ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

- SCIENCES ET TECHNIQUES DU LANGUEDOC -

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ MONTPELLIER II

Discipline : Génie Informatique, Automatique et Traitement du Signal
Formation Doctorale : Systèmes Automatiques et Microélectroniques
École Doctorale : Information, Structures et Systèmes

présentée et soutenue publiquement

par

Huaxi (Yulin) ZHANG

le 7 avril 2010

Titre :

A multi-level architecture description language for forward and reverse evolution of component-based software

---

M. Yamine AIT-AMEUR	Professeur, LISI / ENSMA	Rapporteur
M. Bernard COULETTE	Professeur, IRIT / Université de Toulouse 2	Rapporteur
Mme. Marianne HUCHARD	Professeur, Université Montpellier II	Directeur de Thèse
Mme. Thérèse LIBOUREL	Professeur, Université Montpellier II	Examinateur
M. Mourad OUSSALAH	Professeur, LINA / Université de Nantes	Examinateur
Mme. Christelle URTADO	Maître de Conférence, LGI2P / EMA	Invité
Mr. Sylvain VAUTTIER	Maître de Conférence, LGI2P / EMA	Encadrant

Acknowledgments

As I type this, I have spent three years and four months working in LGI2P / EMA. First of all, many thanks to Pr. Michel Vasquez, head of the laboratory, who helped me solve many times tricky administrative procedures.

Many thanks to Pr. Marianne Huchard, director of my thesis, who helped me during three years in all my academic administrative procedures.

Special thanks to the members of my dissertation committee including Pr. Yamine Aït-Ameur, Pr. Bernard Coulette, Pr. Thérèse Libourel and Pr. Mourad Chabane Oussalah, for showing interest and investing a part of their precious time to evaluate my work.

Thanks to my teachers, Christelle Urtado and Sylvain Vauttier, who led my research work in the software engineering domain during three years, and shared many experiences with me.

Thanks to my family for their support and love.

Yulin ZHANG, Mars 2010

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 CONTEXT

1.1.1 Component-Based software Engineering

Component-based software engineering (Cbse) emerged in the late 1990s as a reuse-based approach to software system development [Sommerville06]. It quickly becomes an important software development approach as it helps satisfy the requirements of complexity, effectiveness and reliability of softwares [Gao03]. Component-based software development (Cbsd) is the major technical response to the increase in software system complexity and the ever growing need to decrease development time and cost without giving up quality [Crnkovic02]. It promotes a reuse-based approach to define, implement and compose loosely coupled independent software components into whole software systems [Sommerville06, Crnkovic06]. In such a context, software architecture-based development processes and evolution support for component-based softwares become important issues.

1.1.2 Software Architecture

As the complexity of software architecture increases, the focus of software development reorients from code to architectures [Shaw96b].

Software architectures are the soul of software development and evolution. They define the structure of a software system and embody the constituent elements of this system and relationships between them [Bass98].

Architecture models are expressed in the form of textual descriptions, which can demonstrate to stakeholders what the software is constituted of and document it [Rozanski05]. Architecture description languages are a popular way to define architecture descriptions. Examples are UML [Booch05], Wright [Allen97b, Allen98], C2 [Medvidovic99b, Taylor96], Darwin[Magee95, Magee96], or xAdl 2.0 [Dashofy01, Dashofy05]. Surprisingly, no architecture description language (Adl) proposes such a detailed description for software architectures at several abstraction levels that correspond to the steps of software lifecycle : its specification, its implementation and its deployment. This hinders processes based on these descriptions from fully supporting component-based software development and evolution.

1.1.3 Software Evolution

Software evolution is the process to change a software system from some version to a newer version to improve its functionality, performance or reliability [Lientz80, Bennett00]. There are two main issues in software evolution. Firstly, evolution is error-prone, as changes in softwares often introduce unsuitabilities or error [Madhavji06, Mens08b]. It is difficult to guarantee the preservation of the functionality and quality of the software. A rigorous description is required to check the correctness of changes. This role can be fulfilled by software architectures. Secondly, software evolution might cause architecture degradation (architectures become obsolete). This problem is known as architecture erosion or drifts [Taylor09]. This often leads to inadequate understanding of architecture changes and incomplete evolution process. Unfortunately, existing work only proposes limited support for evolution control that increases the chances for successful dynamic evolution and often have an incomplete evolution process which often results in architecture degradation.

1.2 CONTRIBUTIONS

The contributions of this thesis include :

1. A three-level ADL named Dedal that supports three description levels of component-based software : abstract architecture specification, concrete architecture configuration and instantiated software component assembly.
2. An architecture-centric evolution process based on the Dedal ADL, which can serve as a support for flexible but controlled evolution that either maintains architecture descriptions consistent or derives new architecture versions thus preventing architecture drift and erosion.
3. A connector model which embeds the necessary mechanisms to monitor and drive architecture modifications. We talk about connector-driven architecture evolution. Connectors are internally designed as expendable and reconfigurable component assemblies, capable of managing various evolution concerns, depending on the requirements.
4. An application of these connectors in the form of gradual evolution of assemblies in which architecture changes are transparently tested before they are effectively connected.

1.3 ORGANIZATION OF DISSERTATION

This dissertation is organized as follows :

– Chapter 2 gives an overview of component-based software engineering by comparing it with traditional software engineering.
– Chapter 3 gives the definitions of the key terms and concepts of software architectures, and presents the levels of software architectures. Then, various ADLs are surveyed to evaluate the strengths and weaknesses of existing ADLs.
– Chapter 4 presents the proposed Dedal ADL – which defines software architectures using the three architecture level descriptions.

- Chapter 5 presents the concepts of software evolution and gives the definitions of architecture-centric evolution. A comparison framework is proposed to compare and evaluate the support of evolution in existing dynamic ADLs. Informed by this framework, many ADLs are surveyed and discussed.
- Chapter 6 presents the evolution aspects of software architecture description : the version model and Dedal change description. Then our architecture-centric evolution process is described using an illustrative example. The runtime evolution framework that manages evolution at runtime and a connector-driven gradual evolution process are then presented.
- Chapter 7 introduces Arch3D tool-suite, an implementation environment which reflects the design principles of Dedal and its architecture-centric evolution process.
- Chapter 8 summarizes our research contributions and suggests themes for future work.

Première partie

Software Architecture

CHAPITRE 2

CONTEXT OF COMPONENT-BASED SOFTWARE ENGINEERING

In the preceding chapter, we introduced the objectives of this thesis and its planning. The objective of this thesis is to provide an ADL which supports the development and evolution of component-based software. Component-based software engineering emerged in the late 1990s. It is a reuse-based approach for software system development. The study of CBSE mostly focuses on the development process of software systems. However, the heart of component-based software systems — architecture — has been less studied. Our work thus aims at studying the architectures in CBSE. In this chapter, we will discuss component-based software development from a software architecture view.

In this chapter, we firstly introduce the principles of CBSE. Then component-based software development will be compared with traditional software development (waterfall). Finally, we present our proposed development process focused on software architectures.

Contents

2.1 COMPONENT-BASED SOFTWARE ENGINEERING

Component-based software engineering (CBSE) emerged in the late 1990s as a reuse-based approach to software system development [Sommerville06]. It quickly becomes an important software development approach. CBSE is emerging so fast for multiple reasons. First, software systems are becoming larger and more complex whereas customers are demanding dependable and cheap software. To reuse components is better than to re-implement to achieve this : shorter time to market, lower cost, higher reliability, and

maintainability [Szyperski97]. Second, several underlying technologies have matured that enable to develop components and build applications from sets of components, such as object oriented and Component technology [Ning97]. For example, Corba [Obj02] enables components to communicate remotely and transparently over a network, and Java Beans [Englander97] and Microsoft COM [Mic95] enables components from different vendors to work together within Windows. Third, the business and organizational context in which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires a technology that will support easy additions. All these reasons speed up the growth of CBSE, and push it to be a mature technique to meet the requirements.

Sommerville [Sommerville06] gives a definition of CBSE as follows :

Definition. "***Component-based software engineering (CBSE)*** *is an approach to software development that relies on software reuse. [Sommerville06]* "

The majors goals of CBSE are as follows :

- To support the development of components as reusable entities.
- To provide support for the development of systems based on reused components from design to implementation.
- To facilitate the maintenance and updating of systems by adding, deleting and replacing their components.

The first two goals concern the development and evolution of entire software systems and will be presented in Section 2.3 and Chapter 5, respectively. The development of individual components is out of the scope of this thesis.

2.2 CHARACTERISTICS OF CBSE

CBSE is distinguished by three characteristics : "reuse in the large", reusability of components and adaptability of systems. In this section, we will present these characteristics one by one.

2.2.1 Reuse In the Large

Reuse varies from fine products such as functions or objects to very large products such as components or component architectures. For CBSE, the reused products are existing components or component architectures. Thus, its reuse is called "reuse in the large".

Furthermore, the "reuse in the large" principle implies that component-based software development is *architecture-centric*, as software architectures capture the "principle" design decisions of software systems based on components [Crnkovic02].

2.2.2 Component Reusability

Components can be reused from legacy application or databases, or recently Commercial Off-The-Shelf (COTS) components. The effective reuse of components often requires their adaptation. Components often are wrapped to adapt their interfaces to their usage context. COTS components are designed to be reused, *i.e.,* to hold generic features which meet many common requirements and to be easily adapted to meet specific ones.

A COTS component means a software component that is provided by vendors [Dubois04]. The entire software system development and evolution are separated from the individual component development and evolution. Reuse is guided by only external component descriptions which hide implementation details and provide architects with only pertinent information, in a uniform way.

- During software development, a well-defined functional description of COTS components is necessary, including their external interfaces and their behavior. the functional description of components and information of versions and changes between versions are also required during maintenance stage [1].
- During maintenance, these information is a criterion to choose the best suitable available version.

2.2.3 Adaptability

The software adaptability refers to the extent to which a software system adapts to change in its environment [Chung01]. Software systems with high adaptability are easy to adapt to the changes of their architectures or environments. CBSE provides a great adaptability to changes. Firstly, adaptability comes from loosely-coupled components. Components are designed to be connected to independent other components through well identified and defined interaction points (interface or ports) that strictly encapsulate the implementations of components and enforce modularity. Secondly, these components are often COTS components, which turn out to be more robust, adaptable and updateable.

In traditional software developed from scratch, components are designed for special purposes and have a tight relationship with other components. When one of these components needs to update, many propagated changes in other components may be required.

In component-based softwares, the components are loosely coupled in the systems. Systems built of loosely coupled components show a higher degree of software reuse and improved handling of the inherently involved complexity. Usually, COTS components are adopted. They are more robust and adaptable than customer components, as they have been designed as independent components, used and tested in many different applications. Thus changes of this kind of components usually do not affect other components.

However, the evolution of COTS components is not controlled by their uses but by their vendors. Developers face the risk that new versions of the used components bring system's failures because of unexpected or unwanted evolutions [D'Souza99].

2.3 COMPONENT-BASED SOFTWARE LIFE CYCLE

A software system can be considered from its life-cycle point of view. A software process is the set of activities and associated results that produce a software product [Sommerville06]. A software process model is an abstract representation of a process. It defines software from some particular perspective[Sommerville06]. Most software process models are based on one of three general models or paradigms of software development :

- *The waterfall approach.* This organizes the fundamental process activities of specification, development, validation and evolution and represents them as successive

1. Details will be described in Chapter 5

process phases such as requirements specification, software design, implementation, testing and so on.

- *Iterative development.* This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from partial specifications. This is, then refined with customer input to produce a system that satisfies the customer's requirements.
- *Component-based software engineering (CBSE).* This approach is based on the existence of a significant number of reusable components. The system development process focuses on the integration of existing components into a system rather than implementing and testing individually new ones. CBSE leverages reuse in order to shorten the development process.

In the following subsection, we will give an overview of traditional and component-based development processes.

2.3.1 Process Models For Traditional Software Life Cycle

The waterfall and iterative approaches are all based on the same activities[Sommerville06] which vary only in the way they are performed (see Fig. 2.1).

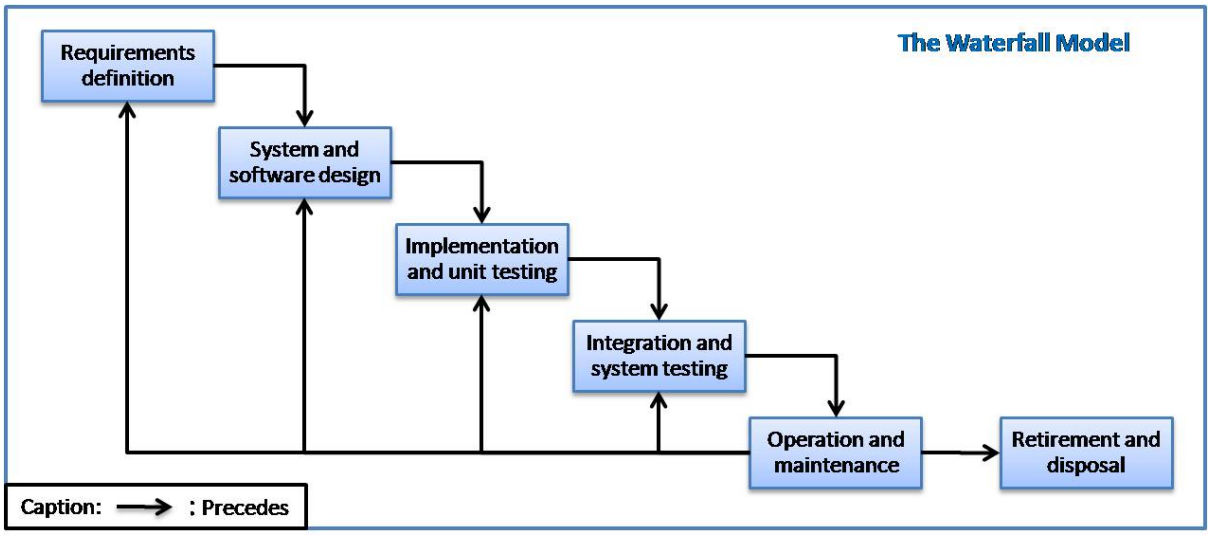


FIGURE 2.1 – The software life cycle of Waterfall model

1. *Requirements analysis and system specification.* The goals of the system are firstly established. Then a specification of a system achieving these goals is proposed. The specification defines the *services* (functional requirements) which must be provided and the *constraints* (no-functional requirements) which must be enforced by the system.
2. *System and software design.* Software design involves identifying and describing a conceptual and technical solution including design decisions answering how to realize the system. An overall system architecture is established. A detailed design follows the coarse-gained design.

3. *Implementation and unit testing.* Implementation translates the design in an executable way, which can be composed of smaller units. The units are verified and tested to meet their specifications.
4. *Integration, system verification, and validation.* System units are integrated ; the complete system is verified, validated, and delivered to the customer.
5. *Operation support and maintenance.* A system in operation requires continuous support and maintenance.
6. *Disposal.* A disposal activity, often implicit in life-cycle models, includes the phasing out of the system, with its possible replacement by another system or a complete termination.

2.3.2 Component-Based Software Life Cycle

Reuse is the better way to deliver quickly a reliable software. CBSE is an intensive reuse development approach supported by an adapted concept : *components*. These two key elements of the CBSE paradigm, reuse and components, lead to the following definition of a specific development process. Figure 2.2 shows this development process.

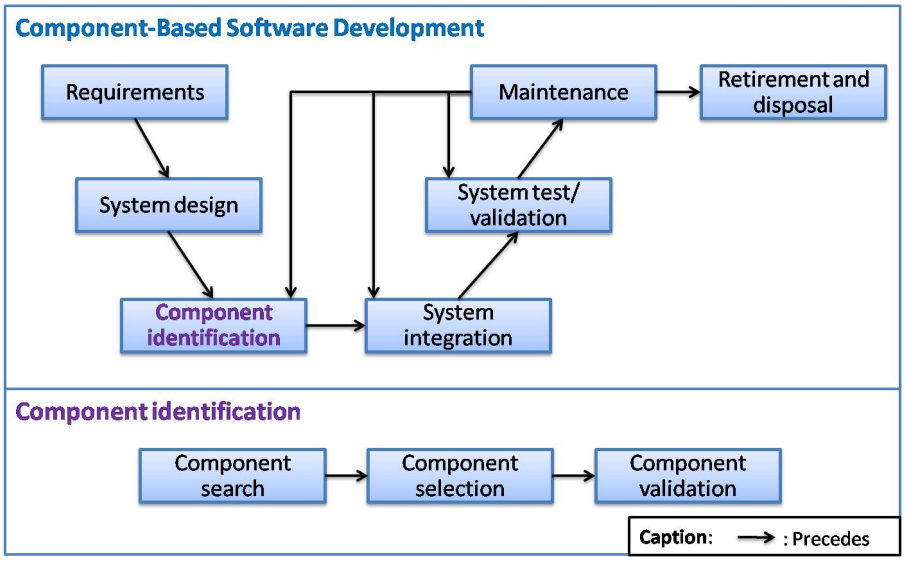


FIGURE 2.2 – The CBSE development process

1. *Requirements.* Once the goals of the system are identified, the set of services and their associated constraints that define the specification of the system are mapped to a set of component descriptions, structured into an architecture. These component descriptions identify the abstract types of components that are required to meet user's requirements. Adapting a hierarchical approach, the specification can be mapped to a unique top component which encapsulates a sub architecture that will be defined during the design step.
2. *System design.* During this stage, a first conceptual architecture is produced as a criterion to choose components for reuse. At this stage, a complete architecture of

the system must be defined. The component set coming from the specification is extended and the types of the components are refined in order to describe a concrete functioning architecture for the system.

3. *Component identification.* This is a specific stage for CBSE. This involves three sub-activities : component search, component selection and component validation, as shown in Fig. 2.2. Indeed a component type may be implemented by many component classes. The chosen implementation must be selected for component instance upon non functional application qualities which meet some defined constraints. This step replaces implementation stage in traditional development process.

   (a) *Component search.* The first stage is to look for components that are available locally or from trusted suppliers. Component search consists of browsing component repositories to find existing components corresponding to the search criterion. Typically, types are used to search existing components which correspond to the conceptual components defined at the design step. Existing types can be retrieved from component repositories and used to complete the design of the architecture. As a background schema, existing component types can be suggested to the architect as an assistance. In this way, the reuse of components starts since the design stage with the reuse of existing types as conceptual component descriptions.

   (b) *Component selection.* Once the component search process has identified candidate components, specific components have to be selected.

   (c) *Component validation.* Once architects have selected components from possible inclusion in a system, components may be verified to check whether they behave as advertised in the component repository. This corresponds to the unit testing step in other development processes. Validation can be omitted when the provider of the components is trusted.

4. *System integration.* The integration step aims at building an executable assembly composed of a set of components. It consists of downloading components from repositories, deploying them into a component container and connecting them together using connectors, according to the definition of the system's architecture.
5. *System test/ validation.* This stage corresponds to the test stage in traditional development process model. The behavior of the whole system is checked against its specification to verify whether users' requirements are met.
6. *Maintenance.* As long as requirements are unchanged, maintenance is mainly achieved by iterating the component identification step to search for new components or new versions of already available components. The goal of maintenance is to update the definition of the architecture, or correct or improve the behavior of the system.

2.4 SUMMARY

In this section, we briefly introduce the background of component-based software engineering. It covers the characters and life-cycle of CBSE by comparing with traditional engineering.

Chapitre 3

Software Architectures and Architecture Description Languages

The preceding chapter introduced the characteristics of component-based software engineering and the life-cycle of component-based software. Since the introduction of the object orientation, software development processes are strongly architecture centric. The component orientation has further increased this approach and has introduced explicit architecture representations, used to manage the whole life-cycle of softwares, from construction to maintenance, through deployment and execution. Thus, in this chapter, we will present the context of software architecture modeling thanks to architecture description languages (ADLs).

Firstly, we formally define the basic concepts of software architectures, namely components and connectors . After this overview of software architectures, we introduce the three conceptual levels which compose software architecture definitions, i.e. specification, configuration and assembly. Existing architecture description languages are then presented and compared, regarding the representation of these three conceptual levels. Finally, a synthesis of these ADLs is given to evaluate their support to architecture management throughout the development process.

Contents

3.1 SOFTWARE ARCHITECTURE

This section presents the key concepts of software architectures and more specifically components and connectors.

3.1.1 Architectures

A software architecture is the backbone of the software system [Taylor09]. It can be considered as a blueprint during the software system's construction and evolution [ISO99]. We give a definition as follows.

> ***Definition.*** *A software architecture defines the structure of a software system. It describes the constituent elements of this system, their relationships and furthermore, some important software design decisions including functional behaviors, interactions and so on. It is produced during the software development as the heart of a software system and plays as the logical underlying to guide its evolution.*

Software architectures comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass98]. Software elements in architectures can be classified in two types : components and connectors. They correspond to three conceptual roles in the functioning of architectures types [Perry92] :

- *Processing elements* are components that supply transformation on Data elements.
- *Data elements* are components that contain information that is used or transformed.
- *Connecting elements* are the glue that hold different pieces of architecture together.

Components refer to processing and data elements while connectors refer to interaction elements. Architectures are thus often represented as graphs of components linked by connectors. They are presented separately in Subsections 3.1.2 and 3.1.3.

3.1.2 Component

Components split software architectures into modules which encapsulate both processing code and data in. They are the basic constituents of software architectures. From an architectural point of view, we give the following definition of software components :

Definition. *A software component is a decoupled architectural element which provides a functionality and/ or a data source. It can be deployed independently or within a software system. A component should contains three constituents : interfaces, an implementation and a descriptive specification.*

This work is focused on the *reuse* of components and the definition of the roles that components play within architectures. It assumes the existence of a component model, such as those proposed by recent works about components [Bruneton06, Desnos06], which provides a clearly defined component structure with separate internal (white box) and external (black box) descriptions.

3.1.2.1 Basic Elements of Software Components

A software component comprises three elements : a set of interfaces, an implementation and a specification. The first two elements are the basic building blocks of a component. The specification of a component defines its semantics as a set of functional and non-functional properties. The combination of these three elements provides a complete view of the component and ensures the correct reusability of the component (integration, maintenance and updating).

Interfaces. An interface defines a communication point which manages the interactions of a component with its environment, i.e. other components [Szyperski97]. A software component may have multiple interfaces, separated into two main categories : required and provided interfaces.

- A provided interface of a component specifies a set of services that are provided by the component to other components. These interfaces define the communication points which enable components to receive service invocations from other components (incoming requests).
- A required interface of a component specifies a set of services that must be provided to the component by another component. These interfaces define the communication points which enable components to send service invocations to other components (outgoing requests).

Interfaces provide abstract external definitions of components (black box), focused on their interactions, which hide the details of their implementations and ease their reuse as the building elements of architectures.

Implementation. Implementations refer to the concrete, executable definitions of components (their code). As opposed to interfaces, implementations are detailed descriptions which are considered internal to components (white box). Implementations are directly manipulated only by the providers of components, during the development of components for reuse. Implementations are then hidden and decoupled thanks to interfaces. Architectures can thus be built by the reuse of components, upon their external definitions, with no required knowledge about their implementations.

Specification. Specification defines functional and non-functional properties of a component, as a part part of its external definition. It extends the syntactical descriptions

provided by interfaces with semantical information that enable to compute global properties about the architectures in order to validate their global behaviors. Specification is thus a key element to built consistent and reliable software from components [Crnkovic02].

- *Functional properties.* The functional properties of a component define its functionalities and its behavior. *Functionalities* are described by syntactical signatures, which include a name and a set of parameters. Functionality signatures are associated with interfaces to specify the syntactical content of the messages that can be exchanged through specific interfaces, in other words to define the signature (type) of the interface. *Component behavior* defines legal interactions of the component with its environment, as sequences of messages sent and received through its interfaces (protocols). Behavior specifications provide external abstract information, about the consistent usages of components. It enable to select and compose components into consistent wholes without any knowledge about their internal behavior or implementation.
- *Non-functional properties.* Another aspect of component specifications is non-functional properties (quality attributes), such as security, performance or reliability. This complementary information enables to select component upon specific criteria and then to calculate global properties of architectures to verify whether they meet quality requirements.

3.1.3 Connector

Software components implement the functionalities that compose the business logic of applications, while connectors provide generic interaction mechanisms. Software connectors bind components together and act as mediators between them [Shaw96b, Taylor09]. This clearly separates the concerns of computation, handled by components, from inter-component communications, handled by connectors. Decoupled this way, the reusability of components is strengthened. However, connector models seldom support specific non-functional tasks that can be useful for architecture management, such as recording or monitoring the data exchanged between two connected components. We give the following definition to connectors.

> ***Definition.*** *Software connectors are architectural elements which perform connection management functions between components and also auxiliary non-functional tasks, such as managing information about component interactions. Usually, connectors define communication rules which govern the interactions between components.*

3.1.3.1 Basic Elements of Connectors

To conform with the preceding definition, connectors should consist of two kinds of elements : ends which define the type of the components that can be linked by the connector and *glue* which define the interactions managed by the connector.

Connector ends. Connector ends enable to reason about architectural configurations and assert the consistency of connections and interactions of components. A connector end is mainly defined by an interface which is the counterpart of the interface of the connected component. This interface specifies the kind of component that can be connected and then

handled through this connector end. It defines first the signatures of the functionalities that can be reached through the connector end. Then, it defines the direction of the connector end. A provided (*resp.* required) connector end provides access to a corresponding (identical or in a wider sense, compatible [Arévalo07]) provided (*resp.* required) interface of a component.

Glue. Glue defines the interaction management services provided by a connector. Glue contains primitives for managing the *flow of control* and the *flow of data* of interactions in order to implement protocols. It defines the functional roles of the connector in the architecture (see next section).

Non-functional Properties. As first-class architectural elements, connectors bear non-functional properties which provide for instance information about their performance, footprint, resource consumption. As for components, non-functional properties are used as extra criteria to select more precisely functionally equivalent connectors, when specific qualities must be meet by architectures.

3.1.3.2 Roles and Types of Software Connectors

A software connector can play one or many different roles in a system. Each role is identified by the service that it provides. There are four generic kinds of services provided by connectors : communication, coordination, conversion, and facilitation [Mehta00]. Every connector provides services that belong to at least one of these categories. It is also possible to have multi-category connectors to satisfy requirements for richer sets of interaction services. For example, it is possible to have a connector that provides both communication and coordination services. Based on this service categories, the connectors are further classified into eight types, depending on the way they realize interaction services [Mehta00] : procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor.

3.2 SOFTWARE ARCHITECTURE LEVELS

Software architectures can be considered at three abstraction levels : specification, configuration and assembly. Each level captures one aspect of design decisions, corresponding to the different steps of the development process : design, implementation and deployment. These different representations of an architecture share the same architecture building blocks, components and connectors, but model them in different ways. Moreover, these different representations are strongly interrelated, as architecture elements, at a given abstraction level, correspond to refinements of the same elements defined on higher abstraction levels.

3.2.1 Architecture Specification

A software architecture specification translates a set of functionality requirements into a practical software solution. This top level definition of a software architecture gives an abstract overview of its constituting components and connectors, of its organization and

its behavior. Its purpose is to identify the types of a set of components and connectors which can be reused and assembled to perform the required functionalities.

3.2.1.1 Abstract Component Type

Abstract component types, like abstract types in other paradigms, provide external, conceptual definitions of component kinds. An abstract component type can be implemented by several component classes, which provide conformable and executable definitions of its functionalities and behavior. Conversely, a component class can implement several component types, grouping together multiple subjects into a versatile entity. From the component reuse perspective, abstract component types are thus used to classify and regroup component classes that can be considered as comparable candidates, as they implement the same kind of functionalities and behavior. From the development process point of view, abstract component types enable to keep architecture specifications generic and conceptual. Selecting a specific component class is a design decision that corresponds to the implementation of an architecture. To specify an architecture consists in defining the component types which correspond to the roles of components in this architecture. These abstract types only declare the functionalities and behaviors which define the peculiar usage of components in the considered architecture. This way, a wider set of components classes can match the specification and be later selected and used to compose the implementation of the architecture. An abstract component type consists of a functional definition, as a set of interface specifications, and a behavioral definition, as protocols of messages exchanged through the interfaces.

Interface specification. An interface specification defines an interface that must be implemented by a component to cooperate properly with other components within an architecture. An interface specification defines the type of this interface as a set of operation syntactical signatures and a direction (required or provided), describing the types of the messages which should be sent or received through this interface.

Component behavior specification. Component behavior specification defines the behaviors that must be implemented by a component to conforms to its expected roles within the architecture. Those behaviors are externally defined as interaction protocols, describing the activity of the component as sequences of accepted and emitted method calls through its interfaces.

3.2.1.2 Connector Type

Connector specifications can be implicit or explicit. Most of the time, connections between pairs of interfaces are only defined by bindings. The explicit definition of the connectors is postponed to the implementation or the deployment of the architecture. This corresponds to situations in which no other service than default message transport is required from connectors. Explicit connector specifications enable to define the specific interaction management services which are expected from the connector classes that will be chosen to implement the architecture. Connector specifications are close to component specifications (some models, like Fractal [Bruneton06], consider connectors as a specific

kind of components) and consist of a set of interface specifications, defining the types of the connector ends, and a behavior specification, defining declaratively the services which must be implemented by the glue of the connector (protocol control or adaptation, communication facilitation, ...).

3.2.1.3 Topology

Architectural topologies define how components and connectors are composed into graphs that describe the structure of architectures. In many works, topologies are named configurations. In our work, configuration has a different meaning, related to product line management [Choi06, Crnkovic06, van Ommering02] : a configuration does not refer to the structure of an architecture but to a specific set of component classes which define a specific implementation of this architecture (see next section). An architecture topology defines the structure of an architecture as a list of the connections (bindings) that links pairs of component interfaces [1]. Explicit connectors types may be used to specify how the connection must be handled (implemented).

3.2.1.4 Architecture Behavior

The *architecture behavior* defines the behavior that is expected from the whole application. It describes the functional objectives that must be achieved by the architecture built with the selected components and connectors. In other words, architecture behavior transcribes users' requirements into a formal architecture definition which can be used to verify whether a proposed architecture topology meets these requirements. The architecture behavior is less studied compared to component behavior. The SOFA component model [Plasil02] proposes comparable concepts. SOFA enables to specify *frame* behavior protocols, describing the expected behaviors of target architectures to be built, and then to calculate and compare *architecture* behavior protocols, the behaviors of the effective architectures built with components and connectors. Architecture behavior is defined the same way as component behavior, as protocols of interactions between its constituent components and connectors.

3.2.2 Architecture Configuration

An architecture configuration describes an implementation of a software architecture. An architecture is the blueprint of a system from its inception to retirement. As a second step of the development process, an architecture specification is refined into a conform architecture configuration which constitutes a second level of architecture definition, at a more concrete abstraction level. An architecture configuration is defined by a set of component and connector classes which implements the component and connector types declared in the corresponding architecture specification. This way, different configurations can be designed for the same architecture (specification), for instance to take into account the availability of component and connector classes (implementation on multiple platforms, release of new component or connector classes) or to update configurations (new versions

1. In this thesis, we do not cover architecture style but the detailed information can be found in [Abowd93, Shaw95a]

of constituents). Another motivation may be to design configurations for specific non-functional qualities, such as a minimal footprint or a maximal performance.

3.2.2.1 Component Class

Component classes refer to concrete code modules that can be effectively deployed and executed on runtime platforms. Component classes are also, but not only, described as component types, which enable to get definitions of their capabilities and dependencies with no required knowledge about the details of their implementation. Unlike (purely) abstract component types declared in architecture specifications, which aims at defining specific usages of components in the context of specific architectures, the type of a component class contains a complete description of its interfaces and behavior. The non-functional information associated with the description of the component class describes the effective qualities of its implementation. A component class can be primitive or composite. A primitive component class embeds directly a set of functionalities. A composite component embeds an internal architecture, i.e is hierarchically composed of a set of inner components. Composite components enable to reuse whole architectures, which are designed to achieve given functional objectives, as single components in larger architectures. In both case, the component class refers to an entity (generally an object class or a configuration) that defines its implementation. The types are used to search and select the component classes that can be reused to build an architecture. The type of the available component classes is matched against the abstract component types declared in an architecture specification. To be a candidate for the architecture configuration, the type of the component class must be a specialization of an abstract component type, so that it can be substituted to this abstract component type in the definition of the architecture specification while keeping the architecture behavior specification coherent. More details about applicable specialization rules between component interfaces and component behaviors can be found in [Plasil02, Desnos08, Arévalo09].

3.2.2.2 Connector Class

As presented in the previous section (architecture specification), explicit connectors that handle specific interaction management services (protocol adaptation or control, communication facilitation, ...) can be considered as special components. Thus, the definition of a connector class is close to the definition of a component class and consist of a type and a reference to an implementation. The type of a connector class is described as a pair of interfaces and a behavior protocol. As for component classes, the type of a connector type provides a complete description of the interfaces and behaviors implemented by the connector class, whereas abstract connector types in architecture specifications only describe specific usages. The implementation of a connector class is generally defined as a reference (qualified name, uri,..) to the object class that provides an implementation of the connector class. It is often a generic factory class that is able to generate connector classes according their type descriptions (interfaces and protocols) during the deployment of architectures.

3.2.3 Architecture Assembly

Architecture assemblies define the deployment, assembling and execution of architectures on execution environments. This is the third level of architecture descriptions. Architecture assemblies define how architecture configurations are instantiated as sets of fully parametrized components. These design decisions relate to defining specific identities, locations, preferences or attributes values of components in order to customize architectures to given users. This architecture description level is often ignored in many works, as they consider that the definition of architectures at runtime is out of the scope of design decisions but is a matter of architecture management by component frameworks. However, much information at this level affects the deployment and later the evolution of architectures. When this level is missing, a gap exists between the higher level definitions of architectures and their runtime representations.

3.2.3.1 Component Instance

Component instances define the individual components that compose a specific deployment of an architecture. Each component instance is an instance of the corresponding component class in the architecture configuration. As component classes enable to choose specific implementations for the component types declared in the architecture specification, component instances enable to choose and define the identities, preferences and attribute values of individual components. Sometimes, the explicit definition of the component instance is replaced by assembly constraints, a set of rules that must be enforced when an architecture is instantiated and assembled by the runtime environment to be considered valid.

3.2.4 Architecture Description Levels in Different Development Processes

Architectures form the underpinning of software systems from their inception to their retirement. Development methods focus on capturing and combining the many design decisions that conduct the definition of softwares into well defined and organized models. In traditional object-oriented approaches, softwares are developed from scratch. Their architectures are blueprints that define how to code softwares. Their design is not aimed at producing reusable assets but architectures that meet given functional requirements. A strong coherence and coupling exists thus within and between the different models that define softwares along their development life-cycle. In component-based approaches, softwares are intended to be constructed by the reuse of COTS components. Architectures are used to define how to search, select and assemble existing components into new softwares. The many elements and models that define architectures are generic and decoupled, in order to enforce reusability. The main differences between the architecture definitions in these two approaches are summed up in Table 3.1.

Architecture specifications. In classic approaches, an architecture specification is defined with component types that correspond directly to component classes. In a forward engineering approach, component types in architecture specification define the functiona-

Architectures	Traditional development	CBSE
Specification	Complete component types	Abstract (partial) component types.
Configuration	Components are (1) programmed according to the complete component type, (2) isomorphic with component types.	Components are (1) selected among existing COTS components component types, (2) have their own concrete component types, (3) are compatible with (matches) the required abstract (partial) component types.
Assemblies	Instantiation with few customization	Instantiation with customization. Reuse of existing components (presets, samples, profiles,...).

TABLE 3.1 – The architectural comparison between two software developments

lity and behavior that is to be implemented in corresponding component classes. These (still abstract) component types can thus be named as complete as they fully describe the set of the functionalities and behavior implemented in component classes. Components and architectures are designed for a specific software (requirements). Component types are often described in a rather detailed way, as they are used not as an abstract documentation of implemented features, but as a definition of the features that must be implemented. Moreover, these component types tend to be more coupled as specific concrete types are used in their definitions and as they are designed together, in the context of the same software development. In component-based approaches, architecture specifications are defined with abstract component types that do not correspond to component classes. These purely abstract component types are used as criteria to search and select for the component classes to be reused in the implementation of the architecture. This way, decoupled component classes, are reused to build different software architectures.

Architecture configurations. In classic approaches, the essence of the implementation of an architecture is to be faithful to the corresponding specification. The simplest understanding of a faithful implementation is a bijective relationship between the model elements defined in an architecture specification and the model elements coded in its implementation. However, this conception of the relationship between architecture specifications and configurations is not suitable for reuse-oriented development. In these approaches, existing component classes are selected as candidate implementations when they are compatible with the abstract component types declared in architecture specifications. This does not imply that the selected component classes are isomorphic implementations of these component types. Component classes may have extra or specialized features (interfaces or behaviors) or may implement as a single entity several abstract types. Applicable compatibility rules can be found in [Plasil02, Desnos08, Arévalo09].

Architecture assemblies. In classic approaches, components are designed and developed for specific requirements. Components are rather specialized and provide thus few parametrization features to customize them for specific usages. Conversely, COTS components are designed and programmed for generic purposes. The customization of the component instances is required to use them properly. Besides, the customization process can be complex enough to justify the existence of libraries of reusable component instances, that propose default component instantiations, presets for specific usages, ...

3.3 ARCHITECTURAL MODELING

An *architectural model* is an artifact that captures the design decisions that define an architecture. *Architectural modeling* is the process that manages the production of an architecture model to document those design decisions. An *architectural modeling notation* is a set of languages in which an architecture model can expressed. Two kinds of notations for modeling software architectures are proposed : dedicated architecture description languages (ADLs) and the Unified Modeling Language (UML). UML 1.x is object-oriented and does not provide notations to define component-based architectures [Garlan02]. But UML 2.x has more recently introduced new concepts and notations which enable to define full-fledge component-based models. So UML is now also considered as a candidate ADL and the mapping of the concepts of more developed existing ADLs are studied [Roh04, Inverardi05, Oquendo06, Choi06]. This section describe the criteria used to compare the architectural modeling notations proposed by a set of influential ADLs.

3.3.1 Architecture Description Language

A software system architecture [Taylor09] gathers design decisions made about the system. It is expressed using an ADL which, in most cases, provides information on the structure of the software system listing the components and connectors the system is composed of. Quality attributes are sometimes provided (*e.g.,* xADL2.0 [Dashofy01, Dashofy02a]). The dynamic behavior of systems is often described (*e.g.,* C2SADEL [Medvidovic99b], Wright [Allen97b, Allen98], SOFA 2.0 [Plásil98, Plasil02]) but their descriptions are not homogeneous as various means (*e.g.,* message-based communication, CSPs, regular expressions) are used.

Definition. "*An **architecture description language (ADL)** 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 [Medvidovic07].* "

ADLs usually describe :

– a class of systems as composed of component classes and connector classes. This description is called an architecture specification. In some systems, connector types are not listed explicitly but deduced from component ports (*e.g.,*Darwin [Magee95, Magee96]). The types of the component classes that are used can either be defined inside of the architecture specification (*e.g.,*Wright [Allen97b, Allen98]) or separately, (*e.g.,*C2SADEL [Medvidovic99b]), what is better as component types are then reusable.

– an instance of system as composed of component instances and connector instances. This definition is called an architecture configuration.

This scheme is not aligned with the three level models we advocate for architectures designed with a component-based approach.

3.3.2 The Criterion of Comparison

In the Section 3.2, we have defined the three representation levels which should be proposed by ADLs to define architectures. The support for each of these three levels and their combinations into architecture definitions are used in the remainder of this chapter to compare and evaluate different ADLs.

3.3.2.1 Modeling Architecture Specifications

Modeling architecture specifications requires at least three basic elements : abstract component types, connections and architecture behaviors.

– *Abstract component type.* Does the ADL support this concept and enable the partial definition of interfaces and behaviors which describe specific usages of components ?
– *Connections.* How are defined connections ? Do they enable to use abstract component types ?
– *Architecture behavior.* Does the ADL model architecture behavior ? Is it considered as an independent definition of the expected global behavior of an architecture, used to verify that the combination of the behavior of its components is coherent ? Is it rather a calculation of the global behavior that results from the interactions of the components within the architecture ?

3.3.2.2 Modeling Architecture Configurations

Architecture configuration descriptions refer to the component classes and connector classes used to implement architectures. Each component or connector class is described by a concrete type which provide an abstract and external definition of the functionalities and behaviors it implements. Separating concrete type definitions from component class definition enable to reuse them to describe classes and regroup them as implementations of comparable functionalities and behavior.

