\ comp.(comp \in compRole \wedge comp \in \mathbf{dom}(\{se\}) \wedge comp \in spec\_components(spec)))) \wedge$

DEFINITIONS

$comp\_providedInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = PROVIDED\};$

$comp\_requiredInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = REQUIRED\};$

$all\_role\_compatible == \{c1, c2 \mid c1 \in compRole \land c2 \in compRole \land c1 \neq c2 \land \exists (int1, int2).(int1 \in interface \land int1 \in comp\_interfaces(c1) \land int2 \in interface \land int2 \in comp\_interfaces(c2) \land (int1, int2) \in int\_compatible)\};$

$all\_role\_substitution == \{c1, c2 \mid c1 \in compRole \land c2 \in compRole \land c1 \neq c2 \land \exists (inj1, inj2).(inj1 \in comp\_providedInterfaces(c1)$ æ $comp\_providedInterfaces(c2) \land \forall (int).(int \in interface \land \ int \in comp\_providedInterfaces(c1) \ \Rightarrow inj1(int) \in int\_substitution[\{int\}]) \land inj2 \in comp\_requiredInterfaces(c2)$ æ $comp\_requiredInterfaces(c1) \land \forall (int).(int \in interface \land \ int \in comp\_requiredInterfaces(c2) \Rightarrow int \in int\_substitution[\{inj2(int)\}]) \land (inj1 \neq \varnothing \lor inj2 \neq \varnothing))\};$

$clientInterfaceElem == \{cl, int \mid cl \in client \land \exists (comp, rint).(comp \in component \land rint \in interface \land cl = (comp,rint) \land int = rint)\};$

$clientComponentElem == \{cl, cr \mid cl \in client \land \exists (comp, rint).(comp \in component \land rint \in interface \land cl = (comp,rint) \land cr = comp)\};$

$serverInterfaceElem == \{se, int \mid se \in server \land \exists (comp, pint).(comp \in component \land pint \in interface \land se = (comp,pint) \land int = pint)\};$

$serverComponentElem == \{se, cr \mid se \in server \land \exists (comp, pint).(comp \in component \land pint \in interface \land se = (comp,pint) \land cr = comp)\};$

$clients(comp) == \{cl \mid comp \in component \land cl \in client \land clientComponentElem(cl) = comp\};$

$servers(comp) == \{se \mid comp \in component \land se \in server \land serverComponentElem(se) = comp\};$

INITIALISATION

$compRole := \{cr3a, cr4, cr3, cr1, cr2\} \ ||$

$specification := \{HASSpec\} \ ||$

$spec\_components := \{HASSpec \mapsto \{cr3,cr1,cr2\}\} \ ||$

$spec\_connections := \{HASSpec \mapsto \{((cr3 \mapsto rintILight) \mapsto (cr1 \mapsto pintILight))$

$,((cr3 \mapsto rintITime) \mapsto (cr2 \mapsto pintItime))\}\} \ ||$

$spec\_clients := \{HASSpec \mapsto \{(cr3 \mapsto rintILight)$

$,(cr3 \mapsto rintITime)\}\} \ ||$

$spec\_servers := \{HASSpec \mapsto \{(cr1 \mapsto pintILight), (cr2 \mapsto pintItime)\}\} \ ||$

$role\_substitution := \varnothing \ ||$

$role\_compatible := \varnothing$

OPERATIONS

/*initialization operations*/

initialzeArchSpecModel =

BEGIN

role_substitution := *all_role_substitution* ||

role_compatible := *all_role_compatible*

END;

s_initializeParameterSubtype =

BEGIN

initializeParameterSubtype

END;

s_initializesignatureSubtype =

BEGIN

initializesignatureSubtype

END;

s_initializeInterfaceSubtype =

BEGIN

initializeInterfaceSubtype

END;

s_initializeBasicModel =

BEGIN

initializeBasicModel

END;

s_initializeArchModel =

BEGIN

initializeArchModel

END;

s_initializeConnections =

BEGIN

initializeConnections

END;

/* Model manipulation operations*/

addRole(*spec*, *newRole*) =

PRE

$spec \in specification \wedge newRole \in compRole \wedge newRole \notin spec\_components(spec) \wedge$

$\forall\ cr.(cr \in compRole \wedge cr \in spec\_components(spec) \Rightarrow comp\_name(cr) \neq comp\_name(newRole))$

THEN

spec_components$(spec) := spec\_components(spec) \cup \{newRole\}$ ||

spec_servers$(spec) := spec\_servers(spec) \cup servers(newRole)$ ||

spec_clients$(spec) := spec\_clients(spec) \cup clients(newRole)$

END;

connect$(cl, se, spec) =$

PRE

$spec \in specification \wedge cl \in client \wedge se \in server \wedge cl \in spec\_clients(spec) \wedge se \in spec\_servers(spec) \wedge$

$(cl \mapsto se) \in connection \wedge (cl \mapsto se) \notin spec\_connections(spec)$

THEN

spec_connections$(spec) := spec\_connections(spec) \cup \{(cl \mapsto se)\}$

END;

removeRole$(spec, role) =$

PRE

$spec \in specification \wedge role \in compRole \wedge role \in spec\_components(spec) \wedge$

$\forall\ cl.(cl \in client \wedge cl \in clients(role) \Rightarrow \forall\ se.(se \in server \wedge se \in spec\_servers(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec))) \wedge$

$\forall\ (se).(se \in server \wedge se \in servers(role) \Rightarrow \forall\ cl.(cl \in client \wedge cl \in spec\_clients(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec)))$

THEN

spec_clients$(spec) := spec\_clients(spec)$ - $clients(role)$ ||

spec_servers$(spec) := spec\_servers(spec)$ - $servers(role)$ ||

spec_components$(spec) := spec\_components(spec)$ - $\{role\}$

END;

disconnect$(spec, se, cl) =$

PRE

$spec \in specification \wedge cl \in client \wedge se \in server \wedge cl \in spec\_clients(spec) \wedge se \in spec\_servers(spec) \wedge$

$(cl \mapsto se) \in connection \wedge (cl \mapsto se) \in spec\_connections(spec)$

THEN

spec_connections($spec$) := $spec\_connections(spec)$ - $\{(cl \mapsto se)\}$

END;

replaceRole($spec$, $oldRole$, $newRole$) =

PRE

$oldRole \in compRole \wedge newRole \in compRole \wedge spec \in specification \wedge oldRole \in spec\_components(spec) \wedge$

$newRole \notin spec\_components(spec) \wedge (oldRole, newRole) \in role\_substitution \wedge$

$\forall cl.(cl \in client \wedge cl \in clients(oldRole) \Rightarrow \forall se.(se \in server \wedge se \in spec\_servers(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec))) \wedge$

$\forall (se).(se \in server \wedge se \in servers(oldRole) \Rightarrow \forall cl.(cl \in client \wedge cl \in spec\_clients(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec)))$

THEN

spec_components($spec$) := ($spec\_components(spec)$ - $\{oldRole\}$) $\cup$ $\{newRole\}$ ||

spec_servers($spec$) := ($spec\_servers(spec)$ - $servers(oldRole)$) $\cup$ $servers(newRole)$ ||

spec_clients($spec$) := ($spec\_clients(spec)$ - $clients(oldRole)$) $\cup$ $clients(newRole)$

END

END

A.4 Arch_configuration Machine

MACHINE

Arch_configuration

USES

Arch_specification

SETS

$COMP\_CLASS = \{eClass, cl3a, cl2a, cl3, cl2, cl1\}$;

$CLASS\_NAME = \{AndroidOrchestrator, EmbeddedClock, Orchestrator, Clock, AdjustableLamp\}$;

$ATTRIBUTES$;

$CONFIGURATIONS = \{eConfig, HASConfig\}$;

ATT_NAMES

CONSTANTS

$COMP\_TYPES$

PROPERTIES

$COMP\_TYPES \subseteq COMPONENTS \wedge$

$COMP\_TYPES = \{ct3a, ct2a, ct3, ct2, ct1\}$

VARIABLES

configuration, *config_components*, *config_connections*, *implements*,

compType, *compType_substitution*, *compType_compatible*, *matches*,

compClass, *class_name*, *class_implements*, *class_attributes*, *realizes*,

compositeComp, *delegatedInterface*, *delegation*, *composite_uses*,

config_clients, *config_servers*, *documents*,

attribute, *attribute_name*, *attribute_type*, *initSpec*

INVARIANT

$compType \subseteq COMP\_TYPES \wedge$

/*matching rule*/

$matches \in compType \leftrightarrow compRole \wedge$

$\forall (CR, CT).(CR \in compRole \wedge CT \in compType \Rightarrow$

$((CT, CR) \in matches \Leftrightarrow (CR, CT) \in comp\_substitution)) \wedge$

$compClass \subseteq COMP\_CLASS \wedge$

$class\_name \in compClass \rightarrow CLASS\_NAME \wedge$

$attribute \subseteq ATTRIBUTES \wedge$

$attribute\_name \in attribute \rightarrow ATT\_NAMES \wedge$

$attribute\_type \in attribute \rightarrow TYPES \wedge$

$class\_attributes \in compClass \rightarrow \mathcal{P}\ (attribute) \wedge$

$class\_implements \in compClass \rightarrow compType \wedge$

/*realization rule*/

$realizes \in compClass \leftrightarrow compRole \wedge$

$\forall\ (CL,\ CR).(CL \in compClass \wedge CR \in compRole \Rightarrow$

$((CL,\ CR) \in realizes \Leftrightarrow \exists\ CT.(CT \in compType \wedge (CT,\ CR) \in matches \wedge (CL,\ CT) \in class\_implements))) \wedge$

$configuration \subseteq \quad CONFIGURATIONS \wedge$

$config\_components \in configuration \rightarrow \mathcal{P}_1\ (compClass) \wedge$

$\forall\ (cl1,\ cl2,\ C1).(cl1 \in compClass \wedge cl2 \in compClass \wedge cl1 \neq cl2 \wedge C1 \in configuration \wedge \{cl1,\ cl2\} \subseteq config\_components(C1) \Rightarrow class\_name(cl1) \neq class\_name(cl2)) \wedge$

$implements \in configuration \leftrightarrow specification \wedge$

$documents \in specification \leftrightarrow configuration \wedge$

$compositeComp \subseteq compClass \wedge$

$delegatedInterface \subset interface \wedge$

$delegation \in delegatedInterface\ æ\ interface \wedge$

$\forall\ (int1,\ int2).(int1 \in interface \wedge int2 \in interface \wedge int1 \neq int2 \wedge (int1,\ int2) \in delegation \Rightarrow int\_type(int1) = int\_type(int2) \wedge int\_direction(int1) = int\_direction(int2)) \wedge$

$composite\_uses \in compositeComp \rightarrow configuration \wedge$

$config\_connections \in configuration \rightarrow \mathcal{P}\ (connection) \wedge$

$config\_clients \in configuration \rightarrow \mathcal{P}\ (client) \wedge$

$config\_servers \in configuration \rightarrow \mathcal{P}\ (server) \wedge$

$initSpec \in \mathbf{BOOL}$

DEFINITIONS

$comp\_providedInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = PROVIDED\};$

$comp\_requiredInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = REQUIRED\}$;

$clientInterfaceElem == \{cl, int \mid cl \in client \wedge \exists (comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp,rint) \wedge int = rint)\}$;

$clientComponentElem == \{cl, cr \mid cl \in client \wedge \exists (comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp,rint) \wedge cr = comp)\}$;

$serverInterfaceElem == \{se, int \mid se \in server \wedge \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp,pint) \wedge int = pint)\}$;

$serverComponentElem == \{se, cr \mid se \in server \wedge \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp,pint) \wedge cr = comp)\}$;