– *Component type.* Does the ADL support a separate description of component types in order to classify component classes ? How do component types support the definition of interfaces and component behavior ?
– *Component class.* Does the ADL support the definition of composite component classes ?
  – *Primitive component class.* How does the ADL define the implementation, the configurable attributes and the non-functional information of primitive component classes ?
  – *Composite component class.* How does the ADL define the hierarchical structure of composite component classes ? How are established the delegation links between the interfaces of the composite components and the interfaces of their internal components ?

– *Connector class.* How does the ADL model connector classes – explicitly or implicitly ?

3.3.2.3 Modeling Architecture Assemblies

The architecture assemblies describe the runtime architectures of software systems. They should be modeled with component instances, connector instances along with assembly constraints.

– *Component instance.* How does the ADL support the definition of the customization information used to instantiate specific components ?
– *Connector instance.* Does the ADL support the explicit definition of connector instances ?
– *Assembly constraints.* Does the ADL support declarative (implicit) definitions of architecture assemblies as sets of constraints (assertions) that must be enforced by the component and connector instances created at runtime ?

Architecture Specification	Architecture Configuration	Architecture Assembly
Component role	Component class	Component instance
+Interfaces	+Component type	+Attributes
+Component Behavior	+Interfaces	Connector instance
Connection	+Component behavior	Assembly constraint
Architecture behavior	▷Primitive component	
	+Implementation	
	+Attributes	
	▷Composite component	
	+Composition	
	+Delegation	
	Connector class	
	+Connector type	
	+Interfaces	
	+Connector behavior	
	+Implementation	

TABLE 3.2 – Expected expressiveness of ADLs in the different representation models

3.3.2.4 Applying the Comparison Criteria

Each ADL proposes specific means to model architectures. First, a concise presentation of the original concepts of each ADL are presented. Then their suitability to define architecture specifications, configurations and assemblies is discussed.

A conceptual example is used to concretely compare the concepts of the different ADLs. Its structure is depicted in Fig. 3.1. This simple system called *ArchExmaple* is composed of three components (*CompA*, *CompB* and *CompC*) and two connectors (*Connector1* and *Connector2*). *CompB* is a composite component which embeds an inner architecture composed of two components (*CompB1* and *CompB2*) connected by *Connector3*.

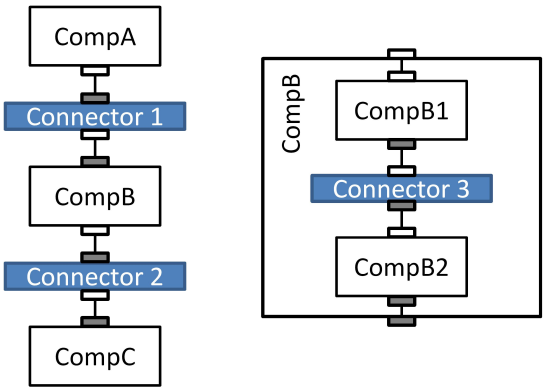


FIGURE 3.1 – Structure of the *ArchExample* system

3.4 A COMPARISON OF EXISTING ADLS

This section examines a set of influential ADLs including C2SADEL, Wright, Darwin, Unicon, SOFA 2.0, FractalADL and xADL2.0. The total number of existing ADLs are more than twenty. The ADLs in our comparison are chosen based on two criteria : (1) their representativeness, (2) the way they model architectures. Considering our selection of ADLs, first of all, they are all widely referenced by academic and industrial works. Secondly, they cover the many ways to model architectures which can be found in existing ADLs. For instance, UML 2.0 cannot yet be considered, to our opinion, as a significantly used nor innovative ADL and was not selected. A perspective is to extend UML 2.0 with the concepts proposed in Dedal to promote its usage as an ADL.

The original concepts proposed by these ADLs are highlighted. The concrete usage of these concepts to model architectures is illustrated and discussed using the *ArchExample* system.

3.4.1 C2SADEL

C2SADEL [Taylor96, Medvidovic99b] is the language proposed for defining architectures built according to the C2 style. C2 is a component and message based "architecture style". C2SADEL models the specification and configuration levels of software architectures. Besides, it is one of the ADLs that clearly separates architecture specifications from configurations.

3.4.1.1 Architecture Specification

In C2SADEL, architecture specifications are composed of (1) sets of component types (called *"conceptual_components"*) [2], a set of connectors and a set of connections (called "*architecture_topology*" in C2SADEL) (cf. Figure 3.3), as shown in Table 3.3.

Component type ("conceptual_component"). Component types in C2SADEL are defined by two interfaces (the top and bottom *domains*) and a behavior. Each ("*domain*") interface

2. The word in italic in the brackets represents the name given in each ADL.

Architecture Specification	Concept name	Characteristics
Specification	"architecture"	Explicit, independent specification
Component Type	"conceptual_component"	Concrete component types
Connector	"connector"	Four built-in types. Connector implicitly managed by the runtime framework.
Connection	"architecture_topology"	Defined by specifying the connections of connectors.
Architecture Behavior	—	—
Component Type	**Concept name**	**Characteristics**
Interfaces	"top_domain", "bottom_domain"	Each component have one *top_domain* and one *bottom_domain*.
Component Behaviors	"behavior"	Message-based communication.
Connector Type	**Concept name**	**Characteristics**
Builtin connector type	"connector type"	Four types : *no_filtering*, *notification_filtering*, *prioritized*, and *msg_sink*.
Interfaces	"top_ports", "bottom_ports"	Ports are dynamically generated according to connected components.

TABLE 3.3 – Expressiveness of C2SADEL for architecture specification

is defined by the set of messages that may be sent and received through the interface. Secondly, the component behavior is defined message-based as a message-based protocol. It contains the *startup*, *cleanup* and *received_messages* sub-protocols. *Startup* and *cleanup* are optional parts of the specification that indicate any special processing needed after the component is instantiated and before it is removed from a system, respectively. *Received_messages* defines the interactions that are triggered by the reception of messages.

Connector type. C2SADEL provides a fixed set of built-in connector types which are dedicated to the management of interaction protocols : *no_filtering*, *notification_filtering*, *message_filtering*, *prioritized* and *msg_sink* [Taylor96]. The interfaces of connectors are not explicitly defined in the specification but calculated as *context-reflection* features : the interfaces of a connector are determined dynamically by the interfaces of the components that communicate through it. The interfaces of connectors are grouped in two sets called *top_ports* and *bottom_ports*, which correspond conceptually to the layered architectural style of C2.

Connections ("architectural_topology"). The topology of architecture is specified by describing the connections of connectors. In C2SADEL, the top (interface) of a component may be connected to the bottom (ports) of a single connector and the bottom (interface) of a component may be connected to the top (ports) of a single connector. A component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors.

Example of an architecture specification in C2SADEL. The architecture specification of the *ArchExemple* system in C2SADEL syntax is shown in Fig 3.3. In C2, the composite component is explicitly defined by wrapped them with two additional interfaces in its top and bottom (see Fig. 3.2).

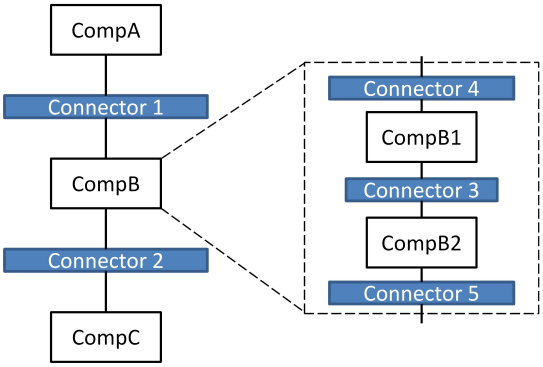


FIGURE 3.2 – Graphic specification of the ArchExample in C2SADEL

```
1 architecture ArchExample is
2  conceptual_components
3    CompA; CompB1; CompB2; CompC;
4  connectors
5   connector Connector1 is message_filter no_filtering;
6   connector Connector2 is message_filter no_filtering;
7   connector Connector3 is message_filter no_filtering;
8   connector Connector4 is message_filter no_filtering;
9   connector Connector5 is message_filter no_filtering;
10 architectural_topology
11  connector Connector1 connections
12   top_ports CompA;
13   bottom_ports connector4;
14  connector Connector4 connections
15   top_ports Connector1;
16   bottom_ports CompB1;
17  connector Connector3 connections
18   top_ports CompB1;
19   bottom_ports CompB2;
20  connector Connector5 connections
21   top_ports CompB2;
22   bottom_ports Connector2;
23  connector Connector2 connections
24   top_ports Connector5;
25   bottom_ports CompC;
26 end ArchExample;
```

FIGURE 3.3 – Specification description of *ArchExample* in C2SADEL

```
component CompA is
 interface
  bottom_domain_is
   out FindName();
   in Name(p : String);
  behavior
   startup always generate
   FindName;
end CompA
```

FIGURE 3.4 – Component type *CompA* description in C2SADEL

3.4.1.2 Architecture Configuration

In C2SADEL, a configuration is called a *system*. It is specified from architecture specification by instantiating the component types. It contains two parts including a reference to the refined architecture specification, and a set of components that are called *component instances*. Information about connector instances is omitted, as connectors are implicitly managed by the runtime framework. The overview of configuration is shown in Table 3.4.

Architecture configuration	Concept name	Characteristics
Configuration	"system"	Instantiated from specification ("*architecture*").
Component	"instance"	Instantiated from component types ("*conceptual_components*").
Connector	–	–

TABLE 3.4 – Expressiveness of C2SADEL for architecture configurations

Component ("component instances"). In C2SADEL, a component type in specification can instantiated into many component instances in configuration.

Example of a configuration in C2SADEL. Figure 3.5 presents the configuration of the *ArchExample* system in C2SADEL syntax. The content of a C2SADEL configuration is very simple, just defining the set of component instances that form a given system.

```
system ArchExample_1 is
  architecture ArchExample with
  CompA instance a;
  CompB1 instance b1;
  CompB2 instance b2;
  CompC instance c;
end ArchExample_1;
```

FIGURE 3.5 – Configuration of *ArchExample_1* in C2SADEL

3.4.1.3 Discussion

C2SADEL proposes only two abstraction levels to define architectures. They lack a more detailed configuration definition which would enable to explicitly represent different versions of the implementation of an architecture, recording the design decisions of selecting and combining specific sets of component classes among many other possible choices. Moreover, as C2SADEL does not support the definition of abstract component types (as opposed to the concrete component types defined by component classes), it is difficult to model architectures in which several distinct instances of a same component type are used to play different roles. As all the different roles of these instances are represented in the specification by a unique component type, the instantiation of some architecture specifications may become ambiguous.

3.4.2 Wright

Wright [Allen97b, Allen97c, Allen97a] is focused on formalizing the behaviors of components and connectors. Component and connector behaviors are specified using concepts derived from the Communicating Sequential Processes (CSP) formalism [Hoare78]. Architecture models have only one definition level — configuration.

3.4.2.1 Type System

In Wright, components and connectors are typed. Wright defines component and connector types in two ways : (1) as a configuration description, or (2) as a *style* description. A *style* defines a set of properties that are shared by the configurations that are members of the style. These properties include a common set of elements (component and connector types) and restrictions (connection *constraints*) on the way they can be used in configurations. The overview of type system of Wright is shown in Table 3.5.

Component Type	**Concept name**	**Characteristics**
Interfaces	"Port"	CSP-based.
Component behavior	"Computation"	A behavior specification (primitive component) or an architectural description (composite component).
Connector Type	**Concept name**	**Characteristics**
Interfaces	"Role"	CSP-based.
Connector behavior	"Glue"	CSP-based.
Composite component[c]	**Concept name**	**Characteristics**
Composition[c]	"Configuration"	Embedded in *Computation* description.
Delegation[c]	"Bindings"	Embedded in *Computation* description.

TABLE 3.5 – Wright type system

Component type. In Wright, a component type holds two parts : interfaces (called *ports*) and component behavior (called *computation*). A *port* process defines the local protocol with which the component interacts with its environment through that port. Wright

supports the hierarchical definition of composite components. The behavior of a primitive component type is defined in its description by different *computations*. The behavior of a composite component type is defined by its architectural description.

- *Primitive component computation.* A computation defines the behavior of a component as protocols (regular expression) of accepted and emitted events through its ports. $e \rightarrow P$ represents prefixing. $\overline{p.request}$ represents a request on port p. $p.reply$ represents a reply on port p. §represents successful termination. $\sqcap$ represents the internal choice (decision). $\square$ represents the external choice (alternative).
- *Composite component computation.* A composite component defines an abstraction boundary (black box) that encapsulates its nested architecture. The computation contains a nested architecture configuration description and an associated set of delegations. Delegations are called *bindings* in Wright. They define how the unattached ports of the inner components are linked to ports of their enclosing composite component.

Connector type. In Wright, a connector type is modeled by a set of interfaces (called *roles*), and a connector protocol (called *glue*). A *role* specifies the protocol that must be satisfied by any port that is attached to that role. The *glue* describes how the roles of a connector interact with each other.

3.4.2.2 Architecture Configuration

In Wright, a configuration is defined by three elements : a specification ("*style*"), a set of component and connector instances and the way they are assembled. The components and connectors are named *instances*. The connections are named *attachments*. The overview of architecture configuration modeled in Wright is shown in Table 3.6.

Architecture configuration	Concept name	Characteristics
Configuration	"Configuration"	Instantiated from specification ("*Style*").
Component	"Instance"	Instantiated from component types ("*Component*").
Connector	"Instance"	Instantiated from connector types ("*Connector*").
Connections	"Attachments"	

TABLE 3.6 – Expressiveness of Wright for architecture configurations

Connections ("attachments"). The attachments define the topology of the configurations, showing which components participate in which interactions. This is done by associating a component's *port* with a connector's *role*. The attachment declaration *Comp.client as Conn.server* indicates that the port *Client* of the component *Comp* will play the role *Server* of Connector *Conn* in the interaction.

Example of a configuration in Wright. Figure 3.6 presents the configuration of the *ArchExample* system in Wright syntax.

```
1  Configuration ArchExample
2   Component CompA
3    Port r = request → reply → r⊓§
4    Computation= internalCompute→ r.request→r.reply → Computation⊓§
5   Connector connector1
6    Role r = request → reply → r⊓§
7    Role p = request→ reply → p□§
8    Glue = r.request→p.request→Glue
9           □ p.reply→r.reply→Glue
10          □ §
11  ...
12  Component CompB
13   Port r
14   Port p
15   Computation
16    Configuration CompBConfig
17     Component CompB1
18     ...
19    Instances
20     b1 :CompB1
21     ...
22    Attachments
23     b1.r as l3.p
24     ...
25   EndCompBConfig
26   Bindings
27    B1.p=p; B2.r=r
28   End Bindings
29 Instances a :CompA; conn1 :Connector1; b :CompB; conn2 :Connector2;
30           c :CompC
31 Attachments a.r as conn1.p; b.p as conn1.r; b.r as conn2.p;
32             c.p as conn2.p
33 EndConfiguration
```

FIGURE 3.6 – Example of *ArchExmaple* configuration in Wright

3.4.2.3 Discussion

Wright models an architecture with a unique representation. It enable to describe component types, their multiple instances (roles) used in the architectures and the topology of their connections. Wright was clearly not designed for development processes based on reuse. Component types are not defined as independent, reusable model elements. Moreover, without any independent architecture configuration and assembly descriptions, Wright do not support the definition of abstract architecture types (what we call architecture specification) that could be reused to define different implementations (versioning, product line) or different instantiations (customization, localization) of a same conceptual architecture. In other words, a Wright configuration is designed to define one specific architecture.

3.4.3 Darwin

Darwin [Magee95, Magee96] is a general-purpose architecture description language for specifying the structure of systems composed from components that communicate through explicit interfaces. Darwin has a canonical textual representation in which components and their interconnections are described. There is an associated graphical representation that depicts Darwin configurations in a more accessible but less precise way. A Darwin architecture model consists of only one architecture definition — configuration. Darwin provides a hierarchical composition scheme to define components. There is no formally defined "configuration" keyword in language. An architecture configuration is modeled as a root composite component that encapsulates all the other components of the architecture. In Darwin, architectures are described as sets of interconnected components. There is no explicit connector concept, but components that facilitate interactions are regarded as connectors.

3.4.3.1 Architecture Configuration

In Darwin, an architecture is defined as a hierarchically structured composite component. The architecture configuration is described by a set of component instances and their interconnections.

Primitive components types. Darwin primitive component types expose a set of provided and required *services*, sometimes also called *ports*. *Services* in Darwin correspond to our notion of interfaces.

Composite components types. Composite component types are constructed from the primitive components and can in turn be used to define more complex composite components. Composite component types are described by a set of interfaces, a set of sub-component instances and a set of bindings between these sub-components.

The differences between configurations (root composite components) and (regular) composite components stand in two points : (1) composite components have interfaces and (2) composite components have hierarchic bindings. Hierarchic bindings (delegation connections) bind the interface provisions and requirements of a composite component to the interfaces of its constituent components.

The overview of configuration modeled in Darwin is shown in Table 3.7.

Example of an architecture configuration in Darwin. Figure 3.7 presents the configuration of the *ArchExample* system in the Darwin syntax.

3.4.3.2 Discussion

Darwin provides a set of constructs with which dynamic constructions of architectures can be specified, such as loops. Like Wright, the purpose of Darwin is clearly to manage the deployment of architectures by describing the instantiation and the assembly of their components. Reuse is improved in Darwin, as the many component types are independently defined and thus can be used to define different architectures. However, the lack of explicit

Architecture Configuration	Concept name	Characteristics
Configuration	"component"	The configuration is represented by a composite root component type.
Component	"***inst***ance"	Component instances refer to externally defined primitive or composite component types.
Connector	—	Connectors are defined as specialized components.
Connections	"***bind***ing"	No explicit connector declaration. Bindings between component interfaces.
Primitive Component	**Concept name**	**Characteristics**
Interface	"service" or "port"	Provided or required services.
Behavior	—	—
Composite Component	**Given name**	**Characters of an ADL**
Delegation	"***bind***ing"	Hierarchic binding between ports of the composite components and ports of its components.

TABLE 3.7 – Expressiveness of Darwin for architecture configurations

```
component ArchExample {
 inst
  A : CompA;
  B : CompB;
  C : CompC;
 bind
  A.r - B.p
  B.r - C.p
}
component CompA {
  require r;
}
```

```
component CompB {
 require r;
 provide p;
 inst
  B1 : CompB1;
  B2 : CompB2;
 bind
  B1.p - B2.r
  B1.r - r
  B2.p - p
}
```

FIGURE 3.7 – Configuration of *ArchExample* and component of *CompB* in Darwin

and separate architecture specification definitions does not enable to represent the alternate multiple implementations of a same conceptual architecture. The same way, there is no support to represent different specific deployments of an architecture, as explicit separate architecture assembly definitions.

3.4.4 Unicon

Unicon [Shaw95b, Shaw96a] is an architecture description language which aims to model the architectures of distributed systems. Like Darwin, Unicon supports the hierarchical construction of composite components. It models just one definition of software architectures which can mainly be considered as a configuration description.

3.4.4.1 Type System

Primitive component types. In Unicon, the definition of a primitive component types consist of a description of its "interface" and of its implementation (see Fig. 3.8).

```
1 COMPONENT CompA
2   INTERFACE IS
3     TYPE Module
4     PLAYER r IS RoutineDef
5       SIGNATURE "rItf"
6       END
7   END INTERFACE
8   IMPLEMENTATION IS
9     VARIANT aImpl IN "CompAImpl"
10      IMPLTYPE source
11    END
12  END IMPLEMENTATION
13 END CompA
```

FIGURE 3.8 – Description of the primitive component type *CompA* in Unicon

The component type "interface" defines its abstract, external representation (as a Java interface defines an abstract object type). It consists of the three following parts :

- *Built-in component type.* Unicon provides a set of generic component built-in types which are used to identify the main responsibility of components and thus the general characteristics of their implementations. The proposed built-in types are *Module*, *Computation*, *SharedData*, *SeqFile*, *Filter*, *Process*, *SchedProcess*, and *General*.
- *Specific properties.* They are attributes the values of which are used to specialize built-in component types.
- *A list of players.* Players define the interaction points of the component (its interfaces, in a more common component vocabulary). Unicon specifies of the cardinality of the connections in which a player can be involved. Unicon supports a predefined set of common player types, including *RoutineDef*, *RoutineCall*, *GlobalDataDef*, *GlobalDataUse*, *ReadFile*, *WriteFile*, *PRCDef*, and *PRCCall*. A signature can be used to define the type of the input and output data exchanged through a player.

A primitive component type definition also identifies its available implementations. Each implementation is called a "variant". Each variant bears a distinctive name and refers to the source code or to an executable definition (object code) of the implementation of the component type. A set of properties may be used to specify deployment informations (for instance compilation options).

Composite component types. Composite components provide the mechanism for building up subsystems from primitive components and hierarchically complete architecture (top composite components) from smaller subsystems (nested components). The definition of a composite component type consists of three kinds of information (see Fig. 3.9 :

- *The parts.* Component and connector instances from which the composite component is constructed. Component instances are defined in the ***CompUses*** clause. Connector instances are defined in the ***ConnUses*** clause. ***CompUses*** and ***ConnUses***

```
1  COMPONENT CompB
2    INTERFACE IS
3      TYPE Module
4      PLAYER r IS RoutineDef
5        SIGNATURE "rItf"
6        END
7      PLAYER p IS RoutineCall
8        SIGNATURE "pItf"
9        END
10   END INTERFACE
11   IMPLEMENTATION IS
12     USES b1 INTERFACE CompB1
13     USES b2 INTERFACE CompB2
14     BIND p TO b1.p
15     BIND r TO b2.r
16     CONNECT b1.r TO b2.p
17   END IMPLEMENTATION
18 END CompB
```

FIGURE 3.9 – Description of the composite component type *CompB* in Unicon

have the same syntax, using the *USES* keyword to designate the used component and connector instances.

- *The configuration.* The set of connections that link the players of the components, eventually using explicit connector instances. Connections are enumerated in the ***Connect*** clause.
- *The abstraction.* The set of delegation connections that link the players of the composite component with the players of its nested components. Delegations are established in the ***Bind*** clause.

Connector types. Unicon supports the explicit definition of connector types. The definition of a connector type is close to the definition of a primitive component type. It consists of (1) a protocol definition (connector type) and (2) an implementation definition (see Figure 3.10). The protocol definition contains three parts :

- *Built-in connector type.* Unicon provides *built-in connector types* to express the general purposes of designed connector types. The proposed built-in types are : *Pipe*, *FileIO*, *ProcedureCall*, *DataAccess*, *PLBundler*, *RemoteProCall* and *RTScheduler*.
- *Specific properties.* Optional or mandatory attributes are be used to customize the use of the built-in types.
- *Interfaces.* The interfaces of connectors are called *roles* in Darwin. Unicon supports a predefined set of generic role types including : *Source*, *Sink*, *Reader*, *Readee*, *Writer*, *Writee*, *Caller* and *Definer*. Each type of *role* has an associated property list which can be used to customize the definition of a specific connector role type.

The overview of types modeled in Unicon is shown in Table 3.8.

```
1 CONNECTOR Connector1
2   PROTOCOL IS
3     TYPE ProcedureCall
4     ROLE r IS Definer
5       MAXCONNS 1
6       END
7     ROLE p IS Caller
8       MAXCONNS 1
9       END
10  END PROTOCOL
11  IMPLEMENTATION IS
12    BUILTIN
13  END IMPLEMENTATION
14 END Connector1
```

FIGURE 3.10 – Description of the connector type *Connector1* in Unicon

Component type	**Concept name**	**Characteristics**
Component type	"interface"	derived from built-in generic component types
Interfaces	"players"	derived from built-in generic interaction point types
Connector type	**Concept name**	**Characteristics**
Connector type	"protocol"	derived from built-in generic connector types
Interfaces	"roles"	derived from built-in generic connector end types

TABLE 3.8 – The component type model of Unicon

3.4.4.2 Architecture Configuration

As in Darwin, architecture configurations can be defined in Unicon thanks to top composite components embedding the whole architecture (see Fig. 3.11). An architecture configuration is defined in the implementation clause of a component type definition as the set of component and connector instances used to build the architecture. The structure of the architecture is defined by a set of connections between the component instance players and the connector instance roles. Table 3.9 gives an overview of configuration modeled in Unicon.

Example of a configuration in Unicon. Figure 3.11 presents the configuration of the *ArchExample* system in the Unicon syntax.

3.4.4.3 Discussion

The features of Unicon are closed to those of Darwin. Component types can be defined separately and flexibly reused to build many different architectures, as hierarchically composed composite components. Unicon provides a more powerful architecture configuration definition and enable to regroup a set of architecture configurations as alternative

Architecture configuration	Concept name	Characteristics
Configuration	"implementation"	Configurations are represented as top composite components
Component	"component uses"	Component instances are defined as uses of a component type.
Connector	"connector uses"	Connector instances are defined as uses of a connector type.
Connection	"***Connect***ion"	
Primitive component	**Concept name**	**Characteristics**
Primitive component type	"component"	
Component type	"interface"	embedded in the concrete component type description
Implementation	"implementation"	refers to the different available implementations (executable code) of the component type
Attribute	—	—
Composite component	**Concept name**	**Characteristics**
Composite component	"component"	contains a definition of an embedded architecture (a set of component and connector instances, structured by connections)
Component type	"interface"	embedded in the concrete component type description
Delegation	"***Bind***ing"	specific connections (bindings) between players of the composite component and players of its inner components

TABLE 3.9 – The architecture configuration expressiveness of Unicon

implementations (variants) of a same component type (interface). However, the type definition (interface) of composite component types cannot be (re)used as conceptual definitions of architectures (architecture specifications). Indeed, these component type definitions contain no information about the related conceptual architecture topology. Each architecture configuration described in the implementation clause of the component type can define its own structure, provided it meets the requirements defined by the interface of the component. Moreover, the definition of the architecture assemblies is mixed with the definition of the architecture configuration in the implementation clauses. This does not enable the definition of multiple instances of a given architecture configuration.

3.4.5 SOFA 2.0

SOFA 2.0 [Hnetynka05, Bures06] is a component-based development platform which features : hierarchically composed components, ADL-based design, behavior specification using behavior protocols, automatically generated connectors supporting seamless and transparent distribution of applications and a distributed runtime environment with dynamic update of components.

SOFA 2.0 covers two levels of architecture definition : configuration and non descriptive

```
1 COMPONENT ArchExample
2   INTERFACE IS
3     TYPE General
4   END INTERFACE

5   IMPLEMENTATION IS
6     USES a INTERFACE CompA
7     USES b INTERFACE CompB
8     USES c INTERFACE CompC

9     USES conn1 INTERFACE Connetcor1
10    USES conn2 INTERFACE Connector2

11    CONNECT a.r TO conn1.p
12    CONNECT conn1.r TO b.p
13    CONNECT b.r TO conn2.p
14    CONNECT conn2.r TO c.p
15  END IMPLEMENTATION
16 END ArchExample
```

FIGURE 3.11 – An architecture configuration of *ArchExmaple* in Unicon

assembly.

3.4.5.1 Type System

In SOFA 2.0, an architecture is represented as a composite component type. A composite component type description can be considered as an architecture configuration. Table 3.10 gives an overview of types supported by SOFA 2.0.

Component type	Concept name	Characteristics
Component type	"frame"	Primitive or composite component type definition
Interface	"interface"	Specified by interface types. Defined in ***provides*** and ***requires*** clauses.
Behavior	"protocol"	Defined by behavior protocols (regular expressions).

TABLE 3.10 – Component type definitions in SOFA 2.0

Component types. In SOFA 2.0, component types are called *frames* (see Fig. 3.12. A frame provides black-box definition of a component by defining (1) its provided and required interfaces, which are defined using interface types, and (2) a behavior that is formally described by behavior protocols [Plásil98].

Behavior protocols are used for behavior specification in the SOFA [Plásil98] and the Fractal [Bruneton06] component models. A behavior protocol is a regular expression describing the behavior of a component as sequences (traces) of accepted and emitted events through the interfaces of the component.

– `!i.m` (*resp.*) denotes an outgoing call of method `m` on interface `i`.

```
1 frame CompB {
2   provides :Server p;
3   requires :Client r;
4   protocol :(?p.findName{?r.findName};)
5 };
```

FIGURE 3.12 – Example of a component type *CompB* in SOFA 2.0

- `?i.m` denotes an incoming call of method `m` on interface `i`.
- `A+B` is for `A` or `B` (exclusive or).
- `A;B` for `B` after `A` (sequencing).

Connector types. In SOFA 2.0, connector types are called *connector frames*. A frame provides black-box view of a connector by defining its provided and required roles [Bálek01] (the types of the connector ends).

3.4.5.2 Architecture Configuration

SOFA 2.0, like Darwin and Unicon, supports the hierarchical construction of composite components. An architecture configuration is modeled thanks to a composite component type. In SOFA 2.0, a composite component is called an *architecture*. An *architecture* defines the implementation of a *frame* (component type), as shown in Table 3.11.

Architecture configuration	**Concept name**	**Characteristics**
Configuration	"architecture"	An architecture configuration is modeled as a composite component.
Component	"***inst***ance"	Nested in the inner architecture of a composite component
Connector	—	No explicit definition ; dynamically generated
Connection	"***bind***ing"	Connection between peer components in the inner architecture of a composite component
Composite component	**Concept name**	**Characteristics**
Composite component	"architecture"	Implements a component type as an inner architecture (set of component instances).
Delegation	"***Delegat(e)***ion" and "***Subsum(e)***ption"	Connections between the interfaces of the composite component and the interfaces of its inner components. *Delegation* denotes connections between provided interfaces. *Subsupmtion* denotes connections between required interfaces.

TABLE 3.11 – Expressiveness of SOFA 2.0 for architecture configurations

Primitive component. The architecture definitions of primitive components are left empty. The declaration of a primitive component is used to map the name of a primitive

component class with the name of the frame (component type). The primitive component class contains directly code that implements the interfaces and behaviors described by the frame (it is not defined by an inner architecture).

Composite component. A composite component defines the implementation of a frame as an inner architecture. This architecture is described by a set of constituent subcomponents instances, bindings among interfaces of the subcomponents, and delegation between the interfaces of the internal subcomponents to the external interfaces of the enclosing composite component. SOFA 2.0 distinguishes two kinds delegation links : *delegations* and *subsumption* (see Fig. 3.13). *Delegation* connects a provided interface of the component to one of its subcomponent's provided interface and *subsumption* connects a subcomponent's required interface to a required interface of the component. SOFA allows to use both frames and architectures to define the types of the subcomponents. Frames are used as variation points in the definition of architectures. The choice of an effective implementation (architecture) for such a subcomponent is postponed to the deployment time and is documented in a specific deployment descriptor.

```
1 architecture CompBConfig implements CompB {
2   inst :CompB1 b1;
3   inst :CompB2 b2;
4   bind b1 :r to b2 :p;
5   delegate p to b1.p;
6   subsume b2 :r to r;
7 };
```

FIGURE 3.13 – Example of a composite component *ComBConfig* in SOFA 2.0

Connectors. There are two ways to define connectors in SOFA 2.0 : using the extensible connector definition or the implicit connector generation.

- *Implicit.* The connector instances are dynamically generated according to the bindings defined in the architecture descriptions. The generation is performed at deployment time, just before the application is prepared to be launched.
- *Extensible.* As in many component models, connector types can be defined as specialized component types. Thus, concrete connector types can be defined as primitive or composite ***architectures*** implementing connector frames.

Example of a configuration in SOFA 2.0. Figure 3.14 presents the configuration of the *ArchExample* system in the SOFA 2.0 syntax.

3.4.5.3 Discussion

Like Unicon and Darwin, SOFA 2.0 promotes the reuse of independently defined component types by the hierarchic construction of composite components described by nested architectures. SOFA 2.0 improves this scheme thanks to the effective separation of the component type definitions (the *frames*) from the definition of their implementations (the *architectures*). Like in Darwin, the component type definitions do not describe

```
frame ArchExampleType {
};

architecture ArchExample implements ArchExampleType {
  inst :CompA a;
  inst :CompB b;
  inst :CompC c;
  bind a :r to b :p;
  bind b :r to c :p;
};
```

FIGURE 3.14 – Frame and architecture description of the *ArchExample* in SOFA 2.0

architecture specifications. They are black-box specifications which do not describe the internal structures of components. Considering the development process of an architecture, the role of these architecture definitions is not clearly positioned. When an architecture description is partially or completely abstract, concrete definitions of the components must be provided by a deployment descriptor. These deployment descriptors contain then both configuration and assembly information, in a mixed way. Indeed, assembly description is more considered in SOFA as a runtime activity, which aims at defining deployment (installation) information, than a conceptual activity, which aims at predefining remarkable architecture instantiations.

3.4.6 Fractal ADL

The Fractal Architecture Description Language (ADL) [Leclercq07] is an open and extensible language to define component architectures for the Fractal component model, which is itself open and extensible. More precisely, the Fractal ADL is made of an open and extensible set of ADL modules, which define the abstract syntax of different architectural "aspects" (such as interfaces, bindings, attributes or containment relationships). Users are then free to define their own modules to add new aspects.

3.4.6.1 Type System

The basic concepts of Fractal ADL are component *definitions* (see Fig. 3.15 and Fig. 3.16. Depending on its content, a component definition may represent a component type, a concrete component type or even component assemblies. A component type definition contains only a set of interface declarations which provide a blackbox description of a component type. A concrete component type definition also contains an executable implementation. A component assembly definition contains a set of property value initializations. A component definition can "extend" another component definition, i.e. inherit and refine the elements of another component definition. This way, a component type definition can be refined into a concrete component definition and further refined into a parameterized assembly.

Another way of combining component definitions is the hierarchical construction of composite components. A composite component definition contains nested component definitions. These nested component definitions may be integrally embedded or may refer

to external reused component definitions. A composite component definition contains also a set of interface bindings which define the structure of the inner architecture of the composite component. In Fractal too, architectures are defined as the inner architectures of top composite components.

Contrary to a composite component, the content of a primitive component is not defined as a nested architecture. A primitive components contains directly an executable implementation, defined as a reference to an object class.

3.4.6.2 Architecture Configuration

In Fractal ADL, an architecture configuration is described as the inner architecture of a concrete composite component definition. It is defined by a set of components and as set of bindings between their interfaces, as listed in Table 3.12.

Architecture configuration	**Concept name**	**Characteristics**
Configuration	"definition"	A configuration is represented as a concrete composite component definition
Component	"component"	Nested component definition. May refer to an external component definition.
Connector	—	—
Connection	"binding"	connect server (provided) and client (required) interfaces.
Primitive component	**Concept name**	**Characteristics**
Primitive component	"definition"	no nested component definition
Implementation	"content class"	reference to an object class as an executable implementation
Composite component	**Concept name**	**Characteristics**
Composite component	"definition"	nested component definitions, using references to external component definitions
Delegation	"binding"	*bindings* between interfaces of the composite component and interfaces of its inner components.

TABLE 3.12 – Expressiveness of Fractal for architecture configurations

Example of a configuration in Fractal ADL. Figure 3.15 presents the configuration of the *ArchExample* system in the Fractal ADL syntax.

3.4.6.3 Discussion

Fractal ADL provide a very flexible component definition concept that enables to define components at three abstraction levels : component types, component class and component instantiations. But like Unicon and Darwin, it lacks a formal definition of architectures in three levels.

```
<definition name="ArchExample">
  <component name="a">
    <interface name="r" role="server"
      signature="Service"/>
    <interface name="p" role="client"
      signature="Service"/>
    <content class="CompAImpl"/>
  </component>
  <component name="b"
    definition="CompBConfig"/>
      %simple reference
  <component name="c"
    definition="CompC">
      %Inheritance reference
    <content class="CompCImpl"/>
  </component>
  <binding client="a.r" server="b.p"/>
  <binding client="b.r" server="c.p"/>
</definition>
```

FIGURE 3.15 – Configuration of the *ArchExample* architecture in FractalADL

```
<definition name="CompBConfig">
  <interface name="r" role="server"
    signature="Service"/>
  <interface name="p" role="client"
    signature="Service"/>
  <component name="b1"
    definition="CompB1">
    <content class="CompB1Impl"/>
  </component>
  <component name="b2"
    definition="CompB2">
    <content class="CompB2Impl"/>
  </component>
  <binding client="b1.p"
    server="this.p"/>
  <binding client="this.r"
    server="b2.r"/>
</definition>
```

FIGURE 3.16 – Configuration of the *CompB* composite component in FractalADL

3.4.7 xADL 2.0

xADL 2.0 [Dashofy01, Dashofy02a, Dashofy05] is an extensible XML-based architecture description language. It is an attempt to provide a platform upon which common modeling features can be reused from domain to domain and new features can be created and added to the language as first-class entities. Every xADL model is a well-formed and valid XML document. xADL 2.0 covers two architecture levels : architecture configuration and assembly, which are called the STRUCTURE and INSTANCE schemas respectively.

3.4.7.1 Type model

xADL 2.0 provide a first XML schema, named "xArch Type XML Schema", which contains a metamodel that enables to define architectural element types. These type definitions are regrouped in the *"archTypes"* element of an "xArch" XML descriptor. It contains the definitions of architectural element types including component types, connector types and interface types (see Fig. 3.17). Table 3.13 gives an overview of types modeled in xADL 2.0.

Component and connector types. A component or connector type is defined by a unique identifier, a textual description along with a set of signatures. *Signatures* are the prescribed interfaces of this type : components or connectors instances of the type must hold corresponding interfaces. Signatures contain references to the interface types that define their syntactical properties.

Composite component type. Composite component types are specified in the *archTypes* schema. A composite component definition consists of two parts : *signatures* and *subArchitecture*. The *subArchitecture* contains in turn :

Architecture specification	**Concept name**	**Characteristics**
Specification	"archType"	Partial specification. Contain only element type definitions
Component type	"componentType"	Primitive and composite component type
Connector type	"connectorType"	Connector type definitions are similar to component type definitions
Interface type	"interfaceType"	Used to define the syntactical type of interfaces
Primitive component type	**Concept name**	**Characteristics**
Interface	"signature"	Direction can be *in*, *out*, *inout*, and *none*.
Behavior	—	—
Composite component type	**Concept name**	**Characteristics**
Content	"subArchitecture"	Defined by an embedded architecture configuration (*archStructure*) and delegations (*signatureInterfaceMapping*).
Delegation	"signature-InterfaceMapping"	Mapping between the signatures (interfaces) of the enclosing component and the signatures of its inner components.

TABLE 3.13 – The type model of xADL 2.0

- An *archStructure* element : the internal architecture of a composite component type is defined as a regular architecture configuration,
- A set of *signatureInterfaceMapping* elements : these declarations map the signatures of the composite component type with signatures of the components defined in its internal architecture.

3.4.7.2 Architecture Configuration

The "xArch Type XML Schema" metamodel also enables the definition of architectures, described by instances of the defined architecture element types. However, these architecture descriptions are not intended to be deployable runtime architectures. They are design-time prescriptions for a type of architectures, though not explicitly identified as such, which must be thus considered as architecture configurations. These architecture configurations are defined by *archStructure* elements of the xArch XML descriptors. An architecture configuration contains a set of components, connectors and links declarations (see Fig. 3.17), as listed in Table 3.14.

Components. The components have a unique identifier and a short textual description, along with a set of interfaces. In xADL 2.0, an interface has unique identifier, a textual description, and a direction (an indicator of whether the interface is provided, required, or both). Both component and interface description can refer to a type, which specify the elements that the component or the interface must hold to be a valid instance of this type.

Architecture configuration	Concept name	Characteristics
Configuration	"archStructure"	
Component	"component"	refers to a component type definition
Connector	"connector"	refers to a connector type definition
Connection	"links"	connections between component and connector interfaces.

TABLE 3.14 – Expressiveness of xADL 2.0 for architecture configurations

Connector. They have only a unique identifier and a short textual description, along with a set of interfaces. A connector description can refer to a type, which specifies the elements that the connector must hold to be a valid instance of this type.

Connections (links). Connections are called links in xADL 2.0. *Links* connect the interfaces of components to the interfaces of connectors. They define the topology of the architecture.

3.4.7.3 Architecture Assembly

xADL 2.0 proposes another XML schema, named "xArch Instances XML Schema". This schema defines a metamodel that enables to describe runtime instances of architectures, in other words architecture assemblies (see Fig. 3.17). Architecture assemblies are defined by the archInstance elements of xArch XML descriptors.

An architecture assembly is defined by a set of component instances, connector instances, interface instances and link instances, as listed in Table 3.15. Each instance definition may contain a "structure" element that refers to the conceptual entity which it is an instance of. Nonetheless, every instance definition is standalone (like all the architectural element definition), i.e. it contains a full definition of the features of the instance.

Composite component or connector instances. The definition of a composite component or connector instance has two parts : a set of *interfaceInstances*, which defines its external representation (as a blackbox), and a sub-architecture, which defines its content. The *subArchitecture* definition contains in turn :

- an *archInstance* element : a nested full architecture assembly definition describing the internal architecture of the composite.
- a set of *interface* mappings : delegation links between the interfaces of the enclosing composite and the interfaces of the constituents of its internal architecture.

Example of an architecture configuration in xADL. Figure 3.17 presents the partial definition of the *ArchExample* architecture configuration in the xADL syntax. The presented descriptor combines the definition of the used types and the actual definition of the architecture configuration.

```
<xArch>
  <archStructure id="ArchExample">
    <description>example</description>
    <component id="CompA">
      <description>CompA</description>
      <interface id="CompA.r">
        <description>
          r Itf (out)
        </description>
        <direction>out</direction>
        <type href="#r_type"/>
      </interface>
    </component>
    ...
    <connector id="Connector1">
      <description>Connector1</description>
      <interface id="Connector1.r">
        <description>
          Connector1 r interface (out)
        </description>
        <direction>out</direction>
        <type href="#r_type"/>
      </interface>
      <interface id="Connector1.p">
        <description>
          Connector1 p interface (in)
        </description>
        <direction>in</direction>
        <type href="#p_type"/>
      </interface>
    </connector>
    ...
    <link id="link1">
      <description>
        ComponentA to Connector1
      </description>
      <point>
        <anchor href="#CompA.r"/>
      </point>
      <point>
        <anchor href="#Connector1.p"/>
      </point>
    </link>
    ...
  </archStructure>
  <archTypes>
    <componentType id="CompB_type">
      <description>
        Composite ComponentB Type
      </description>
      <signature id="CompB_type.r">
        <description>
          r Sig (out)</description>
        <direction>out</direction>
        <type href="#r_type"/>
      </signature>
      <signature id="CompB_type.p">
        <description>
          p Signature (in)
        </description>
        <direction>in</direction>
        <type href="#p_type"/>
      </signature>
      <subArchitecture>
        <archStructure
          href="#compB_Structure"/>
        <signatureInterfaceMapping id="r">
          <outerSignature
            href="#r_type"/>
          <innerInterface href="#r"/>
        </signatureInterfaceMapping>
        <signatureInterfaceMapping id="p">
          <outerSignature href="#p_type"/>
          <innerInterface href="#p"/>
        </signatureInterfaceMapping>
      </subArchitecture>
    </componentType>
    ...
    <connectorType id="Connector1_type">
      <description>
        Connector1 Type
      </description>
      <signature id="Connector1_type.p">
        <description>
          p Signature (in)
        </description>
        <direction>in</direction>
        <type href="#p_type"/>
        </signature>
      <signature id="Connector1_type.r">
        <description>
          r Signature (out)
        </description>
        <direction>out</direction>
        <type href="#r_type"/>
      </signature>
    </connectorType>
    ...
    <interfaceType id="r_type">
      <description>
        Change Channel Interface
      </description>
    </interfaceType>
  <archTypes>
</xArch>
```

FIGURE 3.17 – Example of a configuration *ArchExample* in xADL 2.0

3.4.7.4 Discussion

The metamodel proposed by xADL is rather classic. But its philosophy and modular design are much more original. The core of xADL is the metamodel proposed to define architecture assemblies. Thus, the first purpose of xADL is the description of the runtime architectures. Companion meta-models are provided as extensions that enable to model other aspects of architectures definitions. The architecture type metamodel is proposed to model design-time architecture configurations, including the definition of the component

Architecture assembly	Concept name	Characteristics
Assembly	"archInstance"	definition of a runtime architecture
Component instance	"componentInstance"	
Connector instance	"connectorInstance"	
Link instance	"linkInstance"	connection between component and connector instances
Interface instance	"interfaceInstance"	interface of a component or a connector instance
Primitive component instance	**Concept name**	**Characteristics**
Interface	"interfaceInstance"	Direction can be *in*, *out*, *inout*, and *none*.
Behavior	—	
Composite component instance	**Concept name**	**Characteristics**
Content	"subarchitecture"	defined by an embedded assembly
Delegation	"interface-InstanceMapping"	mapping between the interfaces of the composite component and the interfaces of its inner components

TABLE 3.15 – Expressiveness of xADL 2.0 for architecture assemblies

and connector types they use. Another metamodel extension can be used to support the definition of concrete component and connector types that contain explicit references to their executable implementations. Using this set of meta-models, xADL enable to model concrete architecture configurations and architecture assemblies. This is close to the style proposed by Wright. However, they fail to model the abstract architectures (specifications).

3.5 SYNTHESIS OF THE ADL STUDY

3.5.1 Views in Architecture Representations

When systems are too complex to be easily described, two "classical" mechanisms can be used to split descriptions into smaller ones :

- Hierarchical decomposition that enables to view systems at various granularities (*e.g.,* Darwin [Magee95, Magee96], SOFA 2.0 [Plásil98, Plasil02] or Fractal ADL [Bruneton06]). Using this principle, sub-system description can be further detailed at a more precise hierarchical level. A system description is the result of the composition of all its sub-system descriptions. A car, for example, would be described as the composition of an engine and a car body. The engine can then further be detailed.
- System description decomposition according to viewpoints (e.g. syntactic and behavioral diagrams of UML [Booch05]). Using this principle, several descriptions of the system coexist, each of which describes the whole system from a partial viewpoint. The description of the whole system can be seen as some kind of synthesis (juxtaposition) of all its partial descriptions. A car, for example, would be described as a whole but with an electrical or a mechanical viewpoint.

```
archInstance{
  componentInstance{
    (attr) id = "CompA"
    description = "CompA instance"
    interfaceInstance{
      (attr) id = "CompA.r"
      description = "r interface"
      direction = "out"
    }
  }
  ...
  connectorInstance{
    (attr) id = "Connector1"
    description = "Connector1 instance"
    interfaceInstance{
      (attr) id = "Connector1.r"
      description = "r interface"
      direction = "out"
    }
    interfaceInstance{
      (attr) id = "Connector1.p"
      description = "p interface"
      direction = "in"
    }
  }
  linkInstance{
    (attr) id = "link1"
    description =
      "Comp1 to Conn1 Link"
    point{
      (link) anchorOnInterface =
        "#CompA.r"
    }
    point{
      (link) anchorOnInterface =
        "#Connector1.p"
    }
  }
}
```

FIGURE 3.18 – Example of an architecture assembly *ArchExample* in xADL 2.0

Systems can also be described at various steps of their life-cycles. To our knowledge, no ADL really includes this "process" dimension. Some works such as UML [Booch05] or Taylor *et al.* [Taylor09] implement or describe close notions. UML allows to describe object-oriented software as multiple models corresponding to various abstraction levels, but this is not transposed in their more recent component-based models. Taylor *et al.* [Taylor09] defines two description levels for architectures at design and programming time. They respectively call them perspective (or as-intended) and descriptive (or as-realized) architectures.

3.5.2 Support of the Three Architecture Abstraction Levels

In this section, we will give a synthesis from the three architecture level views of the existing ADLs, specially the ones we presented in previous section. The overview of the support of the three architecture levels in these ADLs can be found in Tab. 3.16.

ADL	Specification	Configuration	Assembly
C2SADEL	✓	✓	×
Wright	×	✓	×
Darwin	×	✓	×
Unicon	×	✓	×
SOFA 2.0	×	✓	×
Fractal ADL	×	✓	×
xADL 2.0	×	✓	✓

TABLE 3.16 – Comparison of three architectural levels in ADLs

3.5.2.1 Architecture Specification

Most ADLs do not support a separated architecture specification in their architecture models, except C2SADEL. However, C2SADEL lacks a corresponding support for configuration representations.

Abstract component type. Abstract component types are used in architecture specifications to specify the required usages of components. They are not intended to define complete abstract component types. Most ADLs support the definition of abstract component types but they often lack mechanisms to refine abstract component types into concrete implemented component types. C2SADEL provides a subtyping mechanism, but the definitions of the component types remain abstract and do not refer to implementations.

Connection. Most of the ADL supply information about connections at the configuration level, except C2SADEL. The more abstract level defines generally only abstract component types. At this level, the composite component types are represented as blackboxes. The definitions of their internal architectures are considered as implementation details and thus defined as an architecture configuration. Only C2SADEL, which models only architecture specifications, supports the definition of connections at this abstraction level.

Architecture Behavior. Most works do not support any architecture behavior definition nor even support architecture behavior analysis. C2SADEL, Wright and SOFA 2.0 enable to define formally the behavior of component types and then to calculate the behavior of the architectures they compose. They also support examining the correctness of interface communications of these composed components [Bessam09]. Only SOFA 2.0 provides a means to define the expected behavior of an architecture, independently from the component types that compose it, in order to verify that the proposed architecture meets requirements. This kind of behavior protocol is called a frame protocol and is associated

to the blackbox definition of a composite component type. The nested architecture configurations that are proposed to implemented this component type must have a conform behavior [Hne06].

ADL	Specification	Abstract component type	Connections	Architecture behavior
C2SADEL	***architecture*** : full architecture topology description	***conceptual_component*** : references to external component type declarations	***architecture_topology*** : implicitly defined by the connector type descriptions	calculated from the behaviors defined in the component types
Wright	—	—	—	—
Darwin	—	—	—	—
Unicon	—	—	—	—
SOFA 2.0	—	—	—	—
Fractal ADL	—	—	—	—
xADL 2.0	—	—	—	—

TABLE 3.17 – Comparison of architecture specification in ADLs

3.5.2.2 Type systems

Component Type. ADLs support the definition of concrete component types. A component type is specified by the set of its interfaces. Some of ADLs support also the definition of behaviors, like C2SADEL, Wright and SOFA 2.0.

Connector type. ADLs support three kind of connector type definitions : (1) implicit, such as Darwin and SOFA 2.0, (2) explicit and predefined, such as C2SADEL and Unicon, and (3) explicit and customized, such xADL2.0 and Wright.

ADL	Component type	Interfaces	Component behavior	Connector type	Connector ends	Connector behavior
C2SADEL	***conceptual_component*** : Extensible, component type.	***top_domain, bottom_domain*** : each component have only these two interfaces (layered architecture style)	***behavior*** : Message-based protocols.	***connector*** : Built-in connector types.	***top_port, bottom_port*** : Generated according to connected components.	—
Wright	***Component*** : Extensible, concrete component type.	***Port*** : CSP-based behavior.	***Computation*** : CSP-based behavior.	***Connector*** : Extensible connector type.	***Role*** : CSP-based behavior.	***Glue*** : CSP-based behavior.
Darwin	***Component*** : Extensible, concrete component type.	***service*** or ***port***	—	Implicit connectors.	—	—
Unicon	***interface*** : Built-in, concrete component type.	***player*** : Built-in extensible types.	—	***protocol*** : Built-in extensible types.	***role*** : Built-in extensible extensible types.	—
SOFA 2.0	***frame*** : Extensible, concrete component type.	set of required and provided interfaces. Refer to external interface type definitions.	***protocol*** : Defined by behavior protocols	Implicit	—	—
Fractal ADL	No explicit component types. But component definitions reused and extended in other definitions		set of client and server interfaces. Signature defined as a reference to an implemented type	—	No connectors. Direct bindings of component interfaces.	—
xADL 2.0	***componentType*** : Extensible, concrete component type.	***signature*** : prescribe the definition of an interface of a given type in the instances of this component type. Refers to an external ***interfaceType***	—	***connectorType*** : Explicit connector type definitions. Similar component type definition.	***signature*** : refers to an external ***interfaceType***	—

TABLE 3.18 – Comparison of component and connector types in ADLs

3.5.2.3 Architecture Configuration

The purpose of most ADLs is to describe deployable architectures. Thus most ADLs provide means to define architecture configurations. At the configuration abstraction level, modeling issues relate to the concrete definition of the components : the internal architecture of composite component types (hierarchical composition), implementation constraints and parametrization attributes. Many ADLs do not support directly the definition of architectures but as the internal structures of composite components.

ADL	Configuration	Component class	Connector class
C2SADEL	***System*** : instance of an architecture specification.	No concrete type definitions. Component instances defined with types	topology defined in the architecture specification.
Wright	***Configuration*** : Independent architecture configuration. Full topology definition	***Component*** : Concrete component types. No references to implementations.	***Connector*** : Concrete connector types. No references to implementations.
Darwin	***component*** : Composite component internal definition. Full topology definition	***component*** : Concrete component types. No references to implementations.	—
Unicon	***component*** : Composite component internal definition. Full topology definition	***Component*** : Concrete component types, with multiple references to implementations (variants).	***Connector*** : Concrete connector types, with multiple references to implementations (variants)
SOFA 2.0	***architecture*** : Composite component internal definition. Full topology definition.	Concrete component types. Support references to implementations.	Implicit connectors.
Fractal ADL	***definition*** :Independent architecture configuration(definition containing a set of components). Full topology definition. Can be reused as a composite component	***definition*** : component definition (set of interfaces). Can be reused to model other components (as a class of component)	Implicit connectors.
xADL 2.0	***archStructure*** : Independent architecture configuration	***componentType*** : Concrete component type definition. Can refer to an implementation	***connectorType*** : Concrete connector type definition. Can refer to an implementation

TABLE 3.19 – Comparison of architecture configuration models in ADLs

Component classes. Component classes are most of the time described as types that provide a complete definition of the functionalities and of the external observable behaviors and an implemented component type. We call these types concrete component types. Component classes definitions often contain references to their actual executable implementations (generally object classes). The concrete component types of component classes

describe them as a blackboxes. Composite component classes must provide definitions of their implementations as inner architecture configurations.

Implementation. Implementation information is often ignored by ADLs, as they first intend to be conceptual design time languages. However, support for implementation information is mandatory to use architecture descriptions as means to manage architecture deployment. Unicon, SOFA, Fractal ADL and xADL enable to specify the implementation of component classes. Unicon is the more advanced, as the various alternative implementations of a given component type can be modeled.

Attributes. Few models support the definition of attributes in component type descriptions. These attributes correspond to the definition of externally accessible properties, provided to customize component instances. FractalADL and xADL 2.0, as XML-based ADLs, support the definition of extensible sets of attributes.

ADL	Primitive component	Implementation	Attributes	Composite component	Delegation
C2SADEL	Independent definition	—	—	—	—
Wright	Embedded in architecture configuration	—	—	Implicit composition : embedded architecture configuration	Specific bindings
Darwin	Independent definition	—	—	Implicit composition : embedded architecture configuration	Regular bindings
Unicon	Independent definition	References to several implementations (variants)	—	Implicit composition : embedded architecture configuration	specific bindings
SOFA 2.0	Independent definition	Content definition consisting of a reference to an executable implementation	—	Implicit composition : content defined by an embedded architecture	***Delegate*** and ***subsume*** bindings.
Fractal ADL	Independent refined definition	Reference to a Java class	Explicitly defined as interface features	Implicit composition : embedded architecture configuration	Regular bindings.
xADL 2.0	Independent refined definition	Reference to an executable implementation	Defined as features by metamodel extensions	Implicit composition :embedded architecture configuration	Definition of interface mappings

TABLE 3.20 – Comparison of primitive and composite component models in ADLs

3.5.2.4 Architecture Assembly

The assembly (runtime) description of software architectures is an important feature to support architecture deployment and runtime dynamic evolution. Architecture assemblies can be modeled as explicit architecture representations or managed as architecture configuration instantiations. Most dynamic ADLs support the definition of architecture assemblies, such as C2SADEL, SOFA 2.0, FractalADL or xADL 2.0. Without explicit architecture assemblies, the mapping between the design-time and the runtime architecture representations are much more difficult to manage. Specific tool support are then required to manage the relations between assemblies and configurations (for instance to check that an assembly is coherent with a configuration). These relations are then difficult to preserve and update. This often leads to erosion and drift between these architecture representations. Furthermore, the configuration cannot easily represent all the facets of the many different runtime architectures, such as specific parameterization of component attributes.

ADL	Assembly	Component instance	Parameterization
C2SADEL	Instance of a configuration	—	—
Wright	—	—	—
Darwin	—	—	—
Unicon	—	—	—
Sofa 2.0	—	—.	—
Fractal ADL	Independent definition	Independent definition. Can refine a base definition used as a type	Definable attributes
xADL 2.0	Independent definition	Independent definition. Can contain a reference to a component type	—

TABLE 3.21 – Comparison of assembly models in ADLs

3.5.3 Synthesis of Connector Models

Representative connectors models are proposed by Unicon [Shaw96a], SOFA [Bálek01], Wright [Allen97b] and COSA [Oussalah04]. Beyond the basic binding function of connectors, each of these models has its specificity. Unicon has a predefined taxonomy of connectors, either simple or complex (pipes, remote calls, etc.). SOFA has a composite connector model that enable to define connector types which hold inner architectures. Wright models the behavior of connectors (their glue) as event trace specifications (CSP). COSA eases the building of complex connectors by the composition of simpler connectors.

As connectors separate concerns of component computation and inter-component communication, they can be designed to meet non-functional requirements for architectures, such as adding security (encrypting messages) or life-cycle related services (starting, stopping or suspending components). Our work, for instance, proposes a connector model which provide support for architecture dynamic evolution.

3.5.4 Conclusion

This analysis of state-of-the-art ADLs focused on some qualities that we consider to be necessary for building a new ADL. This results in requirements for the language proposed in this thesis :

1. Components in architecture configurations should not be strictly identical to component types described in their architecture specification. As components come from some repository, the specification should define abstract (ideal) component types while configurations describe concrete (satisfying) components that are going to be used (as claimed by Taylor *et al.* [Taylor09]).
2. Connectors should be explicit in architecture specifications.
3. ADLs should include some description on how components classes are implemented and what are the characteristics of the running assemblies (constraints on component state values).
4. Components should possibly be primitive (implemented by an implementation class) or hierarchically composed of components (implemented by a configuration).
5. Component types should be reusable (described outside architecture descriptions).
6. Both structural and behavioral viewpoints should be provided for (abstract or concrete) components.

Moreover, most existing ADLs are not tailored to component-based software development. Switching to such a reuse centered development life-cycle as previously claimed in Chapter 1 and illustrated in Fig. 2.2 shall impact the description language. Figure 3.19 illustrates our vision of such a development life-cycle, which is comparable to those proposed by Crnkovic, Chaudron *et al.* [Crnkovic06, Chaudron08]. This illustration focuses on the produced artifacts for each development step. For simplicity's sake the proposed life-cycle is "reuse-centered" and thus does not describe how components should be :

- Adapted if no existing component perfectly matches specifications,
- developed from scratch if no component is found that matches or closely matches specification,
- Tested and integrated,
- Physically deployed.

In this development process, applications are mainly produced by reusing existing components, stored and indexed in a component repository. After a classical requirement analysis step, architects design architecture specifications. They define which functionalities should be supplied by components, which interfaces should be exported by components, and how interfaces connect to build a software system they meets the requirements. In a second step, architects create architecture configurations that define the sets of component implementations (classes) by searching and selecting from the component repository. Abstract component types from the architecture specification then become concrete component types in architecture configurations. In a third step, configurations are instantiated into component instance assemblies and deployed to executable software applications.

One of the claims developed in this thesis is that some software engineering production (architecture description) should correspond to each of the three steps of the component-based software development life-cycle. In other words, architectures should be described

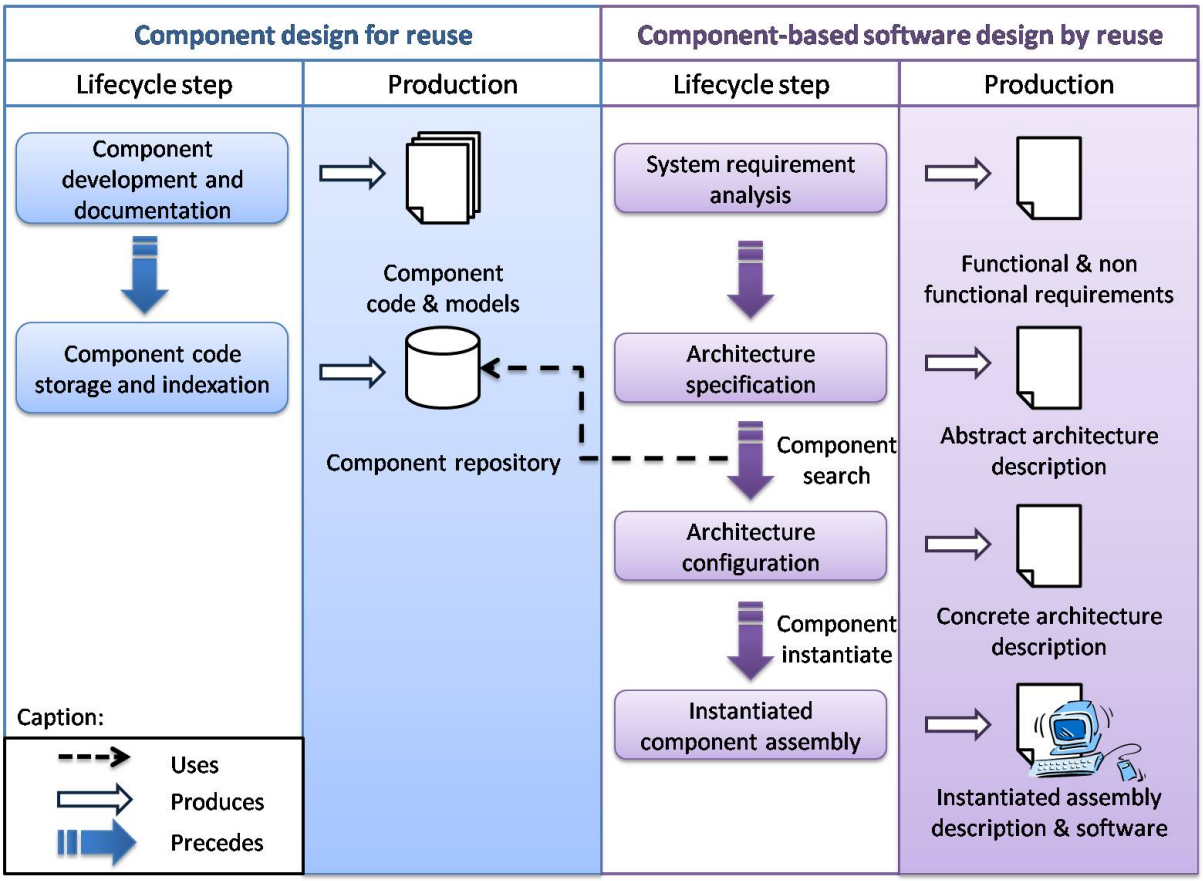


FIGURE 3.19 – Component-based software development process

from a specification, configuration and assembly point of views. These three descriptions should reflect the design decisions of the architect at each step of the development cycle and be expressed using an adequate ADL.

Next chapter presents Dedal, the ADL proposed in this thesis to fulfill these objectives.

Chapitre 4

Dedal Model

In Chapter 3, we gave a definition of software architectures that spans three abstraction levels : *abstract architecture specification*, *concrete architecture configuration* and *instantiated component assembly*. These three abstraction levels correspond to the different steps of the component-based software life-cycle presented in Chapter 2, from design down to execution through configuration. They precisely capture design decisions throughout software life-cycle and thus enable to manage advanced evolution processes.

After studying state-of-the-art ADLs, we found that there is a need for an ADL that (1) clearly and separately models three abstraction levels of architectures, and (2) is suitable for component-based software development.

We propose such an ADL, called Dedal, which supports these different architecture representations. Its syntax and usage is presented in this chapter, level by level using a XML BNF (Backus-Naur form) [1] definition. Syntax is explained by using our illustrative example. At the end of each section, we present the special characteristics of each level and their roles to support evolution.

Contents

1. In XML BNF, " : :=" means "is defined as", "|" means "or". It uses regular expressions to define options and repetitions, "a ?" means optional item, "a*" or "a+" means repetitive item. In order to simplify, all literals, including single-character literals are bold (*e.g.,*'**(**' or '**,**'). Spacing and indentation are used solely for readability.

4.1 OVERVIEW

Dedal is a three-level ADL aimed at modeling the component-based software systems. The two objectives of Dedal are to enable the expression of architecture design and to help support architecture evolution. Firstly, it enables the description of abstract architecture specifications, concrete architecture configurations and instantiated component assemblies. Then it supports a controlled process that helps build, test and record architecture evolution. The evolution of architectures described in Dedal will be presented in Chapter 6.

Dedal models architectures at three separate abstraction levels, which correspond to the main stages of component-based development lifecycle. Each architecture definition on an abstraction level consists of components and connectors in different forms. An overview of their relationships is shown in Fig. 4.1. Each abstraction level tries to capture and reflect the design decisions at a given stage (design, development/construction and deployment/execution).

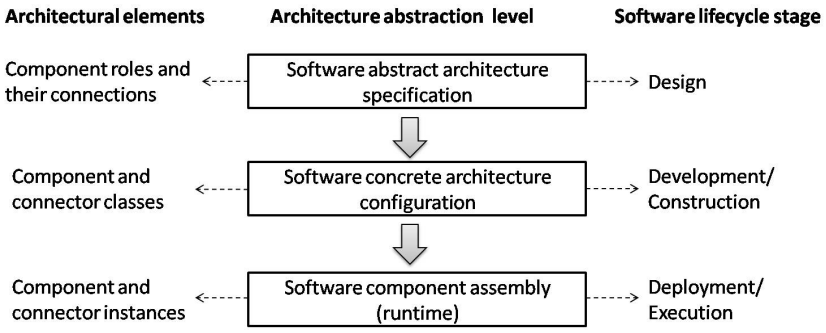


FIGURE 4.1 – Three architecture description levels

4.2 ABSTRACT ARCHITECTURE SPECIFICATIONS (AAS)

Abstract architecture specifications (AAS) are the first level of software architecture descriptions. They provide a formal and generic definition of the global structure and behavior of software systems, which are designed to achieve previously identified functional requirements. They are defined during the design step of software and serve as a basis to derive concrete architecture configurations.

Architecture specifications are abstract : they do not identify concrete software components (classes) that are going to be deployed and executed in some component-based environment. On the contrary, abstract architecture specifications only describe "ideal" component types from the application viewpoint (users' requirements). Abstract architecture specifications correspond to perspective architectures as denoted by Taylor *et*

al. [Taylor09]. Abstract architecture specifications capture the intention of the system, without being limited by the effective available components.

In Dedal, an abstract architecture specification is composed of a set of component roles, a set of connections and its architecture behaviors. The architecture version number and versioning information are also part of the abstract architecture specification description. This aspect of the ADL syntax is presented in Chapter 6, relative to evolution. [2] Its syntax is described in Fig. 4.2. Figures 4.3 and 4.4 provide a graphical view and a text description of the abstract architecture specification for the BRS example.

```
1  specification : :=
2   specification specification_name
3    component_roles component_role_list
4    ( connections connection_list ) ?
5    architecture_behavior architecture_behavior
6    versionID revision_numb
7    ( pre_version pre_version ) ?
8    ( by change ) ?
9  component_role_list : :=
10   component_role_name ( ; component_role_name )*
11 connection_list : :=
12   connection ( connection )*
```

FIGURE 4.2 – Syntax of abstract architecture specification

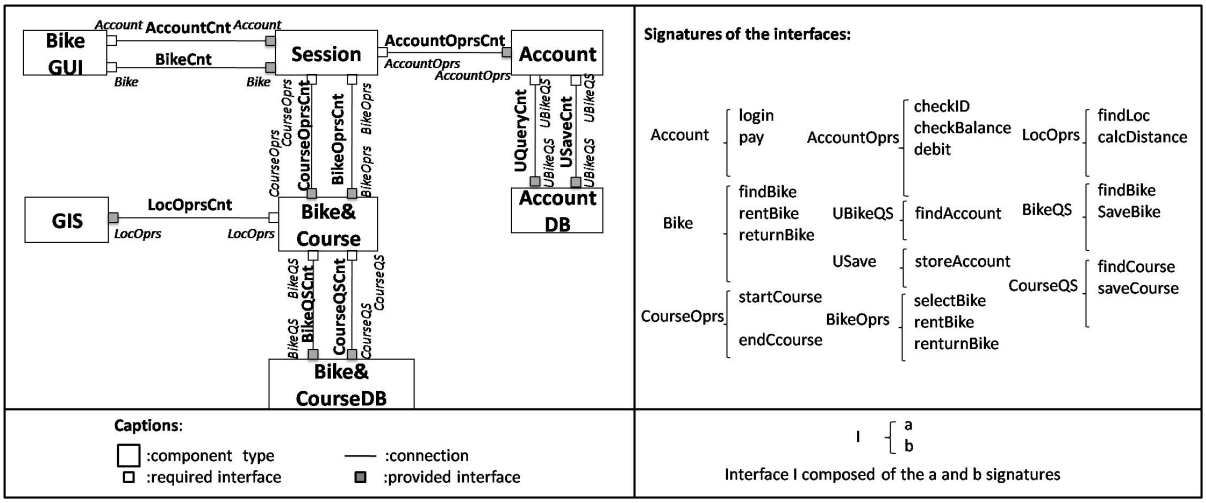


FIGURE 4.3 – Abstract architecture specification graphical view for the BRS

4.2.1 Component Roles

Component roles model components in an abstract way by describing the roles components are expected to play in the system, instead of the direct definition of the component

2. The version information of all the architecture elements, such as abstract architecture specifications or component roles, will be presented in Chapter 6, relative to evolution.

```
specification BRSSpec
  component_roles
    BikeCourse ; BikeCourseDB
  connections
    connection BQueryCnt
      client BikeCourse.BQuery
      server BikeCourseDB.BQuery
    connection BSaveCnt
      client BikeCourse.BSave
      server BikeCourseDB.BSave
  architecture_behavior
     ( !BikeCourse.BikeOprs.selectBike ; ?BikeCourse.BikeQS.findBike ;
!BikeCourseDB.BikeQS.findBike ; )
     +
     ( !BikeCourse.CourseOprs.startC ; ?BikeCourse.CourseQS.findCourse ;
!BikeCourseDB.CourseQS.findCourse ; )
  versionID 1.0
```

FIGURE 4.4 – Abstract architecture specification description of the BRS

classes used in the system. Roles represent the functions that abstract components must supply in a given execution context, *i.e.,* a specific usage of a component. Component roles can be considered as definitions of abstract component types. But we call them roles because they also correspond to the identification of the different component instances which must be created when the architecture is deployed. A component role consists of a set of *interfaces* and its expected component behavior (called *role_behavior in Dedal*). The syntax can be found in Fig. 4.5.

```
component_role : :=
  component_role component_role_name
    ( required_interfaces interface_list ) ?
    ( provided_interfaces interface_list ) ?
    ( role_behavior component_behavior ) ?
    ( MinInstanceNbr PositiveInteger ) ?
    ( MaxInstanceNbr PositiveInteger ) ?

interface_list : :=
   interface_name ( ; interface_name )*
```

FIGURE 4.5 – Syntax of component role

- *Interface.* The *interfaces* are the connection points that the component should expose. They can be *provided* or *required*. An interface is composed by its name, direction (provided or required) and its implementation class, as shown in Fig. 4.6.
- *Role behavior.* A *role behavior* is the protocol that describes the expected behavior of a component in an architecture (the behavior protocol is often referred to as the dynamics of the architectures). Dedal uses the protocol syntax of SOFA [Plasil02] to describe component role behavior as regular expressions [3]. Other formalisms could

3. `!i.m` (*resp.* `?i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i`. `A+B` is for `A` or `B` (exclusive or) and `A;B` for `B` after `A` (sequencing).

```
interface : :=
  interface interface_name
    interface_direction interface_direction
    implementation implementation_class
interface_direction : := provided | required
```

FIGURE 4.6 – Syntax of interface

have been used instead ; the notation chosen is interesting as it is compact and is implemented as an extension of the Fractal component model we use for our experimentations (see Chapter. 7), with companion verification tools. Component protocols capture the behavior of components describing all valid sequences of emitted function calls (emitted by the component and addressed to neighbor components) and received function calls (received by the component from neighbor components).

– *Cardinality.* The precise cardinality of component instances are described in component role descriptions using **minInstances** and **maxInstances**. They define the minimum and maximum numbers of component instances that are to be instantiated from the component class which implements this component role. For example, the *BikeGUI* component role has a maximum number of component instances *15*, as shown in Fig. 4.7.

Dedal chooses to describe component roles outside abstract architecture specifications, so as they can be reused from a specification to another (this would not be possible if they were embedded). Figure 4.7 shows the descriptions of the *BikeCourse* and *BikeCourseDB* component roles. They contain the SOFA-like descriptions of their behavior.

4.2.2 Connections

Connections describe interactions between two components by defining which component interfaces are bound together. The connection links a required interface from a component role to a provided interface from another. These interfaces are declared using the **client** and **server** keywords, respectively (as shown in Fig. 4.8). In Dedal, components must be connected through connectors (except for delegation as described in Section 4.3.1.3). Moreover two connectors cannot connect directly with one another. *BQ* and *connection2* in Fig. 4.4 are examples of such connections.

4.2.3 Architecture Behaviors

Architecture behaviors describe the protocol of the entire architecture – meaning all required interactions between its constituent components. As for component protocols, the syntax used is that of SOFA protocols [4]. Compatibility between individual component protocols and the protocol of their containing architecture as well as compatibility between the protocols of two connected components is not studied in this work, because we interface our language with corresponding mechanisms (trace inclusion) from SOFA [Plasil02]. Figure 4.4, that describes the BRS specification, contains the BRS architecture protocol.

4. `!c.i.m` (*resp.* `?c.i.m`) denotes an outgoing (*resp.* incoming) call of method `m` on interface `i` of component `c`.

```
component_role BikeCourse
  required_interfaces BikeQS ; CourseQS
  provided_interfaces BikeOprs ; CourseOprs
  component_behavior
    ( !BikeCourse.BikeOprs.selectBike,
?BikeCourse.BikeQS.findBike ;)
    +
    ( !BikeCourse.CourseOprs.startC,
?BikeCourse.CourseQS.findCourse ;)
    MaxInstanceNbr 3

component_role BikeCourseDB
  provided_interfaces BikeQS ; CourseQS
  component_behavior
!BikeCourseDB.BikeQS.findBike ;
    +
!BikeCourseDB.BikeOprs.findCourse ;

component_role BikeGUI
  required_interfaces Account ; Bike
  component_behavior
?BikeGUI.Account.login ;
...    MaxInstanceNbr 15
```

FIGURE 4.7 – Component role descriptions of *BikeCourse* and *BikeCourseDB*

```
connection : :=
  connection connection_name
    client component_role_name . interface_name
    server component_role_name . interface_name
```

FIGURE 4.8 – Syntax of connection

The reader can intuitively see that the protocol of the *BikeCourse* component role (see Fig. 4.7) is compatible with ("included" in) the protocol of the BRS architecture.

However, in our model, the architecture behavior is not automatically generated according its sub components, not like Wright and SOFA 2.0, which generate the architecture behavior automatically. Manually defining architecture behavior avoids confusions compared with dynamically generating architecture behavior. For example, if a behavior trace is not used in this architecture, it can nonetheless be supported by behaviors of sub-components possessed by the specification.

4.2.4 Analysis : Consistency of Abstract Specification

The inner-analysis of specification level is to check the consistencies of abstract specification to ensure that different elements of specification do not contradict one another. The consistencies of specification concerns three types : name consistency, connection consistency and behavior consistency. Each of them will be checked after designs or changes of a specification in order to preserve the consistencies of the entire specification. The rules of inconsistency checking are presented as follows :

Name consistency. Name consistency checking respects one rule : Two different architectural elements (including the component role and the connection) cannot have the same name.

Connection consistency. Connection consistency checks whether the connection description is correct. Connection consistency checking conforms with three rules as follows.

1. Two connected component roles in a connection should exist in and be contained by the specification.
2. The two connected interfaces should be possessed by two connected component roles defined in the connection.
3. The two connected interfaces should have the same signature.
4. The two connected interfaces should have different interface directions : one required and the other provided.

Behavior consistency. The behavior consistency checking examines whether component role behaviors conform to the architecture behavior. The rule is : The architecture behavior should be supported by component role behaviors, that is the part of architecture behavior concerning this component role should be contained by this component role behavior.

4.2.5 Summary

The abstract architecture specification of a software provides a formal and generic definition of its global structure and behavior. It is specified during the design step of the software and then used during the later implementation and execution steps to control evolution (see Fig. 4.1). In our approach, the many different existing versions of a software are described as concrete architecture configurations, as presented in the next section.

4.3 CONCRETE ARCHITECTURE CONFIGURATIONS (CAC)

Concrete architecture configurations (CAC) are the second level of system architecture descriptions. They result from the choice of real component types and implementations (component classes) from a component repository. This choice is the result of a search and selection process. These component classes must match abstract component descriptions from the abstract architecture specification but need not be identical ; compatibility is sufficient. Concrete architecture configurations describe the architecture from an implementation viewpoint (by assigning existing component classes to component roles). Concrete architecture configurations correspond to descriptive architectures as denoted by Taylor *et al.* [Taylor09].

Concrete architecture configurations list the concrete component and connector classes. The architecture of a given software is thus defined by a unique abstract architecture specification and possibly several concrete architecture configurations. Each architecture configuration of the software application must conform to its abstract architecture specification. This means that every component or connector class used in a concrete architecture configuration must be a legal implementation of the corresponding component role or connection in the abstract architecture specification. The configuration version number and versioning information are also part of the concrete architecture configuration description. This aspect of the ADL syntax is further presented in Chapter 6, along with concepts

related to evolution. The syntax of connector type is described in Fig. 4.9. An excerpt of the BRS configuration derived from the abstract architecture specification presented on Fig. 4.3 and 4.4 is described in Fig. 4.10 and Fig. 4.11.

```
configuration : :=
  configuration configuration_name
    implements specification_identifier
    component_classes component_class_list
    ( connector_classes connector_class_list ) ?
    versionID revision_numb
    ( pre_version pre_version ) ?
    ( by change ) ?
specification_identifier : :=
  specification_name ( revision_numb )
component_class_list : :=
  component_class_name ( revision_numb ) as component_role_name
  ( ; component_class_name ( revision_numb ) as component_role_name )*
connector_class_list : :=
  connector_class_name as connection_name
  ( ; connector_class_name as connection_name )*
```

FIGURE 4.9 – Syntax of an architecture configuration

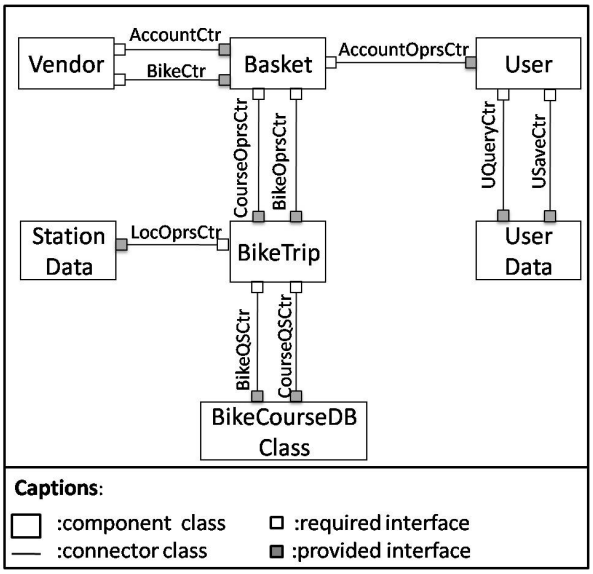


FIGURE 4.10 – Concrete architecture configuration graphical view of BRS

```
configuration BRSConfig
  implements BRSSpec (1.0)
  component_classes
    BikeTrip (1.0) as BikeCourse;
    BikeCourseDBClass (1.0) as
      BikeCourseDB
  versionID 1.0
```

FIGURE 4.11 – Concrete architecture configuration description of BRS

4.3.1 Component Classes

The component classes used in a concrete architecture configuration are listed. Each component class name is associated to the name of the component role the component class implements. Component classes can either be primitive or composite. Each component class

is associated with a component type that the component class supports. This component type is the concrete component type, which describes the interfaces and behaviors of this component class.