$clients(class) == \{cl \mid class \in compClass \wedge cl \in client \wedge clientComponentElem(cl) = class\_implements(class)\}$;

$servers(class) == \{se \mid class \in compClass \wedge se \in server \wedge serverComponentElem(se) = class\_implements(class)\}$;

$class\_substitution == (class\_implements ; compType\_substitution ; class\_implements^{-1})$;

$all\_compType\_compatible == \{c1, c2 \mid c1 \in compType \wedge c2 \in compType \wedge c1 \neq c2 \wedge \exists (int1, int2).(int1 \in interface \wedge int1 \in comp\_interfaces(c1) \wedge int2 \in interface \wedge int2 \in comp\_interfaces(c2) \wedge (int1, int2) \in int\_compatible)\}$;

$all\_compType\_substitution == \{c1, c2 \mid c1 \in compType \wedge c2 \in compType \wedge c1 \neq c2 \wedge \exists (inj1, inj2).(inj1 \in comp\_providedInterfaces(c1) \text{ æ } comp\_providedInterfaces(c2) \wedge \forall (int).(int \in interface \wedge int \in comp\_providedInterfaces(c1) \Rightarrow inj1(int) \in int\_substitution[\{int\}]) \wedge inj2 \in comp\_requiredInterfaces(c2) \text{ æ } comp\_requiredInterfaces(c1) \wedge \forall (int).(int \in interface \wedge int \in comp\_requiredInterfaces(c2) \Rightarrow inj2(int) \in int\_substitution[\{int\}]) \wedge (inj1 \neq \varnothing \vee inj2 \neq \varnothing))\}$;

$all\_matches == \{ct, cr \mid ct \in compType \wedge cr \in compRole \wedge \exists (inj1, inj2).(inj1 \in comp\_providedInterfaces(cr) \text{ æ } comp\_providedInterfaces(ct) \wedge \forall (int).(int \in interface \wedge int \in comp\_providedInterfaces(cr) \Rightarrow inj1(int) \in int\_substitution[\{int\}]) \wedge inj2 \in comp\_requiredInterfaces(cr) \text{ æ } comp\_requiredInterfaces(ct) \wedge \forall (int).(int \in interface \wedge int \in comp\_requiredInterfaces(cr) \Rightarrow int \in int\_substitution[\{inj2(int)\}]))\}$;

$all\_realizes == \{cl, cr \mid cl \in compClass \wedge cr \in compRole \wedge \exists ct.(ct \in compType \wedge (ct, cr) \in matches \wedge (cl, ct) \in class\_implements)\}$;

INITIALISATION

$configuration := \{HASConfig\} \parallel$

$config\_components := \{HASConfig \mapsto \{cl3,cl2,cl1\}\} \parallel$

$implements := \{(HASConfig \mapsto HASSpec)\} \parallel$

$documents := \{HASSpec \mapsto HASConfig\} \parallel$

$compType := \{ct3a,ct2a,ct3,ct2,ct1\} \parallel$

compType_substitution := ∅ ||

compType_compatible := ∅ ||

matches := ∅ ||

compClass := {*cl2a*,*cl3a*,*cl3*,*cl2*,*cl1*} ||

class_name := {*cl3a* ↦ *AndroidOrchestrator*

,*cl2a* ↦ *EmbeddedClock*

,*cl3* ↦ *Orchestrator*

,*cl2* ↦ *Clock*

,*cl1* ↦ *AdjustableLamp*} ||

class_implements := {*cl3a* ↦ *ct3a*, *cl2a* ↦ *ct2a*, *cl3* ↦ *ct3*, *cl2* ↦ *ct2*, *cl1* ↦ *ct1*} ||

class_attributes := ∅ ||

realizes := ∅ ||

attribute := ∅ ||

attribute_name := ∅ ||

attribute_type := ∅ ||

compositeComp := ∅ ||

delegatedInterface := ∅ ||

delegation := ∅ ||

composite_uses := ∅ ||

config_connections := {*HASConfig* ↦ {((*ct3* ↦ *rintIClock*) ↦ (*ct2* ↦ *pintIClock*))

,((*ct3* ↦ *rintILamp*) ↦ (*ct1* ↦ *pintILamp2*))}} ||

config_clients := {*HASConfig* ↦ {(*ct3* ↦ *rintIClock*)

,(*ct3* ↦ *rintILamp*)}} ||

config_servers := {*HASConfig* ↦ {(*ct2* ↦ *pintIClock*)

,(*ct1* ↦ *pintILamp2*)}} ||

initSpec := **FALSE**

OPERATIONS

/*initialization operations*/

initializeArchConfig =

BEGIN

compType_compatible := *all_compType_compatible* ||

compType_substitution := *all_compType_substitution* ||

matches := *all_matches*

END;

computeRealizations =

BEGIN

realizes := *all_realizes*

END;

/*model manipulation operations*/

addClass(*config*, *newClass*) =

PRE

$config \in configuration \wedge newClass \in compClass \wedge newClass \notin config\_components(config) \wedge$

$\forall\ cl.(cl \in compClass \wedge cl \in config\_components(config) \Rightarrow class\_name(cl) \neq class\_name(newClass) \wedge class\_implements(newClass) \neq class\_implements(cl))$

THEN

config_clients(*config*) := *config_clients*(*config*) $\cup$ *clients*(*newClass*) ||

config_components(*config*) := *config_components*(*config*) $\cup$ {*newClass*}

END;

addServer(*config*, *se*) =

PRE

$config \in configuration \wedge se \in server \wedge se \notin config\_servers(config) \wedge serverComponentElem(se) \in class\_implements[config\_components(config)]$

THEN

config_servers(*config*) := *config_servers*(*config*) $\cup$ {*se*}

END;

class_connect(*cl*, *se*, *config*) =

PRE

$config \in configuration \wedge cl \in client \wedge se \in server \wedge cl \in config\_clients(config) \wedge se \in config\_servers(config) \wedge$

$(cl \mapsto se) \in connection \wedge (cl \mapsto se) \notin config\_connections(config)$

THEN

config_connections$(config) := config\_connections(config) \cup \{(cl \mapsto se)\}$

END;

removeClass$(config,\ class) =$

PRE

$config \in configuration \wedge class \in compClass \wedge class \in config\_components(config) \wedge$

$\forall\ cl.(cl \in client \wedge cl \in clients(class) \Rightarrow \forall\ se.(se \in server \wedge se \in config\_servers(config) \Rightarrow (cl \mapsto se) \notin config\_connections(config))) \wedge$

$\forall\ se.(se \in server \wedge se \in servers(class) \Rightarrow se \notin config\_servers(config))$

THEN

config_clients$(config) := config\_clients(config)$ - $clients(class)$ ||

config_components$(config) := config\_components(config)$ - $\{class\}$

END;

deleteServer$(config,\ se) =$

PRE

$config \in configuration \wedge se \in server \wedge se \in config\_servers(config) \wedge \forall\ cl.(cl \in client \wedge cl \in config\_clients(config) \Rightarrow$

$(cl \mapsto se) \notin config\_connections(config))$

THEN

config_servers$(config) := config\_servers(config)$ - $\{se\}$

END;

class_disconnect$(config,\ se,\ cl) =$

PRE

$config \in configuration \wedge cl \in client \wedge se \in server \wedge cl \in config\_clients(config) \wedge se \in config\_servers(config) \wedge$

$(cl \mapsto se) \in connection \wedge (cl \mapsto se) \in config\_connections(config)$

THEN

config_connections$(config) := config\_connections(config)$ - $\{(cl \mapsto se)\}$

END;

replaceClass(*config*, *oldClass*, *newClass*) =

PRE

$oldClass \in compClass \wedge newClass \in compClass \wedge config \in configuration \wedge oldClass \in config\_components(config) \wedge$

$newClass \notin config\_components(config) \wedge (oldClass, newClass) \in class\_substitution \wedge$

$\forall\ cl.(cl \in client \wedge cl \in clients(oldClass) \Rightarrow \forall\ se.(se \in server \wedge se \in config\_servers(config) \Rightarrow (cl \mapsto se) \notin config\_connections(config))) \wedge$

$\forall\ (se).(se \in server \wedge se \in servers(oldClass) \Rightarrow se \notin config\_servers(config))$

THEN

config_components(*config*) := (*config_components*(*config*) - {*oldClass*}) ∪ {*newClass*} ||

config_clients(*config*) := (*config_clients*(*config*) - *clients*(*oldClass*)) ∪ *clients*(*newClass*)

END

END

A.5 Arch_assembly Machine

MACHINE

Arch_assembly

USES

Arch_specification, *Arch_configuration*

SETS

$COMP\_INSTANCES = \{eInst, ci3a, ci2a, ci3, ci12, ci12, ci2\}$;

$INTERFACE\_INSTANCES = \{rintILamp2Inst1, rintIIntensityInst1, rintIClock2Inst, pintIClock2Inst, rintILampInst1, rintIClockInst1, pintILamp2Inst2, pintIItensityInst2, pintILamp2Inst1, pintIIntensityInst1, pintIClockInst1\}$;

$INSTANCE\_NAME = \{androidOrchestrator, embeddedClock, orchestrator1, lampDesk, lampSitting, clock\}$;

$ASSEMBLIES = \{eAssm, HASAssm\}$;

$ATTRIBUTES\_VALUES$

VARIABLES

compInstance, *compInstance_name*, *interfaceInstance*, *int_instantiates*, *instInterfaces*, *comp_instantiates*,

initiation_state, *current_state*, *attribute_value*, *assembly*, *assm_components*, *instantiates*, *binding*, *assm_connections*,

client_instance, *server_instance*, *max_instances*, *nb_instances*, *assm_servers*, *assm_clients*

INVARIANT

$compInstance \subseteq COMP\_INSTANCES \wedge$

$compInstance\_name \in compInstance \rightarrow INSTANCE\_NAME \wedge$

$nb\_instances \in compClass \rightarrow \mathbf{NAT} \wedge$

$max\_instances \in compClass \rightarrow \mathbf{NAT} \wedge$

$interfaceInstance \subseteq INTERFACE\_INSTANCES \wedge$

$int\_instantiates \in interfaceInstance \rightarrow interface \wedge$

$instInterfaces \in compInstance \rightarrow \mathcal{P}_1\ (interfaceInstance) \wedge$

$comp\_instantiates \in compInstance \rightarrow compClass \wedge$

$\forall\ (ci, cl).(ci \in compInstance \wedge cl \in compClass \Rightarrow$

$(cl = comp\_instantiates(ci) \Rightarrow \exists\ surj.(surj \in instInterfaces(ci)\ \imath\ class\_interfaces(cl) \wedge$

$\forall\ int.(int \in interfaceInstance \wedge int \in instInterfaces(ci) \Rightarrow surj(int) = int\_instantiates(int))))) \wedge$

$attribute\_value \in attribute \rightarrow \quad ATTRIBUTES\_VALUES \wedge$

$initiation\_state \in compInstance \rightarrow \mathcal{P}\ (attribute\_value) \wedge$

$\forall\ (ci,\ rl).(ci \in compInstance \wedge rl \in \mathcal{P}\ (attribute\_value) \wedge (ci,\ rl) \in initiation\_state \Rightarrow \forall\ att.(att \in attribute \wedge att \in \mathbf{dom}(rl) \Rightarrow$

$\exists\ cc.(cc \in compClass \wedge (ci,cc) \in comp\_instantiates \wedge att \in class\_attributes(cc)))) \wedge$

$current\_state \in compInstance \rightarrow \mathcal{P}\ (attribute\_value) \wedge$

$\forall\ (ci,\ rl).(ci \in compInstance \wedge rl \in \mathcal{P}\ (attribute\_value) \wedge (ci,\ rl) \in current\_state \Rightarrow \forall\ att.(att \in attribute \wedge att \in \mathbf{dom}(rl) \Rightarrow$