4.3.1.1 Component Types

Component types are defined by describing the interfaces and behavior of a set of component classes. Its syntax is described in Fig. 4.12. The component types are reusable blocks which can be implemented by multiple component classes which possess the same interfaces and component behavior. Dedal provides an extensible type system (including component types and connector types). The *BikeTripType* component type description of Fig. 4.13 is an example of component type description.

```
component_type : :=
  component_type component_type_name
    provided_interfaces interface_list
    required_interfaces interface_list
    component_behavior component_behavior
```

FIGURE 4.12 – Syntax of a component type

```
component_type BikeTripType
  required_interfaces
    BikeQS ; CourseQS ; LocOprs
  provided_interfaces
    BikeOprs ; CourseOprs
  component_behavior
    ( ?BikeTrip.BikeOprs.selectBike ;
 ?BikeTrip.LocOprs.findStation ;
 !BikeTrip.BikeQS.findBike ; )
    +
    ( ?BikeTrip.CourseOprs.startC,
 !BikeTrip.CourseQS.findCourse ; )
```

FIGURE 4.13 – Description of the component type *BikeTripType*

4.3.1.2 Primitive Component Classes

Primitive component classes are defined by describing its component type, version and implementation class. Its syntax is described in Fig. 4.14. Component class versions are documented by their revision numbers, their previous versions' revision numbers and by the motivations of the changes that entail their derivation from their previous versions. Motivations can either be *corrective* if the evolution aims at fixing some bug or *perfective* if the evolution aims at increasing the performance of the component [5]. An implementation class is also mentioned for each primitive component. Existing models usually do not include this information as primitive components of same type are considered having a

5. Motivations are used for gradual component version substitution as described in [Zhang08, Zhang09].

single possible implementation. In Dedal, mentioning implementation class enables to have identical component type with distinct implementation classes. The *BikeTrip* component class description of Fig. 4.15 is an example of primitive component class description.

```
primitive_component_class : :=
  primitive_component_class component_class_name
    implements component_type_name
    content implementation_class
    ( attributes attribute_list )?
    versionID revision_numb
    ( pre_version pre_version )?
    ( motivation motivation )?
    ( condition condition )?

attribute_list : :=
  attribute ( ; attribute )*

attribute : :=
  type attribute_name
```

FIGURE 4.14 – Syntax of primitive component class

```
component_class BikeTrip
  implements BikeTripType
  content fr.ema.BikeTripImpl
  attributes string company ; string currency
  versionID 1.0
```

FIGURE 4.15 – Description of the primitive component class *BikeTrip*

4.3.1.3 Composite Component Classes

Composite component classes differ from primitive components in that their implementation is not defined by a single class but by an embedded architecture configuration, *i.e.,* a set of connected inner components. The composite component class definition further defines how the interfaces of the composite component are mapped to corresponding unconnected interfaces of its inner components thanks to delegation declarations. As for simple provided interfaces and required interfaces in composite components, delegated interfaces are implementations of the corresponding provided and required interfaces in the corresponding component role. The explicit delegation declarations can be found in almost all the hierarchical ADL models, such as Darwin [Shaw95b], Unicon [Magee96], and SOFA2.0 [Bures06]. Its syntax can be found in Fig 4.16. Figure 4.18 gives an example of the composite component class *BikeCourseDBClass* where the *BikeQS* provided interface of the *BikeData* component inside the *BikeCourseDBConfig* configuration is delegated as a provided interface of the composite component that implements the *BikeQS* interface of the *BikeCourseDB* component role. Figure 4.17 shows a graphical representation of the same *BikeCourseDBClass* component.

```
composite_component_class : :=
   composite_component_class component_class_name
    implements component_type_name
    content configuration_identifier
    delegated_interfaces delegated_interface_list
    ( attributes attribute_list ) ?
    versionID revision_numb
    ( pre_version pre_version )
    ( motivation motivation ) ?
    ( condition condition ) ?

configuration_identifier : :=
 configuration_name ( revision_numb )

delegated_interface_list : :=
 provided | required inner_interface as outer_interface
 ( ; provided | required inner_interface as outer_interface )*

inner_interface : :=
 component_class_identifier [ component_role_name ] . interface_name

outer_interface : :=
 component_type_name . interface_name

component_class_identifier : :=
 component_class_name ( revision_numb )
```

FIGURE 4.16 – Syntax of the composite component class

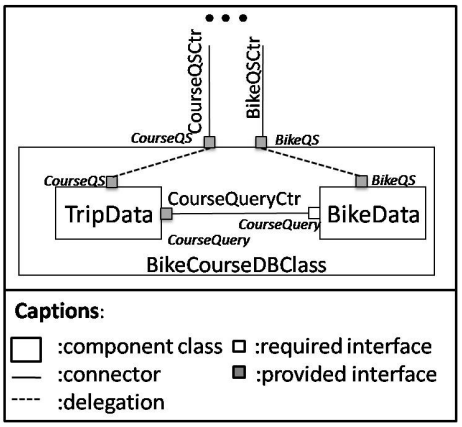


FIGURE 4.17 – Graphic view of the *BikeCourseDBClass* composite component class and inner configuration

4.3.1.4 Attributes

Both primitive and composite component classes can have an attribute list. Attributes are not mandatory but can be declared as visible properties of component classes. They serve two functions. Firstly, they enable to express the initiation state of component instance (attribute with its instantiated value). Secondly, they enable to add constraints on attribute values of component instances in the instantiated component assembly level.

```
component_class BikeCourseDBClass
  implements BikeCourseDBType
  using BikeCourseDBConfig (1.0)
  delegated_interfaces
    provided BikeData(1.0)[BikeDB].BikeQS as BikeCourseDBType.BikeQS;
    provided TripData(1.0)[CourseDB].CourseQS as
BikeCourseDBType.CourseQS
  attributes string company
  versionID 1.0

configuration BikeCourseDBConfig
  implements BikeCourseDBSpec (1.0)
  component_classes
    BikeData (1.0) as BikeDB;
    TripData (1.0) as CourseDB
  connector_classes
    CourseQueryCtr as CourseQueryCnt;
  versionID 1.0

specification BikeCourseDBSpec
  component_roles
    BikeDB; CourseDB
  connections
    connection CourseQueryCnt;
      client BikeDB.CourseQuery
      server CourseDB.CourseQuery
  versionID 1.0
```

FIGURE 4.18 – Description of the *BikeCourseDBClass* composite component class and inner configuration

4.3.2 Connector Classes

Connector classes can be of two types in Dedal : explicit as extensible type systems, defined in terms of interaction protocols (behavior protocols), or implicit as built-in based on basic communication implementation mechanisms.

4.3.2.1 Connector Types

Connector types are defined by describing the interfaces and interaction protocols of a set of connector classes. Its syntax and example are described in Fig. 4.19 and 4.20. Connector types are applicable for explicit connector classes solely.

```
connector_type : :=
  connector_type connector_type_name
    provided_interfaces interface_list
    required_interfaces interface_list
    connector_protocol connector_protocol
```

FIGURE 4.19 – Syntax of the connector type

```
connector_type BQueryCtrType
  provided_interfaces BQueryP
  required_interfaces BQueryR
  connector_protocol !BQueryP ; ?BQueryR ;
```

FIGURE 4.20 – Example of the connector type

4.3.2.2 Implicit Connector Classes

Connector classes are often implicit in concrete architecture configurations as exemplified on Fig. 4.18. In cases where they are implicit, we consider them as generic entities which are created in deployment time and provided by containers (execution environments) in which configurations are deployed.

4.3.2.3 Explicit Connector Classes

The explicit description of connectors is possible but not mandatory to define configurations. The syntax and example for explicit connectors are described in Fig. 4.21 and 4.22. In cases where connectors are explicitly added, their descriptions define the specific connector classes that reflect design choices made to manage special communication, coordination, and mediation schemes in specific connections between components.

```
connector_class : :=
  connector_class connector_class_name
    implements connector_type_name
    content implementation_class
```

FIGURE 4.21 – Syntax of the connector class

```
connector_class BQueryCtr
  implements BQueryCtrType
  content ArchExmaple.BQueryCtr
```

FIGURE 4.22 – Example of the connector class

4.3.3 Conformance Between Abstract Specification and Concrete Configuration.

Conformance between an abstract architecture specification and a concrete architecture configuration is a matter of (1) conformance between component roles and the component classes that supposedly implement them, (2) conformance between connections and the explicit connector classes, and cardinality conformance.

Component conformance. Many conformance relations could be defined, from stricter to very loose ones [Medvidovic98]. On the one hand, we defend that the type of the reused

component classes in a configuration need not be exactly identical to the corresponding component role in a specification because being too strict in this matter might seriously decrease the number of reuse opportunities. On the other hand, it is expected from a conformance relation that the completeness of abstract architecture specification is preserved in this derived configuration. Rule of thumb, component classes must provide at least what is specified and require less than what is required by the specification. This can be translated into rules for interfaces and rules for behavior protocols :

1. The provided interface list of the concrete component class must contain all the provided interfaces specified in the component role.
2. All the required interfaces of the concrete component class must contain all the required interfaces specified in the component role.
3. The behavior of a component class includes (in the sense of trace inclusions) the behavior specified in the component role.

Connector conformance. The explicit connector class possesses at least two interfaces to connect with the interfaces indicated in the implemented connection specification. The two interfaces should possess the same signature with its connected interfaces and have an opposite direction.

Variations on these rules can further consider interface specialization rules as in [Arévalo09]. Figure 4.15 shows an example of a component class (*BikeTrip*) conforms to the three rules described above and which has a required interface (*LocOprs*) not in the specification (*BikeCourse* component role).

In the case of composite components, delegated interfaces of provided (*resp.* required) direction are considered exactly as if they were provided (*resp.* required) interface of primitive components. Figure 4.18 provides an example of the *BikeCourseDBClass* composite component class, that conforms to the specification of the *BikeCourseDB* component role.

As all the three rules concern interface and behavior conformances between the component role and the component class, this conformance can be considered as the conformance between the component role and the component type which this class is implemented. Thus, the conformance is checked between component role and component type.

4.4 INSTANTIATED COMPONENT ASSEMBLIES (ICA)

Instantiated software component assemblies (ICA) are the third level of system architecture descriptions. They result from the instantiation of the component classes from a configuration. They provide a description of runtime software systems and gather information on identity and state-dependent design decisions about the component instances of the assemblies [Shaw96b]

Instantiated software component assemblies list the component and connector instances that compose runtime software system descriptions, their attributes and the assembly constraints the component instances are constrained by. Figure 4.24 gives the description of a software assembly that instantiates the BRS architecture configuration of Fig. 4.10 and Fig. 4.11.

```
assembly : :=
  assembly assembly_name
    instance_of configuration_identifier
    component_instances component_instance_list
    ( assembly_constraints assembly_constraint_list )?
    versionID revision_numb
    ( pre_version version )?
    ( by change )?

component_instance_list : :=
  component_instance_name as component_role_name
  (; component_instance_name as component_role_name )*

assembly_constraint_list : :=
  assembly_constraint (; assembly_constraint )*
```

FIGURE 4.23 – Syntax of component assembly

```
assembly BRSAss
  instance_of BRSConfig (1.0)
  component_instances
    BikeTripC1 as BikeCourse; BikeCourseDBClassC1 as BikeCourseDB
  assembly_constraints
    BikeTripC1.currency=="Euro.";
    BikeCourseDBClassC1.company==BikeTripC1.company;
    MaxInstanceNbr(BikeGUI)==8 && MinInstanceNbr(BikeGUI)==5
  versionID 1.0
```

FIGURE 4.24 – Component assembly description of the BRS

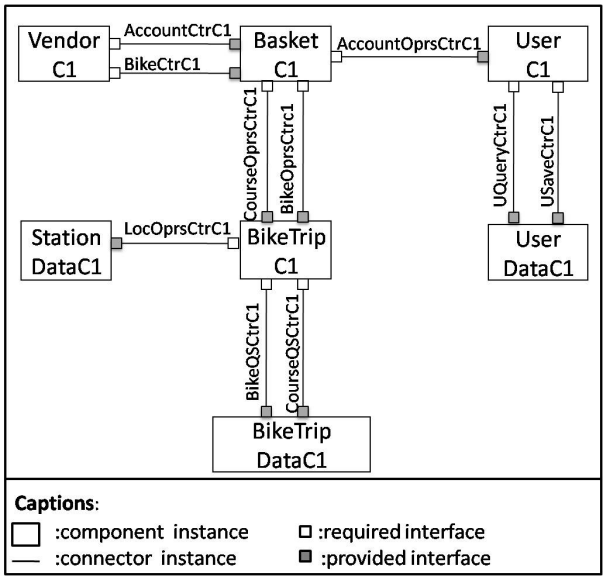


FIGURE 4.25 – Component assembly graphic view of the BRS

4.4.1 Component Instances

Component instances document the instances of components that are connected together in an assembly at runtime. They are instantiated from the corresponding component classes of the architecture configuration. They might define constraints on components' attributes that reflect design decisions impacting component states (attribute values). They also give their current component status by describing the attributes with their values.

```
component_instance : :=
  component_instance component_instance_name
    instance_of component_class_identifier
    initiation_state attribute_value_list
    current_state attribute_value_list

attribute_value_list : :=
  attribute_name = attribute_value
  ( ; attribute_name = attribute_value )*

attribute_value : :=
  "an attribute value of the correct type"
```

FIGURE 4.26 – Syntax of component instance

```
component_instance BikeTripC1
  instance_of BikeTrip (1.0)
  initiation_state currency=="Euro."

component_instance BikeCourseDBClassC1
  instance_of BikeCourseDBClass (1.0)
```

FIGURE 4.27 – Component instance descriptions of *BikeTripC1* and *BikeCourseDB-ClassC1*

By default, component classes can be instantiated into multiple component instances. When more precise cardinality information is needed, it is expressed in component role descriptions using **minInstances** and **maxInstances** that define the minimum and maximum numbers of component instances that are permitted to instantiate from the component class which implements this component role. By this means, component classes do not include this configuration-dependent information and remain reusable. In the assembly level, assembly constraints that restrain the valid number of instances will be checked against the cardinality information defined in the component role (in the specification level). There is no rule to constrain the name of component instances of a given component class.

4.4.2 Assembly Constraints

Assembly constraints define conditions that must be verified by attributes of some component instances of the assembly, to enforce its consistency. Such assembly constraints

are not mandatory. Dedal permits to define two types of constraints that must all be enforced and that either.

- *Logical constraint.* Logical constraints are regular expressions that are written using one or more logical operators among *and* (&&), *or* (||) and *not* (!) in our Dedal definition. To be verified, logical constraints must be valued as true.
- *Relational constraint.* Relational constraints can be used in two situations : (1) declare the relation between an attribute and a given constant value, or (2) specify the relation of the values of two distinct attributes. The relation operators are admissible as *less than* (<), *greater than* (>), *less than or equal to* (<=), *greater than or equal to* (>=), *equals* (==) and *different from* (! =).
- *Instance constraint.* The number of component instance for a component role can be refined in the assembly constraint to meet the different requirements of different runtime systems. They are expressed using **MinInstanceNbr**, **MaxInstanceNbr** and **InstanceNbr**, that represent the minimum, maximum and exact number of component instance.

Such simple assembly constraints are illustrated on the example of Fig. 4.24 where the value of the *currency* attribute of component *BikeTripC1* is fixed to *Euro* and where the value of the attribute *company* of the *BikeCourseClassDBC1* component must be maintained identical to the value of attribute *company* of component *BikeTripC1*. Another example that involves cardinalities would be expressed as the assembly constraint *InstanceNbr(BikeGUI)==10* that means that exactly ten component instances of the *BikeGUI* component role should be instantiated in this system. The cardinality information of the *BikeCourse* component role is stored in its specification (see Fig. 4.7).

```
1 assembly_constraint : :=
2   logical_constraint | relational_constraint

3 logical_constraint : :=
4   ( ! assembly_constraint ) |
5     ( assembly_constraint ( || | && ) assembly_constraint )

6 relational_constraint : :=
7   ( instance_attribute ( == | != | > | < | >= | <= )
8     ( instance_attribute | attribute_value ) )

9 instance_constraint : :=
10  ( ( MinInstanceNbr | MaxInstanceNbr | InstanceNbr )
11    ( Component_role_name ) == PositiveInteger )

12 instance_attribute : :=
13  component_instance_name . attribute_name
```

FIGURE 4.28 – Syntax of assembly constraint

However, in our work, assembly constraints are only listed without conflict detection among them, such as the logical conflict or the relational conflict.

4.4.3 Analysis

The analysis of the assembly is the most complicated one, as it covers three kinds of checking : consistency checking inner-level and two kinds of conformance checking intra-level.

4.4.3.1 Consistency of Instantiated Component Assembly

For instantiated component assemblies, two consistencies should be checked :
- *Name consistency.* Two component/ connector instances cannot have the same name.
- *Cardinality consistency.* The component instances of assembly should respect the cardinality defined in the assembly constraints or component role specification.

4.4.3.2 Conformance Between Specification and Assembly

The conformance between abstract architecture specifications and instantiated component assemblies focuses on the *cardinality conformance*. The cardinality constraints defined in the assembly constraints should conform to the cardinality constraints specified in the component role specification. The cardinality defined in the assembly should be a subset of cardinality defined in the specification. For example, the cardinality of the *BikeGUI* component role in assembly is defined between 5 and 8 (in Fig. 4.24), and this is a subset of cardinality defined in the *BikeGUI* component role specification which is defined the maximum number of instance should be 15 (in Fig. 4.7).

4.4.3.3 Conformance Between Configuration and Assembly

Conformance between a concrete architecture configuration and an instantiated component assembly is quite straightforward. Two rules should be preserved.

1. All component / connector instances of the assembly must be an instance of a corresponding component / connector class from its source configuration. Fig. 4.24 depicts an architecture assembly that conforms to the configuration presented on Fig. 4.11.
2. Conformance also includes the verification that attribute names used in an assembly constraint of some component assembly pertain to the component classes the components of the assembly are instances of. For example, the assembly constraint *BikeTripC1.currency=="Euro."* of Fig. 4.24 has the conformance process check whether the *BikeTrip* component class (from which *BikeTripC1* is instantiated) possesses a *currency* attribute.

4.5 SYNTHESIS OF DEDAL

In this chapter, we describe Dedal, a three-level ADL. It covers three views of architectures : abstract architecture specification, concrete architecture configuration and instantiated component assembly.

4.5.1 Modelisation of Architecture Description

Following the comparison of Chapter 3, we sum up and evaluate the architecture modeling elements of Dedal using the same comparison criteria as in Section 3.3.2.

For specifications, we propose a separated specification level which is defined by component roles (abstract component types), connections and architecture behavior. Component roles are often used in industry design, however, they are proposed for the first time as a model element in an ADL. Architecture behaviors exist in works like Wright and SOFA 2.0 (generated from component behavior), however we permit the customized architecture behavior which can be defined by architects.

Architecture specification	**Given name**	**Characteristics of an ADL**
Specification	"specification"	Explicit, independent specification
Component Role	"component_role"	Abstract component types
Connections	"connection"	Definition of the connected interfaces of components.
Architecture Behavior	"architecture_behavior"	Based on the behavior protocols of SOFA 2.0 [Plásil98].
Component Role	**Given name**	**Characteristics of an ADL**
Interfaces	"interface"	Set of *provided* or *required* interfaces.
Component behaviors	"role_behavior"	Using the behavior protocols of SOFA 2.0 [Plásil98].

TABLE 4.1 – Specification expressiveness in Dedal

At the configuration level, the contribution of our work is to clearly isolate the definition of the implementation of the architecture from the definition of the class design of the architecture. The implementation is the choice of the component instances that effectively compose component assemblies at runtime. The class design is the choice of the component classes that compose the component configurations.

Architecture configuration	**Given name**	**Characteristics of an ADL**
Configuration	"configuration"	Implement a specification.
Component	"component_class"	Component class which match the corresponding component role.
Connector	"connector_class"	Optional. Connector class which manages a given connection.

TABLE 4.2 – Configuration expressiveness in Dedal

The type system of Dedal is explicit and extensible. It supports the definition of interfaces, primitive components, composite components and connectors.

At the assembly level, Dedal integrates *assembly constraints* which describe the specific rules applied at runtime to build assemblies. This information is very important in the evolution phase, as it can be considered as a measure to validate the evolution of assemblies at runtime.

Component Type	Given name	Characteristics of an ADL
Interfaces	"interface"	Can be *provided* or *required*.
Component behavior	"connector_behavior"	Using the behavior protocols of SOFA 2.0 [Plásil98].
Connector Type	**Given name**	**Characteristics of an ADL**
Interfaces	"interface"	Can be *provided* or *required*.
Connector behavior	"connector_behavior"	Using the behavior protocols of SOFA 2.0 [Plásil98].
Composite component	**Given name**	**Characteristics of an ADL**
Composition	"composite_component"	The information is specified in ***using***.
Delegation	"delegated_interface"	Differentiated into two types : ***provided*** and ***required***.

TABLE 4.3 – Type system in Dedal

Architecture assembly	Given name	Characteristics of an ADL
Assembly	"assembly"	Independent level with its own description.
Component instance	"component_instance"	Specified by different attributes.
Connector instance	"connector_instance"	Can be automatically generated.
Assembly constraint	"assembly_constraint"	Constraints of the runtime system.

TABLE 4.4 – Component type expressiveness in Dedal

4.5.2 Our contributions

The main contributions of Dedal is twofold : modeling architectures in three views, and providing support of component-based software evolution [Zhang10].

- *Modeling architectures in three levels.* The proposed three views for architecture representations clearly identify the elements of the meta-model that underlays architecture definitions and classify the architecture information in separate models to enable the reuse and evolution of those designs. Other ADLs often propose architecture models that have the same names but are not clearly related to the different steps of architecture development processes. System, configuration or architecture are often used to name the same concepts. Furthermore, model elements are often given misleading names, which often conflict with other paradigms or domains. For instance, component instances should be used to name the instances of component classes which are created and executed at runtime, in other words used to describe the structure of effective component assemblies while in most ADLs, the definition of component instances is done along with the ! choice of component classes in the configuration of architectures. With such confusion, the evolution process and the versioning of the architecture are thus limited or complex to manage.
- *Design for component reuse.* Dedal aims to support software development based on reused components. Firstly, it supports architecture element reuse including components, connectors and even architectures. As reuse is the essence of component-based software development and evolution, it is perfectly suitable for component-based software development. As we mentioned in Chapter 3, very few existing ADLs

support component-based software development. Dedal fills this gap. Furthermore, Dedal's three levels perfectly correspond to component-based software development (see Fig. 3.19 in previous section 3.5.4).

Deuxième partie

Software Evolution

Chapitre 5

Architecture-Centric Software Evolution

In preceding chapters, we presented the state-of-the-art of CBSE and software architectures, and proposed Dedal, a three level ADL. These three chapters focus on how software architectures should be modeled for software development. Software maintenance and evolution require a special focus.

In this and the following chapters, we change our focus from software architecture to software evolution, precisely architecture-centric software evolution. This means that we are going to study what needs to be documented in architecture to manage evolution. In this chapter, we try to find the answers of following questions.

- How the software evolution affects on software architectures ?
- How the software architecture helps examine software changes ?
- Is there an ADL supporting the software evolution ?

In the beginning of this chapter, we will give the context of software evolution, including its definition, taxonomies and evolution process. Then we present architecture-centric software evolution and its weight in the context of software evolution. At last, we examine the existing ADLs to evaluate how they support evolution.

Contents

5.1 SOFTWARE EVOLUTION

In this section, we will briefly introduce the background of software evolution and the strategies of research in this domain. This information will help us to study and evaluate a more precise sub-domain of evolution, *architecture-centric evolution*, which will be presented in Section 5.2.

5.1.1 Definition of Software Evolution

Software evolution plays an important role in the software development lifecycle, because organizations that have invested in software systems are ready to invest in system change to maintain the value of these assets. Erlikh [Erlikh00] suggests that 90% of software costs are evolution costs. There is quite a lot of uncertainty in this percentage, however, at least the cost is high enough to concentrate research and engineering efforts [Sommerville06].

Definition. "***Software evolution*** *[Lehman00a] 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****.* "

5.1.2 Dimensions of Software Evolution

The study of software evolution has three *dimensions* : the theoretical study of software evolution, the pragmatic evolution support and the study of types of changes. The first two dimensions are proposed by Lehman *et al.* [Lehman00b] from the two prevalent views : the *what* and *why* (the theory study of software evolution) versus the *how* (the pragmatic evolution support) perspectives [Lehman00b]. The other dimension of software evolution that focuses on taxonomies of changes is pointed by Mens [Mens08a].

5.1.2.1 Theoretical Study of Software Evolution

The theoretical study of software evolution studies the nature of the software evolution phenomenon, and seeks to understand its driving factor, its impact, etc. An important insight that has been gained from these studies is the theories of evolution processes.

Evolution [Bennett00] can be considered as a process that either introduces new requirements into an existing system, or modifies the system if the requirements were not correctly implemented, or moves the system into a new operating environment. Based on this view of evolution, Yau *et al.* [Yau93] propose a *change mini-cycle* which consists of the five main phases as shown in Fig. 5.1.

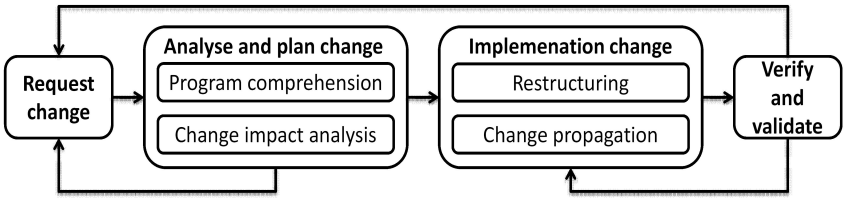


FIGURE 5.1 – The staged process model for evolution (adapted from [Yau93])

5.1.2.2 Pragmatic Evolution Support

Pragmatic evolution support studies the more pragmatic aspects that aid the software developer or project manager in his day-to-day tasks. Hence, this view primarily focuses on technology, methods, tools and activities that provide the *means* to direct, implement and control software evolution.

5.1.2.3 Taxonomies of Software Changes

Many taxonomies of software change [Lientz80, Chapin01, Buckley05] are proposed trying to answer the *why, how, what, when* and *where* of software evolution. Categorizing software changes can help better understand the nature of the changes in order to prioritize effectively the change request [Lientz80, Chapin01] or position concrete software evolution [Buckley05].

Lientz and Swanson [Lientz80] proposed a mutually exclusive and exhaustive software maintenance typology, including three categories : corrective, perfective and adaptive. This definition was later updated in the ISO/IEC Standard for Software Maintenance [ISO99] to include four categories : corrective, perfective, adaptive and preventive. This typology was further refined by Chapin *et al.* [Chapin01] into an evidence-based classification, that includes non-technical issues such as documentation, consulting, training and so on. [1]

The more complex taxonomy of evolution is given by Buckley *et al.* [Buckley05], which is focusing more on the technical aspects. It is a taxonomy of characteristics of software change mechanisms and the factors that influence these mechanisms, i.e., the *how*, *when*, *what*, *where*, of software change. In general, the dimensions of evolution is classified as characterizing the mechanism or as influencing factors. Four categories are presented : Temporal properties, object of change, system properties and change support, as shown in Fig. 5.2.

5.1.3 Synthesis of Software Evolution

Software evolution begins to shift its underlying basis from source code to software architecture models. Source code cannot supply a complete and synthetic view of software to reflect its original design decisions. On the contrary, architecture models supply a more manageable and effective basis for software changes. Thus, the architecture-centric evolution as a sub-domain of software evolution concentrates more attention these years [Oreizy98b].

1. In this thesis, we use the typology of Lientz & Swanson, because the refined typology of Chapin *et al.*covers organization aspects we do not take into account.

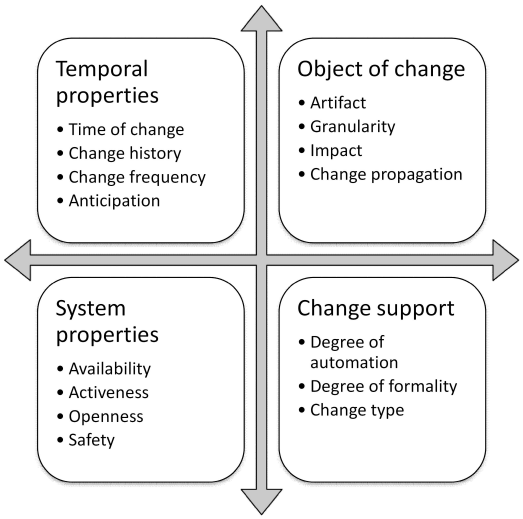


FIGURE 5.2 – Themes and dimensions of software change

Architecture provides the alternative of using architectural models as the basis for maintenance tasks. This makes the software evolution more efficient and effective by just changing the coarse gained software elements, such as components, connectors and connections between them.

– These is no architecture-centric evolution process proposed to that implements the mini change cycles of architecture-centric evolution.
– There is no taxonomy of architecture-centric software evolution that focuses its classification on the characteristics of architecture changes.

Evolution should be studied globally from change types, to its evolution process and to its implementation tools. Hence, in the following section, we try to firstly give some definitions for architecture-centric software evolution and propose a taxonomy of architecture changes. In the reminder of this chapter, we examine how various ADLs support architecture-centric evolution.

5.2 ARCHITECTURE-CENTRIC EVOLUTION

5.2.1 Context

5.2.1.1 Definition

Architecture-centric evolution concerns the evolutions which are activated by architecture changes. Software architectures may change over time, due to the need to provide a more dependable system, the need to remove identified deficiencies, or the need to handle dynamically-evolving collections of components at runtime [Garlan00]. We give the definition of architecture-centric evolution as follows.

> ***Definition. Architecture-centric evolution*** *is the collection of software architectural activities to change a software from its older version to the new version, which is activated by architecture changes. The architectural activities concern the modifications of both the software architecture model and its runtime software counterpart.*

Architecture-centric evolution can make software evolution more easy and controllable than code evolution. Updating a software component to use a newly available library, for example, can be achieved by changing its architectural connections so that it is linked to this new library. Focusing on this level of abstraction spares developers from having to delve into low-level component source code, which is harder to understand and modify [Georgas06].

5.2.1.2 Architecture Changes

Architecture changes concern the changes applied on architectures, including their components, connectors or their configuration. Architecture changes have nature characteristics, including change time, change type, etc. However, a better understanding of architecture changes is the prerequisite to plan the evolution process and enact the changes.

5.2.1.3 Architecture-Centric Evolution Process

Architecture-centric evolution has requirements for its evolution process, as the evolution is based on architecture models. For example, change propagation should be applied on architecture models not source code.

5.2.1.4 Architecture Mismatches

Software evolution might cause architecture mismatches. Perry and Wolf [Perry92] have identified the phenomenons of architectural erosion and drift as any detrimental deviation over time of a software system's architecture from its original design conception. This covers any mismatch between design and implementation and any change to an architecture that results in a loss of design quality. Precisely, architectural drift and erosion can be defined as follows.

- *Architectural drift.* Architectural drift is introduction of principal design decisions into a system's lower level of software architecture that (a) are not included in, encompassed by, or implied by the higher level of architecture.
- *Architectural erosion.* Architecture erosion is the introduction of architecture design decisions into a system's lower level of architecture that violate its higher level of architecture.

The architectural drift and erosion are both bottom-up inconsistencies, that lower level introduce new changes that are inconsistent with higher level of architecture. However, are there top-down inconsistencies in architectures ? We define a new concept called architecture pendency.

- *Architecture pendency.* Architecture pendency is the introduction of new design decisions into a higher level of architecture that are not implemented by its lower level architecture.

These three categories of architecture mismatches are error-prone. To avoid these three situations is our objective during the architecture-centric evolution process.

5.2.2 Our Taxonomy of Architecture Evolution

In our thesis, we study software architecture-centric evolution from its two aspects : architecture changes and architecture-centric evolution process.

- *Architecture changes.* Architecture changes have their own semantics which differs from source code changes. Its concerns the elements of architecture : component, and connectors changes, and changes in connections between them. To study architecture changes is a prerequisite to plan the evolution process and implement the changes.
- *Architecture-centric evolution process.* Architecture-centric evolution is based on architecture models. Its main difference with source code evolution is that it must preserve the coherence of the architecture models after evolution. The above three kinds of architecture mismatches should be searched for and avoided in software evolution process.

Based on this view, we propose a taxonomy of software architecture evolution. It is inspired from the taxonomy of Buckley et *al.* [Buckley05]. The purpose of this taxonomy is : (1) to better understand architecture change nature and, thus, describe them correctly, (2) to evaluate the support of evolution by existing ADLs tools. It classifies evolution factors into two dimensions : characteristics of architecture changes and activities of architecture-centric evolution, corresponding to two aspects of architecture-centric evolution.

Characteristics of architecture change. characteristics of software change are the factors of software change which can be used to describe their nature. We use a simple heuristic to determine whether this factor is characteristics of software change. It is to put the factor in a simple sentence of the form : "The change is <factor>" like in [Buckley05]. For example, the change is a semantic change or structure change. Hence, change type is one of the characteristics of software change. According to this strategy, we determine the characteristics of change : *time of change*, *change type*, *purpose of change*, *artifact subject to change* and *anticipation*.

Activities of architecture evolution. In the taxonomy of Buckley *et al.* [Buckley05], there are many activities of software evolution such as change propagation, or impact analysis, but they are classified into different dimensions. In this thesis, we try to distinguish the evolution activities from the characteristics of changes, and classify them as a new independent dimension. These activities are indispensable in an architecture-centric evolution process, as they affect the result of change. The heuristics we use to distinguish them are : (1) whether they should be included in evolution process and (2) whether they affect the result of architecture evolution. The activity list we come out with is : *completeness and consistency checking*, *impact analysis*, *evolution test*, *change propagation*, and *versioning*.

Table 5.1 lists the factors of architecture-centric evolution included in this taxonomy. The definitions and details of these characters and activities will be discussed in the rest of this section.

Characteristics of architecture change	Activities of architecture evolution
Time of change	Completeness and consistency checking
Change type	Impact analysis
Change purpose	Evolution test
Artifact subject to change	Change propagation
Initial level of change	Versioning
Change operation	
Anticipation	

TABLE 5.1 – The characteristics and activities of change

5.2.2.1 Characteristics of Architecture Change

Time of change. Software changes can be intervene at three different phases of the software life-cycle : compile-time, load-time and run-time. Based on *when* the change happens, three categories of changes can be deduced as follows.

- *Static*. The software change concerns the source code of the system. Consequently, the software needs to be recompiled for the changes to become effective.
- *Load-time*. The software change occurs while software elements are loaded into an executable system.
- *Dynamic*. The software change occurs during the execution of the software.

In this thesis, we study the architecture-centric evolution focusing on dynamic changes.

Change type. Architecture changes might affect the semantics of the software or its structure. Thus, architecture changes can be typed in two categories : semantic change and structure change.

- *Semantic changes* might be confined to the interior of a component or connector. They modify the interior of individual components or connectors but preserve their exterior interface skeleton (component type). In other words, the entire architecture structure remains unchanged. It sometimes corresponds to replacing one or more components or connectors by their newer versions.
- *Structure changes* alter the structure of the configuration by adding or removing a component or connector that adds or removes functionalities to the system.

Change operation. The operations of architecture changes indicates the actions of *adding*, *removing* or *replacing* the architecture elements (components or connectors). The adding and removing operations always cause structure changes. The replacing operations might cause the semantic changes or structure changes (if the component is replaced by another component with different component type).

Artifact subject to change. The artifacts subject to change in a software architecture are its building blocks : components, connectors or connections (links) between them. At different levels of software architectures, these elements take specific forms : for example, component roles in the abstract specification level.

Initial level of change. Software architectures have three levels : specification, configuration and assembly, as mentioned in Chapter 3. Each of them can be changed. For example, if the change is initiated at specification level, then the change will be propagated to lower levels, but if the change is initiated in the assembly level, then the change will be propagated to upper levels.

Anticipation. This dimension refers to the time when the requirements for a software change are foreseen. *Anticipated changes* can be foreseen during the initial development of the system and, as such, can be accommodated in the design decisions taken. *Unanticipated changes* are those that are not foreseen during the development phase. These may arise because of new system requirements [Kniesel05].

- *Anticipated dynamic change* is a system property that indicates the system is architected specifically to accommodate to expected dynamic changes ;
- *Unanticipated dynamic change* is a quality that refers to a system's general suitability for dynamic change.

Purpose of change. According to the ISO/IEC 14764 Standard for Software Maintenance [ISO99], the purpose of changes can be categorized into four types : corrective, perfective, adaptive and preventive. We adapt their definitions to software architecture changes as follows :

- *Corrective changes* means that the changed software corrects errors / bugs identified in the old one.
- *Perfective changes* means that the changed software alters the code of a component or a connector to improve its non-functional qualities (such as performance).
- *Adaptive changes* adapt components or connectors to meet changes in their environment or in requirements on the software.
- *Preventive change* mean that the change is undertaken to prevent the ripple (side) effects of other changes. For example, adding a new component often requires that its connected component be adapted.

5.2.2.2 Activities of Architecture-Centric Evolution

Consistency checking. Consistency checking aims to check whether the system architecture will still be consistent after the requested changes. It plays an important role in software evolution. Software architecture as a formalized model of a software system is a support for reasoning and preserving the completeness of architecture, while direct evolution of source code cannot do this. *Consistency* is an internal property of an architectural model, which is intended to ensure that different elements of that model do not contradict one another. Taylor *et al.* [Taylor09] propose five kinds of inconsistencies that can happen in software architectures :

- *Name inconsistency.* A component or connector has the same name as a newly introduced component or connector.
- *Interface inconsistency.* Two connected interfaces are not type-compatible.
- *Behavior inconsistency.* Behavior inconsistencies occur between components that request and provide services which names and signature match, but which behaviors (expressed as protocols, state machines or regular expressions) do not.

- *Interaction inconsistency.* Interaction inconsistencies can occur between connected architectural elements when they violate the interaction protocols defined by their running environment. For example, two components cannot connect directly in C2SADEL.
- *Refinement inconsistency.* Refinement inconsistencies come from the fact that a system's architecture can be captured at multiple levels of abstraction that are not consistent with one another anymore after changes. For example, the architecture configuration is altered, but the architecture specification and assembly are not updated.

Impact analysis. Change impact analysis [Bohner96] aims to predict the system-wide impact of a change request before actually carry out all the required modifications to the system. The impact of a change can vary from the small scale of a component to the entire architecture. Lassing *et al.* [Lassing99b, Lassing99a] created a subjective impact scale in their study of architecture flexibility. The scale included four levels : *no impact*, *affect one component*, *affect several components*, and *affect entire architecture*.

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

Evolution test. Evolution test is an activity that guarantees the security of changes. The disadvantage of changes are their unfaithful reliability. For component-based softwares, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization that use them (COTS components are often developped externally). When introducing changes at the application level such as component updating or addition, developers face the risk that the introduced changes cause a system failure [D'Souza99]. Test can be useful in two cases : (1) to guarantee the correctness of newly introduced components or connectors, (2) to ensure the entire system execution after changes.

In the case of architecture change, test rely on the underlying architecture models of software systems. This kind of test can use richer change semantics than transitional source-code based software tests.

Change propagation. Change propagation is a key activity to avoid architectural drift and erosion. It is the process of ensuring that a change to an entity is propagated to all related entities in the software system. Propagation can be performed in two directions : vertical, or horizontal, like the impact directions. *Vertical propagation* affects the same level of architecture abstraction and *horizontal propagation* is done across different levels of architecture abstraction. In this thesis, we are more interested in horizontal propagation among architecture levels. This propagation can be top-down or bottom-up.

Versioning. The addition of versioning information to software architecture has the potential to control and support software architecture evolution. Firstly, architectures evolve like software systems, they need to track and manage their changes by multiple versions of an architecture [van der Hoek98b]. Secondly, each level of software architecture

can independently have its own versions. For example, one versioning of higher level might be implemented by multiple versions in a lower level of architecture.

In software architecture models, each architecture dimension and its basic building blocks can be versioned. The version model [Conradi98] defines items to be versioned, the common properties shared by all versions of an item, and the deltas, that is, the differences between them. For the version model of software architecture, it can be state-based versioning or change-based versioning.

- *State-based versioning.* Versions are described in terms of revisions and variants.
- *Change-based versioning.* Versions are described in terms of changes applied to some baselines, such as extensional or intentional [Haake96]. *Extensional change-based versioning* annotates the changes inside the version. *Intentional change-based versioning* describes the changes separately from the versions, in terms of the operations or transformations that have been used.

5.2.3 Synthesis of Change Taxonomy

The objectives of this taxonomy is twofold. The first is to better understand the nature of architecture changes and the activities of architecture-centric evolution process. The second is to serve as an evaluation grid to position the support of evolution in existing ADLs. It can be used to evaluate evolution tools from two perspectives : (1) which type of changes they can support, (2) how many activities their evolution process contains. In the following section, we will use this taxonomy to evaluate the existing ADLs from these two perspectives. In conclusion, the detailed factors of changes and their possible values are summed up in Table 5.2.

Characteristics of Architecture Change	**Value**
Time of change	▶static, ▶load-time, ▶dynamic
Anticipation	▶anticipated change, ▶unanticipated change
Change type	▶semantic, ▶structure
Change purpose	▶corrective, ▶perfective, ▶adaptive, ▶preventive
Artifact subject to change	▶component, ▶connector, ▶connection
Initial level of change	▶specification, ▶configuration, ▶assembly
Change operation	▶addition, ▶removal and ▶substitution
Activities of architecture change	**Value**
Consistency checking	▶behavior, ▶interaction, ▶refinement consistency checking
Impact analysis	▶vertical impact, ▶horizontal impact
Evolution test	▶yes, ▶no
Change propagation	▶vertical propagation, ▶horizontal propagation
Versioning	▶state-based versioning, ▶change-based versioning (extensional, or intentional)

TABLE 5.2 – The characteristics and activities of architecture change with their possible values

5.3 EVOLUTION SUPPORT IN ADLS

Software architecture evolution refers to static, load-time and dynamic (runtime) changes. Dynamic evolution [2] refers to modifying the architecture and enacting those modifications in the system while the system is executing. Support for dynamism is important in the case of certain safety- and mission-critical systems, such as air traffic control, telephone switching, and high availability public information systems. Shutting down and restarting such systems for upgrades may incur unacceptable delays, increased cost, and risk [Medvidovic00]. To support architecture- based run-time evolution, ADLs need to provide specific features for modeling dynamic changes and techniques for making them effective in the running system [Oreizy98b].

We have argued for the need to model dynamic changes at the architecture level. Dynamic evolution support also imposes many requirements to ADLs. Firstly, ADLs should be able to model architecture changes. Secondly, ADLs need to analyze the correctness of the changes and the consistency of the system architecture after the changes. Furthermore, changes should be correctly applied at the implementation level to guarantee a correct execution of the system after changes and preserve the continuity of execution.

In the following section, we examine a set of dynamic ADLs including C2SADEL, Dynamic Wright, Darwin, SOFA 2.0, xADL2.0 and MAE. This selection of ADLs is different from the ADLs used in the architecture modeling comparison. There are two reasons for this : (1) not all ADLs presented in previous chapter support evolution, such as Unicon, (2) some ADLs, like MAE, are in fact extensions of other ADLs, with additional functions to support evolution.

We try to analyze the existing dynamic ADLs from two views : (1) which kind of architecture changes they support (using our taxonomy) and (2) how they implement the architecture changes (the method, the process or special mechanism).

Illustrative example. In this section, we will continue to use the example from Chapter 3. The architecture change is to replace the component *CompC* by the component *CompD*.

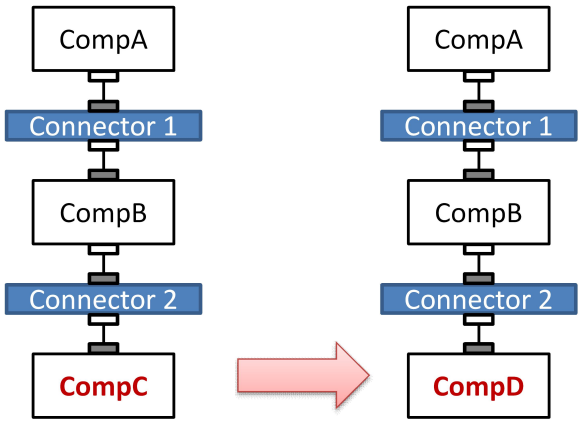


FIGURE 5.3 – The illustrative example of component substitution (replace the component *compC* by the component *compD*)

2. Often called dynamism in ADL

5.3.1 C2SADEL

C2SADEL supports unanticipated software change without any restrictions on the types of permitted changes. Instead, arbitrary modifications are allowed in principle. C2SADEL realizes software evolution with two important techniques : an (sub) architecture modification language (AML) [Medvidovic96, Oreizy98b] and a relatively mature evolution process [Oreizy98b, Oreizy99]. The prototype to support runtime reconfiguration is implemented by the *ArchStudio* 1.0 and 2.0 [3] tool suite.

5.3.1.1 AML

In C2SADEL, they define a (sub) architecture modification language to describe the architecture changes, called AML. This language [Oreizy98a] can describe addition and deletion of components or connectors, connection and disconnection of components and connectors.

In order to support the architecture change operations, components, connectors and systems are all wrapped to support the dynamic reconfiguration using pre-defined external interfaces [Medvidovic99a]. The methods of these interfaces are used to support the dynamic operations such as *addComponent(component)* for *architecture* control interface

In C2SADEL, two interactive tools are used to describe software architectures and their changes : *Argo* (graphic) and *ArchShell* (text). The example of component addition using ArchShell can be found in Fig. 5.4.

```
1 >add component
2 ClassName : c2.ArchExample.CompD
3 Name? CompD
4 >unweld
5 Top entity : Connector2
6 Bottom entity : CompC
7 >remove component
8 Name? CompC
9 >weld
10 Top entity : Connector2
11 Bottom entity : CompD
12 >start
13 Entity : CompD
```

FIGURE 5.4 – The ArchShell commands used to replace a component *CompC* by component *CompD*

5.3.1.2 Change Process of C2SADEL

In C2SADEL, the general approach to support dynamic reconfiguration is shown in Fig. 5.5. There are three main steps in this process. Firstly, architecture changes are applied to architectural models. Secondly, the runtime architecture infrastructure reifies changes from the architectural model to the implementation and thus maintains the

3. These are the early versions of ArchStudio. Now, ArchStudio is updated to version 4, which is used by xADL 2.0.

consistency between the architectural model and the implementation. At last, the changes are implemented.

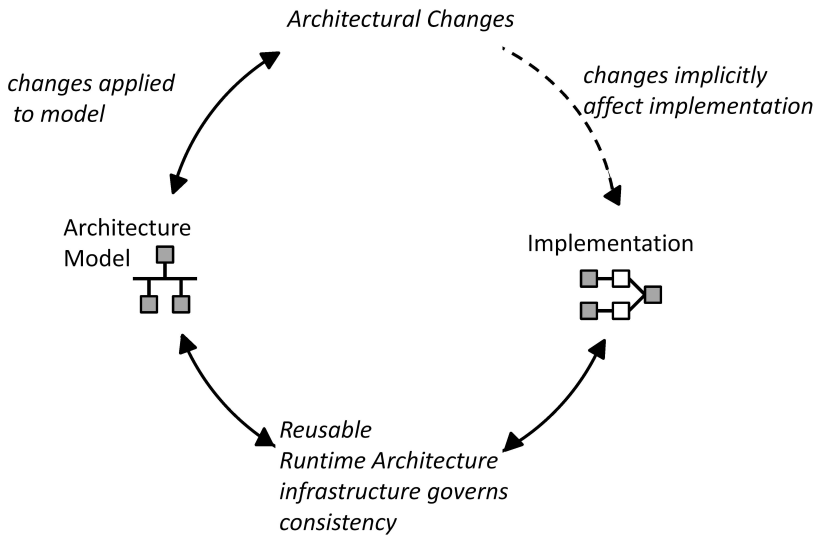


FIGURE 5.5 – High-level processes in a comprehensive, general-purpose approach to self-adaptive software systems (adapted from [Oreizy98a])

5.3.1.3 Discussion

The C2SADEL has a relatively complete evolution process, but the relationship between architectures and implementation is done by a mapping (one-way). This makes C2SADEL unable to support reverse evolution (evolve the runtime system and propagate the changes to software architectures). Table 5.3 gives an overview of evolution supported by C2SADEL.

5.3.2 Darwin

Dynamic evolution in Darwin is realized by two techniques : dynamic instantiation to realize the anticipated changes [Magee96] and a declarative language to accomplish unanticipated change [Kramer90, Kramer98].

5.3.2.1 Dynamic Instantiation

Darwin has features which permit the description of dynamic structures which evolve as execution progresses. Structural evolution includes changes in both the bindings (connections) between components and the set of component instances. These structural changes can be expressed without violating the declarative nature of Darwin, thereby facilitating both a semantic description and reasoning. There are three techniques used in Darwin to capture dynamic structures : lazy instantiation, direct dynamic instantiation, and dynamic binding.

Characteristics of Architecture Change	Value
Time of change	Dynamic
Anticipation	Unanticipated change
Change type	Structure change
Change purpose	—
Initial level of change	Configuration
Change operation and subject artifact	▶ Adding and removing a component, ▶ Adding and removing a connector, ▶ Adding and removing a connection
Activities of architecture evolution	**Value**
Consistency checking	Refinement consistency checking (to-down)
Impact analysis	—
Evolution test	—
Change propagation	Horizontal propagation (top-down)
Versioning	—

TABLE 5.3 – The characteristics and activities of change in C2SADEL

- *Lazy instantiation.* The component providing a service is not instantiated until a user of that service attempts to access to the service. This is specified by the keyword ***dym***.
- *Direct dynamic instantiation.* Direct dynamic instantiation enables structures to evolve in arbitrary ways. The dynamic component instance is defined by ***bind***, and is triggered to instantiate by a supervisor component.
- *Dynamic binding.* Dynamic binding enables the one to many binding, which is controlled by event mechanisms. The one to many means that one component interface can connect with different interfaces of different components. Each of them is active in different situations according the control of event mechanisms.

5.3.2.2 Structure-Based Change Model

In Darwin, the entire evolution process is controlled by reconfiguration management, that is responsible for making decisions regarding the change application policy and its scope. In order to support dynamic unanticipated changes, Darwin uses a change script language to describe the changes, and a node-state transition model to preserve the consistency of the system during dynamic evolution.

A configuration [4] consists of processing nodes interconnected using bidirectional communication links. A *node* is a processing entity (component). A *link* is a connection between nodes. When a runtime change is required, reconfiguration management will require all the nodes affected by changes (either directly or propagatedly) to enter the *quiescent state*. Quiescent state of a node indicates that the node is passive and has no outstanding transactions which it must accept and service, as shown in Fig. 5.6. To enter the passive state is in order to avoid side effects when the evolution is effectively performed.

4. The description of Darwin architectures can be found in Section 3.4.3.

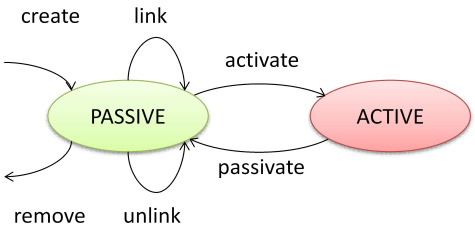


FIGURE 5.6 – Node state transition diagram in Darwin (adapted from [Kramer90])

In Darwin, unanticipated changes are described in terms of change scripts. Those scripts support the addition and deletion of components and the connection and disconnection of components. Change scripts contain two parts : change specification and change transactions. *Change specification* specifies architecture changes using structural actions :

- **create** N :T [**at** L] : Create node N of type T, optionally at physical location L.
- **remove** N : Remove node N.
- **link** N1 **to** N2 : Create a connection from node N1 to node N2.
- **unlink** N1 **from** N2 : Remove a connection between node N1 and node N2.

Change transactions are automatically derived from change specifications. They include two extra actions (**activate** and **passivate**) and the ordering of execution. Figure 5.7 shows the change specification and the change transactions of the illustrative example, in Darwin.

```
1  Change specification :
2    remove CompC <and unlink dangling connections>
3    create CompD
4    link CompB to CompD

5  Change transaction :
6    passivate CompA, CompB, CompC
7    create CompD
8    unlink CompC from CompB <unlink dangling connections>
9    remove CompC
10   link CompB to CompD
11   activate CompA, CompB, CompD
```

FIGURE 5.7 – The change example in Darwin (replace the component *CompC* by the component *CompD*)

5.3.2.3 Discussion

Darwin is an ADL which supports all kinds of evolution including unanticipated and anticipated. It is the first ADL proposing to support dynamic evolution. Many later ADLs inspired from its ideas. However, Darwin (1) does no support of reverse evolution like C2SADEL, (2) has an incomplete evolution process missing many important phases, like evolution test and re-engineering, and (3) makes no architecture level consistency checking (the only consistency checking supported by Darwin is runtime state checking). Table 5.4 gives an overview of evolution supported by Darwin with our proposed taxonomy.

Characteristics of Architecture Change	Value
Time of change	►Static, ►load-time, ►dynamic
Anticipation	Anticipated change, unanticipated change
Change type	Structure change
Change purpose	—
Initial level of change	Configuration
Change operation and subject artifact	►Addition and removal of component, ►Addition and removal of connection
Activities of architecture change	**Value**
Consistency checking	State consistency checking
Impact analysis	Horizontal impact in configuration
Evolution test	—
Change propagation	Horizontal propagation (top-down)
Versioning	—

TABLE 5.4 – The characteristics and activities of change in Darwin

5.3.3 Dynamic Wright

Dynamic Wright [Allen97c, Allen98] supports anticipated changes using control events and has a relatively complete consistency checking system.

5.3.3.1 Control Approach

Dynamic Wright distinguishes two types of events in software systems : communication and control events. A special *control* event type is introduced to specify conditions under which dynamic changes are allowed.

Firstly, special *control* events are introduced into a component's alphabet, and allowed to occur in port descriptions. By this means, the interface of a component is extended to describe when reconfigurations are permitted in each protocol in which it participates. Secondly, these control events are used in a separate view of the architecture, the *configuration program*, which describes how these events trigger reconfigurations. The reconfiguration actions that are triggered in response to control events are *new*, *del*, *attach* and *detach*.

Figure 5.8 is an illustrative example. In this example, the *Configuror* is responsible for achieving the changes to the architectural topology (triggered by up and down) using instances of component types (e.g., CompB, Connector2, CompC, CompD), and *new*, *del*, *attach*, and *detach* actions, as illustrated in Fig 5.9.

5.3.3.2 Consistency Checking.

Wright enables three kinds of inconsistency checking : connector consistency, attachment consistency and configuror consistency checking.

– *Connector consistency checking* aims to check whether there is no deadlock among the interactions of connectors and internal protocols of connectors.

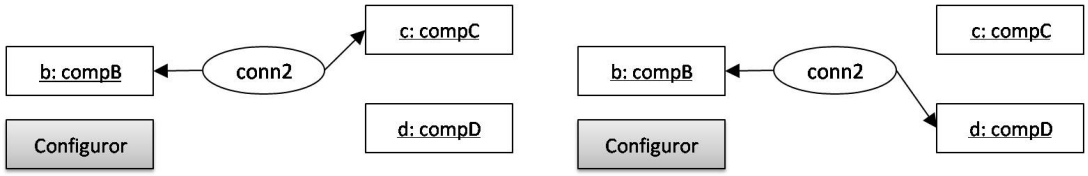


FIGURE 5.8 – The example of dynamic topology in Dynamic Wright (adapted from [Allen98])

```
1  Configuror DynamicClient-Server
2    Style Fault-Tolerant-Client-Server[5]
3          new.b :CompB
4          →new.c :CompC
5          →new.d :CompD
6          →new.conn2 : Connector2
7          →attach.b.r.to.conn2.p
8          →attach.c.p.to.conn2.r → WaitForDown
9    where
10   WaitForDown = (c.control.down→d.control.on→conn2.control.changeOk→
11                        Style Fault-Tolerant-Client-Server
12                              detach.c.p.from.conn2.s
13                              → attach.d.p.to.conn2.s
14                              → WaitForUp)
15                  []
16   WaitForUp = (c.control.up→d.control.off→conn2.control.changeOk→
17                        Style Fault-Tolerant-Client-Server
18                              detach.d.p.from.conn2.s
19                              → attach.c.p.to.conn2.s
20                              → WaitForDown)
21                  []
```

FIGURE 5.9 – The anticipated change example in Wright (replace the component *CompC* by the component *CompD*)

- *Attachment consistency checking* aims to check whether the port of a component is "consistent" with respect to a role to which it is attached. It corresponds to *interface inconsistency checking* in our taxonomy.
- *Configuror consistency checking* aims to guarantee that when the event *new.C* occurs, *C* doest not already belong to the current configuration. It corresponds to *name inconsistency checking* in our taxonomy.

5.3.3.3 Discussion

Dynamic Wright as a CSP-based ADL examines the evolution with many interaction consistency checking. However, Dynamic Wright only supports anticipated changes. Table 5.5 gives an overview of evolution supported by Dynamic Wright.

Characteristics of Architecture Change	Value
Time of change	Dynamic
Anticipation	Anticipated change
Change type	Structure change
Change purpose	—
Initial level of change	Configuration
Change operation and subject artifact	▶Addition and deletion of component instance, ▶Addition and deletion of connector instance, ▶Addition and deletion of connection
Activities of architecture change	**Value**
Consistency checking	Name consistency, interaction consistency, and deadlock consistency checking
Impact analysis	—
Evolution test	—
Change propagation	—
Versioning	—

TABLE 5.5 – The characteristics and activities of change in Dynamic Wright

5.3.4 SOFA2.0

SOFA2.0 [Hne06, Bures06] considers that arbitrary software change leads to *uncontrolled architecture modification*.[6] SOFA2.0 thus limits dynamic reconfigurations to those compliant with specific reconfiguration patterns. There are three authorized patterns : (1) nested factory, (2) component removal, and (3) addition or removal of interface.

5.3.4.1 Nested Factory

The nested factory pattern covers adding a new component and a new connection to an architecture. The new component is created by a *factory component* as a result of method invocation on this factory. The newly created component becomes a sibling of the component that initiated the creation. Figure 5.10 shows an example that adds a component *CompC* to the system which is created by a *CompCFactory* factory component. From the example, we can find that the new created component is treated as a sibling of the connected *CompB* component. This setting is specifically used to make the factory component independent of the entire working system.

5.3.4.2 Component Removal

Component removal is often seen as the most difficult evolution problem, as it might lead unbound (pending) interfaces. SOFA2.0 components have pre-defined behavior protocols [Plásil98] (see section 3.4.5). SOFA2.0 checks the possibility of component removal by a *behavior consistency checking* to see whether the required interfaces can be left unbound or not.

6. This uncontrolled architecture modification comes from the gap between architecture description and assembly level. There can be three kinds of mismatches as mentioned in Section 5.2.1.4.

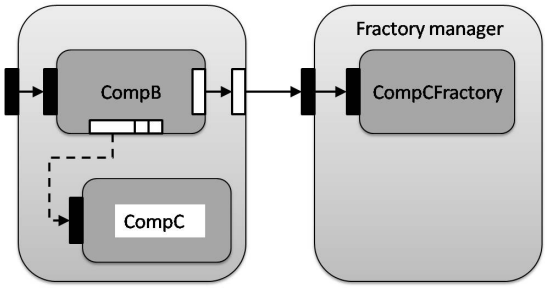


FIGURE 5.10 – Dynamic application example of SOFA2.0

5.3.4.3 Addition and Removal of Composite Component's Interface

Another interesting contribution of SOFA2.0 is the addition and removal of the interfaces of composite components. Adding or deleting an interface of primitive component is often unreasonable. However, composite components, as boxes of components, often require to add or remove a sub-component. If the addition or removal component provides a functionality to a delegated interface of the composite component, there is a need to be able to add or remove an interface to the composite component. The addition or removal of interfaces are then quite important to reconfigure hierarchical components [Cross02].

5.3.4.4 Versioning

SOFA 2.0 supports the versioning of architecture elements. The versionned architecture elements can be composite components or primitive components. As composite components are also represented as an architecture description, we can say SOFA 2.0 also supports versioning of software architectures. However, the version model in SOFA 2.0 is limited to a revision number which is concatenated to the name of components. For example, the component *DataBase* is described as *frame Database rev 2.1.0*.

5.3.4.5 Discussion

SOFA 2.0 is an ADL extended from Fractal ADL, thus it has a tight relationship between the architecture description and the runtime system. It enables to version software architectures. However, its disadvantages are (1) an incomplete evolution process and (2) a too simple version model. The overview of its support for evolution can be found in Table 5.6.

5.3.5 xADL2.0

xADL2.0 [Dashofy02b, Dashofy07] controls and realizes dynamic evolution with ArchStudio 4 [7] [Arc]. In ArchStudio 4, change descriptions are specified in terms of *architectural diffs*, and the architecture evolution process is based on ArchStudio 4.

7. The most recent version of ArchStudio.

Characteristics of Architecture Change	Value
Time of change	Dynamic
Anticipation	Anticipated change
Change type	Structure
Change purpose	—
Initial level of change	Configuration
Change operation and subject artifact	▶Adding and removing a component, ▶Adding and removing a connection, ▶Adding and removing a composite component's interface
Activities of architecture change	**Value**
Consistency checking	Behavior consistency checking
Impact analysis	—
Evolution test	—
Change propagation	—
Versioning	Revision number for each component

TABLE 5.6 – The characteristics and activities of change in SOFA2.0

5.3.5.1 Architectural Differences

Architecture changes in xADL2.0 are described as *architectural diffs*. The architectural diffs are the differences between two given architectures. Architecture diffs are caculated using the tool *ArchDiff*. ArchDiff takes two architectures as input, finds the differences between them, and outputs their differences as the required architecture changes that should be applied to the pointed architecture. Figure 5.11 shows the XML description of the architecture changes of the illustrative example. Chen *et al.* [Chen03] developed two tools *ArchDiff* and *ArchMerge* to find the architecture differences and merge the architectural diff with the architecture model.

```
1  <diff>
2    <diffPart><remove id="Link4"/></diffPart>
3    <diffPart><remove id="CompC"/></diffPart>
4    <diffPart><add>
5      <component id="CompD">
6      <type id="CompD_type"/>
7      </component>
8    </add></diffPart>
9    <diffPart><add>
10     <link id="Link5">
11     </link>
12   </add></diffPart>
13 </diff>
```

FIGURE 5.11 – The anticipated change example of XADL2.0 (replace the component *CompC* by the component *CompD*)

5.3.5.2 Architecture Evolution Process

The architecture evolution process of xADL2.0 is realized using ArchStudio 3 and 4 [8]. ArchStudio is an architecture-centric development and evolution environment. It employs an architecture-implementation framework to establish a mapping between architecture and implementation as C2SADEL. *Architectural diffs* are then used as change descriptions to evolve the system at run-time. The process for evolving a system decomposes into four steps (shown in Fig 5.12) :

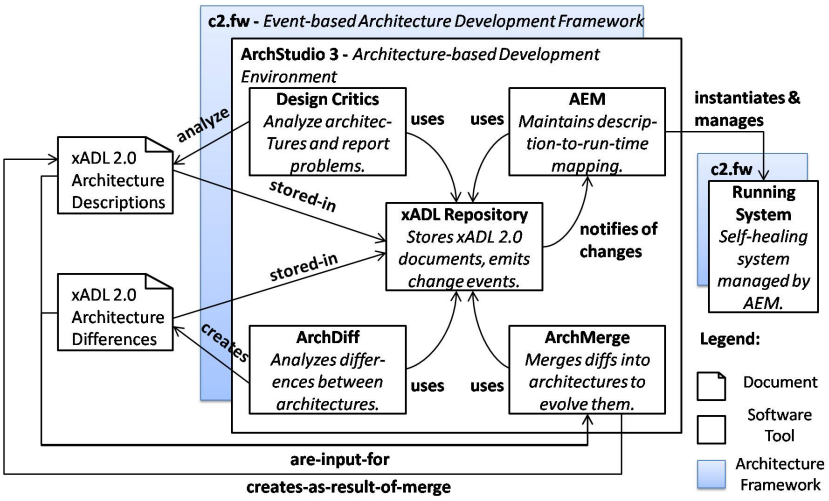


FIGURE 5.12 – Tools in ArchStudio to support dynamic evolution

1. The architecture model described in xADL 2.0 is instantiated into C2SADEL framework by the Architecture Evolution Manager (AEM).
2. The objective architecture model and the old one are inputs of the *ArchDiff* tool. ArchDiff creates an *architecture diff* that describes the differences between the two architectures.
3. The original architecture model is modified with the *architectural diff* using *ArchMerge*.
4. Changes in the new architecture model are applied to the running architecture system by AEM.

5.3.5.3 Version Model

In xADL2.0, the versioning of elements is not explicitly described in the architecture descriptions. The version information appears in each element type description (component type, connector type, and interface type). The version graph of these types expresses the relationship between these versions. Each version node in the version graph contains two elements : versionID and parent.

8. ArchStudio 3 is an outgrowth of the Arch- Studio 2.0 project [Khare01]

5.3.5.4 Discussion

xAdl 2.0 models two levels of software architectures : configuration and assembly. It really propagates changes from architecture configuration to assembly description. Its evolution is triggered by a given target architecture, which facilitates the support of adaptive architecture evolution. Unfortunately, as a suite project of C2sadel, it also fails to support reverse evolution (propagation from assembly to configuration). Table 5.7 gives an overview of evolution supported by xAdl 2.0.

Characteristics of Architecture Change	**Value**
Time of change	Dynamic
Anticipation	Unanticipated change
Change type	Structure
change purpose	—
Initial level of change	Configuration
Change operation and subject artifact	▶Adding and removing a component, ▶Adding and removing a connector, ▶Adding and removing a connection
Activities of architecture change	**Value**
Consistency checking	Sub-type checking
Impact analysis	—
Evolution test	—
Change propagation	Vertical propagation (top-down)
Versioning	State-based versioning

Table 5.7 – The characteristics and activities of change in xAdl2.0

5.3.6 Mae

Mae [Roshandel04, Westhuizen02] is an Adl merging with configuration management techniques to control and manage software architecture evolution. Its conceptual model is concretely realized using xAdl2.0. Beyond the characters of xAdl2.0, there are two main contributions of Mae : (1) using multi-version connectors (Mvc) to support component substitution, and (2) adding version models to the entire architectures to capture architecture evolution.

5.3.6.1 Component Substitution Based on MVC

Cook and Dage [Cook99] propose to keep multiple versions of a component running in a system and use arbiters to present to the system the image of a single component. Furthermore, Rakic [Rakic01] introduces this approach into the Mae Adl and uses connectors instead of arbiters. This multi-version approach improves the evolution success rate of component substitution. Firstly, it connects multiple versions of components at the same time and then it determines which one will be used. Secondly, it employs the sub-typing theory of Medvidovic *et al.* [Medvidovic98] as the substitution constraint. The substitution operation is then permitted only when a newer version of a component type

is both an interface and behavioral subtype of its predecessor. This constraint can be considered as a type consistency checking and guarantee that the component substitution will at least not violate the previous design decision. Figure 5.13 shows the example of changing the component *CompC* by the component *CompD*.

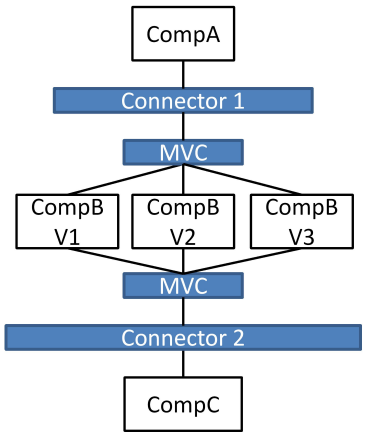


FIGURE 5.13 – The example of multi-versioned connector in MAE

5.3.6.2 Version Model

MAE supports two types of evolution versioning : linear evolution and diverging paths of evolution. Unfortunately, this version information is just applied to element types (component types, connector types and interface types).

For linear evolution, it uses versioning and sub-typing to capture this kind of evolution. Versioning is used to identify different (historical) incarnations of the same element. Sub-typing is used to annotate each change to indicate the nature of that change.

For diverging paths of evolution, it adopts interfile branching [Seiwald96] as applied to each of the types in the system model. Each architectural element definition is extended with the fields *ascendant* and *descendant*. Upon creation of a new branch, the ascendant of the new type is set to the original type. In addition, the set of descendants of the original type is updated with the new type.

Figure 5.14 gives an example of versioning for component type *CompC*.

```
1 name = CompC
2 revision = 3
3 ...
4 ascendant = { CompC 2 }
5 descendant = { CompD 1 }
6 subtype = { beh and int, CompC 2 }
```

FIGURE 5.14 – The *CompC* component type description focusing on version information

5.3.6.3 Discussion

MAE is qualified in two aspects : its version model for element types and multi-versioning connectors. MAE fails to model changes in architectures, and is thus not able to maintain a change-based version history. Furthermore, the evolution process concentrates on evolution testing but misses consistency checking and re-engineering. The synthesis of MAE can found in Table 5.8.

Characteristics of Architecture Change	Value
Time of change	Dynamic
Anticipation	Unanticipated change
Change type	Semantic
Change purpose	Perfective
Initial level of change	Configuration
Change operation and subject artifact	Replacing a component
Activities of architecture change	**Value**
Consistency checking	Sub-type consistency checking
Impact analysis	—
Evolution test	Yes
Change propagation	—
versioning	Available just for component type, connector type and interface type

TABLE 5.8 – The characteristics and activities of change in MAE

5.4 SYNTHESIS

After examination of the support of evolution in existing ADLs, we try to give a synthesis of our review. We consider two perspectives : evolution expressiveness, the support changes classified using the proposed taxonomy.

5.4.1 Evolution Expressiveness in ADLs.

To describe the evolution and software changes in ADLs, there are two important aspects that should be modeled : versions and architecture changes.

5.4.1.1 Version Model

Versioning problems including variants, evolving artifacts, forking and branching has been studied by software engineering for years, especially targeted to source code versioning. The main resolution is *configuration management*. However, versions of software architectures is less studied. Ménage [van der Hoek98a] firstly propose to introduce configuration management into ADLs to control the versioning of architectural artifacts. MAE and xADL2.0 both use the version model proposed by Ménage. However, they focus on the version control of architectural elements, such as component types and connector types.

The version of entire architectures is not studied. SOFA 2.0 enables to version software architectures which is treated as composite components, but their version model is too simple that cannot capture the enough information of evolution. A simple solution proposed by Taylor*et al.* [Taylor09] is to use CVS [Fogel99] or Subversion (SVN) [Pilato08]. But, this approach does not work well for creating or maintaining component-based software architectures, they are designed to capture the versioning of source codes.

From another side only MAE and xADL can express the version history with a branching version tree. This version tree is unfortunately applied only on architecture element types.

5.4.1.2 Change Description

The majority of existing ADLs do not support change description. Unicon [Shaw95b, Shaw96a] and C2SADEL [Medvidovic99b] are exceptions. Unicon uses change specifications to describe changes (create, remove, link, unlink). C2SADEL uses AML (architecture modification language) which focuses on describing changes to architecture descriptions [Oreizy98b, Medvidovic96, Agnew94]. It has more actions than Unicon (createComponent, createConnector, weld, unweld, etc.) [Oreizy98a]. However, these languages are imperative languages for modifying architecture, but no ADL, which supplies a full and formal description language of evolution covering its characteristic factors [Buckley05].

Furthermore, these existing ADLs miss another important point : no treat changes as first-class entities. The advantages of treating changes as first-class entities are : (1) trace software architectures evolution over time and (2) keep the change relationships between successive versions.

5.4.2 Evolution Supported

We compare existing ADLs with our proposed taxonomy from two perspectives : characteristics of architecture changes and activities of evolution process. Table 5.9 gives a comparison of ADLs presented in this chapter based on the proposed taxonomy.

Characteristics of Architecture Change	C2	Darwin	Dynamic Wright	SOFA2.0	xADL2.0	MAE
Time of change	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Anticipation	Unanticipated	Anticipated, Unanticipated	Anticipated	Anticipated	Unanticipated	Unanticipated
Change type	Structure	Structure	Structure	Structure	Structure	Semantic
Change purpose	—	—	—	—	—	Perfective
Level of change	Configuration	Configuration	Configuration	Configuration	Configuration	Configuration
Change operation and subject artifact	Adding and removing components, connectors or connections	Addition and removal of components or connections	Addition and removal of components or connections	Addition and removal of components, connections or interfaces of composite components	Addition and removal of components, connectors or connections	substitution of components
Activities of architecture change	**C2**	**Darwin**	**Dynamic Wright**	**SOFA2.0**	**xADL2.0**	**MAE**
Consistency checking	Refinement consistency checking	State consistency checking	Name, interaction and deadlock consistency checking	Behavior consistency checking	—	Sub-type consistency checking
Impact analysis	—	Horizontal impact analysis	—	—	—	—
Evolution test	—	–	—	—	—	Perfective test for component substitution
Change propagation	Horizontal propagation (top-down)	Horizontal propagation (to-down)	—	—	Horizontal propagation (top-donw)	—
versioning	—	—	—	State-based versioning	—	Change-based versioning

TABLE 5.9 – The comparison of characters and activities of change in existing ADLs

5.4.2.1 Architecture Changes Supported

By comparing existing ADLs from architecture changes they support, we find two problems :

- *Change type.* Most ADLs except MAE support only structure changes (substitution operation is considered as a combination of removal and addition). Without support of semantic changes, changes like component substitution cannot be applied correctly with evolution test according their evolution purposes.
- *Level of change.* The initial level of change is always restricted in the configuration level for existing ADLs. This hinders changes initiate in specification and assembly levels.

5.4.2.2 Evolution Process Supported

By examining existing ADLs through the activities of software evolution they support, we find that there is no ADL that has a complete evolution process.

- *Consistency checking.* The consistency checking of these ADLs is incomplete, none of them checks all consistencies of architectures : name, behavior, interface, interaction and refinement.
- *Evolution test.* Most of them do not support evolution test except MAE. However, MAE only tests component substitution.
- *Change propagation.* All existing ADLs support top-down propagation. None of them support bottom-up propagation.

In conclusion, there are important weak points in these works : (1) missing a full-scale change description embedded in the ADL as first-class entities and (2) lacking a complete evolution process covering from evolution planning, test, implementation, propagation to its re-versioning.

Chapitre 6

Evolution Process Based On Dedal

In the preceding Chapter 5, we presented the context of architecture changes and architecture-centric evolution. Then, we compared the existing dynamic ADLs in order to survey the nature of architecture changes they support, and the way these changes are supported.

In this chapter, we present our approach to support dynamic architecture changes. It has two objectives : provide a clearer description of software architecture changes that conveys the semantics of changes, and implement the corresponding architecture-centric evolution process based on Dedal. This chapter contains three sections. In the first section, we introduce the Dedal change description and how the architecture changes described in Dedal are implemented in an architecture-centric evolution process. In the second section, we present a runtime evolution framework to support the dynamic evolution. In the last section, we propose a gradual evolution process based on this runtime evolution framework.

Contents

6.1 ARCHITECTURE-CENTRIC EVOLUTION

6.1.1 Evolution Expression

The objective of this work is to be able to trace evolution and obtain a change-based software architecture version history. To do so, we define a version model that enables to represent each architecture element version with regards to its predecessor. Dedal change description is proposed that enables to trace the deltas between two successive versions of a given architecture element.

6.1.1.1 Version Model

Four types of artifacts can be versionned : abstract architecture specification, concrete architecture configuration, instantiated component assembly and component class. Version model of Dedal is applied to three architecture levels including architecture specification, configuration and assembly and one architecture constituent element : component class. The reason for versioning entire architectures has been already discussed in Section 5.4.1.1. Versioning architectures as a whole can help track the evolution of architectures over time. In our work, we can find that only versioning component class has two reasons. Firstly, our work targets to component-based softwares, connector classes are often automatically generated. Secondly, the versioning of component class can ensure the evolution test by explicitly defining the differences between two versions. For example, the version 2.0 is modified to perform better than version 1.0. This will be used in test the component substitution, which will presented in Section 6.3.2. However, our work is expressed in XML (see Chapter 7), thus the version model can be easily added to every element which is required to be versionned.

The versioning of architectural elements requires to record two pieces of information : the *versionID* and the *pre_version*. The versionID identifies the version. The information of pre_version links a version to its predecessor (the one it derives from). Pre_version is predecessor composed by the version name and versionID. If the version's name is the same as its predecessor's, the name can be omitted. The syntax for this version model is given in Fig. 6.1.

Figure 6.2 gives an example of version model for architecture configuration *BRSConfig*. In this description, we can see that the version name of the *pre_version* is omitted as they have the same name.

6.1.1.2 Dedal Change Description

In this section, we will present Dedal change description. A software architecture is derived from two phases : the development phase produces the original architecture, and the evolution phase produces the changed architecture. However, the evolution and architecture changes are not visualized directly in ADL description, as in most existing

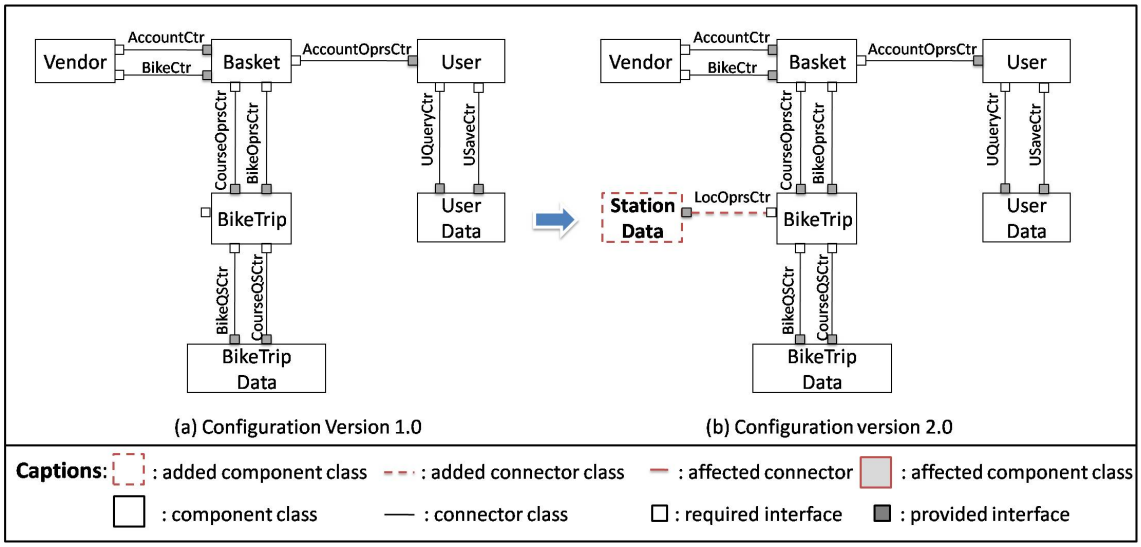
```
version : :=
   element_name ( revision_numb )

pre_version : :=
   ( element_name )? ( revision_numb )

element_name : :=
  specification_name | configuration_name | assembly_name
  |component_class_name

revision_numb : :=
  nonZero ( digit )* . ( digit )+
```

FIGURE 6.1 – Syntax of version




```
configuration BRSConfig
 implements BRSSpec (1.0)
 component_classes
  BikeTrip (1.0) as BikeCourse;
  BikeCourseDBClass (1.0) as
     BikeCourseDB
 versionID 1.0
```

```
configuration BRSConfig
 implements BRSSpec (2.0)
 component_classes
  BikeTrip (1.0) as BikeCourse;
  BikeCourseDBClass (1.0) as
   BikeCourseDB;
  StationData (1.0) as GIS
 versionID 2.0;
 pre_version 1.0;
 by additionStationDataList;
```

FIGURE 6.2 – Example of version modeling for the *BRSConfig* architecture configuration *BRSConfig*

ADLs, architecture changes are not treated as first-class entities. The reasons motivated the definition of Dedal change description are :

1. Having changes described in the same language as the first-class entities in order to (1) trace the evolution of software architecture and (2) make changes explicitly, and (3) enable the reuse of changes.

2. A full-scale change description engaging in software evolution is the prerequisite of change planning. The complete information of changes helps to find the impact of changes and then to plan the entire change process.

3. Architecture changes trace the changes that occurred between two successive versions enabling architecture changes visualization.

Thus, we propose Dedal change description which tries to fill this gap. Dedal change description has the following characteristics.

– Modeling software architecture by treating changes as first-class entities.
– Describing changes using the same syntax as that of the ADL language.
– Modeling changes from a semantic view rather than an operational view.

In order to model architecture changes from a semantic view rather than an operational view, we try to model the architecture change using the semantic of its nature characters described in our proposed taxonomy (see 5.2.2.1).

Change descriptions collect information of changes. Its syntax is defined in Fig 6.3. Figure 6.4 gives a change description of the change *additionStationData*. In this example, we can see this change aims to add the *additionStationData* component class.

```
1  change : :=
2   change change_name
3     time change_time
4     level initial_level
5     operation change_operation
6     artifact architecture_element is element_name
7     purpose change_purpose
8     origin change_origin ( from change ) ?
9  time : := static | dynamic
10 level : := specification | configuration | assembly
11 operation : := addition | removal |substitution | modification
12 architecture_element : := specification_element| configuration_element
   | assembly_element
13 specification_element : := component_role | connection |
   architecture_behavior
14 configuration_element : := component_class | connector_class
15 assembly_element : := component_instance | connector_instance |
   assembly_constraint
16 purpose : := corrective | perfective | adaptive
17 origin : := given | generated | propagated
```

FIGURE 6.3 – Syntax of change

The details of change aspects are in following :

```
change additionStationData
  time dynamic
  level configuration
  operation addition
  artifact component_class is StationData
  purpose perfective
  origin given
```

FIGURE 6.4 – The example of change description for change *additionStationData*

Change name. In Dedal change description, a change can be described in an independent file. This makes changes be able to reuse in different architectures.

Change time. Change time can be *static* or *dynamic*. When the changes are triggered at specification time during the development process, the architecture changes are static. Otherwise, they are considered dynamic.

Change level. Changes can be initiated from any of the three levels of architecture description : *specification*, *configuration* and *assembly*.