$\exists\ cc.(cc \in compClass \wedge (ci,cc) \in comp\_instantiates \wedge att \in class\_attributes(cc)))) \wedge$

$client\_instance \in compInstance \leftrightarrow interfaceInstance \wedge$

$\forall\ (ci,\ int).(ci \in compInstance \wedge int \in interfaceInstance \wedge (ci,\ int) \in client\_instance \Rightarrow inst\_direction(int) = REQUIRED) \wedge$

$server\_instance \in compInstance \leftrightarrow interfaceInstance \wedge$

$\forall\ (ci,\ int).(ci \in compInstance \wedge int \in interfaceInstance \wedge (ci,\ int) \in server\_instance \Rightarrow inst\_direction(int) = PROVIDED) \wedge$

$binding \in client\_instance \leftrightarrow server\_instance \wedge$

$assembly \subseteq ASSEMBLIES \wedge$

$assm\_components \in assembly \rightarrow \mathcal{P}_1\ (compInstance) \wedge$

$assm\_servers \in assembly \rightarrow \mathcal{P}\ (server\_instance) \wedge$

$assm\_clients \in assembly \rightarrow \mathcal{P}\ (client\_instance) \wedge$

$assm\_connections \in assembly \rightarrow \mathcal{P}\ (binding) \wedge$

$instantiates \in assembly \rightarrow configuration \wedge$

DEFINITIONS

$inst\_direction(int) == int\_direction(int\_instantiates(int));$

$class\_interfaces(cl) == comp\_interfaces(class\_implements(cl));$

$clientInterfaceElem == \{cl,\ int \mid cl \in clientInstance \wedge \exists\ (inst,\ rint).(inst \in compInstance \wedge rint \in interfaceInstance \wedge cl = (inst,rint) \wedge int = rint)\};$

$clientComponentElem == \{cl,\ ci \mid cl \in client\_instance \wedge \exists\ (inst,\ rint).(inst \in compInstance \wedge rint \in interfaceInstance \wedge cl = (inst,rint) \wedge ci = inst)\};$

$serverInterfaceElem == \{se,\ int \mid se \in server\_instance \wedge \exists\ (inst,\ pint).(inst \in compInstance \wedge pint \in interfaceInstance \wedge se = (inst,pint) \wedge int = pint)\};$

$serverComponentElem == \{se,\ ci \mid se \in server\_instance \wedge \exists\ (inst,\ pint).(inst \in compInstance \wedge pint \in interfaceInstance \wedge se = (inst,pint) \wedge ci = inst)\};$

$clients(inst) == \{cl \mid inst \in compInstance \wedge cl \in client\_instance \wedge clientComponentElem(cl) = inst\};$

$servers(inst) == \{se \mid inst \in compInstance \wedge se \in server\_instance \wedge serverComponentElem(se) = inst\};$

$inst\_substitution == (comp\_instantiates;class\_implements\ ;\ compType\_substitution\ ;$
$class\_implements^{-1}\ ;comp\_instantiates^{-1}\ );$

$all\_clientInstances == \{ci, int \mid ci \in compInstance \wedge int \in interfaceInstance \wedge$
$int\_direction(int\_instantiates(int)) = REQUIRED \wedge int \in instInterfaces(ci)\};$

$all\_serverInstances == \{ci, int \mid ci \in compInstance \wedge int \in interfaceInstance \wedge$
$int\_direction(int\_instantiates(int)) = PROVIDED \wedge int \in instInterfaces(ci)\};$

$all\_bindings == \{cl, se \mid cl \in client\_instance \wedge se \in server\_instance \wedge \exists\ (c1, c2, int1, int2).(c1 \in$
$compInstance \wedge c2 \in compInstance \wedge c1 \neq c2 \wedge int1 \in interfaceInstance \wedge int2 \in interfaceInstance \wedge (c1 \mapsto$
$int1)=cl \wedge (c2 \mapsto int2)=se \wedge (int\_instantiates(int1), int\_instantiates(int2)) \in int\_compatible)\}$

INITIALISATION

$compInstance := \{ci2a, ci3a, ci3, ci11, ci12, ci2\} \parallel$

$compInstance\_name := \{(ci3a \mapsto androidOrchestrator)$

$,(ci2a \mapsto embeddedClock)$

$,(ci3 \mapsto orchestrator1)$

$,(ci11 \mapsto lampDesk)$

$,(ci12 \mapsto lampSitting)$

$,(ci2 \mapsto clock) \quad \} \parallel$

$max\_instances := \{cl3a \mapsto 1,\ cl2a \mapsto 1,\ cl3 \mapsto 1,\ cl2 \mapsto 1,\ cl1 \mapsto 2\} \parallel$

$nb\_instances := \{cl3a \mapsto 0,\ cl2a \mapsto 0,\ cl3 \mapsto 1,\ cl2 \mapsto 1,\ cl1 \mapsto 2\} \parallel$

$interfaceInstance := \{rintILamp2Inst1, rintIIntensityInst1, rintIClock2Inst, pintIClock2Inst,$
$rintILampInst1, rintIClockInst1, pintILamp2Inst2, pintIItensityInst2, pintILamp2Inst1, pintIIntensityInst1,$
$pintIClockInst1\} \parallel$

$int\_instantiates := \{(rintILamp2Inst1 \mapsto rintILamp2)$

$,(rintIIntensityInst1 \mapsto rintIIntensity)$

$,(rintIClock2Inst \mapsto rintIClock2)$

$,(pintIClock2Inst \mapsto pintIClock2)$

$,(rintILampInst1 \mapsto rintILamp)$

$,(rintIClockInst1 \mapsto rintIClock)$

$,(pintILamp2Inst2 \mapsto pintILamp2)$

$,(pintIItensityInst2 \mapsto pintIIntensity)$

,($pintILamp2Inst1 \mapsto pintILamp2$)

,($pintIIntensityInst1 \mapsto pintIIntensity$)

,($pintIClockInst1 \mapsto pintIClock$)} ||

$instInterfaces := \{ci3a \mapsto \{rintILamp2Inst1,rintIIntensityInst1,rintIClock2Inst\}$

,$ci2a \mapsto \{pintIClock2Inst\}$

,$ci3 \mapsto \{rintILampInst1,rintIClockInst1\}$

,$ci11 \mapsto \{pintILamp2Inst2,pintIItensityInst2\}$

,$ci12 \mapsto \{pintILamp2Inst1,pintIIntensityInst1\}$

,$ci2 \mapsto \{pintIClockInst1\}\}$ ||

$initiation\_state := \varnothing$ ||

$current\_state := \varnothing$ ||

$comp\_instantiates := \{(ci3a \mapsto cl3a),(ci2a \mapsto cl2a), (ci3 \mapsto cl3), (ci11 \mapsto cl1), (ci12 \mapsto cl1), (ci2 \mapsto cl2)\}$ ||

$attribute\_value := \varnothing$ ||

$client\_instance := \{(ci3 \mapsto rintILampInst1)$

,($ci3 \mapsto rintIClockInst1$)} ||

$server\_instance := \{(ci11 \mapsto pintILamp2Inst2)$

,($ci12 \mapsto pintILamp2Inst1$)

,($ci2 \mapsto pintIClockInst1$)} ||

$binding := \{((ci3 \mapsto rintILampInst1) \mapsto (ci11 \mapsto pintILamp2Inst2))$

,$((ci3 \mapsto rintILampInst1) \mapsto (ci12 \mapsto pintILamp2Inst1))$

,$((ci3 \mapsto rintIClockInst1) \mapsto (ci2 \mapsto pintIClockInst1))\}$ ||

$assembly := \{HASAssm\}$ ||

$assm\_components := \{HASAssm \mapsto \{ci3, ci11, ci12, ci2\}\}$ ||

$assm\_connections := \{HASAssm \mapsto \{((ci3 \mapsto rintILampInst1) \mapsto (ci11 \mapsto pintILamp2Inst2))$

,$((ci3 \mapsto rintIClockInst1) \mapsto (ci2 \mapsto pintIClockInst1))$

,$((ci3 \mapsto rintILampInst1) \mapsto (ci12 \mapsto pintILamp2Inst1))\}\}$ ||

$assm\_servers := \{HASAssm \mapsto \{(ci11 \mapsto pintILamp2Inst2), (ci12 \mapsto pintILamp2Inst1), (ci2 \mapsto pintIClockInst1)\}\}$ ||

$assm\_clients := \{HASAssm \mapsto \{(ci3 \mapsto rintILampInst1), (ci3 \mapsto rintIClockInst1)\}\}$ ||

$instantiates := \{HASAssm \mapsto HASConfig\}$

OPERATIONS

/*Initialization operations*/

initializeBindings =

BEGIN

binding := *all_bindings*

END;

initializeArchAssembly =

BEGIN

client_instance := *all_clientInstances* ||

server_instance := *all_serverInstances*

END;

/*Model manipulation operations*/

deployInstance(*asm*, *inst*, *class*) =

PRE

$asm \in assembly \wedge class \in compClass \wedge inst \in compInstance \wedge class = comp\_instantiates(inst) \wedge inst \notin assm\_components(asm) \wedge$

$nb\_instances(class) < max\_instances(class)$

THEN

nb_instances(*class*) := $nb\_instances(class) + 1$ ||

assm_components(*asm*) := $assm\_components(asm) \cup \{inst\}$ ||

assm_clients(*asm*) := $assm\_clients(asm) \cup clients(inst)$

END;

addServerInstance(*asm*, *se*) =

PRE

$asm \in assembly \wedge se \in server\_instance \wedge se \notin assm\_servers(asm) \wedge serverComponentElem(se) \in assm\_components(asm)$

THEN

assm_servers(*asm*) := $assm\_servers(asm) \cup \{se\}$

END;

bind(*cl*, *se*, *asm*) =

PRE

$asm \in assembly \wedge cl \in client\_instance \wedge se \in server\_instance \wedge cl \in assm\_clients(asm) \wedge se \in assm\_servers(asm) \wedge$

$(cl \mapsto se) \in binding \wedge (cl \mapsto se) \notin assm\_connections(asm)$

THEN

assm_connections$(asm) := assm\_connections(asm) \cup \{(cl \mapsto se)\}$

END;

deleteServerInstance$(asm, se) =$

PRE

$asm \in assembly \wedge se \in server\_instance \wedge se \in assm\_servers(asm) \wedge \forall\ cl.(cl \in client\_instance \wedge cl \in assm\_clients(asm) \Rightarrow$

$(cl \mapsto se) \notin assm\_connections(asm))$

THEN

assm_servers$(asm) := assm\_servers(asm)$ - $\{se\}$

END;

removeInstance$(asm, inst, class) =$

PRE

$asm \in assembly \wedge inst \in compInstance \wedge inst \in assm\_components(asm) \wedge class \in compClass \wedge class = comp\_instantiates(inst) \wedge$

$\forall\ cl.(cl \in client\_instance \wedge cl \in clients(inst) \Rightarrow \forall\ se.(se \in server\_instance \wedge se \in assm\_servers(asm) \Rightarrow (cl \mapsto se) \notin assm\_connections(asm))) \wedge$

$\forall\ se.(se \in server\_instance \wedge se \in servers(inst) \Rightarrow se \notin assm\_servers(asm))$

THEN

nb_instances$(class) :=$ $nb\_instances(class)$ - 1 ||

assm_clients$(asm) :=$ $assm\_clients(asm)$ - $clients(inst)$ ||

assm_components$(asm) :=$ $assm\_components(asm)$ - $\{inst\}$

END;

unbind$(asm, se, cl) =$

PRE

$asm \in assembly \wedge cl \in client\_instance \wedge se \in server\_instance \wedge cl \in assm\_clients(asm) \wedge se \in assm\_servers(asm) \wedge$

$(cl \mapsto se) \in binding \wedge (cl \mapsto se) \in assm\_connections(asm)$

THEN

assm_connections(asm) := $assm\_connections(asm)$ - $\{(cl \mapsto se)\}$

END;

replaceInstance$(asm, oldInst, newInst)$ =

PRE

$oldInst \in compInstance \wedge newInst \in compInstance \wedge asm \in assembly \wedge oldInst \in assm\_components(asm) \wedge$

$newInst \notin assm\_components(asm) \wedge (oldInst, newInst) \in inst\_substitution \wedge$

$\forall cl.(cl \in client\_instance \wedge cl \in clients(oldInst) \Rightarrow \forall se.(se \in server\_instance \wedge se \in assm\_servers(asm) \Rightarrow (cl \mapsto se) \notin assm\_connections(asm))) \wedge$

$\forall (se).(se \in server\_instance \wedge se \in servers(oldInst) \Rightarrow se \notin assm\_servers(asm))$

THEN

assm_components(asm) := $(assm\_components(asm)$ - $\{oldInst\}) \cup \{newInst\}$ ||

assm_clients(asm) := $(assm\_clients(asm)$ - $clients(oldInst)) \cup clients(newInst)$

END

END

Appendix B

EvolutionManager machine

MACHINE

EvolutionManager

INCLUDES

Arch_specification, *Arch_configuration*, *Arch_assembly*

SETS

/*Enumerated set to designate the level of change*/

CHANGE_LEVEL = {*eLevel*, *specLevel*, *configLevel*, *asmLevel*}

VARIABLES

/*Variable to designate the level of change*/
changeLevel,

/*Variables used to store the history of manipulated artifacts (*i.e.*, added and deleted artifacts)*/

addedRoles, *deletedRoles*, *addedConnections*, *deletedConnections*, *selectedArch*,

addedClassConnections, *deletedClassConnections*, *addedServers*, *deletedServers*, *addedClasses*, *deletedClasses*,

selectedConfig,

deletedInstConnections, *addedInstConnections*, *addedInstances*, *deletedInstances*, *selectedAsm*,

addedServerInstances, *deletedServerInstances*,

/*Boolean variables used to initialize the included machines*/
init1, *init2*, *init3*, *init4*, *init5*, *init6*, *init7*, *init8*, *init9*, *init10*, *init11*, *init12*,

initialisation

INVARIANT

$changeLevel \in CHANGE\_LEVEL \wedge$

/*Variables related to the specification level*/

$addedRoles \subseteq compRole \wedge$

$deletedRoles \subseteq compRole \wedge$

$addedConnections \subseteq connection \wedge$

$deletedConnections \subseteq connection \wedge$

$selectedArch \in ARCH\_SPEC \wedge$

/*Variables related to the configuration level*/

$addedClassConnections \subseteq connection \wedge$

$deletedClassConnections \subseteq connection \wedge$

$addedServers \subseteq server \wedge$

$deletedServers \subseteq server \wedge$

$addedClasses \subseteq compClass \wedge$

$deletedClasses \subseteq compClass \wedge$

$selectedConfig \in CONFIGURATIONS \wedge$

/*Variables related to the assembly level*/

$deletedInstConnections \subseteq binding \wedge$

$addedInstConnections \subseteq binding \wedge$

$addedInstances \subseteq compInstance \wedge$

$deletedInstances \subseteq compInstance \wedge$

$selectedAsm \in ASSEMBLIES \wedge$

$addedServerInstances \subseteq server\_instance \wedge$

$deletedServerInstances \subseteq server\_instance \wedge$

/*Boolean variables used for initialization*/

$\{init1, init2, init3, init4, init5, init6, init7, init8, init9, init10, init11, init12\} \subset \mathcal{P}($ **BOOL** $) \wedge$

$initialisation \in$ **BOOL**

DEFINITIONS

/*generic macro definitions*/

$comp\_providedInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge$
$int\_direction(int) = PROVIDED\};$

$comp\_requiredInterfaces(comp) == \{int \mid int \in interface \wedge int \in comp\_interfaces(comp) \wedge int\_direction(int) = REQUIRED\}$;

/*Macros related to the specification level*/

$clientInterfaceElem == \{cl, int \mid cl \in client \wedge \exists (comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp,rint) \wedge int = rint)\}$;

$clientComponentElem == \{cl, cr \mid cl \in client \wedge \exists (comp, rint).(comp \in component \wedge rint \in interface \wedge cl = (comp,rint) \wedge cr = comp)\}$;

$serverInterfaceElem == \{se, int \mid se \in server \wedge \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp,pint) \wedge int = pint)\}$;

$serverComponentElem == \{se, cr \mid se \in server \wedge \exists (comp, pint).(comp \in component \wedge pint \in interface \wedge se = (comp,pint) \wedge cr = comp)\}$;

$roleClients(role) == \{cl \mid role \in component \wedge cl \in client \wedge clientComponentElem(cl) = role\}$;

$roleServers(role) == \{se \mid role \in component \wedge se \in server \wedge serverComponentElem(se) = role\}$;

/*Architecture specification consistency*/

$spec\_consistency == \forall (cl, spec).(cl \in client \wedge spec \in specification \wedge cl \in spec\_clients(spec) \Rightarrow \exists (conn, se).(conn \in connection \wedge conn \in spec\_connections(spec) \wedge$

$se \in server \wedge se \in spec\_servers(spec) \wedge conn = (cl \mapsto se))) \wedge$

$\forall (se, spec).(se \in server \wedge spec \in specification \wedge se \in spec\_servers(spec) \Rightarrow \exists (conn, cl).(conn \in connection \wedge conn \in spec\_connections(spec) \wedge$

$cl \in client \wedge cl \in spec\_clients(spec) \wedge conn = (cl \mapsto se))) \wedge$

$\forall (conn, spec).(conn \in connection \wedge spec \in specification \wedge conn \in spec\_connections(spec) \Rightarrow \exists (cl, se).(cl \in client \wedge se \in server \wedge cl \in spec\_clients(spec) \wedge$

$se \in spec\_servers(spec) \wedge conn=(cl \mapsto se))) \wedge$

$\forall (cr, spec).(cr \in compRole \wedge spec \in specification \wedge cr \in spec\_components(spec) \Rightarrow \exists conn.(conn \in connection \wedge conn \in spec\_connections(spec) \wedge cr \in \mathbf{dom}(\mathbf{dom}(\{conn\})) \cup \mathbf{dom}(\mathbf{ran}(\{conn\})))) \wedge$

$\forall (cr, spec).(cr \in compRole \wedge spec \in specification \wedge cr \in spec\_components(spec) \Rightarrow \forall cl.(cl \in client \wedge clientComponentElem(cl) = cr \Rightarrow cl \in spec\_clients(spec))) \wedge$

$\forall (cr, spec).(cr \in compRole \wedge spec \in specification \wedge cr \in spec\_components(spec) \Rightarrow \forall se.(se \in server \wedge serverComponentElem(se) = cr \Rightarrow se \in spec\_servers(spec)))$;

/*Macros related to the configuration level*/

$class\_interfaces(cl) == comp\_interfaces(class\_implements(cl))$;

$classClients(class) == \{cl \mid class \in compClass \wedge cl \in client \wedge clientComponentElem(cl) = class\_implements(class)\}$;

$classServers(class) == \{se \mid class \in compClass \wedge se \in server \wedge serverComponentElem(se) = class\_implements(class)\}$;

$class\_substitution == (class\_implements\ ;\ compType\_substitution\ ;\ class\_implements^{-1})$;

/*Architecture configuration consistency*/

$config\_consistency == \forall\ (cl,\ config).(cl \in client \wedge config \in configuration \wedge cl \in config\_clients(config) \Rightarrow \exists\ (conn,\ se).(conn \in connection \wedge conn \in config\_connections(config) \wedge$

$se \in server \wedge se \in config\_servers(config) \wedge conn = (cl \mapsto se))) \wedge$

$\forall\ (se,\ config).(se \in server \wedge config \in configuration \wedge se \in config\_servers(config) \Rightarrow \exists\ (conn,\ cl).(conn \in connection \wedge conn \in config\_connections(config) \wedge$

$cl \in client \wedge cl \in config\_clients(config) \wedge conn = (cl \mapsto se))) \wedge$

$\forall\ (conn,\ config).(conn \in connection \wedge config \in configuration \wedge conn \in config\_connections(config) \Rightarrow \exists\ (cl,\ se).(cl \in client \wedge se \in server \wedge cl \in config\_clients(config) \wedge$

$se \in config\_servers(config) \wedge conn{=}(cl \mapsto se))) \wedge$

$\forall\ (class,\ config).(class \in compClass \wedge config \in configuration \wedge class \in config\_components(config) \Rightarrow \exists\ conn.(conn \in connection \wedge conn \in config\_connections(config) \wedge class\_implements(class) \in \mathbf{dom}(\mathbf{dom}(\{conn\})) \cup \mathbf{dom}(\mathbf{ran}(\{conn\})))) \wedge$

$\forall\ (class,\ config).(class \in compClass \wedge config \in configuration \wedge class \in config\_components(config) \Rightarrow \forall\ cl.(cl \in client \wedge clientComponentElem(cl) = class\_implements(class) \Rightarrow cl \in config\_clients(config))) \wedge$

$\forall\ (class1,\ class2,\ config).(class1 \in compClass \wedge class2 \in compClass \wedge config \in configuration \wedge \{class1,class2\} \subseteq config\_components(config) \Rightarrow ((class1,\ class2) \notin class\_substitution \wedge (class2,\ class1) \notin class\_substitution))$;

/*Macros related to the assembly level*/

$inst\_direction(int) == int\_direction(int\_instantiates(int))$;

$serverInterfaceInstElem == \{se,\ int \mid se \in server\_instance \wedge \exists\ (inst,\ pint).(inst \in compInstance \wedge pint \in interfaceInstance \wedge se = (inst,pint) \wedge int = pint)\}$;

$clientInstanceElem == \{cl,\ inst \mid inst \in compInstance \wedge cl \in client\_instance \wedge \exists\ (ci,\ rint).(ci \in compInstance \wedge rint \in interfaceInstance \wedge cl = (ci,rint) \wedge ci = inst)\}$;

$serverInstanceElem == \{se,\ inst \mid inst \in compInstance \wedge se \in server\_instance \wedge \exists\ (ci,\ pint).(ci \in compInstance \wedge pint \in interfaceInstance \wedge se = (ci,pint) \wedge ci = inst)\}$;

$instanceClients(inst) == \{cl \mid inst \in compInstance \wedge cl \in client\_instance \wedge clientInstanceElem(cl) = inst\}$;

$instanceServers(inst) == \{se \mid inst \in compInstance \wedge se \in server\_instance \wedge serverInstanceElem(se) = inst\}$;

$inst\_substitution == (comp\_instantiates;class\_implements\ ;\ compType\_substitution\ ;\ class\_implements^{-1}\ ;comp\_instantiates^{-1})$;

/*Architecture assembly consistency*/

$assm\_consistency == \forall\ (cl,\ asm).(cl \in client\_instance \wedge asm \in assembly \wedge cl \in assm\_clients(asm) \Rightarrow \exists\ (conn,\ se).(conn \in binding \wedge conn \in assm\_connections(asm) \wedge$

$se \in server\_instance \wedge se \in assm\_servers(asm) \wedge conn = (cl \mapsto se))) \wedge$