Change operations. Change operations can be either addition, removal or substitution, which can be applied in architectural elements, such as components. However, for the architecture behavior and the assembly constraint, as they are not independent description elements as components and connectors, thus authorized operations for them are modification. For components, we do not enter the detailed modification of their descriptions, we treat all these changes as component substitution.

Change artifact. Change artifacts are the elements which can be affected by change operations. Artifacts include all the architectural elements modeled in Dedal. In the specification level, they are *component roles*, *connections* and *architecture behavior* ; in the configuration level, they are *component classes* and *connector classes* ; in the assembly level, they are *component instances*, *connector instances* and *assembly constraints*.

Among these characters of changes, change level and change artifact should be coherent, that is, the change artifact should be included in this level. For example, in specification level, the change artifact must be component role, connection or architecture behavior. Secondly, different artifacts have their own permitted operations. For example architecture behavior can solely be modified. Their relationship can be found in Table 6.1.

Change purpose. The change purpose documents the motivation of change. It can be *corrective*, *perfective* or *preventive*. For component substitution, it can be used as the test criterion to evaluate whether the new version meets the change request. For the other kinds of changes, it is an important information which indicates the motivation of changes.

Architecture level	Change operation	Change artifact
Specification	Addition, removal and substitution	Component role
	Addition and removal	Connection
	Addition, removal and modification	Architecture behavior
Configuration	Addition, removal and substitution	Component class
	Addition, removal and substitution	Connector class
Assembly	Addition, removal and substitution	Component instance
	Addition, removal and substitution	Connector instance
	Addition, removal and modification	Assembly constraint

TABLE 6.1 – The change characters and its value defined in CDL

Change origin. Change origin indicates the source of the change.

- *Given changes.* When the change is prescribed by architects or maintainers, they are said to be *given* changes.
- *Generated changes.* When changes are consequences of given changes, they are said to be *generated* changes. It is the case, for example, when the addition of a new component role causes the addition of a connection between the new component role and the old component role. Table 6.2 lists all the possible generated changes for each kind of request change.

Architecture level	Request change	Generated changes
Specification	Addition of a component role	Addition of a connection, modification of the architecture behavior
	Removal of a component role	Removal of a connection, modification of the architecture behavior
Configuration	Addition of a component class	Addition of a connector class
	Removal of a component class	Removal of a connector class
Assembly	Addition of a component instance	Addition of a connector instance
	Removal of a component instance	Removal of a connector instance

TABLE 6.2 – The generated changes for different kind of request change

- *Propagated changes.* When the changes are propagated by given or generated changes, called *propagated* changes, such as an addition of a component class is propagated by a given change – addition of a component role. For the generated and propagated changes, the source changes should be indicated, using key word **from** (as shown in Fig. 6.3). Table 6.3 lists the propagated relationships between components and connectors in different levels.

6.1.2 Change-based Version Graph

By making the changes explicit and first-class, version graphs can be change-based. A version graph consists of nodes and edges which correspond to versions and their relationships, respectively. In change-based models, a version is described in terms of changes applied to some baseline. To this end, changes characterize the reasons and the nature of a change. Change-based versioning provides a nice link to change requests : A

Architecture level	Component	Connector
Specification	Addition/ removal/ substitution of a component role	Addition/ removal/ substitution of a connection
Configuration	Addition/ removal/ substitution of a component class	Addition/ removal/ substitution of a connector class
Assembly	Addition/ removal/ substitution of a component instance	Addition/ removal/ substitution of a connector instance

TABLE 6.3 – The propagated relationships between components and connectors in different levels

change request is implemented by a (possibly composite) change. Thus, a version may be described in terms of changes. A version v is constructed by applying a sequence of changes c_1 ... c_n to a baseline b : $v = c_1 \circ ... \circ c_n(b)$.

In our model, we try to build a change-based version graph for software architectures. In this version graph, there are two important relationships : (1) the predecessor relationship between two versions (sometimes called derivation) and (2) the implementation relationship between different levels of software architectures. The changes are expressed by its operation symbol (including addition, deletion, substitution and modification) and change name. The change with underline indicates this change is a request change. Fig. 6.5 shows an example of such a version tree for the BRS [1].

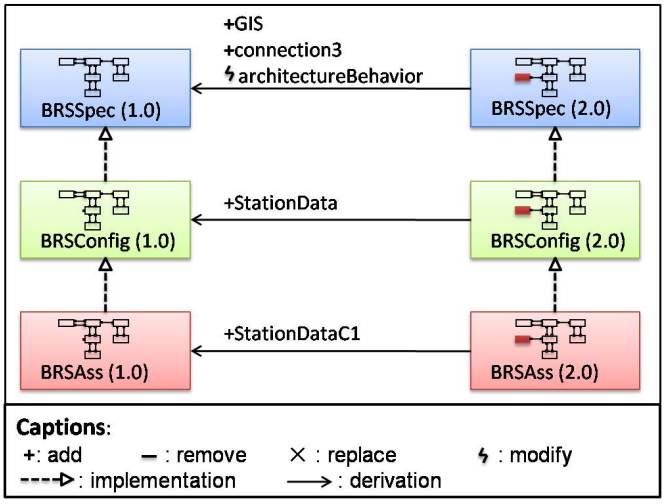


FIGURE 6.5 – The example of version for the BRS architectures

6.1.3 Evolution Process

In this section, we present an architecture-centric software evolution process based on Dedal. In this work, we highlight evolution from two aspects.

1. The studies about change-based version graph are our perspectives. As we know, there are many aspects of version graph that we are not considered, such as introducing branches etc.

– *Architecture-centric evolution.* The evolution is based on architecture models (three levels). Architectures are the underlying logical foundation of software evolution. Evolution is not only applied on runtime system but also can be initiated at any of the three levels of software architecture.
– *Autonomous evolution.* Evolution is autonomous, indicating that human involvement in the evolution process is absent, or at least greatly minimized.

The evolution process (shown in Fig. 6.6) contains three phases :

1. *Evolution planning* to analyze the change impact and check its consistency in each abstraction level of software,
2. *Evolution implementation* to prepare, test the change and implement it in implementation environment,
3. *Evolution re-engineering* to propagate change to **unchanged** levels and version software if necessary.

This evolution process is controlled by evolution management which contains architecture evolution management module and implementation evolution management module to govern the architecture models and implementation respectively.

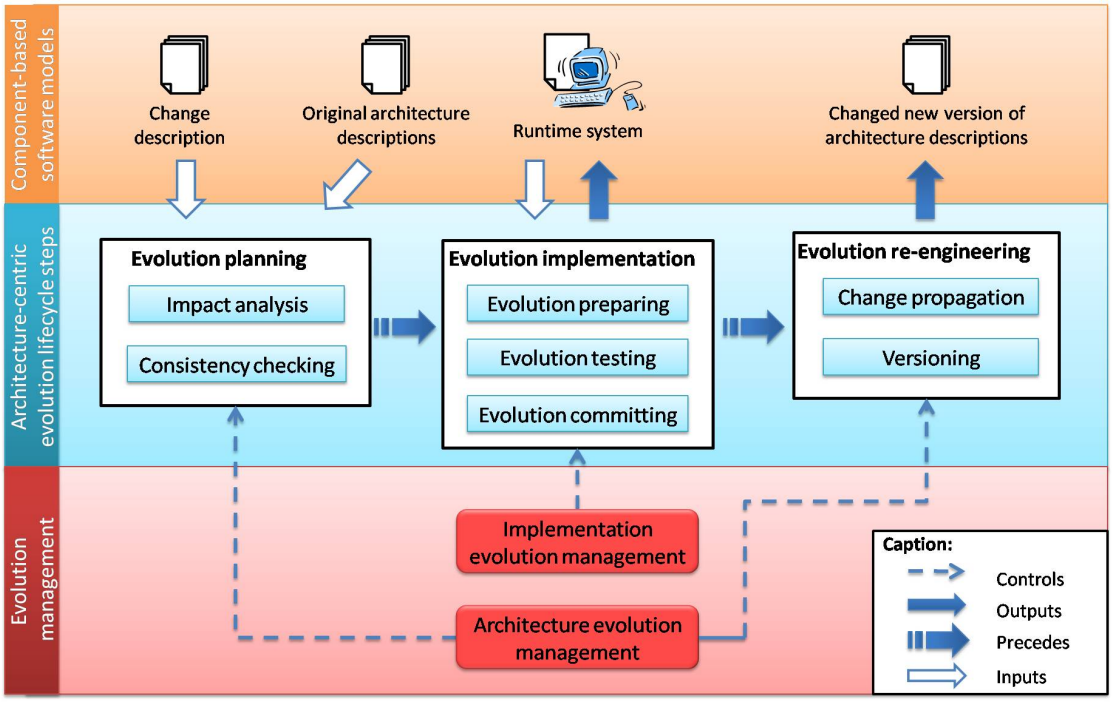


FIGURE 6.6 – Component-based software evolution process

6.1.3.1 Evolution planning

Evolution planning is the first phase of software evolution. It analyzes the change impact and checks the completeness and consistency of the target architecture by architecture evolution manager, as shown in Fig. 6.7. The first step of evolution planning is the *impact analysis*, which transforms the change request (expressed using Dedal change description)

into change lists by analyzing it and adding the necessary propagated changes. Secondly, the changes from the change lists are checked so as to see whether they are consistent with given architecture models, the feasibility of the thought evolution. If the changes are feasible, the evolution process will enter into the next step. If not, changes will be forbidden.

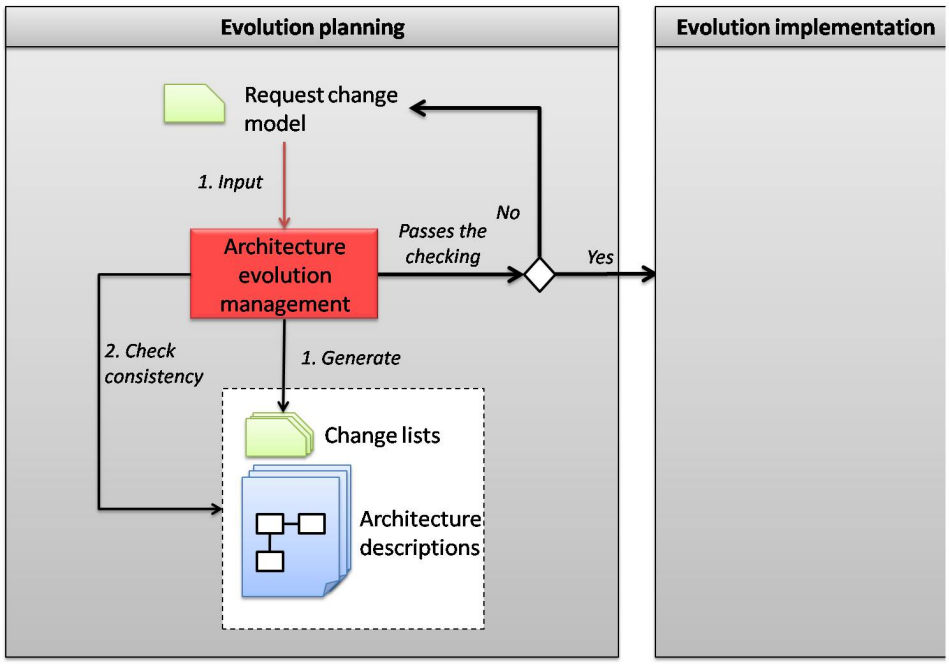


FIGURE 6.7 – The evolution planning phase

Impact analysis. During impact analysis, the architecture evolution management module produces the change lists from its input change request. A *change list* is the list of changes which should be performed by the evolution manager to modify the architectures. A change request can produce three change lists for architecture specification, architecture configuration and component assembly separately. Firstly, the request change is analyzed vertically and changes are generated for the architecture level in which the change request is performed. Secondly, this change list is analyzed horizontally to produce two propagated change lists for the other two levels. To propagate changes, we restraint the propagation only authorizes between the successive levels, as shown in Fig. 6.8.

For example, with the given change *addtionStationData* described in Fig. 6.4, the generated change list and the propagated change lists are listed in Table. 6.4.

Furthermore, the architecture evolution manager module needs to find the affected scope of the request change. Affected scope is a set of elements (components or connectors) of the system that are most likely to be affected by the change request. Such as, for addition of a new component, the components which connect with new component have the closest relationship with it, thus they are most possibly to be affected by added component. This information of affected scope serves to test evolution in evolution implementation phase.

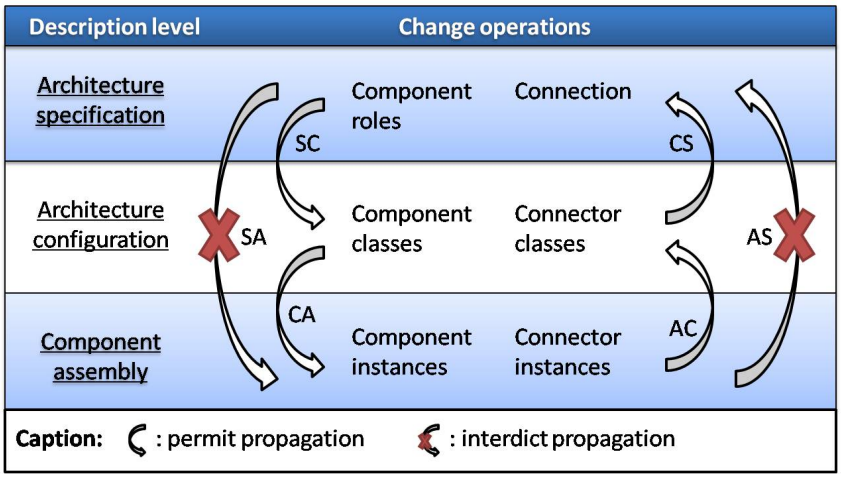


FIGURE 6.8 – Propagation of changes between the three description levels

Change lists	Changes
Given change	Addition of a *StationData* component class
Generated change list	Addition of a *LocOprsCtr* connector class
Propagated change list	Specification :
	Addition of a *GIS* component role
	Addition of a *LocOprscnt* connection
	Modification of the architecture behavior
Propagated change list	Assembly :
	Addition of a *StationDataC1* component instance
	Addition of a *LocOprsCtrC1* connector instance

TABLE 6.4 – The propagated change lists for change request *additionStationData*

Consistency checking. The analysis of architecture model can have different goals, including early estimation of system size, complexity, and cost ; adherence of the architectural model to design guidelines and constraints ; satisfaction of requirements, both functional and non-functional ; assessment of the implemented system's correctness with respect to its documented architecture ; and so forth [Taylor09]. It can be categorized into four goals : completeness, correctness, consistency and compatibility [Taylor09]. In our work, we try to examine architecture descriptions with regards to the consistency goal.

Consistency is an internal property of an architecture model, which intends to ensure that different elements of that model do not contradict with one another [Taylor09]. The aim of consistency checking is to predict whether changes induce inconsistencies inside and among the three levels of a given architecture. We talk about intra-level (vertical) consistency checks and inter-level (horizontal) consistency checks. If changes preserve consistency, the thought evolution will be permitted. If not, it will either be forbidden or trigger the derivation of a new architecture version for which consistency will be ensured (using change propagation).

Consistency checking covers five types of inconsistencies which can be consequences of changes :

– *Name inconsistency.* The components or connectors have the same name as after

the addition of a new component or connector. This is an intra-level consistency checking.

- *Interface inconsistencies.* Connect interfaces are inconsistent, when their types are not compatible. Interfaces consistency calculus can be automated as previously studied in Arévalo *et al.* [Arévalo09], for example. This is an intra-level consistency checking.
- *Behavior inconsistency.* Behavior inconsistencies can occur when the behavior of newly added component calls an unconnected interface. Furthermore, when behavior of two connected components are not compatible, behavior inconsistencies can occur also. For the second situation, we reuse the work of Plasil *et al.* [Plasil02] on behavior protocol comparisons. This is an intra-level consistency checking.
- *Attributes inconsistencies.* They are detected automatically at the assembly level, to check whether the current state of the component violates the assembly constraint defined in assembly description. For example, if an assembly constraint defines that the attribute *Bank* of component instance should be *BNP*, the newly added component instance cannot possess another value. This is an intra-level consistency checking.
- *Interaction inconsistency.* Interaction inconsistencies can occur between connected architectural elements, when they violate the interaction protocols of systems (often defined in architecture style). For example, two client components cannot connect directly in client-server architecture style. By default, three interaction rules are checked in Dedal : (1) Two components cannot connect directly besides the delegation situation, they must be connected through connectors ; (2) In the delegation situation, two connected interfaces should be the same type ; (3) Besides delegation situation, a provided interface must connect with a required interface, and vice versa. This is an intra-level consistency checking.
- *Mapping inconsistencies.* Mapping inconsistencies occur between two successive levels of the description of level software architectures. Mapping inconsistencies checking that occurs between specification and configuration levels, are detected using the component specialization rule exposed in Sect. 4.3 Mapping inconsistencies checking that occurs between configuration and assembly levels, are detected using the component specialization rule exposed in Sect. 4.4.3.3 and Sect. 4.4.3.2. This is an inter-level consistency checking.

6.1.3.2 Evolution Implementation

Evolution implementation aims to test evolution at runtime to assure the feasibility of changes for the running system. There are two steps (shown in Fig. 6.10) : (1) transform the change description list for the assembly into change transactions and (2) implement gradual evolution process.

Change transaction. Firstly, the change list of the assembly level expressed in Dedal change description, is transformed into change transactions. *Change transactions* are the operations that can be directly applied to the runtime system. The permitted change transactions are addition and removal of components and connectors, and component substitution, as shown in Fig. 6.9.

```
1 add(ComponentInstance A);
2 delete(ComponentInstance A);
3 replace(ComponentInstance newA, ComponentInstance oldB);
4 add(ConnectorInstance A);
5 delete(ConnectorInstance A);
```

FIGURE 6.9 – The change transactions

Gradual evolution process. Gradual evolution process concerns evolution preparing, testing and committing. It is controlled by the implementation evolution manager. The main idea is to get the original system evolve into a target system through a transitional step. During the transitional step, a transitional assembly is produced to test the proposed changes in the real execution environment and either validate the evolution or invalidate it to return to the original state.

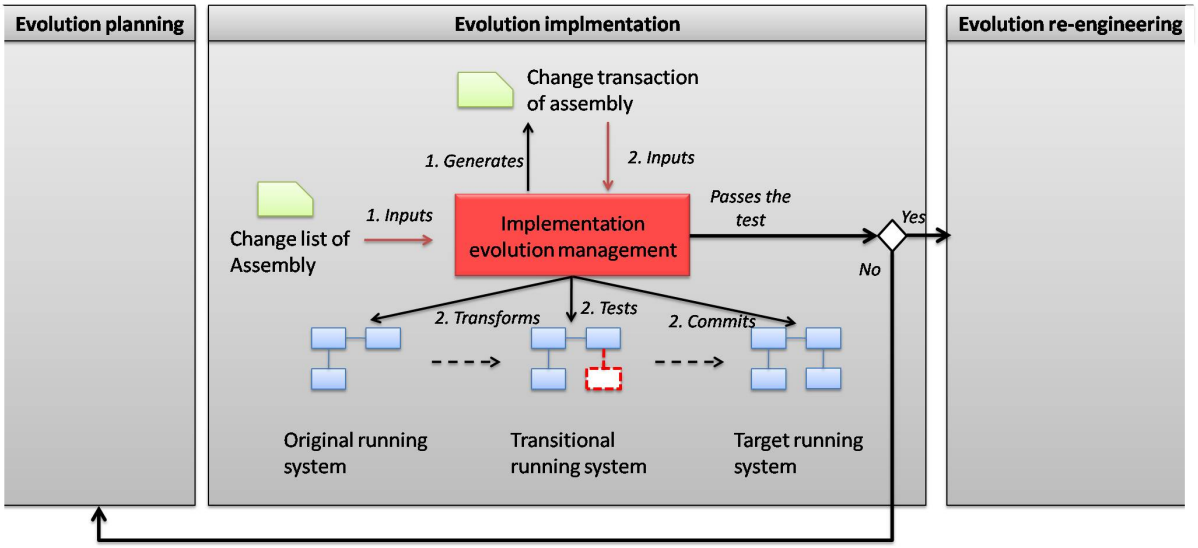


FIGURE 6.10 – The evolution implementation phase

The objective of the evolution preparing step is to change the original system into the transitional system. The deployment of change description first transforms change description into change transaction – the executive programs that operate on a system to make changes. It is composed of basic operations : deploy (deploy the component into runtime environment), mutate (mutate the component instances according to the assembly rules from the assembly level or directly from old existing component instance), add, delete, connect, disconnect. Evolution test uses the connector-driven gradual assembly evolution process [Zhang08, Zhang09]. If the changes are validated by the implementation evolution management module after evolution has been tested, changes will be committed in the running software system. Otherwise, the software system will roll back to the recorded safe state, and the evolution will roll back to the evolution planning phase.

6.1.3.3 Evolution Re-engineering

The architecture-centric evolution re-engineering involves modifying architecture descriptions, versioning the modified architectures, as shown in Fig. 6.11.

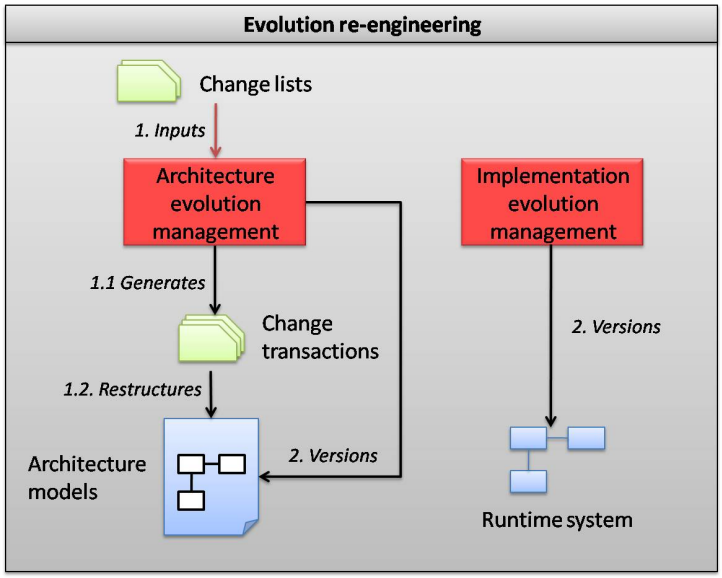


FIGURE 6.11 – The evolution Re-engineering phase

Change propagation. Architecture evolution manager module treats the thought evolution as being part of a new architecture version. This is solved using change propagation techniques [Rumbaugh88, Urtado98]. The new architecture version is derived and its content inferred so as to be consistent with the thought change. The information necessary to derive new versions on each of its levels is extracted from lower to higher levels as shown in Fig. 6.8.

Versioning. The version model of Dedal is a changed-based version model which maintains version trees for component roles and component classes and component instances and for specifications, configurations and assemblies. In each level, versions record three types of information : version ID, previous version ID and change operations. Few ADLs enable to describe changes, except C2 [Oreizy98b, Medvidovic96] and Darwin [Magee95, Magee96] which both use change transactions that apply on the runtime system. They have no formal description of changes as first class information in ADL syntax. In our work, changes can be described as first class information with change operations. Changes from a version to the next (delta) are described as a list of change operations. Permitted change operations are listed in Table 6.1.

Figure 6.13 shows the version model of *BRSSpec (2.0)* which is composed by a version ID *2.0*, previous version ID *1.0* and a list of change operations *addition of StationData*.

As changes are first class information, it becomes easy to draw a change-based version tree of software systems according to their architecture descriptions. Fig. 6.5 shows an example of such a version tree for the BRS.

6.1.3.4 Illustrative evolution scenario for the BRS

This evolution scenario aims to add a component class (*StationData*) to the BRS software configuration. This new component class connects with *BikeTrip* by interfaces *LocOprs* (see Fig. 6.12).

```
1 component_class StationData
2   implements GISType
3   using fr.ema.StationDataImpl
4   provided_interfaces LocOprs
5   versionID 1.0
```

FIGURE 6.12 – The description of component class *StationData*

Consistency checking. Consistency is checked for the configuration. Interface consistency checking analyses that the unused *LocOprs* interface of *Biketrip* component class is compatible with the *LocOprs* interface of *StationData* component class. Behavior consistency checking concludes that behaviors are consistent. Indeed, the *BikeTrip* component behavior has the capability of using the *findStation* method of the *LocOprs* provided interface that is described in the *StationData* component class and the *StationData_1.0* component class behavior imposes no extra constraints.

The consistency of the configuration towards its specification is also checked. The new configuration proves to be inconsistent with its specification. As we want the thought change to be committed, propagation is needed.

Change propagation. Change propagation results in adding elements to the newly derived specification versions : the *GIS* component role, the *connection3* connection and the *LocOprs* required interface to the *BikeCourse* component role. Changes are also propagated to the assembly and result in the addition of the *StationDataC1* component instance and of the *LocOprsCon* connector instance (which is automatically generated). Changes are shown in Fig. 6.13, Fig. 6.14 and Fig. 6.15 (underlined text highlight modifications).

Versioning. The three levels of the BRS architecture are versioned as shown in Fig. 6.5. Possible changes include addition, removal and substitution (of components and connectors, either they be described by their roles, classes or instances) which impact the structure of software architectures, and modification of the architecture behaviors.

6.1.4 Summary

In this section, we present the evolution expression of Dedal and the proposed architecture-centric evolution process based on Dedal. The objective is to version global architectures and to prevent architecture erosion and drift by the architecture-centric evolution process.

```
specification BRSSpec
  component_roles
    BikeCourse; BikeCourseDB; GIS
  connections
    connection BQueryCnt
      client BikeCourse.BQuery
      server BikeCourseDB.BQuery
    connection BSaveCnt
      client BikeCourse.BSave
      server BikeCourseDB.BSave
    connection connection3
      client BikeCourse.LocOprs
      server GIS.LocOprs
  architecture_behavior
    (!BikeCourse.BikeOprs.selectBike;
    {?BikeCourse.LocOprs.findLoc;
    !GIS.LocOprs.findLoc;}
 ?BikeCourse.BikeQS.findBike;
 !BikeCourseDB.BikeQS.findBike;)
    +
    (!BikeCourse.CourseOprs.startC;
 ?BikeCourse.BQuery.findCourse;
 !BikeCourseDB.BQuery.findCourse;)
  versionID 2.0;
  pre_version 1.0;
  by additionGISList;
```

FIGURE 6.13 – The abstract architecture specification description after evolution

```
configuration BRSConfig
  implements BRSSpec (2.0)
  component_classes
    BikeTrip (1.0) as BikeCourse;
    BikeCourseDBClass (1.0) as BikeCourseDB;
    StationData (1.0) as GIS
  versionID 2.0;
  pre_version 1.0;
  by additionStationDataList;
```

FIGURE 6.14 – The concrete architecture configuration description after evolution

Evolution expression in Dedal. Dedal enables to version the global architectures and describe the versions and changes of architectures as first-class entities. This facilitates to trace the evolution history of architectures, and also makes Dedal be nature to support evolution.

Architecture-centric evolution process. Our architecture-centric evolution process supports forward and reverse evolution. It is composed by three phases : evolution planning, implementation and re-engineering. Evolution planning ensures the correctness and the feasibility of changes by examining the consistencies of architectures. Evolution implementation tests changes on the running system to ensure the adaptabilities of changes.

```
1 assembly BRSAss
2   instance_of BRSConfig (2.0)
3   component_instances
4     BikeTripC1 as BikeCourse; BikeCourseDBClassC1 as BikeCourseDB;
5     StationDataC1 as GIS
6   assembly_constraints
7     BikeTripC1.currency=="Euro.";
8     BikeCourseDBClassC1.company==BikeTripC1.company;
9     MaxInstanceNbr(BikeGUI)==8 && MinInstanceNbr(BikeGUI)==5
10  versionID 2.0
11  pre_version 1.0;
12  by additionStationDataC1List;
```

FIGURE 6.15 – The instantiated component assembly description after evolution

Evolution re-engineering propagates changes to architectures in three levels and reversions them when necessary, in order to prevent architecture erosion and drift.

6.2 RUNTIME EVOLUTION FRAMEWORK

In previous section, we discussed architecture-centric evolution process with the proposed ADL. The descriptions of architectures provide an abstract view. The following issues arise :

- How to connect software runtime system with the architecture descriptions (the configuration level) ?
- How to make runtime softwares support the evolution process we proposed ?

In this section, we will present our runtime evolution framework, which the meta-level framework that provides components & connectors with the capability to evolve dynamically.

6.2.1 Overview of Runtime Evolution Framework

The runtime evolution framework (REF) is the meta-level information that components & connectors embed for the management of runtime evolution. The objectives of this framework are to preserve the coherence between architecture models and runtime system after evolution, and to support and control the dynamic evolution process.

In this framework, a software system is defined as a collection of *components* and *connectors* which are contained in and managed by a *container*. These three types of entities are defined as *framework entities* in the REF. The container can be seen as the top component in the system composition hierarchy. Components and connectors are the basic constituent blocks of softwares. They are runtime counterparts of component and connector instances defined in the corresponding Dedal description.

Framework entities in the REF are composed of two parts : *core* and *extension*. *Core elements* consist of classes which describe the basic, invariable part of the different kinds of architectural entities, *i.e.,* their functional concerns. *Extension elements* are added to architectural entities in order to support the non-functional concerns which apply specifically, depending on the management policies defined by ADL descriptors. Extension elements consist of meta-objects (*controllers* and *descriptors*). Figure 6.16 shows the class of framework elements.

In our runtime model, controllers are structured into three levels. *Evolution manager* is a top-level controller that controls the entire configuration evolution process by managing connectors and components. *Connector controllers* and *StateTrace controllers* each control local evolution actions (respectively on connectors and on components). *Dataflow* and *time controllers* are third level controllers that are used by *Connector controllers* to control and collect data on connectors.

The descriptors are distributed to each element to store their descriptions. These descriptions can be automatically updated by the descriptors when a change occurs for the element.

6.2.2 Container

A *container* can be seen as the component at the top of the composition hierarchy. It contains the description of the entire runtime software system and thus contains a set of *connectors* and *components* which describe a system as an assembly (see Section. 4.4). The role of the container is to control and manage the system's evolution and serve as a bridge between the configuration level and the assembly level. It is extended by a *container*

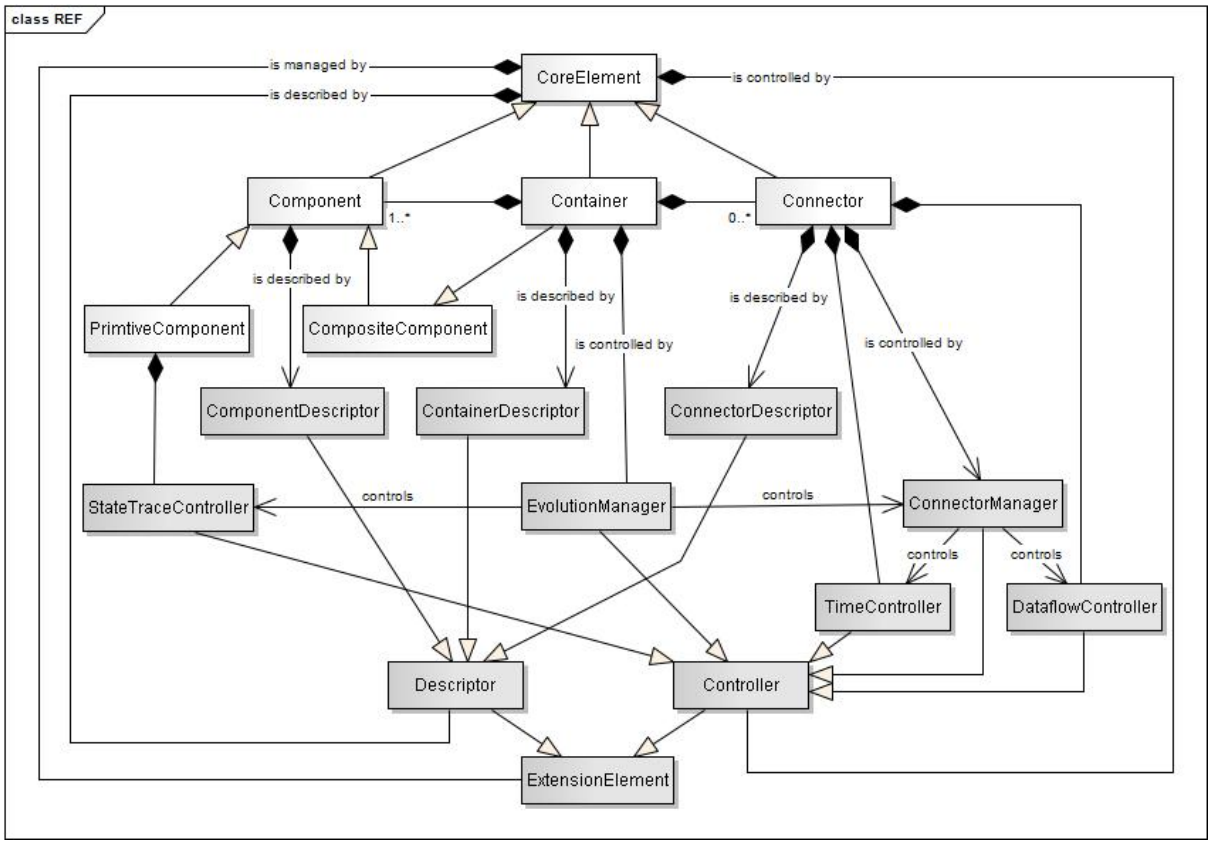
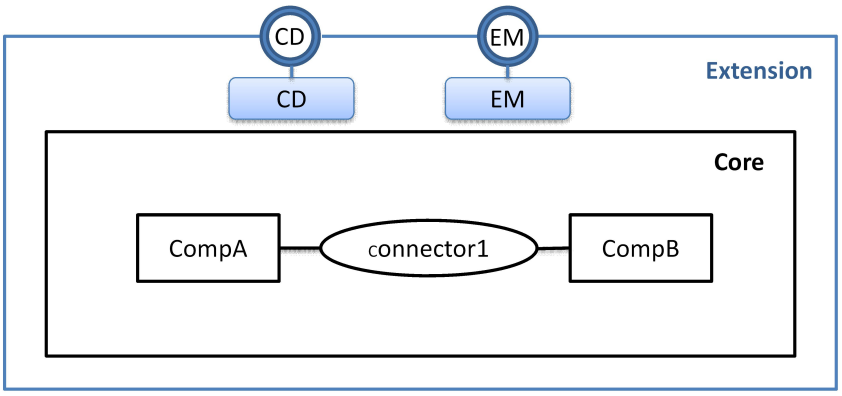


FIGURE 6.16 – Class model of connectors

descriptor and an *evolution manager*. Fig. 6.17 gives a graphic view of container.



Caption:
CD: container descriptor EM: evolution manager

FIGURE 6.17 – Model of container

The *Container descriptor* embeds information on the runtime software system (the assembly description). It can automatically re-generate this description after each change of the software runtime system.

The *evolution manager* manages the evolution of its inner components and evaluates the feedback from its inner connectors. It controls the evolution process from the top view

and manages the changes of the entire system. Its tasks obey two objectives : control the evolution and perform changes.

- *Control evolution.* It should be able to manipulate the entire evolution from planning to test and verification, which bases on assembly level.
- *Perform changes.* It can perform changes on the runtime system, including adding, deleting and replacing components, connectors or connections.

The functions provided by evolution manager can be found in Table 6.5.

The functions of evolution manager	
Control evolution	**Perform changes**
Changes checking	Instantiate component
Evolution planning	Component/connector/connection addition
Evolution test	Component/connector/connection removal
Evolution verification	Component/connector/connection substitution

TABLE 6.5 – Functions provided by evolution manager

6.2.3 Components

Components are the bases of hierarchical composition. There are two kinds of components : *primitive* and *composite*. A primitive component is a leaf component that is directly implemented. A composite component is built from the interconnecting of smaller components and connectors as a substructure. Each component is extended by a component descriptor. A *component descriptor* contains the component ADL description of a component, either primitive or composite. The primitive components are extended by *StateTrace controllers*. A component instance is depicted in Fig. 6.18.

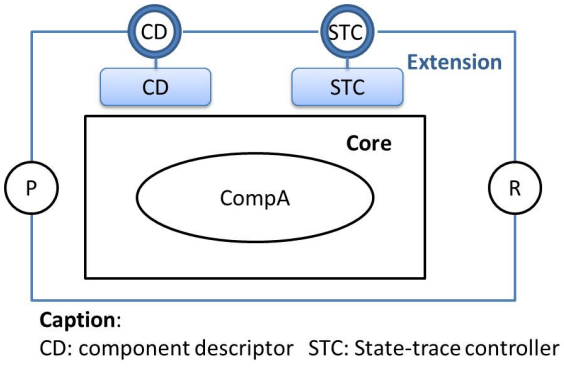


FIGURE 6.18 – Instance view of component

6.2.3.1 StateTrace Controller

StateTrace controllers are used to prevent component breakdown during the evolution process. In order to achieve this objective, the controllers perform two main functions :

- They keep track of the successive states of primitive components each time they are modified, when this function is active.

– They act so as to recover from a failure and put back the components into a sound state to prevent system breakdown. The user can choose the state in which he/she want components to roll back to.

The state trace is realized by recording and re-assigning the values of component instances' attributes. The state of component instance at the moment t is composed by the attributes and their values at the moment t. StateTrace controllers record the state of component instances at each time when one or more of its attribute values are changed at the moment t. When component instances are required to roll back to state of the moment t_i, the stateTrace controllers re-assign the attributes with their values at the moment t_i.

6.2.4 Connectors

Connectors govern communications between components. A connector instance is depicted in Fig. 6.19. Connectors can be automatically added a new interface when a new connection has to be established. A connection is the binding between provided and required interfaces of components and connectors. Indeed, components should not communicate directly by referencing each other. Instead, they should use connectors which minimize coupling between components and enable binding decisions to change without requiring component modification. The interfaces of two components which are connected by a connector need to be compatible [Arévalo09]. The connection should be chained as is : a component's required interface connects to a connector's provided interface and the required interface of the connector connects with another component's provided interface.

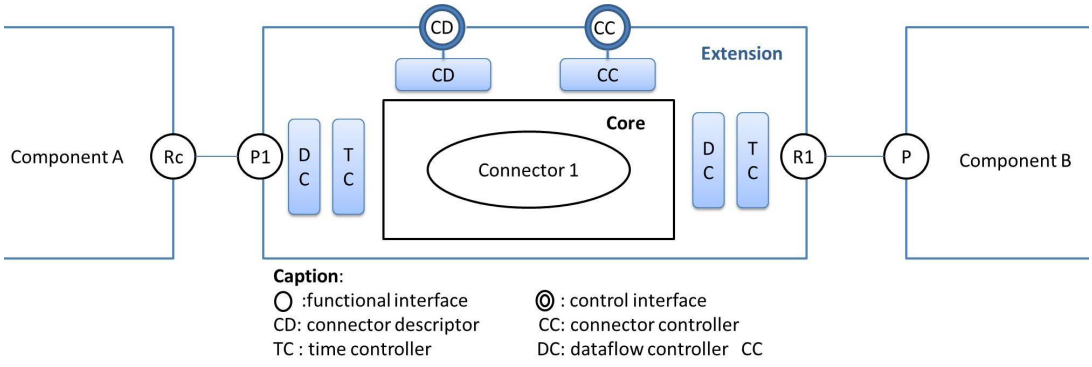


FIGURE 6.19 – Instance view of connector

Connectors are extended to manage and control the evolution of each connected component, control the dataflow between components and collect test samples of component inputs and outputs. The extension of connectors consists of a connector descriptor and three controllers : the *connector controller*, the *dataflow controller* and the *time controller*. Their functions are listed in Table 6.6 and more throughly described below.

6.2.4.1 Connector Controller

The connector controller controls evolution procedures for the connected components. It is the control center of this connector, which directs the executions of dataflow and time controllers. The main objective of this controller is to control and manage the

Connector controller	Dataflow controller	Time controller
Add new interface	Control dataflow of the connector	Record input/output time of passing messages
Delete unused interface	Record messages passing the connector	
Control the local evolution of connected components		

TABLE 6.6 – Functions provided by connector, dataflow and time controllers

local evolution of its connected components. It directly communicates with the evolution manager to receive beginning and finishing commands and send back the collection information from dataflow and time controllers.

It can add a new interface to the connector at runtime, to enable its connection with two versions of a same component at a time. It also can delete an interface from the connector at runtime. It controls the evolution process of the connected components, such as starting and finishing the test and evolution. It controls the evolution test by controlling the dataflow and time controllers to start and finish data collections.