$\forall$ (*se*, *asm*).(*se* $\in$ *server_instance* $\wedge$ *asm* $\in$ *assembly* $\wedge$ *se* $\in$ *assm_servers*(*asm*) $\Rightarrow \exists$ (*conn*, *cl*).(*conn* $\in$ *binding* $\wedge$ *conn* $\in$ *assm_connections*(*asm*) $\wedge$

cl $\in$ *client_instance* $\wedge$ *cl* $\in$ *assm_clients*(*asm*) $\wedge$ *conn* = (*cl* $\mapsto$ *se*))) $\wedge$

$\forall$ (*conn*, *asm*).(*conn* $\in$ *binding* $\wedge$ *asm* $\in$ *assembly* $\wedge$ *conn* $\in$ *assm_connections*(*asm*) $\Rightarrow \exists$ (*cl*, *se*).(*cl* $\in$ *client_instance* $\wedge$ *se* $\in$ *server_instance* $\wedge$ *cl* $\in$ *assm_clients*(*asm*) $\wedge$

se $\in$ *assm_servers*(*asm*) $\wedge$ *conn*=(*cl* $\mapsto$ *se*))) $\wedge$

$\forall$ (*inst*, *asm*).(*inst* $\in$ *compInstance* $\wedge$ *asm* $\in$ *assembly* $\wedge$ *inst* $\in$ *assm_components*(*asm*) $\Rightarrow \exists$ *conn*.(*conn* $\in$ *binding* $\wedge$ *conn* $\in$ *assm_connections*(*asm*) $\wedge$ *inst* $\in$ **dom**(**dom**({*conn*})) $\cup$ **dom**(**ran**({*conn*}))))) $\wedge$

$\forall$ (*inst*, *asm*).(*inst* $\in$ *compInstance* $\wedge$ *asm* $\in$ *assembly* $\wedge$ *inst* $\in$ *assm_components*(*asm*) $\Rightarrow \forall$ *cl*.(*cl* $\in$ *client_instance* $\wedge$ *clientInstanceElem*(*cl*) = *inst* $\Rightarrow$ *cl* $\in$ *assm_clients*(*asm*)));

/*Coherence between specification and configuration*/

specConfigCoherence == $\forall$ (*spec*, *config*).(*spec* $\in$ *specification* $\wedge$ *config* $\in$ *configuration* $\Rightarrow$ ((*config*, *spec*) $\in$ *implements* $\Leftrightarrow \forall$ *CR*.(*CR* $\in$ *compRole* $\wedge$ *CR* $\in$ *spec_components*(*spec*) $\Rightarrow \exists$ *CL*.(*CL* $\in$ *compClass* $\wedge$ *CL* $\in$ *config_components*(*config*) $\wedge$ (*CL*,*CR*) $\in$ *realizes*))) $\wedge$ (*spec*, *config*) $\in$ *documents* $\Leftrightarrow \forall$ *se1*.(*se1* $\in$ *server* $\wedge$ *se1* $\in$ *config_servers*(*config*) $\Rightarrow \exists$ *se2*.(*se2* $\in$ *server* $\wedge$ *se2* $\in$ *spec_servers*(*spec*) $\wedge$ (*serverInterfaceElem*(*se2*), *serverInterfaceElem*(*se1*)) $\in$ *int_substitution*)));

/*Coherence between configuration and assembly*/

configAssemblyCoherence == $\forall$ (*asm*, *config*).(*asm* $\in$ *assembly* $\wedge$ *config* $\in$ *configuration* $\Rightarrow$ ((*asm*, *config*) $\in$ *instantiates* $\Leftrightarrow$ ($\forall$ *cl*.(*cl* $\in$ *compClass* $\wedge$ *cl* $\in$ *config_components*(*config*) $\Rightarrow \exists$ *ci*.(*ci* $\in$ *compInstance* $\wedge$ *ci* $\in$ *assm_components*(*asm*) $\wedge$ (*ci*, *cl*) $\in$ *comp_instantiates*)) $\wedge$

($\forall$ *ci*.(*ci* $\in$ *compInstance* $\wedge$ *ci* $\in$ *assm_components*(*asm*) $\Rightarrow \exists$ *cl*.(*cl* $\in$ *compClass* $\wedge$ *cl* $\in$ *config_components*(*config*) $\wedge$ (*ci*, *cl*) $\in$ *comp_instantiates*))))));

/*Preconditions of the model manipulation operations*/

/*preconditions related to the specification level*/

roleAdditionPrecondition == *spec* $\in$ *specification* $\wedge$ *newRole* $\in$ *compRole* $\wedge$ *newRole* $\notin$ *spec_components*(*spec*) $\wedge \forall$ *cr*.(*cr* $\in$ *compRole* $\wedge$ *cr* $\in$ *spec_components*(*spec*) $\Rightarrow$ *comp_name*(*cr*) $\neq$ *comp_name*(*newRole*)) ;

roleConnectionPrecondition == *spec* $\in$ *specification* $\wedge$ *cl* $\in$ *client* $\wedge$ *se* $\in$ *server* $\wedge$ *cl* $\in$ *spec_clients*(*spec*) $\wedge$ *se* $\in$ *spec_servers*(*spec*) $\wedge$ (*cl* $\mapsto$ *se*) $\in$ *connection* $\wedge$ (*cl* $\mapsto$ *se*) $\notin$ *spec_connections*(*spec*) $\wedge \forall$ *se1*.(*se1* $\in$ *server* $\wedge$ *se1* $\in$ *spec_servers*(*spec*) $\Rightarrow$ (*cl* $\mapsto$ *se1*) $\notin$ *spec_connections*(*spec*)) $\wedge \forall$ *cl2*.(*cl2* $\in$ *client* $\wedge$ *cl2* $\in$ *spec_clients*(*spec*) $\Rightarrow$ (*cl2* $\mapsto$ *se*) $\notin$ *spec_connections*(*spec*));

roleDeletionPrecondition == *spec* $\in$ *specification* $\wedge$ *role* $\in$ *compRole* $\wedge$ *role* $\in$ *spec_components*(*spec*) $\wedge \forall$ *cl*.(*cl* $\in$ *client* $\wedge$ *cl* $\in$ *roleClients*(*role*) $\Rightarrow \forall$ *se*.(*se* $\in$ *server* $\wedge$ *se* $\in$ *spec_servers*(*spec*) $\Rightarrow$ (*cl* $\mapsto$ *se*) $\notin$ *spec_connections*(*spec*))) $\wedge \forall$ (*se*).(*se* $\in$ *server* $\wedge$ *se* $\in$ *roleServers*(*role*) $\Rightarrow \forall$ *cl*.(*cl* $\in$ *client* $\wedge$ *cl* $\in$ *spec_clients*(*spec*) $\Rightarrow$ (*cl* $\mapsto$ *se*) $\notin$ *spec_connections*(*spec*)));

roleDisconnectionPrecondition == *spec* $\in$ *specification* $\wedge$ *cl* $\in$ *client* $\wedge$ *se* $\in$ *server* $\wedge$ *cl* $\in$ *spec_clients*(*spec*) $\wedge$ *se* $\in$ *spec_servers*(*spec*) $\wedge$ (*cl* $\mapsto$ *se*) $\in$ *connection* $\wedge$ (*cl* $\mapsto$ *se*) $\in$ *spec_connections*(*spec*);

$roleSubstitutionPrecondition == oldRole \in compRole \land newRole \in compRole \land spec \in specification \land oldRole \in spec\_components(spec) \land newRole \notin spec\_components(spec) \land (oldRole, newRole) \in role\_substitution \land \forall cl.(cl \in client \land cl \in roleClients(oldRole) \Rightarrow \forall se.(se \in server \land se \in spec\_servers(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec))) \land \forall se.(se \in server \land se \in roleServers(oldRole) \Rightarrow \forall cl.(cl \in client \land cl \in spec\_clients(spec) \Rightarrow (cl \mapsto se) \notin spec\_connections(spec)))$;

$roleInitSubstitutionPrecondition == oldRole \in compRole \land newRole \in compRole \land spec \in specification \land oldRole \in spec\_components(spec) \land newRole \notin spec\_components(spec) \land (oldRole, newRole) \in role\_substitution$;

$roleInitDeletionPrecondition == spec \in specification \land role \in compRole \land role \in spec\_components(spec)$;

/*preconditions related to the configuration level*/

$classInitAdditionPrecondition == config \in configuration \land newClass \in compClass \land newClass \notin config\_components(config) \land \forall cl.(cl \in compClass \land cl \in config\_components(config) \Rightarrow class\_name(cl) \neq class\_name(newClass) \land class\_implements(newClass) \neq class\_implements(cl)) \land services \subseteq server \land services \subseteq classServers(newClass)$;

$classAdditionPrecondition == config \in configuration \land newClass \in compClass \land newClass \notin config\_components(config) \land \forall cl.(cl \in compClass \land cl \in config\_components(config) \Rightarrow class\_name(cl) \neq class\_name(newClass) \land class\_implements(newClass) \neq class\_implements(cl))$;

$serverAdditionPrecondition == config \in configuration \land se \in server \land se \notin config\_servers(config) \land serverComponentElem(se) \in class\_implements[config\_components(config)]$;

$serverDeletionPrecondition == config \in configuration \land se \in server \land se \in config\_servers(config) \land \forall cl.(cl \in client \land cl \in config\_clients(config) \Rightarrow (cl \mapsto se) \notin config\_connections(config))$;

$classConnectionPrecondition == config \in configuration \land cl \in client \land se \in server \land cl \in config\_clients(config) \land se \in config\_servers(config) \land (cl \mapsto se) \in connection \land (cl \mapsto se) \notin config\_connections(config)$;

$classDeletionPrecondition == config \in configuration \land class \in compClass \land class \in config\_components(config) \land \forall cl.(cl \in client \land cl \in classClients(class) \Rightarrow \forall se.(se \in server \land se \in config\_servers(config) \Rightarrow (cl \mapsto se) \notin config\_connections(config))) \land \forall se.(se \in server \land se \in classServers(class) \Rightarrow se \notin config\_servers(config))$;

$classDisconnectionPrecondition == config \in configuration \land cl \in client \land se \in server \land cl \in config\_clients(config) \land se \in config\_servers(config) \land (cl \mapsto se) \in connection \land (cl \mapsto se) \in config\_connections(config)$;

$classSubstitutionPrecondition == oldClass \in compClass \land newClass \in compClass \land config \in configuration \land oldClass \in config\_components(config) \land newClass \notin config\_components(config) \land (oldClass, newClass) \in class\_substitution \land \forall cl.(cl \in client \land cl \in classClients(oldClass) \Rightarrow \forall se.(se \in server \land se \in config\_servers(config) \Rightarrow (cl \mapsto se) \notin config\_connections(config))) \land \forall (se).(se \in server \land se \in classServers(oldClass) \Rightarrow se \notin config\_servers(config))$;

$classInitSubstitutionPrecondition == oldClass \in compClass \land newClass \in compClass \land config \in configuration \land oldClass \in config\_components(config) \land newClass \notin config\_components(config) \land (oldClass, newClass) \in class\_substitution \land services \subseteq server \land \forall se.(se \in server \land se \in services \Rightarrow se \in classServers(newClass)) \land \forall se1.(se1 \in server \land se1 \in classServers(oldClass) \Rightarrow \exists se2.(se2 \in server \land se2 \in services \land serverInterfaceElem(se2) \in int\_substitution[\{serverInterfaceElem(se1)\}]))$;

$classInitDeletionPrecondition == config \in configuration \wedge class \in compClass \wedge class \in config\_components(config)$;

/*preconditions related to the assembly level*/

$instanceInitAdditionPrecondition == asm \in assembly \wedge class \in compClass \wedge inst \in compInstance \wedge class = comp\_instantiates(inst) \wedge inst \notin assm\_components(asm) \wedge services \subseteq server\_instance \wedge services \subseteq instanceServers(inst) \wedge nb\_instances(class) < max\_instances(class)$;