6.2.4.2 Dataflow Controller

The dataflow controller controls the dataflow that traverses the connector and records messages from each connected component to compare them in case evolution is declared to be corrective (see Chapter 6).

Firstly, it controls the dataflow of this connector by opening and closing the pipelines of its interfaces. For example, it can make two provided interfaces receive the same input messages, but just return a single response to the connected required interface. Secondly, when the evolution purpose is corrective, this dataflow controller will be active to record the messages passing from the connectors. This information can be used to compare different versions of a given component to ensure the new version performs better than the old one.

6.2.4.3 Time Controller

The time controller records the input and output time stamps for each message that traverses the connector to compare them in case evolution is declared to be perfective. It will be active then. It also records the input and output time stamps for the messages.

6.2.5 Summary

In this section, we presented our proposed runtime evolution framework (REF) – a meta-level information that embeds in components and connectors to make them support runtime evolution. The objective of this REF is to preserve coherence between architecture descriptions and runtime system, and to make runtime system be able to automatically control the dynamic evolution and restructure itself.

In this REF, a system is modeled as a collection of components and connectors which are contained in a container. Each of them contains two parts : a *core* and an *extension*. The meta-information is embedded in the extension. The extension is composed by descriptors and controllers. The descriptors contain the architecture information of its contained components or connectors. The controllers are used to control and manage the runtime evolution.

6.3 GRADUAL EVOLUTION PROCESS AT ASSEMBLY LEVEL

In Section 6.1, we talk about the architecture-centric evolution process. In Section 6.2, we present a runtime evolution framework, which enable the elements of runtime system have the ability to support the dynamic evolution. The gradual evolution process presented here is based on this runtime evolution framework. This REF can autonomically handle the reconfiguration of runtime systems including the addition, removal and substitution of components in a gradual, transparent and testable manner. Hence, the system has the choice to commit the evolution after a successful test phase of the software or rollback to the previous state. In this section, we will present this gradual evolution process by describing three main operations of evolution : addition, removal and substitution of component instances.

6.3.1 Overview of Gradual Evolution Process

One of the main ideas exposed in this section is to have the original assembly of the system evolved into an objective assembly through a transitional assembly. The transitional assembly is transformed from the original assembly by merging changes into the assembly without deleting any system elements. For example, to replace one component with its new version, the transitional assembly makes two versions of the component co-existing and connecting in the system at the same time. The transitional assembly aims to test new component versions and either validate the evolution (commit changes) or invalidate it to rollback to the original state.

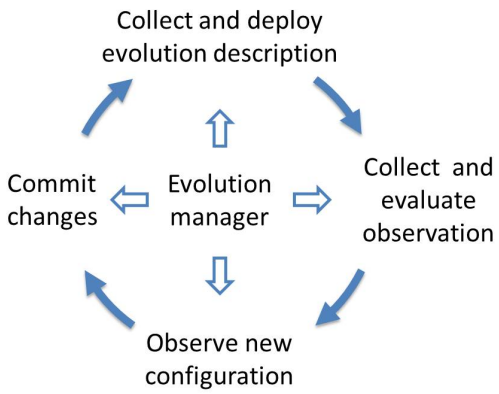


FIGURE 6.20 – Life cycle of evolution actions

To achieve this, we propose a four step evolution process controlled by an evolution manager, depicted in Fig. 6.20 :

- The *preparation stage* collects evolution description and builds the transitional assembly.
- The *test stage* collects observations from newly introduced components and components affected by changes (those directly connected to changing components). By analyzing these observations, it determines :
  - In case of *component addition*, if new components work correctly.
  - In case of *component substitution*, if new component versions meet the requirements

induced by the declared evolution purpose.
  - In case of *component removal*, if the new assembly is stable.
- The *observation stage* maintains original component versions as backups in case new versions cause failure.
- The *commit stage* enacts changes (disconnect the now useless original versions).

Each stage obeys two phases : evolution and validation. The evolution phase defines how connectors drive the evolution process during each stage. The validation phase determines if the activities performed during each stage are valid or not.

6.3.2 Component Substitution

Component version substitution potentially affects four elements of components : their name, interface, behavior protocols or implementation (as described by Palsberg and Schwartzbach [Palsberg92]). In our system, component substitution amounts to replace an old version of a component by one of its newer versions in order to meet the declared evolution requirements (corrective evolution, perfective evolution or corrective&perfective evolution). Thus, the test focuses on the three evolution motivations which affect the measures of our evolution manager, which can perform to guarantee the feasibility of the thought evolution.

Example. The objective of this evolution is to replace the *BasketC1* component (version 1 of the *Basket* component) by the *BasketC2* component (its version 2). It implies changing the component assembly of system from the original assembly represented in Fig. 4.25 in Chapter 4. to the objective assembly represented in Fig. 6.21 (b). Evolution concerns the *Bike* provided interface and its evolution purpose is corrective. Contrary to most evolution approaches, we propose to deploy a transitional assembly (*cf.* Fig. 6.21b) to smoothly switch from an assembly to another through a test period.

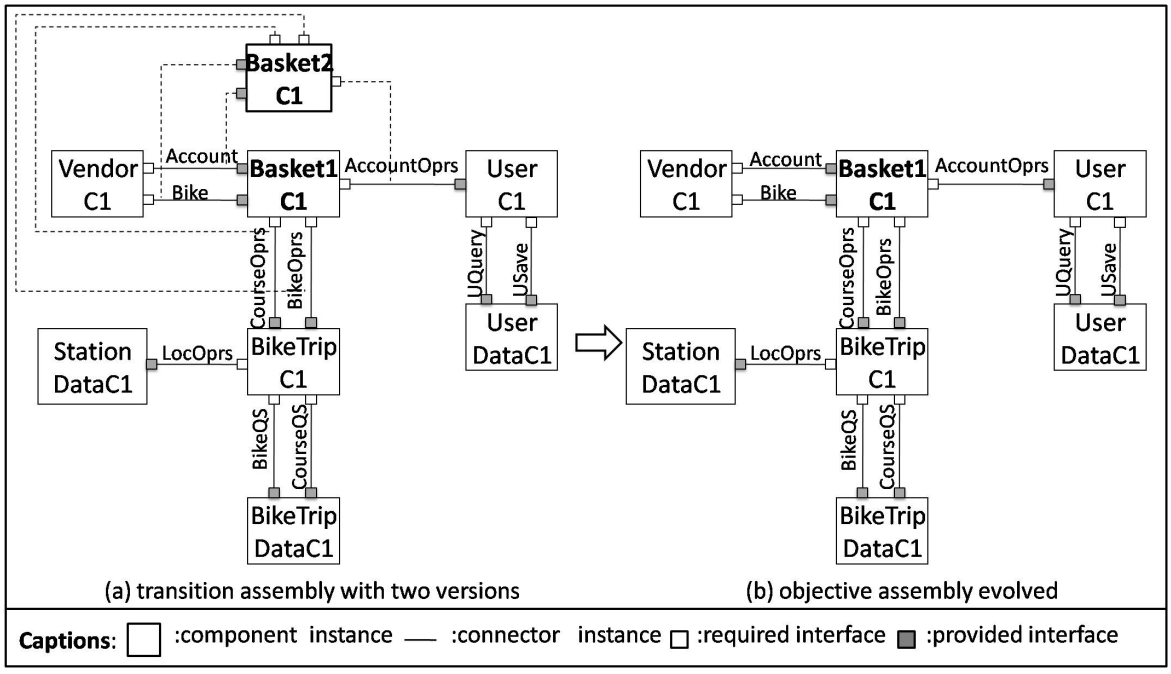


FIGURE 6.21 – Example of component substitution evolution

6.3.2.1 Preparation Stage

Preparation is a vital aspect of the evolution process. Its objective is to change the original assembly into the transitional assembly by collecting and deploying evolution description.

The first step is to collect evolution description. Evolution description must define what to evolve and how. What to evolve refers to (1) references of old and new component versions, (2) interfaces changed and their change purpose (corrective or perfective), and (3) change test condition for corrective evolution. How to evolve refers to the number of collected samples that are to be used during tests. In the example of Fig. 4.25, the *Bike* interface of *Basket* is changed to correct (corrective evolution) its protocol for *Basket* version 2 with the following condition : if type of bike is tandem then price is twice the normal price. The collected evolution description is shown in Table 6.7.

Secondly, the *evolution manager* deploys evolution descriptions according two connector groups : changed group and unchanged group. *Changed group* contains connectors which connect to changed interfaces like *BikeCtrC1*, and *BikeOprsCtrC1*. These connectors drive the evolution process. *Unchanged group* contains all other connectors like *AccountCtrC1*, *CourseOprsCtrC1*, etc.

At the validation step, the evolution manager collects feedback from all connectors. If all connectors connect successfully with the new component instance version, the evolution process passes to next stage.

Components	**Old**	BasketC1 (version 1)
	New	BasketC2 (version 2)
Change interface	**Purpose**	**Condition**
Bike	perfective	The *returnBike* method of new version C2 has a short response time than version C1.
Bike	corrective	**if** bike∈Tandem **then** BasketC2.Bike.returnBike(bike)= 2*BasketC1.Bike.returnBike(bike)

TABLE 6.7 – Evolution description of example

6.3.2.2 Test Stage

Test stage aims to examine the behavior of new component version according to its evolution purpose. Test stage thus involves collecting observations on new component versions in connectors and evaluating these observations in the evolution manager. The measurement of new component versions focuses on functioning and are decomposed into to two sub-stages :

- *Offline test* is focused on the functioning of the new version, keeping it transparent to the system. The old version is *dominant*, which means that connectors just propagate old version's results to the rest of system, while new version is insulated from the system.
- *Online test* aims at adapting the system to new version : new version becomes dominant. As this entails risks of system failure, a state-trace mode is activated. All

components' StateTrace controllers record the state modifications for each component. When critical errors happen, the whole system rolls back to its previous safe state, as every component rolls back to its previous safe state.

Each test of a changed interface is controlled by the connector which connects with this interface. This includes dataflow control and data collection. Thus, according to the direction of interface (provided or required), test and evaluation behave differently. We use a part of the example to explain how dataflow is controlled by connectors (Fig. 6.22). The *Bike* (P1) interface of component *BasketC1* illustrates provided interface evolution, and its *BikeOprs* (R1) interface illustrates required interface evolution.

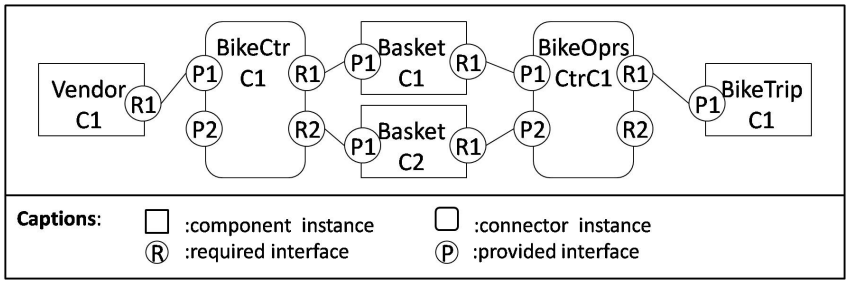


FIGURE 6.22 – The example of component substitution

Provided Interface Evolution. Provided interface evolution means that a component's provided interface is evolved with either a corrective or a perfective purpose. The *BikeCtrC1* connector (*cf.* Fig. 6.22) collects test data and controls the evolution of the *Bike* interface of the *BasketC1* component.

During offline test, component *BasketC1* is dominant. The dataflow controller of the *BikeCtrC1* connector distributes incoming calls to two versions, while all connectors block outgoing calls from *BasketC2* to keep it transparent to the system, as shown in Fig. 6.23. For example, the *BikeCtrC1* connector blocks the response of *BasketC2* and the *BikeOprsCtrC1* connector blocks the incoming calls of *BasketC2*. During online test, the situation is reversed : *BasketC2* becomes dominant, as shown in Fig. 6.23.

Required Interface Evolution. Required interface evolution means that a component's provided interface is evolved with a corrective purpose. The *BikeOprsCtrC1* connector in Fig. 6.22 collects test data and controls the evolution of the *BikeOprs* interface of the *Basket* component.

During offline test, the dataflow controller of the *BikeOprsCtrC1* connector blocks *BasketC2* incomings and let pass *BasketC1* incomings. The outgoings of the *BikeTripC1* component are sent to the two versions of *Basket*. During online test, the dataflow controller reverses the situation to make the new version (*BasketC2*) dominant.

To evaluate a new version, observations encompasses various data, ranging from dataflow (test for corrective evolution) to execution times (test for perfective evolution). The collected data elements are listed in Table 6.8. The dataflow controller is in charge of collecting data from incoming calls which comprises the calling method and inputs and outputs of old and new versions for this calls. The time controller is responsible for collecting time for each input and output. Furthermore, in order to obtain the needed

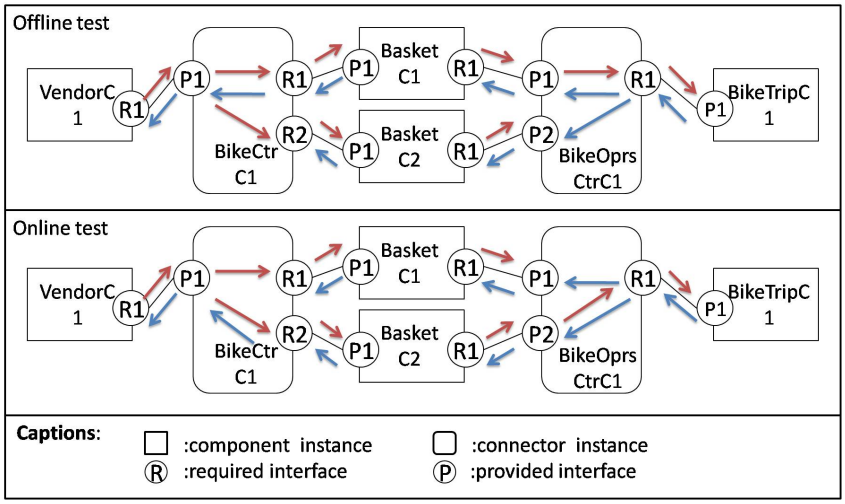


FIGURE 6.23 – The offline and online dataflow control for provided interface evolution

number of observations, each data collection by connectors will continue until all connectors in the changed group have enough observation samples (as empirically defined in the preparation stage).

Controllers	**Provided interface evolution**	**Required interface evolution**
Dataflow Controller	– method name – input message – output message of new version – output message of old version	– method name – input message of new version – input message of old version – output message
Time Controller	– method – time of input message for old version – time of output message for old version – time of input message for new version – time of output message for new version	

TABLE 6.8 – Offline and online data collected by controllers.

The evaluation of observations is handled by the evolution manager which collects all observations from connectors and thus has a global view to precisely calculate the time spent in each function in order to test the behavior of the new version. For corrective evolution, the new version must meet its predefined evolution condition. For perfective evolution, the new version must execute faster than the old version. But the response time measured is not really the response time of the evolved interface. The effective response time should be the calculation time in the evolved component. If this evolved interface calls other required interfaces to finish its task, the response time of required interface should be detracted from response time of the evolved interface. For example, when the *Bike* interface of component *BasketC1* is called by component *VendorC1*, it will call the interface *BikeOprs* before it returns an answer to the *VendorC1* component. Thus, the response time of the *BikeOprs* interface includes the response time of the interface

BikeOprs. In order to get net response times for the component, the formula to calculate the response time is $T_{response} = T_{go} + T_{back}$ in Fig. 6.24. After the new version passes the offline and online evaluation, the evolution process proceeds to next stage.

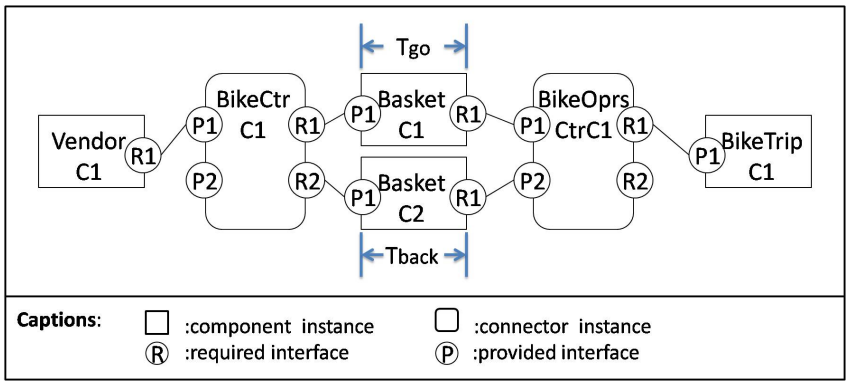


FIGURE 6.24 – Net response time

6.3.2.3 Observation and Commit Stage

Observation stage is another feature used to further secure evolution. In this stage, the old version still remains in the system, but only as a backup (it is not active anymore). The system is still in state-trace mode. If a critical error of new version arises and makes the system fail, the old version will be activated to replace the new version and the whole system will roll back to the previous sound state. The *sound state* means the state in which all components of the system function correctly. This state is recorded before the change test. Finally, the *evolution manager* either commits evolution after the above stages (if the new versions passes all stages) or abandons evolution and disconnects the new version (rollback). The unused version is disconnected and uninstalled.

6.3.3 Component Addition

Component addition aims to add some functionality to the system. This change may cause system failure if there are errors in the new component functionality or if the new component is not compatible with its connected components. The evolution process mainly tests two aspects : the functionality of the new component and the compatibility of the new component to its connected components.

Example. The evolution objective is to add the *Station Data* to the system. It implies changing the assembly from the original architecture represented in Fig.6.25a to the objective assembly Fig.6.25b.

6.3.3.1 Preparation Stage

The preparation stage aims to both introduce a new component into the system and connect it. The addition of a component must be accompanied by the addition of new

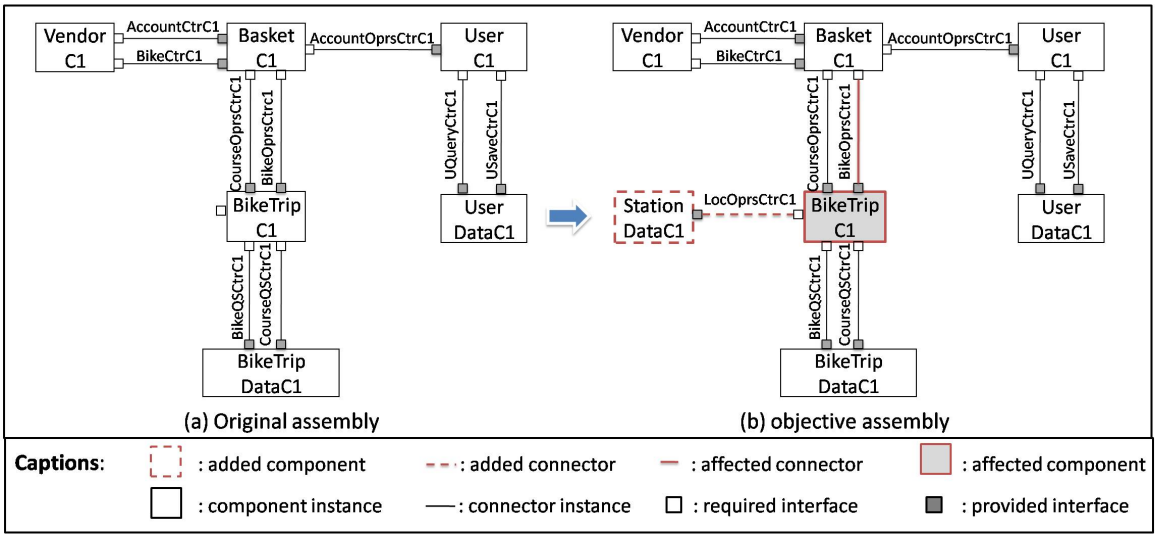


FIGURE 6.25 – Example of component addition

connectors which connect the new component with existing components. At first, the preparation actions for component addition have three steps :

1. Add the component to the system,
2. According to the connected interfaces, generate connectors,
3. Connect the new component and connectors to the existing components.

Then, in order to test the new component, we need to find the information including : its
- *Connected components* which connect with the added component.
- *Affected interfaces* from its connected components, that is the interfaces of the connected components that are indirectly involved in the behavior of the interfaces of the changed component.
- *Affected components* are the components owning the affected interfaces. As we know, changing one interface can affect many components. To test all the affected components is not an efficient choice, thus in our work, we use the techniques of regression test [Binder99]. The directly connected components are the most possibly affected components, thus main efforts are to test these directed connected components.
- *Affected connectors* which connect with affected interfaces to test new version's adaptiveness.
- *Connected connectors* are the new generated connectors which connect new component with other components.

Table 6.9 summarizes this data for the addition of the *Station Data* component of our example.

6.3.3.2 Test Stage

The test stage examines the functionality (offline) and the adaptiveness (online) of the new component.

Connected connectors	LocOprsCtrC1
Connected&affected components	BikeTripC1
Affected connectors	BikeOprsCtrC1

TABLE 6.9 – Affected connectors and components when adding the Station Data component.

Offline test. The offline test examines the functionality of the newly added component. Connected connectors authorize input from the new component but do not return the response to test.

Online test. The online test evaluates the adaptiveness of the newly added component together with its connected components by examining whether the connected components function correctly. During this test stage, each affected connector sends twice the input to the connected component. At first, the controlling connectors block the connections between the new component and connected components. The affected connectors collect responses and send them to the system. In a second phase, controlling connectors authorize the connection between the new component and existing components. The affected connectors collect the response but do not send it to the system. Between these two phases, components record their states and after the first phase, return to their original states to pretend that the first phase did not exist. After test, affected connectors send all this data collection to the evolution manager which compares the two sets of responses and decides whether to effectively add the component or not.

6.3.3.3 Observation and Commit Stage

During this stage, the system remains in recovery alert state. All state controllers record the states of components to prevent errors from the new component. The new component does not cause the system failure during the observation, we consider it *behaves correctly*. If the new component behaves correctly during the observation stage, the system enters its actual working state. The observation time is variant. The predefined observation duration is as long as the test duration. In test stage, test times determine the test duration, thus during this time, the evolved components are called enough times to be tested. Thus to use the test duration as the observation time is the best choice, or architects can customize the observation time.

6.3.4 Component Removal

Component removal aims to suppress an unused component from the system. This change may leave the system in an incomplete state and even prevent it from continuing to work. The evolution test evaluates if the new assembly works correctly without the component to remove.

Example. We reverse the example of the component addition. The evolution objective is to remove the *Station Data* from the system. It implies changing the assembly from the original architecture represented in Fig. 6.25b to the objective assembly Fig. 6.25a.

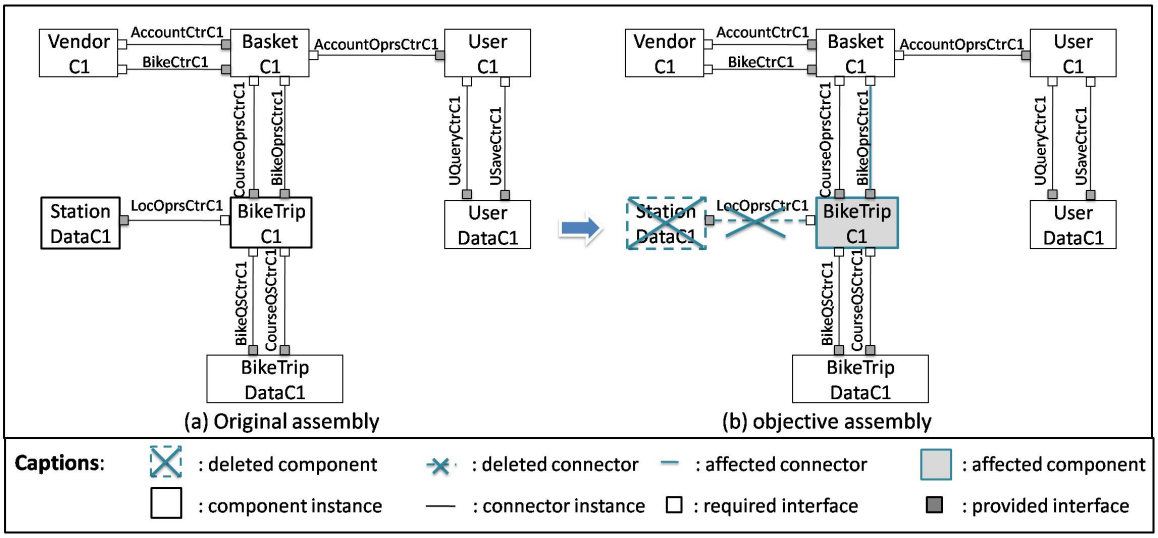


FIGURE 6.26 – Example of component removal

6.3.4.1 Preparation Stage

The preparation stage is similar to component addition's. We need to find the connected connectors, connected components, affected interfaces, affected components and affected connectors. This data for the removal of the *Station Data* component of our example is as same as for the addition of the *Station Data* component as shown in Fig. 6.9.

6.3.4.2 Test Stage

This test focuses on the connected components to evaluate whether they can work correctly without the removed component. All affected connectors send the same input to connected components twice. At first, connected connectors authorize input calls from connected components. Affected connectors record this response and send it back to the system. In a second phase, connected connectors block the input calls from connected components. Affected connectors record the responses of connected components which do not successfully call the removed component. After test, affected connectors send all this data collection to the evolution manager which compares the two sets of responses and decides whether to effectively remove the component or not.

6.3.4.3 Observation and Commit Stage

The removed component will remain in the system but isolated by connected connectors, to prevent connected components from generating instability errors. If the system has no problem during the observation stage, the removed component and connected connectors will be effectively be removed from the system.

6.3.5 Summary

In this section, we discussed our proposed gradual evolution process which realizes software evolution in a gradual, transparent and testable manner. It is the evolution

implementation phase of the architecture-centric evolution process presented in Section 6.1. The main idea of gradual evolution process is to have the original assembly of the system evolve into an objective assembly through a transitional assembly. The transitional assembly is a transitional step used to test the correctness and adaptability of changes. The entire gradual evolution process is decomposed into four steps : evolution preparation, test, observation and commit.

The advantage of this gradual evolution process is to test changes directly at the runtime system with offline and online two steps, which ensure that the newly added components work correctly and meet evolution purpose, and guarantee the system running correctly with changes.

6.4 SYNTHESIS

In this chapter, we presented Dedal version and change description and an architecture-centric evolution process detailed with a gradual evolution process using the runtime evolution framework. The evolution support of Dedal is more complete comparing with other ADLs. Table 6.10 gives the overview of changes supported by Dedal using the taxonomy that we proposed in Section 5.2.2.

Characters of architecture change	**Value**
Time of change	Dynamic
Anticipation	Unanticipated change
Change type	Structure, semantic
Change purpose	Corrective, perfective
Initial level of change	Specification, configuration and assembly
Change operation and subject artifact	▶ Adding, removing and replacing a component, ▶ Adding and removing a connector, ▶ Adding and removing a connection
Activities of architecture evolution	**Value**
Consistency checking	Name, interface, behavior, attributes, interaction and mapping
Impact analysis	Vertical, horizontal impact
Evolution test	Offline and online
Change propagation	Horizontal propagation (top-down and bottom-up), vertical propagation
Versioning	change-based versioning

TABLE 6.10 – The characters and activities of change in Dedal

There are two contributions :

– *Dedal version and change description.* Dedal models versions and changes of global architectures as first-class entities. Versioning of Dedal is change-based. It leads Dedal be nature to support architecture evolution. The detailed and semantic change description supplies a full information to make changes be easily applied and tested during the evolution process.

– *Evolution process.* The objectives of evolution of Dedal is (1) to enable changes be initiated at any level of architectures, (2) to ensure correctness and feasibility by six

consistency checking and a runtime test and (3) to guarantee the coherence between them by horizontal and vertical propagations.

Chapitre 7

Implementation

In previous chapters, we presented our propositions : an architecture description language named Dedal and a related architecture-centric evolution process. A proof-of-concept prototype implementation of these has been implemented. It is called Arch3D – an architecture-based evolution environment. In this chapter, we give an overview of the Arch3D tool suite for architecture modeling, visualization, analysis and evolution management.

Contents

7.1 OVERVIEW OF ARCH3D

An easy to use tool suite is an important aspect to support users of the language and demonstrate the feasibility of our ideas. There are many great tool suits that support ADLs, such as ArchStudio [Arc], Fractal [Fra], ArgoUML [Arg].

We have applied our Dedal model and evolution process into Arch3D. Arch3D is a tool suite for modeling, visualizing, analyzing, implementing and evolving software and systems architectures. Its tools can be divided into three categories : architecture modeling, runtime framework and evolution tools.

1. *Architecture modeling.* We have built our infrastructure to model Dedal that are defined in XML schemas, and Java classes.

- *Dedal XML schema.* The Dedal architecture description is expressed using XML (Extensible Markup Language) [Biron04]. The syntax of Dedal architectures is defined using XML schemas.
- *Arch3D-JDC.* In order to facilitate the edition of the Dedal XML files, the Dedal XML description of software architectures are mapped into Java data classes (abbreviated as JDC in Arch3D) using JAXB[Fialli06]. The Java data classes are synchronized with XML files.
- *Arch3D-JModel.* In order to support evolution and avoid the no-checking operations on architectures, we extend JDC with JModel. It is a Java-based Dedal model with evolution support.

2. *Architecture implementation framework.* The runtime system is modeled using Arch3D-REF (Runtime Evolution Framework).
   - *Arch3D-REF.* The architecture implementation framework is Arch3D-REF (Runtime Evolution Framework) in our environment. It is a runtime environment supporting dynamic software evolution. It embeds Arch3D-IEM (Implementation Evolution Management) to support the dynamic evolution of the runtime system. Arch3D-IEM also plays the communicating role between software architecture description and runtime system.
3. *Tools.* The development and evolution of software architectures based on Dedal is manipulated through specific tools that include Arch3D-Editor, AEC (Architecture Evolution Center), Arch3D-Analyzer (Consistency Checker).
   - *Arch3D-Editor.* Arch3D-Editor is a graphical editor to visualize and modify software architectures. It can show software architectures in formats : tree, tree-table, graph, BNF-based text and XML.
   - *Arch3D-AEC.* Arch3D-AEC (Architecture Evolution Center) is a management center to control software architecture evolution, including evolution planning, implementation and re-engineering.
   - *Arch3D-Analyzer.* Arch3D-Analyzer is an architecture consistency checking tool to assess the correctness and consistency of architecture descriptions. The analysis criteria covers behavior, name, interface, interaction and refinement consistency checking.

Their relationship can be found in Fig. 7.1. The code sizes of various Arch3D modules are shown in Table 7.1. In the rest of this chapter, we will present these tools one by one in detail.

Arch3D tool	Total LOC
Dedal XML Schema	579
JDC	1809
JModel	712
Editor	7445
AEC	638
Analyzer	194
REF	4302

TABLE 7.1 – The LOC of Arch3D tools

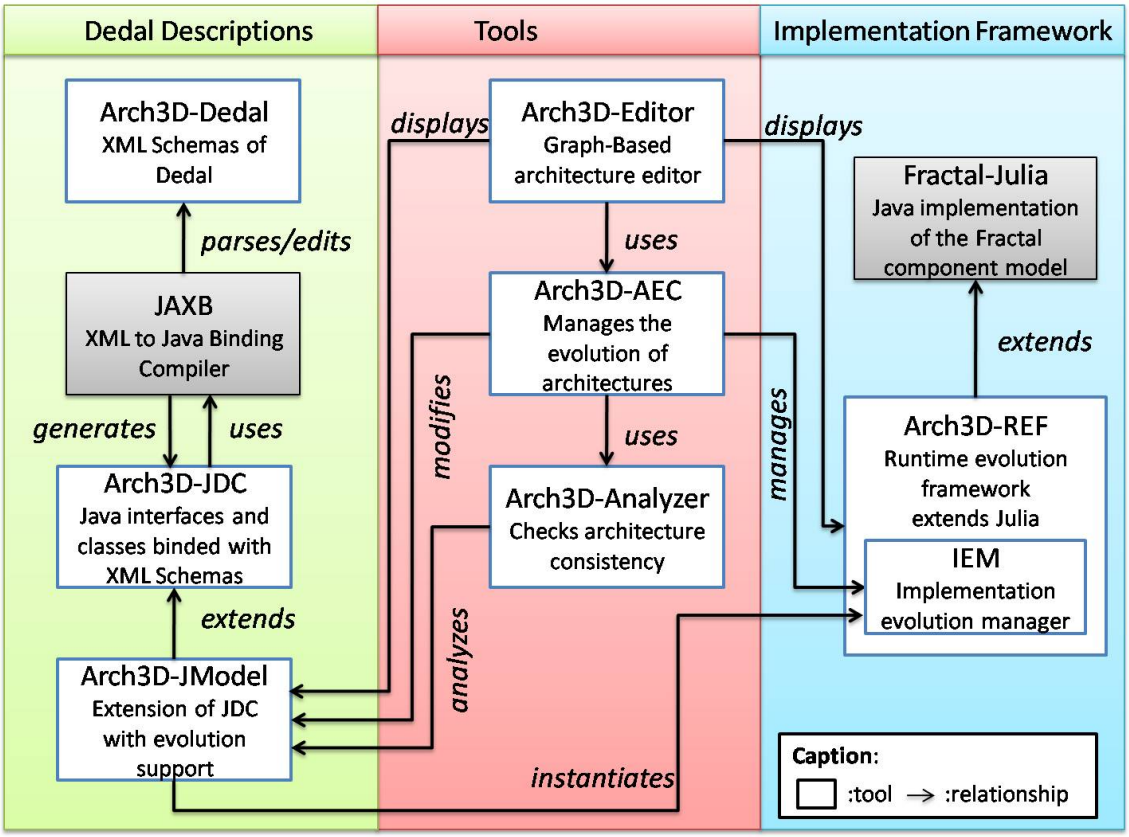


FIGURE 7.1 – Arch3D : an architecture-based modeling and evolution environment

7.2 ARCHITECTURE MODELING

Architecture modeling can be done in various formalisms, including Java classes, XML, BNF-based grammars, etc. In our work, we choose to use XML and Java classes, two methods to model software architectures. The architecture descriptions are stored in XML files. These files compose a XML database. When modifying and using of these descriptions, the mapping Java classes are used in order to prevent error prone modification under XML format and have an easy to use runtime representation of the description.

7.2.1 Dedal XML Schemas

Dedal syntax is defined using XML schemas [Fallside04, Biron04]. Indeed, XML's original metalanguage is the document type definition (DTD) [Bray08] which is used, for example, to define Fractal-ADL [Leclercq07]. The adoption of XML schemas is superior to DTDs in a number of ways. First of all, XML schemas add a type system to XML. This includes basic types like integers and strings, that define the contents of atomic elements and attributes, and complex types, in which elements contain other elements and attributes. In this type system, types can be extended in a manner similar to object-oriented subtyping. This property makes Dedal definition easily extendable. Secondly, XML schemas are richer than DTDs as they possibly include cardinalities for sub-elements.

Figure 7.2 shows the XML schema definition of Dedal abstract architecture specifications.

```
<xs:complexType name="Specification">
 <xs:annotation>
  <xs:appinfo>
   <jaxb:class name="SpecificationADL" />
  </xs:appinfo>
 </xs:annotation>
 <xs:complexContent>
  <xs:extension base="archElement:Architecture">
   <xs:sequence>
    <xs:element name="componentRole" type="ComponentRole"
        maxOccurs="unbounded" minOccurs="0"/>
    <xs:element name="connection" type="Connection"
        maxOccurs="unbounded" minOccurs="0"/>
    <xs:element name="architectureBehavior" type="xs:string"
        maxOccurs="1" minOccurs="0" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

FIGURE 7.2 – XML schema of abstract architecture specification of Dedal

7.2.2 Arch3D-JDC

The XML schemas of Dedal are mapped into a set of Java data interfaces and classes using Java Architecture for XML Binding (JAXB) [Fialli06]. This generated set of Java files constitutes Arch3D-JDC (Java Data Code). JAXB provides Java access to XML data without having to know XML or XML processing. It maps XML elements and attributes into objects, hiding XML details such as namespaces, header tags, sequence ordering, etc. The Java generated objects correspond to the types defined in the XML schemas. Manipulating the objects causes corresponding changes in the underlying XML document.

Figure 7.3 shows the generated code for the SpecificationADL interface.