$instanceAdditionPrecondition == asm \in assembly \wedge class \in compClass \wedge inst \in compInstance \wedge class = comp\_instantiates(inst) \wedge inst \notin assm\_components(asm) \wedge nb\_instances(class) < max\_instances(class)$;

$serverInstanceAdditionPrecondition == asm \in assembly \wedge se \in server\_instance \wedge se \notin assm\_servers(asm) \wedge serverInstanceElem(se) \in assm\_components(asm)$;

$instanceBindingPrecondition == asm \in assembly \wedge cl \in client\_instance \wedge se \in server\_instance \wedge cl \in assm\_clients(asm) \wedge se \in assm\_servers(asm) \wedge (cl \mapsto se) \in binding \wedge (cl \mapsto se) \notin assm\_connections(asm)$;

$serverInstanceDeletionPrecondition == asm \in assembly \wedge se \in server\_instance \wedge se \in assm\_servers(asm) \wedge \forall\ cl.(cl \in client\_instance \wedge cl \in assm\_clients(asm) \Rightarrow (cl \mapsto se) \notin assm\_connections(asm))$;

$instanceDeletionPrecondition == asm \in assembly \wedge inst \in compInstance \wedge inst \in assm\_components(asm) \wedge class \in compClass \wedge class = comp\_instantiates(inst) \wedge \forall\ cl.(cl \in client\_instance \wedge cl \in instanceClients(inst) \Rightarrow \forall\ se.(se \in server\_instance \wedge se \in assm\_servers(asm) \Rightarrow (cl \mapsto se) \notin assm\_connections(asm))) \wedge \forall\ se.(se \in server\_instance \wedge se \in instanceServers(inst) \Rightarrow se \notin assm\_servers(asm))$;

$instanceUnbindigPrecondition == asm \in assembly \wedge cl \in client\_instance \wedge se \in server\_instance \wedge cl \in assm\_clients(asm) \wedge se \in assm\_servers(asm) \wedge (cl \mapsto se) \in binding \wedge (cl \mapsto se) \in assm\_connections(asm)$;

$instanceSubstitutionPrecondition == oldInst \in compInstance \wedge newInst \in compInstance \wedge asm \in assembly \wedge oldInst \in assm\_components(asm) \wedge newInst \notin assm\_components(asm) \wedge (oldInst, newInst) \in inst\_substitution \wedge \forall\ cl.(cl \in client\_instance \wedge cl \in instanceClients(oldInst) \Rightarrow \forall\ se.(se \in server\_instance \wedge se \in assm\_servers(asm) \Rightarrow (cl \mapsto se) \notin assm\_connections(asm))) \wedge \forall\ (se).(se \in server\_instance \wedge se \in instanceServers(oldInst) \Rightarrow se \notin assm\_servers(asm))$;

$instanceInitSubstitutionPrecondition == oldInst \in compInstance \wedge newInst \in compInstance \wedge asm \in assembly \wedge oldInst \in assm\_components(asm) \wedge newInst \notin assm\_components(asm) \wedge (oldInst, newInst) \in inst\_substitution \wedge \forall\ se1.(se1 \in server\_instance \wedge se1 \in instanceServers(oldInst) \Rightarrow \exists\ se2.(se2 \in server\_instance \wedge se2 \in instanceServers(newInst) \wedge int\_instantiates(serverInterfaceInstElem(se2)) \in int\_substitution[\{int\_instantiates(serverInterfaceInstElem(se1))\}]))$;

$instanceInitDeletionPrecondition == \ asm \in assembly \wedge inst \in compInstance \wedge inst \in assm\_components(asm) \wedge class \in compClass \wedge class = comp\_instantiates(inst)$;

INITIALISATION

/* *changeLevel* **is initialized to *eLevel***

$changeLevel := eLevel \ ||$

sets are initialized to empty set */

$addedRoles := \varnothing \ ||$

$deletedRoles := \varnothing \ ||$

addedConnections := ∅ ||

deletedConnections := ∅ ||

selectedArch := *eArch* ||

addedClassConnections := ∅ ||

deletedClassConnections := ∅ ||

addedServers := ∅ ||

deletedServers := ∅ ||

addedClasses := ∅ ||

deletedClasses := ∅ ||

selectedConfig := *eConfig* ||

deletedInstConnections := ∅ ||

addedInstConnections := ∅ ||

addedInstances := ∅ ||

deletedInstances := ∅ ||

selectedAsm := *eAssm* ||

addedServerInstances := ∅ ||

deletedServerInstances := ∅ ||

/*boolean variables are used to manage the order of execution of initialization operations*/
/*init1, associated to the first operation is set to true, all the others are set to false*/

init1 := **TRUE** || *init2* := **FALSE** || *init3* := **FALSE** || *init4* := **FALSE** || *init5* := **FALSE** ||

init6 := **FALSE** || *init7* := **FALSE** || *init8* := **FALSE** || *init9* := **FALSE** || *init10* := **FALSE** ||

init11 := **FALSE** || *init12* := **FALSE** ||

initialisation := **FALSE**

OPERATIONS

/*Initialization operations*/
mng_initializeParameterSubtype =

PRE

init1=**TRUE**

THEN

s_initializeParameterSubtype ||

init1:=**FALSE** ||

init2 := **TRUE**

END;

mng_initializesignatureSubtype =

PRE

init2 = **TRUE**

THEN

s_initializesignatureSubtype ||

init2 := **FALSE** ||

init3 := **TRUE**

END;

mng_initializeInterfaceSubtype =

PRE

init3 = **TRUE**

THEN

s_initializeInterfaceSubtype ||

init3 := **FALSE** ||

init4 := **TRUE**

END;

mng_initializeBasicModel =

PRE

init4 = **TRUE**

THEN

s_initializeBasicModel ||

init4 := **FALSE** ||

init5 := **TRUE**

END;

mng_initializeArchModel =

PRE

init5 = **TRUE**

THEN

s_initializeArchModel ||

init5 := **FALSE** ||

init6 := **TRUE**

END;

mng_initializeConnections =

PRE

init6 = **TRUE**

THEN

s_initializeConnections ||

init6 := **FALSE** ||

init7 := **TRUE**

END;

mng_initialzeArchSpecModel =

PRE

init7 = **TRUE**

THEN

initialzeArchSpecModel ||

init7 := **FALSE** ||

init8 := **TRUE**

END;

mng_initializeArchConfig =

PRE

init8 = **TRUE**

THEN

initializeArchConfig ||

init8 := **FALSE** ||

init9 := **TRUE**

END;

mng_computeRealizations =

PRE

init9 = **TRUE**

THEN

computeRealizations ||

init9 := **FALSE** ||

init10 := **TRUE**

END;

mng_initializeArchAssembly =

PRE

init10 = **TRUE**

THEN

initializeArchAssembly ||

init10 := **FALSE** ||

init11 := **TRUE**

END;

mng_initializeBindings =

PRE

init11 = **TRUE**

THEN

initializeBindings ||

init11 := **FALSE** ||

init12 := **TRUE**

END;

/*Architecture model setter*/

mng_setTargetArchitectures(*spec*, *config*, *assm*) =

PRE

$spec \in specification \wedge config \in configuration \wedge assm \in assembly \wedge$

init12 = **TRUE**

THEN

selectedArch := *spec* ||

selectedConfig := *config* ||

selectedAsm := *assm* ||

init12 := **FALSE** ||

initialisation := **TRUE**

END;

/*Change level setter*/

setChangeLevel(*newChangeLevel*) =

PRE

initialisation = **TRUE** ∧ *newChangeLevel* ∈ *CHANGE_LEVEL* ∧ *newChangeLevel* ≠ *eLevel* ∧ *changeLevel* ≠ *newChangeLevel*

THEN

changeLevel := *newChangeLevel*

END;

/*Evolution rules*/

/*Evolution rules related to the specification level*/

changeArtifact ← **spec_addRole**(*spec*, *newRole*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *specLevel* ∧

roleAdditionPrecondition ∧

selectedArch = *spec* ∧

newRole ∉ (*deletedRoles* ∪ *addedRoles*)

THEN

addRole(*spec*, *newRole*) ||

addedRoles := *addedRoles* ∪ {*newRole*} ||

changeArtifact := *newRole*

END;

artifact1,*artifact2* ← *spec_connect*(*cl*, *se*, *spec*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *specLevel* ∧

selectedArch = *spec* ∧

roleConnectionPrecondition ∧

(*cl* ↦ *se*) ∉ (*deletedConnections* ∪ *addedConnections*)

THEN

connect(*cl*, *se*, *spec*) ||

addedConnections := *addedConnections* ∪ {(*cl* ↦ *se*)} ||

artifact1 := *clientComponentElem*(*cl*) ||

artifact2 := *serverComponentElem*(*se*)

END;

spec_removeRole(*spec*, *role*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *specLevel* ∧

roleDeletionPrecondition ∧

selectedArch = *spec* ∧

role ∉ (*deletedRoles* ∪ *addedRoles*)

THEN

removeRole(*spec*, *role*) ||

deletedRoles := *deletedRoles* ∪ {*role*}

END;

artifact1, *artifact2* ← *spec_disconnect*(*spec*, *se*, *cl*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *specLevel* ∧ *roleDisconnectionPrecondition* ∧

selectedArch = *spec* ∧

$(cl \mapsto se) \notin (deletedConnections \cup addedConnections)$

THEN

disconnect(*spec*, *se*, *cl*) ||

$deletedConnections := deletedConnections \cup \{(cl \mapsto se)\}$ ||

$artifact1 := clientComponentElem(cl)$ ||

$artifact2 := serverComponentElem(se)$

END;

$changeArtifact \leftarrow$ **spec_replaceRole**(*spec*, *oldRole*, *newRole*) =

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = specLevel \wedge$

$roleSubstitutionPrecondition \wedge$

$selectedArch = spec \wedge$

$oldRole \notin (deletedRoles \cup addedRoles) \wedge newRole \notin (deletedRoles \cup addedRoles)$

THEN

replaceRole(*spec*, *oldRole*, *newRole*) ||

$deletedRoles := deletedRoles \cup \{oldRole\}$ ||

$addedRoles := addedRoles \cup \{newRole\}$ ||

$changeArtifact := newRole$

END;

/*Evolution rules related to the configuration level*/

$changeArtifact \leftarrow$ **config_addClass**(*config*, *newClass*) =

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = configLevel \wedge$

$classAdditionPrecondition \wedge$

$newClass \notin (deletedClasses \cup addedClasses) \wedge$

$selectedConfig = config$

THEN

addClass(*config*, *newClass*) ||

addedClasses := *addedClasses* ∪ {*newClass*} ||

changeArtifact := *class_implements*(*newClass*)

END;

changeArtifact ← **config_addServer**(*config*, *se*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *configLevel* ∧

serverAdditionPrecondition ∧

se ∉ (*addedServers* ∪ *deletedServers*) ∧

selectedConfig = *config*

THEN

addServer(*config*, *se*) ||

addedServers := *addedServers* ∪ {*se*} ||

changeArtifact := *serverComponentElem*(*se*)

END;

artifact1, *artifact2* ← *config_class_connect*(*cl*, *se*, *config*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *configLevel* ∧

classConnectionPrecondition ∧

(*cl* ↦ *se*) ∉ (*deletedConnections* ∪ *addedConnections*) ∧

selectedConfig = *config*

THEN

class_connect(*cl*, *se*, *config*) ||

addedConnections := *addedConnections* ∪ {(*cl* ↦ *se*)} ||

artifact1 := *clientComponentElem*(*cl*) ||

artifact2 := *serverComponentElem*(*se*)

END;

$changeArtifact \leftarrow$ **config_deleteServer**$(config,\ se) =$

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = configLevel \wedge$

$serverDeletionPrecondition \wedge$

$se \notin (addedServers \cup deletedServers) \wedge$

$selectedConfig = config$

THEN

deleteServer$(config,\ se)$ ||

$deletedServers := deletedServers \cup \{se\}$ ||

$changeArtifact := serverComponentElem(se)$

END;

config_removeClass$(config,\ class) =$

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = configLevel \wedge$

$classDeletionPrecondition \wedge$

$class \notin (deletedClasses \cup addedClasses) \wedge$

$selectedConfig = config$

THEN

removeClass$(config,\ class)$ ||

$deletedClasses := deletedClasses \cup \{class\}$

END;

$artifact1,\ artifact2 \leftarrow config\_class\_disconnect(config,\ se,\ cl) =$

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = configLevel \wedge$

$classDisconnectionPrecondition \wedge$

$(cl \mapsto se) \notin (deletedConnections \cup addedConnections) \wedge$

$selectedConfig = config$

THEN

class_disconnect($config$, se, cl) ||

$deletedConnections := deletedConnections \cup \{(cl \mapsto se)\}$ ||

$artifact1 := clientComponentElem(cl)$ ||

$artifact2 := serverComponentElem(se)$

END;

$changeArtifact \leftarrow$ **config_replaceClass**($config$, $oldClass$, $newClass$) =

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = configLevel \wedge$

$classSubstitutionPrecondition \wedge$

$oldClass \notin (deletedClasses \cup addedClasses) \wedge newClass \notin (deletedClasses \cup addedClasses) \wedge$

$selectedConfig = config$

THEN

replaceClass($config$, $oldClass$, $newClass$) ||

$deletedClasses := deletedClasses \cup \{oldClass\}$ ||

$addedClasses := addedClasses \cup \{newClass\}$ ||

$changeArtifact := newClass$

END;

/*Evolution rules related to the assembly level*/

$changeArtifact \leftarrow$ **asm_deployInstance**(asm, $inst$, $class$) =

PRE

$initialisation = \mathbf{TRUE} \wedge$

$changeLevel = asmLevel \wedge$

$instanceAdditionPrecondition \wedge$

$inst \notin (addedInstances \cup deletedInstances) \wedge$

$selectedAsm = asm$

THEN

deployInstance(*asm*, *inst*, *class*) ||

addedInstances := *addedInstances* ∪ {*inst*} ||

changeArtifact := *inst*

END;

changeArtifact ← **asm_addServerInstance**(*asm*, *se*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *asmLevel* ∧

selectedAsm = *asm* ∧

serverInstanceAdditionPrecondition ∧

se ∉ (*addedServerInstances* ∪ *deletedServerInstances*)

THEN

addServerInstance(*asm*, *se*) ||

addedServerInstances := *addedServerInstances* ∪ {*se*} ||

changeArtifact := *serverInstanceElem*(*se*)

END;

artifact1, *artifact2* ← *asm_bind*(*cl*, *se*, *asm*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *asmLevel* ∧

instanceBindingPrecondition ∧

(*cl* ↦ *se*) ∉ (*deletedInstConnections* ∪ *addedInstConnections*) ∧

selectedAsm = *asm*

THEN

bind(*cl*, *se*, *asm*) ||

addedInstConnections := *addedInstConnections* ∪ {(*cl* ↦ *se*)} ||

artifact1 := *clientInstanceElem*(*cl*) ||

artifact2 := *serverInstanceElem*(*se*)

END;

asm_removeInstance(*asm*, *inst*, *class*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *asmLevel* ∧

instanceDeletionPrecondition ∧

inst ∉ (*deletedInstances* ∪ *addedInstances*) ∧

selectedAsm = *asm*

THEN

removeInstance(*asm*, *inst*, *class*) ||

deletedInstances := *deletedInstances* ∪ {*inst*}

END;

artifact1, *artifact2* ← *asm_unbind*(*asm*, *se*, *cl*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *asmLevel* ∧

instanceUnbindigPrecondition ∧

(*cl* ↦ *se*) ∉ (*deletedInstConnections* ∪ *addedInstConnections*) ∧

selectedAsm = *asm*

THEN

unbind(*asm*, *se*, *cl*) ||

deletedInstConnections := *deletedInstConnections* ∪ {(*cl* ↦ *se*)} ||

artifact1 := *clientInstanceElem*(*cl*) ||

artifact2 := *serverInstanceElem*(*se*)

END;

changeArtifact ← **asm_deleteServerInstance**(*asm*, *se*) =

PRE

initialisation = **TRUE** ∧

changeLevel = *asmLevel* ∧

selectedAsm = *asm* ∧

$serverInstanceDeletionPrecondition \wedge$

$se \notin (addedServerInstances \cup deletedServerInstances)$

THEN

deleteServerInstance(asm, se) ||

$deletedServerInstances := deletedServerInstances \cup \{se\}$ ||

$changeArtifact := serverInstanceElem(se)$

END;

$changeArtifact \leftarrow$ **asm_replaceInstance**(asm, $oldInst$, $newInst$) =

PRE

$initialisation =$ **TRUE** $\wedge$

$instanceSubstitutionPrecondition \wedge$

$changeLevel = asmLevel \wedge$

$oldInst \notin (deletedInstances \cup addedInstances) \wedge newInst \notin (deletedInstances \cup addedInstances) \wedge$

$selectedAsm = asm$

THEN

replaceInstance(asm, $oldInst$, $newInst$) ||

$addedInstances := addedInstances \cup \{newInst\}$ ||

$deletedInstances := deletedInstances \cup \{oldInst\}$ ||

$changeArtifact := newInst$

END

END

Appendix C

Résumé en français

Gérer l'évolution des logiciels est une tâche complexe mais nécessaire. Tout au long de son cycle de vie, un logiciel doit subir des changements, pour corriger des erreurs, améliorer ses performances et sa qualité, étendre ses fonctionnalités ou s'adapter à son environnement. A défaut d'évoluer, un logiciel se dégrade, devient obsolète ou inadapté et est remplacé. Cependant, sans évaluation de leurs impacts et contrôle de leur réalisation, les changements peuvent être sources d'incohérences et de dysfonctionnements, donc générateurs de dégradations du logiciel.

Cette thèse propose une approche améliorant la gestion de l'évolution des logiciels dans les processus de développement orientés composants. Adoptant une démarche d'ingénierie dirigée par les modèles (IDM), cette approche s'appuie sur Dedal, un langage de description d'architecture (ADL) séparant explicitement trois niveaux d'abstraction dans la définition des architectures logicielles. Ces trois niveaux (spécification, configuration et assemblage) correspondent aux trois étapes principales du développement d'une architecture (conception, implémentation, déploiement) et gardent la trace des décisions architecturales prises au fil du développement. Ces informations sont un support efficace à la gestion de l'évolution : elles permettent de déterminer le niveau d'un changement, d'analyser son impact et de planifier sa réalisation afin d'éviter la survenue d'incohérences dans la définition de l'architecture (érosion, dérive, etc.).

Une gestion rigoureuse de l'évolution nécessite la formalisation, d'une part, des relations intra-niveau liant les composants au sein des modèles correspondant aux différents niveaux de définition d'une architecture et, d'autre part, des relations inter-niveaux liant les modèles décrivant une même architecture aux différents niveaux d'abstraction. Ces relations permettent la définition des propriétés de consistance et de cohérence servant à vérifier la correction d'une architecture. Le processus d'évolution est ainsi décomposé en trois phases : initier le changement de la définition de l'architecture à un niveau d'abstraction donné ; vérifier et rétablir la consistance de cette définition en induisant des changements complémentaires

; vérifier et rétablir la cohérence globale de la définition de l'architecture en propageant éventuellement les changements aux autres niveaux d'abstraction. Ces relations et propriétés sont décrites en B, un langage de modélisation formel basé sur la théorie des ensembles et la logique du premier ordre. Elles s'appliquent à des architectures définies avec un ADL formel écrit en B dont le méta-modèle, aligné avec celui de Dedal, permet d'outiller la transformation de modèles entre les deux langages. Cette intégration permet de proposer un environnement de développement conjuguant les avantages des approches IDM et formelle : la conception d'architectures avec l'outillage de Dedal (modeleur graphique); la vérification des architectures et la gestion de l'évolution avec l'outillage de B (animateur, model-checker, solver).

Nous proposons en particulier d'utiliser un solver B pour calculer automatiquement des plans d'évolution conformes à notre proposition et avons ainsi défini l'ensemble des règles d'évolution décrivant les opérations de modification applicables à la définition d'une architecture. Le solver recherche alors automatiquement une séquence de modifications permettant la réalisation d'un changement cible tout en préservant les propriétés de consistance et de cohérence de l'architecture. Nous avons validé la faisabilité de cette gestion de l'évolution par une implémentation mêlant optimisation et génie logiciel (searchbased software engineering), intégrant notre propre solver pourvu d'heuristiques spécifiques qui améliorent significativement les temps de calcul, pour expérimenter trois scénarios d'évolution permettant de tester la réalisation d'un changement à chacun des trois niveaux d'abstraction.

Mot-clés : architecture logicielle, évolution, composant, niveaux d'abstraction, méthode B, consistance, cohérence

Bibliography

(1997). The B-Toolkit User's Manual. *B-Core (UK) Limited*.

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer, Inc., Secaucus, USA.

Aboud, N. A., Arévalo, G., Falleri, J.-R., Huchard, M., Tibermacine, C., Urtado, C., and Vauttier, S. (2009). Automated architectural component classification using concept lattices. In *Proceedings WICSA/ECSA*, pages 21–31, Cambridge, UK.

Abrial, J.-R. (1996). *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, USA.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition.

Allen, R., Douence, R., and Garlan, D. (1998). Specifying and analyzing dynamic software architectures. In Astesiano, E., editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2010). On challenges of model transformation from uml to alloy. *Software & Systems Modeling*, 9(1):69–86.

Arévalo, G., Desnos, N., Huchard, M., Urtado, C., and Vauttier, S. (2008). FCA-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, pages 427–453.

Barnes, J., Garlan, D., and Schmerl, B. (2014). Evolution styles: foundations and models for software architecture evolution. *Software and Systems Modeling*, 13(2):649–678.

Barnes, J. M., Pandey, A., and Garlan, D. (2013). Automated planning for software architecture evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 213–223.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. (1999). Meteor: A successful application of B in a large project. In Wing, J., Woodcock, J., and Davies, J., editors, *FM'99 — Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, Berlin, Heidelberg.

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA. ACM.

Beugnard, A., Jezequel, J., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *Computer*, 32(7):38–45.

Beugnard, A., Jézéquel, J.-M., and Plouzeau, N. (2010). Contract Aware Components, 10 years after. *Electronic Proceedings in Theoretical Computer Science*, (37):86–100.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5 – 33.

Breivold, H. P., Crnkovic, I., and Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16 – 40.

Brucker, A. and Wolff, B. (2008). Hol-ocl: A formal proof environment for uml/ocl. In Fiadeiro, J. and Inverardi, P., editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer Berlin Heidelberg.

Bruel, J.-M., Lilius, J., Moreira, A., and France, R. (2000). Defining precise semantics for uml. In Goos, G., Hartmanis, J., van Leeuwen, J., Malenfant, J., Moisan, S., and Moreira, A., editors, *Object-Oriented Technology*, volume 1964 of *Lecture Notes in Computer Science*, pages 113–122. Springer Berlin Heidelberg.

Cansell, D., Gopalakrishnan, G., Jones, M., Méry, D., and Weinzoepflen, A. (2002). Incremental proof of the producer/consumer property for the PCI protocol. In Bert, D., Bowen, J., Henson, M., and Robinson, K., editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 22–41. Springer, Berlin, Heidelberg.

Cansell, D. and Méry, D. (2003). Foundations of the B method. *Computers and Informatics*, 22:1–31.

Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.

ClearSy (2001). Atelier B. *Aix-en-Provence (F) http://www.atelierb.eu/ accessed 09/01/2015.*

Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004). A component model for building systems software. In *Software Engineering and Applications (SEA '04)*, pages 684–689, Cambrigde.

Crnkovic, I. (2002). *Building Reliable Component-Based Software Systems.* Artech House, Inc., Norwood, MA, USA.

Crnkovic, I. (2003). Component-based software engineering - new challenges in software development. In *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pages 9–18.

Crnkovic, I., Chaudron, M., and Larsson, S. (2006). Component-based development process and component lifecycle. In *Software Engineering Advances, International Conference on*, pages 44–44.

Crnković, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. (2011). A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615.

Dashofy, E., van der Hoek, A., and Taylor, R. (2001). A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112.

de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151.

Delahaye, D., Dubois, C., Marché, C., and Mentré, D. (2014). The BWare project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In Ait Ameur, Y. and Schewe, K.-D., editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 290–293. Springer, Berlin, Heidelberg.

Derrick, J., North, S., and Simons, A. (2011). Z2sal: a translation-based model checker for z. *Formal Aspects of Computing*, 23(1):43–71.

Dupuy, S., Ledru, Y., and Chabre-Peccoud, M. (2000). An overview of roz : A tool for integrating uml and z specifications. In Wangler, B. and Bergman, L., editors, *Advanced Information Systems Engineering*, volume 1789 of *Lecture Notes in Computer Science*, pages 417–430. Springer Berlin Heidelberg.

Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Int. Res.*, 20(1):61–124.

Garlan, D. (2000). Software architecture: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA. ACM.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26.

Garlan, D., Allen, R., and Ockerbloom, J. (2009). Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66–69.

Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., and Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition.

Garlan, D., Monroe, R., and Wile, D. (1997). ACME: An architecture description interchange language. In *Proceedings of Centre for Advanced Studies Conference*, page 7. IBM Press.

Garlan, D. and Schmerl, B. (2009). Aevol: A tool for defining and planning architecture evolution. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 591–594, Washington, DC, USA. IEEE Computer Society.

Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical report, Pittsburgh, PA, USA.

Georgas, J. C., Dashofy, E. M., and Taylor, R. N. (2006). Architecture-centric development: A different approach to software engineering. *ACM Crossroads, issue on Software Engineering*, 12(4).

Goaer, O. L., Tamzalit, D., Oussalah, M., and Seriai, A. (2008). Evolution shelf: Reusing evolution expertise within component-based software architectures. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, pages 311–318.

Gogolla, M., Büttner, F., and Richters, M. (2007). Use: A uml-based specification environment for validating {UML} and {OCL}. *Science of Computer Programming*, 69(1–3):27 – 34. Special issue on Experimental Software and Toolkits.

Hansen, K. M. and Ingstrup, M. (2010). Modeling and analyzing architectural change with alloy. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2257–2264, New York, NY, USA. ACM.

Heineman, G. T. and Councill, W. T., editors (2001). *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: A survey. *Inf. Softw. Technol.*, 47(10):643–656.

Idani, A. (2006). *B/UML: Bridging the gap between B specifications and UML graphical descriptions to ease external validation of formal B developments.* Theses, Université Joseph-Fourier - Grenoble I.

IEEE1219-1998, S. (1998). IEEE Standard for Software Maintenance.

Ingstrup, M. and Hansen, K. (2009). Modeling architectural change: Architectural scripting and its applications to reconfiguration. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 337–340.

ISO/IEC (2001). Software engineering–product quality–part 1: Quality model. *International Organization For Standardization/International Electrotechnical Commission and others*, 9126:2001.

Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

Jackson, D. and Rinard, M. (2000). Software analysis: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 133–145, New York, NY, USA. ACM.

Jones, C. B. (1990). *Systematic Software Development Using VDM (2nd Ed.).* Prentice-Hall.

Joolia, A., Batista, T., Coulson, G., and Gomes, A. (2005). Mapping adl specifications to an efficient and reconfigurable runtime component platform. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 131–140.

Kramer, J. and Magee, J. (1990). The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306.

Lausdahl, K. (2013). Translating vdm to alloy. In Johnsen, E. and Petre, L., editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg.

Le Goaer, O. (2009). *Evolution styles within software architectures.* Theses, Université de Nantes ; Ecole Centrale de Nantes (ECN) (ECN) (ECN) (ECN).

Ledang, H. and Souquieres, J. (2002). Integration of UML and B specification techniques: systematic transformation from OCL expressions into B. In *Ninth Asia-Pacific Software Engineering Conference*, pages 495–504, Gold Coast, Australia.

Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J.-L., and Labiadh, M.-A. (2011). Taking into Account Functional Models in the Validation of IS Security Policies. In Salinesi, C. and Pastor, O., editors, *Advanced Information Systems Engineering Workshops*, volume 83 of *Lecture Notes in Business Information Processing*, pages 592–606. Springer, Berlin, Heidelberg.

Lehman, M. M. (1984). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of System and Software*, 1:213–221.

Leuschel, M. and Bendisposto, J. (2011). Directed model checking for B: An evaluation and new techniques. In Davies, J., Silva, L., and Simao, A., editors, *Formal Methods: Foundations and Applications*, volume 6527 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg.

Leuschel, M. and Butler, M. (2008). ProB: An Automated Analysis Toolset for the B Method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203.

Lientz, B. P., Swanson, E. B., and Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communication of the ACM*, 21(6):466–471.

Macedo, N., Guimaraes, T., and Cunha, A. (2013). Model repair and transformation with echo. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 694–697.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, UK. Springer-Verlag.

Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14.

McDermott, D. (1998). PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Medvidovic, N. (2006). Moving architectural description from under the technology lamppost. In *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, pages 2–3.

Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. (1998). A type theory for software architectures. Technical report, University of California.

Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. (1999). A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st ICSE*, pages 44–53.

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

Mehta, N. R., Medvidovic, N., and Phadke, S. (2000). Towards a taxonomy of software connectors. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 178–187, New York, NY, USA. ACM.

Mens, T. and Demeyer, S. (2008). *Software Evolution*. Springer.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40.

Mokni, A., Huchard, M., Urtado, C., Vauttier, S., and Zhang, H. Y. (2014). Towards automating the coherence verification of multi-level architecture descriptions. In *Proc. of the 9th ICSEA*, pages 416–421, Nice, France.

Mokni, A., Huchard, M., Urtado, C., Vauttier, S., and Zhang, H. Y. (2015). An evolution management model for multi-level component-based software architectures. In *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*, pages 674–679.

Monroe, R. T. et al. (1998). Capturing software architecture design expertise with armani. Technical report, Carnegie-Mellon University. Department of Computer Science.

Montaghami, V. and Rayside, D. (2012). Extending alloy with partial instances. In Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., and Riccobene, E., editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 122–135. Springer Berlin Heidelberg.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

OCL (2012). 2.3.1 specification. *http://www.omg.org/spec/OCL/2.3.1/ accessed 09/01/2015*.

OMG (2011). Mof 2.0 query/view/transformation (qvt), version 1.1. *http://www.omg.org/spec/QVT/1.1/ accessed 24/07/2015*.

Oquendo, F. (2004a). Pi-ADL: An architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Software Engineering Notes*, 29(3):1–14.

Oquendo, F. (2004b). Pi-arl: An architecture refinement language for formally modelling the stepwise refinement of software architectures. *SIGSOFT Softw. Eng. Notes*, 29(5):1–20.

Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., and Occhipinti, C. (2004). Archware: Architecting evolvable software. In Oquendo, F., Warboys, B., and Morrison, R., editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin Heidelberg.

Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA. IEEE Computer Society.

Oreizy, P. and Taylor, R. (1998). On the role of software architectures in runtime system reconfiguration. In *Proceedings of the International Conference on Configurable Distributed Systems*, CDS '98, pages 61–, Washington, DC, USA. IEEE Computer Society.

OSGi (2015). Osgi alliance. *http://http://www.osgi.org accessed 14/08/2015.*

Oussalah, M., Sadou, N., and Tamzalit, D. (2005). A generic model for managing software architecture evolution. In *Proceedings of the 9th WSEAS International Conference on Systems*, ICS'05, pages 35:1–35:6, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS).

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA. IEEE Computer Society Press.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Pree, W. (1997). Component-based software development-a new paradigm in software engineering? In *Software Engineering Conference, 1997. Asia Pacific and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, pages 523–524.

Riaz, M., Sulayman, M., and Naqvi, H. (2009). Architectural decay during continuous software evolution and impact of 'design for change' on software architecture. In Slezak, D., Kim, T.-h., Kiumi, A., Jiang, T., Verner, J., and Abrahao, S., editors, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 119–126. Springer Berlin Heidelberg.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA. IEEE Computer Society Press.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall.

Saaltink, M. et al. (1999). The z/eves 2.0 user's guide. *Ora Canada*, pages 31–32.

Sadou, N. (2007). *Structural Evolution in Component-based Software Evolution*. Theses, Université de Nantes ; Ecole Centrale de Nantes (ECN) (ECN) (ECN) (ECN).

Sadou, N., Tamzalit, D., and Oussalah, M. (2005). A unified approach for software architecture evolution at different abstraction levels. In *Principles of Software Evolution, Eighth International Workshop on*, pages 65–68.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Smith, G. and Wildman, L. (2005). Model checking z specifications using sal. In Treharne, H., King, S., Henson, M., and Schneider, S., editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 85–103. Springer Berlin Heidelberg.

Snook, C. and Butler, M. (2006). UML-B: Formal Modeling and Design Aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122.

Sommerville, I. (2010). *Software engineering (9th edition)*. Addison-Wesley.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Limited.

Svenningsson, J. and Axelsson, E. (2013). Combining deep and shallow embedding for edsl. In Loidl, H.-W. and Peña, R., editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Tamzalit, D., Oussalah, M., Goaer, O. L., and Seriai, A. (2006). Updating software architectures : A style-based approach. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*, pages 336–342.

Taylor, R., Medvidovic, N., and Dashofy, E. (2009). *Software architecture: Foundations, Theory, and Practice*. Wiley.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, Jr., E. J., and Robbins, J. E. (1995). A component- and message-based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, New York, USA. ACM.

Tibermacine, C., Fleurquin, R., and Sadou, S. (2005). Preserving architectural choices throughout the component-based software development process. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, pages 121–130, Pittsburgh, Pennsylvania, USA. IEEE Computer Society Press.

Tibermacine, C., Fleurquin, R., and Sadou, S. (2006). On-demand quality-oriented assistance in component-based software evolution. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, Vasteras, Sweden. LNCS 4063, Springer-Verlag.

Torlak, E. and Jackson, D. (2007). Kodkod: A relational model finder. In Grumberg, O. and Huth, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer Berlin Heidelberg.

UML (2013). 2.5 specification. *http://www.omg.org/spec/UML/2.5/Beta2/ accessed 09/01/2015*.

Utting, M. (2005). Jaza user manual and tutorial. *http://www.cs.waikato.ac.nz/ marku/jaza/ accessed 03/08/2015*.

Zhang, H. Y. (2010). *A multi-level architecture description language for forward and reverse evolution of component-based software*. PhD thesis, Montpellier II University.

Zhang, H. Y., Urtado, C., and Vauttier, S. (2010). Architecture-centric component-based development needs a three-level ADL. In *Proceedings of the 4th European Conference of Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 295–310, Copenhagen, Denmark. Springer.

Zhang, H. Y., Zhang, L., Urtado, C., Vauttier, S., and Huchard, M. (2012). A three-level component model in component-based software development. In *Proceedings of the 11th of the International Conference on Generative Programming: Concepts and Experiences*, pages 70–79, Dresden, Germany. ACM.