```
public interface SpecificationADL extends ArchitectureADL
{
    List<ComponentRoleADL> getComponentRole();
    List<ConnectionADL> getConnection();
    String getArchitectureBehavior();
    void setArchitectureBehavior(String value);
}
```

FIGURE 7.3 – Java interface *SpecificiationADL* for XML schema complex type *Specification*

7.2.3 Arch3D-JModel

Arch3D-JDC provides the generated interfaces and classes to edit Dedal XML descriptions. However, using JDC has two disadvantages : (1) The modification of architectures is

performed without any correctness checking, and (2) It cannot support the evolution easily and directly with one command. In order to support and realize architecture changes more easily and securely, we extend Arch3D-JDC with Arch3D-JModel, which we design to support architecture changes applied on software architectures. In order to mask the error-prone methods generated in Arch3D-JDC Java interfaces, for each level of architecture, we create an *evolution interface* which contains authorized change operations and an *implementation class* which extends the Arch3D-JDC implementation class and implements the created evolution interface. Figure 7.4 shows the package diagram of Arch3D-JModel and the class relationships with Arch3D-JDC. The interface of architecture specification *SpecifcationADL* is extended by interface *Specification* as shown in Fig 7.5.

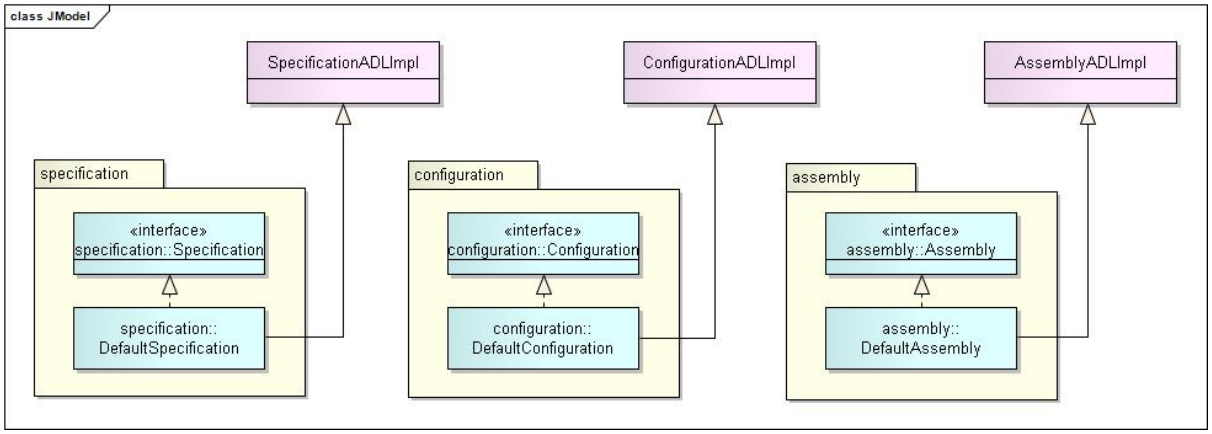


FIGURE 7.4 – Class diagram of Arch3D-JModel

```
public interface Specification
{
    void addComponentRole(ComponentRole cr);
    void removeComponentRole(ComponentRole cr);
    void replaceComponentRole(ComponentRole crOld, ComponentRole crNew);
    void addConnection(Connection c);
    void removeConnection(Connection c);
    void modifyArchitectureBehavior(String ab);
}
```

FIGURE 7.5 – Java interface *Specification*

7.3 ARCHITECTURE IMPLEMENTATION FRAMEWORK

An architecture implementation framework is a piece of software, which acts as a bridge between an architecture modeling notation and a set of implementation technologies [Taylor09]. It provides the key elements of the architecture modeling in code, in a way assists developers in implementing systems that conform to the prescription and constraint of the architecture modeling. The difficult issue is how to synchronize architectures and implementation in a tight manner. In order to solve this problem, we propose

the *Arch3D-REF* (Runtime Evolution Framework) as the architecture implementation framework for Dedal which extends the Julia implementation of Fractal.

7.3.1 Arch3D-REF

Our Arch3D-REF (Runtime Evolution Framework) is implemented as an extension of Julia [Jul], an open-source Java implementation of the Fractal [Fra] component model. We choose the Fractal model to implement Arch3D-REF considering three advantages of the Fractal model.

1. Fractal is a component model that enables to instantiate and manipulate assemblies at runtime.
2. Fractal is also easily extendable. Fractal components are managed by controllers contained in the membrane of components. New controllers can be added to add new control capabilities to components. It thus is straightforward to implement Arch3D-REF to the implementation.
3. It is robust and open source Java implementation and has an active french-speaking community.

7.3.1.1 Quick Facts About The Fractal Component Model

A Fractal component is composed of a *membrane*, which is typically composed of several controller and interceptor objects, and a *content*, which consists in a finite set of inner connected components (called sub-components). The membrane of a Fractal component is used to realize the non-functional activities and which is easily extended. Thus, we try to extend the Fractal membrane to make it support evolution.

The membrane of a Fractal component is composed by four parts : controllers, interceptors, external and internal interfaces. The example of a component model can be found in Fig. 7.6.

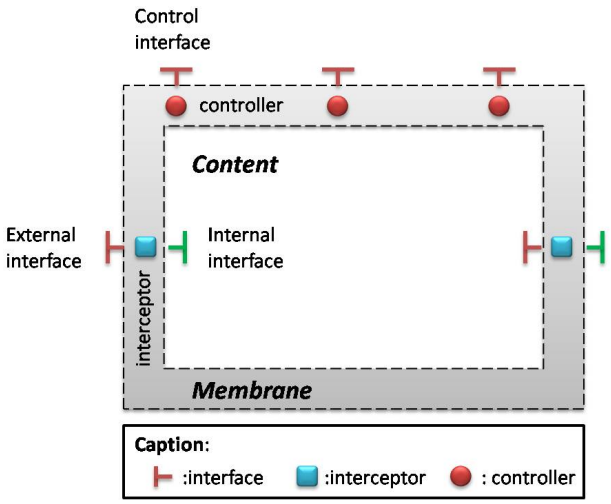


FIGURE 7.6 – Component model in Julia

- External interfaces are accessible from outside the component to introspect and reconfigure its internal features (called control interface access).
- Internal interfaces are only accessible from the component's sub-components.
- Controller. A controller object implements control interfaces. Typically, a controller object can provide an explicit and causally connected representation of the component's sub-components and superpose a control behavior to the behavior of the component's sub-components, including suspending, check pointing and resuming activities of these sub-components. There are four predefined controllers for component model in Fractal : (1) *name controller* to stock the name of the component, (2) *binding controller* to allow binding and unbinding of primitive bindings, (3) *life-cycle controller* to control the component behavior including start and stop methods, (4) *content controller* to list, add and remove subcomponents for composite components. In Julia, most of the controller classes are generated classes, generated from *mixin classes*.
- Interceptor. Interceptor objects are used to export the external interface of a sub-component as an external interface of the parent component. They intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of such invocations (e.g. pre and post-handlers to perform behavior adaptations). In the Julia, interceptor classes are generated classes also, generated from *asm classes*.

The membrane of a component can be personalized by deleting existing controllers / interceptors or adding specific designed controllers / interceptors to make the component support other non-functional activities. Each component membrane thus can be seen as implementing a particular semantics of composition for the component's sub-components. Controller and interceptors can be seen as meta-objects or meta-groups (as they appear in reflective languages and systems).

7.3.1.2 REF

Arch3D-REF extends the Julia implementation of Fractal by implementing additional controllers and interceptors. The class diagram of this extension can be found in Fig 7.7.

In order to adapt Arch3D-REF model to the Fractal component model, we added an *elementTypeController* to specialize Fractal components into three sub-types – components, connectors and container – by storing the type information in this *element type controller*. This was necessary as : Dedal distinguishes the semantic of these three categories of elements. In the Fractal component model, everything has to be Fractal component. Then, we added specific controllers (as described in Chapter 6.2) to each sub-type.

Secondly, as REFramework is mapped with component instantiated assembly of Dedal, thus we add an element-ADL controller to each type of architecture to store their link with the XML file. *ElementDescriptor* stocks the architecture description of container, components and connectors. For the container, it is the Dedal description of instantiated component assembly. For the component, it is the Dedal component instance description. For connector, it is the Dedal connector instance description.

The other controllers we added to Arch3D-REF to support the evolution are presented as follows.

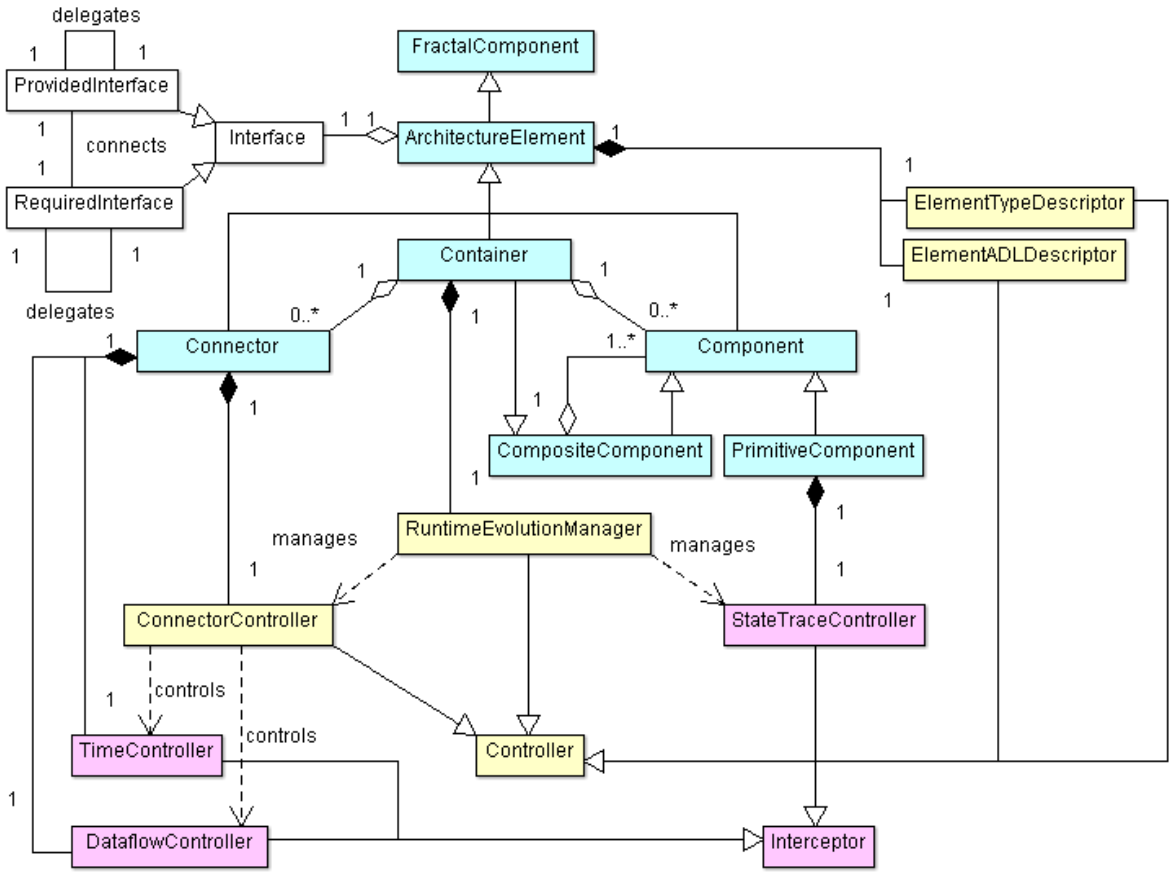


FIGURE 7.7 – Class diagram of Runtime evolution framework

Implementation evolution manager. The implementation evolution manager implements the work of implementation evolution management described in Chapter 6. It is a controller embedded in container to control the dynamic evolution of system composed in this container. It implements the control interface of implementation evolution manager. It controls and realizes the evolution through functions described in Fig. 7.3.1.2.

```
public interface ImplementationEvolutionManager {
    void addComponent(Component cmp);
    void removeComponent(Component cmp);
    void replaceComponent(Component oldCmp, String itfName, Component newCmp,
          String ePurpose, String condition, int testTimes);
    void replaceComponent(Component oldCmp, String itfName, Component newCmp,
          String ePurpose, String condition);
    void connectComponent(Component cmpSource, String itsSource,
          Component cmpTarget, String itfTarget);
    void disconnectComponent(Component cmpSource, String itsSource,
          Component cmpTarget, String itfTarget);
    void disconnectComponent(Component cmp);
          List transformChangeList(List changes);
}
```

Connector, dataflow and time controllers. Connector, dataflow and time controllers in REFramework implement directly functionalities of REF presented in Chapter 6.2. Figure 7.8 illustrates the membrane of the connector model in REFramework with its controllers and interceptors. The controllers and interceptors in Fig. 7.8 with underline are extended by Arch3D-REF.

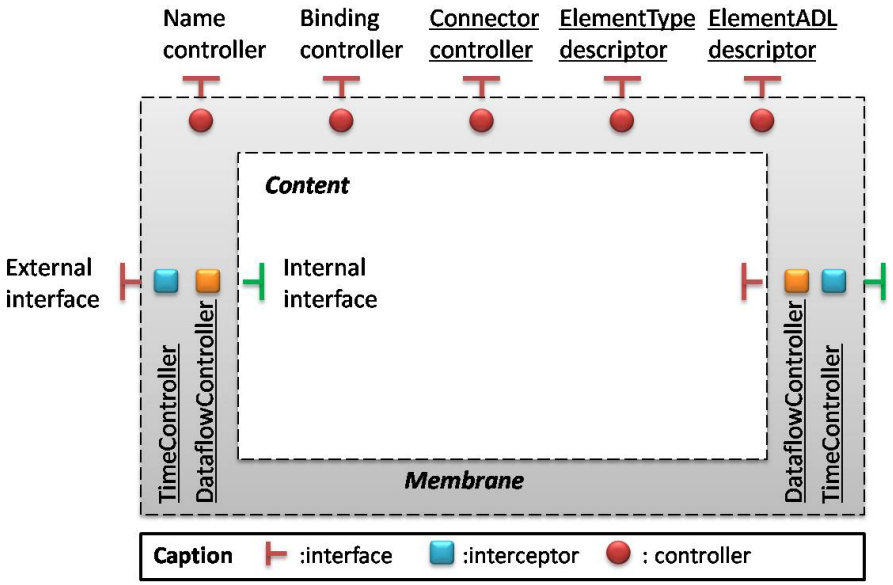


FIGURE 7.8 – Connector model in REF extended from Fractal component model

State trace controller. It is a controller to trace the state of its possessed component according the evolution requirement. When the system requires to recover to the previous state, all of these stateTrace controllers will retrieve the state of their components to previous state.

Figure 7.9 presents the package digram of Arch3D-REF. *GenericFactory* package is used to generate architectural elements of Dedal including container, components and connectors.

7.4 EVOLUTION TOOLS

7.4.1 Arch3D-Analyzer

Arch3D-Analyzer provides a way to automatically test architecture descriptions against many different inconsistencies. The types of inconsistencies checked by analyzer includes all inconsistencies mentioned in Section 6.1.3.1 : name, behavior, interface, interaction, attribute and mapping inconsistencies. Inconsistencies can be checked with a few mouse in Arch3D-Editor, and errors are displayed and inspected. The package diagram of Arch3D-Analyzer is shown in Fig. 7.10. Figure 7.11 gives an example of analyzing result.

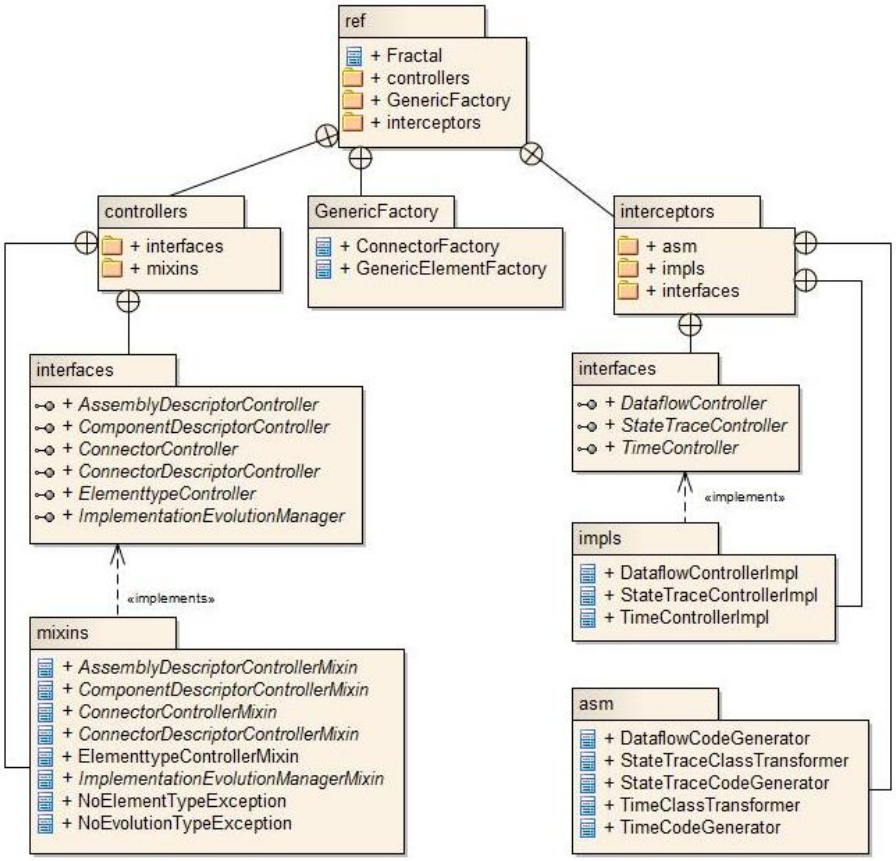


FIGURE 7.9 – Package diagram of Arch3D-REF

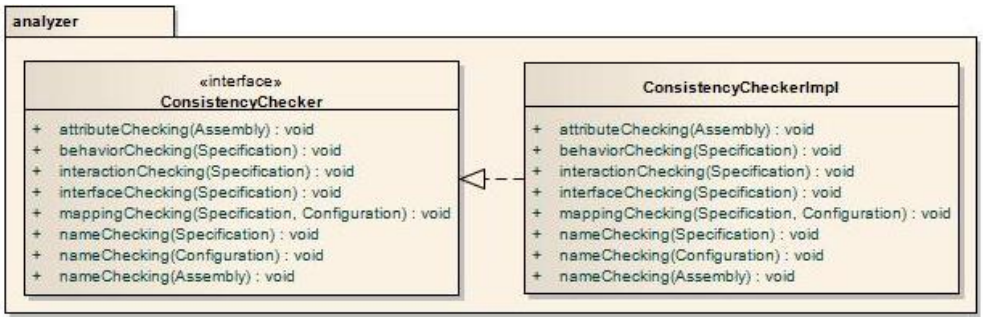


FIGURE 7.10 – The class diagram of Arch3D-Analyzer

7.4.2 Arch3D-AEC (Architecture Evolution Center)

Architecture evolution center (Arch3D-AEC) is a management center to control the entire evolution process. The package diagram of Arch3D-AEC is given in Fig. 7.12. It manages the thought evolution process by three stages : evolution planning, evolution implementation and evolution re-engineering. For Arch3D-AEC, evolution must be triggered by an input change request. Figure 7.13 gives a screenshot of creating new change request.

The evolution process is composed by three phases. Firstly, it propagates the change request into three change lists and analyzes the consistencies of architectures with changes using Arch3D-Analyzer. Secondly, it calls the implementation evolution manager in

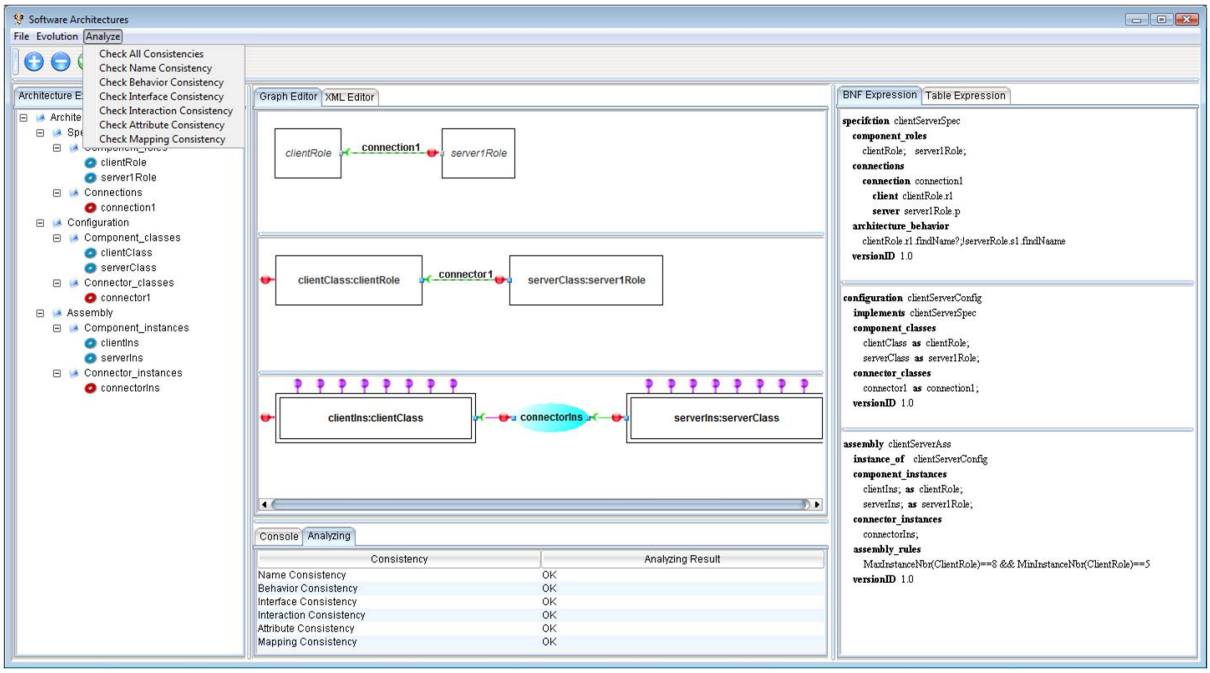


FIGURE 7.11 – The screenshot of Arch3D-Editor for Arch3D-Analyzer

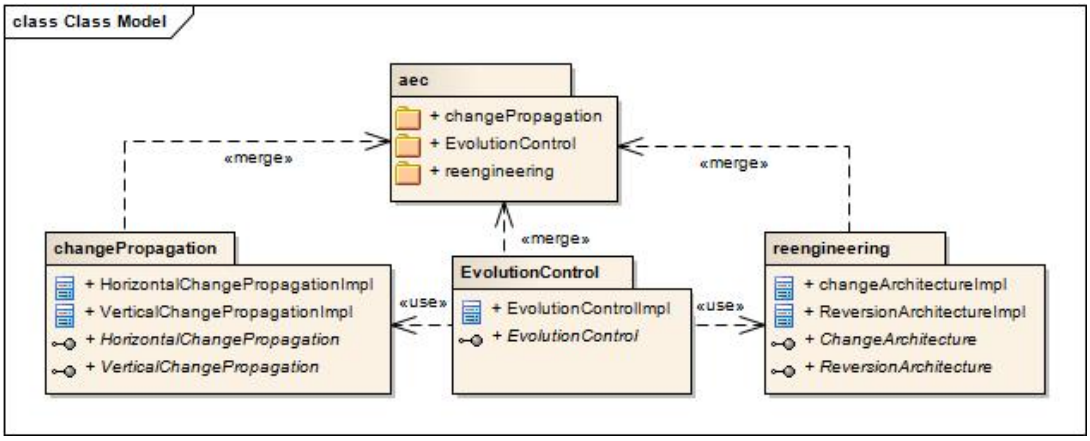


FIGURE 7.12 – The package diagram of Arch3D-AEC

Arch3D-REF to test the changes at runtime. At last, it applies the changes to architecture descriptions and reversions them if required. The procedure of evolution is displayed in Arch3D-Editor (which will be presented in the next section).

7.4.3 Arch3D-Editor : a Dedal Graphical Console

An architecture visualization has two objectives : depiction and interaction [Taylor09]. A single architecture can be visualized in various ways, for example like textually visualization and graphically [Feijs98].

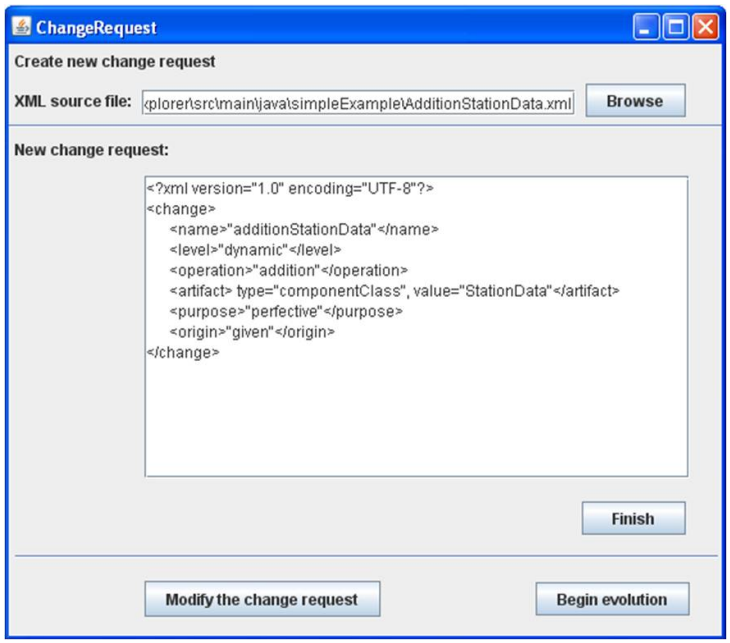


FIGURE 7.13 – The screenshot of creating new change request

Textual visualization. Textual visualizations depict architectures using ordinary text files. There are two ways that are often used. The first is using XML, such as Fractal ADL [Fra] and xADL2.0 [Dashofy01], ADML [Spencer00]. XML visualization facilitates reading, manipulating, and validating using ML tools. The other is using meta-language - many are Backus-Naur Form (BNF) – defines the textual syntax of ADLs, such as C2, SOFA [Bálek02].

Graphical visualization. Graphical visualizations depict architectures using graphical symbols instead of text. Many ADLs, for example Darwin, Koala, and UML, provide or endorse a graphical visualization that captures some or all of the information in the architecture description in a graphical format. In general, these depictions are primarily composed of graphical symbols, annotated with textual labels and other textual decorations.

Arch3D-Editor provides both textual & graphical visualizations. Textual visualization are provided in both XML and BNF forms. Figure 7.14 shows the depiction of an excerpt of the *B*RS architecture descriptions.

The *Arch3D-Editor* provides a graphical console to create, view and modify Dedal-based architectures. The architectures in Arch3D-Editor can be expressed as BNF-based, graph-based, tree-based, or XML-based Dedal description. Architects can simultaneously display the different representation levels of an architecture and conjointly work on them (see Fig. 7.14). The package diagram of Editor can be found in Fig. 7.15.

Arch3D-Editor also provides an interaction to control software evolution based on Dedal. It provides the functional interfaces of Arch3D-AEC and Arch3D-Analyzer, as shown in Fig. 7.10 and 7.13. Furthermore, it supplies an interaction of evolution directly applied on graphical view of software architectures. The results of evolution procedure is displayed one by one, as shown in Fig 7.16.

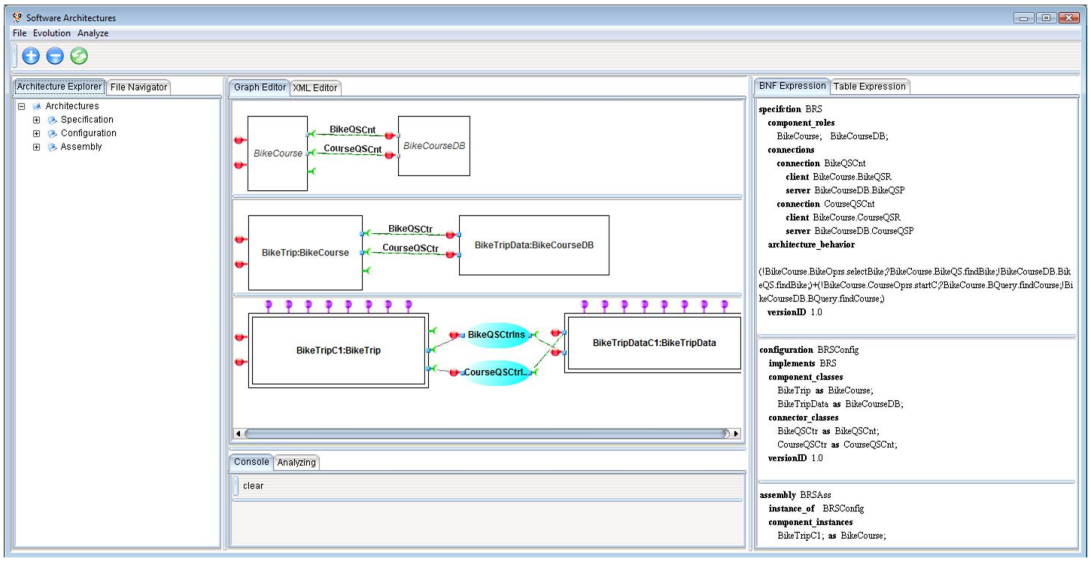


FIGURE 7.14 – Evolution explorer view of an excerpt of BRS example

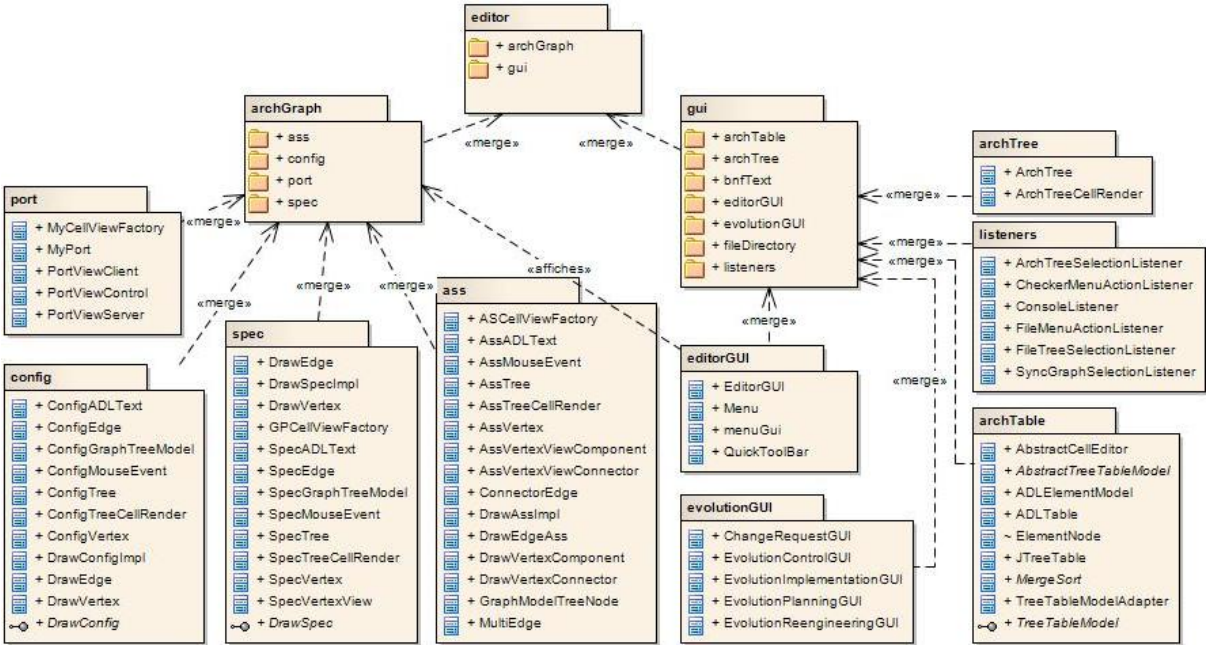


FIGURE 7.15 – Package diagram of Arch-Editor

7.5 SUMMARY

We developed the *Arch3D* tool suite to support architecture modeling and evolution based on Dedal. It consists of three parts : Dedal architecture modeling, architecture implementation framework, and evolution tools. The Arch3D tool suite facilitates the architects evolve the softwares by :

- The direct visualization of architecture descriptions in graphical and textual (XML and BBF-based) views.
- The automatically evolution process control.

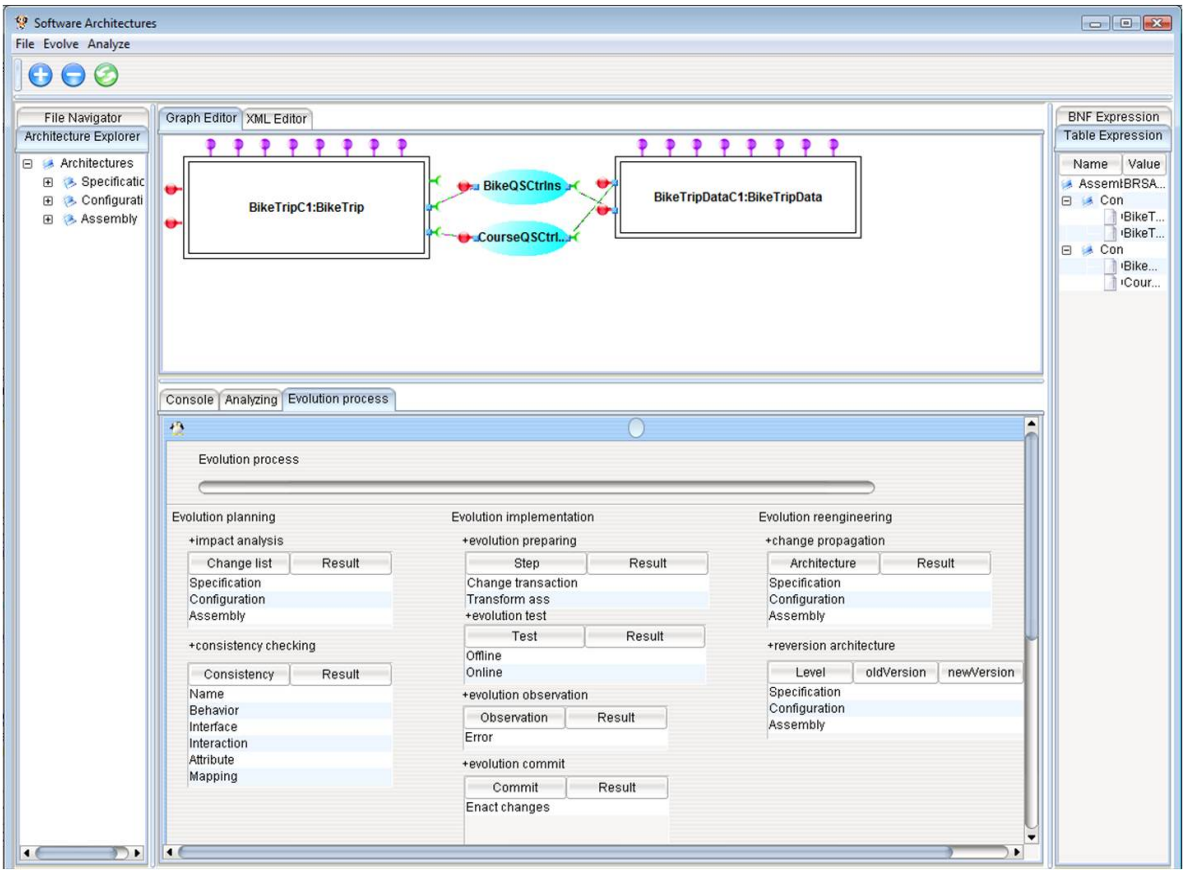


FIGURE 7.16 – Evolution explorer view of the evolution process

– The tight synchronization between architectures and runtime system.
– The architecture consistency checking.
– The runtime evolution test.

CHAPITRE 8

CONCLUSION AND PERSPECTIVE

8.1 CONCLUSION

This dissertation is a contribution of component-based software engineering that addresses two aspects : software architecture and software evolution. It proposes the Dedal ADL to model software architectures in three levels, and an architecture-centric evolution process that bases on it.

As for modeling software architectures for component-based softwares, we found that there is no ADL that motivates descriptions for the three stages of the development cycle (specification, configuration and assembly). They mostly cover only one or two levels of software architectures. In order to fill this gap, we proposed an ADL named Dedal. The objective of the Dedal ADL is to describe software architectures for component-based softwares. Dedal covers software architectures in three levels, respectively named *abstract architecture specification*, *concrete architecture configuration* and *instantiated component assembly*. These three kinds of software architecture descriptions exactly describe three architecture productions produced in three stages of component-based software development.

As for supporting the evolution of component-based softwares, we identified two main issues. Firstly, ADLs generally do not cover the description of versions and changes as first-class entities. This hinders tracing the evolution of software architectures. Secondly, software evolution is often performed directly on runtime systems without a synchronization with architecture descriptions. This leads to software architecture drift and erosion. As the evolution process is incomplete, change propagation misses and software architectures are not versioned. Considering above problems, we propose a combined solution :

- Dedal models versions and changes as first-class entities,
- Evolution is managed completely using an architecture-centric evolution process that covers evolution planning, evolution implementation and re-engineering based Dedal.

8.1.1 Organization

The manuscript was introduced as follows.

Chapter 2. This chapter gives an overview of component-based software engineering by comparing it with traditional software engineering.

Chapter 3. This chapter firstly gives our vision for software architectures. Then it presented a state-of-the-art of ADLs to to identify the strengths and weakness of modeling techniques in existing ADLs. It evaluates these works from three architecture levels : specification, configuration and assembly.

Chapter 4. This chapter presents our proposed ADL – Dedal. It focuses on modeling architectures of component-based softwares by adapting their development and evolution. It models software architectures in three levels :

- *Abstract architecture specifications* provide a formal and generic definition of the global structure and behavior of software systems, which are designed to achieve identified functional requirements. They are composed by abstract component types called component roles.
- *Concrete architecture configurations* describe the architecture from an implementation viewpoint (by assigning component classes to component roles).
- *Instantiated component assemblies* provide a description of runtime software systems and gather information on identity and state-dependent design decisions on the component instances of the assemblies.

Chapter 5. This chapter firstly proposes a taxonomy for software architecture changes. Then its studies existing dynamic ADLs to identify how they support architecture changes. Informed by our proposed taxonomy, a state-of-the-art discussed the characteristics of these existing dynamic ADLs.

Chapter 6. This chapter presents Dedal version and change description. The objective of these descriptions is to describe version information and changes as first-class entities in order to be able to trace evolution and obtain a change-based software architecture version history. It presents the proposed architecture-centric evolution process based on Dedal. This process decomposes into three steps :

- *Evolution planning* ensures the correctness and feasibility of changes by examining the consistencies of architecture.
- *Evolution implementation* tests changes on the running system before making them effective.
- *Evolution re-engineering* propagates changes to the architecture in three levels and reversions them when necessary. This stage is applied in order to prevent architecture erosion and drift.

At last, Chapter 6 describes the runtime evolution framework and a connector-driven gradual evolution process based on it.

Chapter 7. This chapter introduced Arch3D tool-suite for modeling, visualizing, analyzing, implementing and evolving software and systems architectures. Its tools can be divided into three categories :

- *Architecture modeling tools.* Dedal definition is available thanks to that are defined in XML schemas. At runtime, these schemas are automatically synchronized with Java classes (Arch3D-JDC). Arch3D-JModel enhances Arch3D-JDC evolution coarser capabilities and grained functionalities.

– *Architecture implementation framework.* The runtime system is modeled using Arch3D-REF (Runtime Evolution Framework).
– *Evolution Tools.* The development and evolution of software architectures based on Dedal is manipulated through specific tools that include Arch3D-Editor, AEC (Architecture Evolution Center), and Arch3D-Analyzer (Consistency Checker).

8.1.2 Summary

The contributions of this thesis are as follows :

1. Dedal, a three level architecture description language covering abstract architecture specification, concrete architecture configuration and instantiated component assembly.
2. A taxonomy of software architecture changes.
3. The inclusion of change descriptions as first-class entities in the Dedal ADL with a richer change description semantics.
4. Change-based versioning for software architectures.
5. An architecture-centric evolution process based on Dedal supporting forward and reverse evolution.
6. A runtime evolution framework (REF) that supports dynamic evolution with reconfigurable connectors.
7. A connector-driven gradual evolution process based on the REF.
8. The *Arch3D* tool suite.

8.2 PERSPECTIVES

Perspectives for this work are numerous.

8.2.1 Technical Perspectives

Product line. Software product line systems often require a referenced architecture and then specified it into different configurations. Abstract architecture specification from this vision is suitable for serving as a referenced architecture. The point is to model option and variant two concepts in Dedal.

Model-driven architecture and evolution. Dedal models model-driven architecture and its evolution. As UML becomes the most useful technique to model softwares, we would like to study the transformation of Dedal models into UML models.

Versioning. In this thesis, the change-based version graph is the simplest one. It does not cover the complex aspects of version graph like branches, tags. Such enhancements of change-based versioning for software architectures are possible perspectives.

Support other types of evolution. The architecture evolution can be classified into anticipated and unanticipated evolution from anticipation view. However, there is another classification, which is typed evolution into : adaptive architecture evolution by given target architecture, self-evolution without given target architecture, and architecture evolution with given changes. Our work presented in this thesis focuses on the third type of evolution. In our future work, we want to adapt and develop Dedal to support also the other two types.

8.2.2 Applicative Perspectives

Plugin of Eclipse. Arch3D is Java application. We want to transform *Arch3D* as a plug-in of Eclipse in order to integrate with Eclipse. As a plug-in of Eclipse has two advantages : (1) It is integrated with the development environment, and (2) It is easily installed.

Experimentation. We want to evaluate Dedal and Arch3D for industrial softwares to get application experiences. We want to evaluate Dedal and Arch3D on two aspects : (1) Dedal models enough information required in industrial projects, and (2) Evolution process supported by Arch3D can manage complex evolution cases in industrial projects.

Bibliographie

[Abowd93] G. Abowd, R. Allen & D. Garlan. *Using style to understand descriptions of software architecture.* In SIGSOFT '93 : Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering, pages 9–20, New York, NY, USA, 1993. ACM Press.

[Agnew94] B. Agnew, C. Hofmeister & J. Purtilo. *Planning for change : a reconfiguration language for distributed systems.* In Proc. 2nd International Workshop on Configurable Distributed Systems, pages 15–22, 1994.

[Allen97a] R. D. R. Allen & D. Garlan. *Specifying dynamism in software architectures.* In G. T. Leavens & M. Sitaraman, editeurs, Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997, pages 11–22, 1997.

[Allen97b] R. Allen & D. Garlan. *A formal basis for architectural connection.* ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pages 213–249, 1997.

[Allen97c] R. Allen, D. Garlan & R. Douence. *Specifying dynamism in software architectures.* In Proceedings of the Workshop on Foundations of Component-Based Software Engineering, Zurich, Switzerland, September 1997.

[Allen98] R. Allen, R. Douence & D. Garlan. *Specifying and analyzing dynamic software architectures.* In Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), pages 21–37, Lisbon, Portugal, March 1998.

[Arc] ArchStudio 4. *http ://www.isr.uci.edu/projects/archstudio.*

[Arévalo07] G. Arévalo, N. Desnos, M. Huchard, C. Urtado & S. Vauttier. *Precalculating component interface compatibility using FCA.* In J. Diatta, P. Eklund & M. Liquière, editeurs, Proc. of the 5^{th} int'l conf. on Concept Lattices and their Applications. CEUR Workshop Proc. Vol. 331, pages 241–252, Montpellier, France. 2007, October 2007.

[Arévalo09] G. Arévalo, N. Desnos, M. Huchard, C. Urtado & S. Vauttier. *Formal concept analysis-based service classification to dynamically build efficient software component directories.* International Journal of General Systems, vol. 38, no. 4, pages 427–453, May 2009.

[Arg] ArgoUML. *http ://argouml.tigris.org/.*

[Bálek01] D. Bálek & F. Plášil. *Software connectors and their role in component deployment.* In Proceedings of DAIS'01, pages 69–84, Krakow, September 2001. Kluwer.

[Bálek02] D. Bálek. *Connectors in software architectures.* PhD thesis, Charles University, Czech Republic, 2002.

[Bass98] L. Bass, P. Clements & R. Kazman. Software architecture in practice. Addison Wesley, 1998.

[Bennett00] K. H. Bennett & V. T. Rajlich. *Software maintenance and evolution : a roadmap.* In ICSE '00 : Proceedings of the Conference on The Future of Software Engineering, pages 73–87, 2000.

[Bessam09] A. Bessam & M. T. Kimour. *Multi-view metamodeling of software architecture behavior.* Journal of Software, vol. 14, no. 5, pages 478–486, Jul 2009.

[Binder99] R. V. Binder. Testing object-oriented systems : models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Biron04] P. V. Biron & A. Malhotra. *XML Schema Part 2 : Datatypes Second Edition.* W3C Recommendation, October 2004. http ://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[Bohner96] S. A. Bohner & R. S. Arnold. Software change impact analysis. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Booch05] G. Booch, J. Rumbaugh & I. Jacobson. Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series). Addison-Wesley Professional, 2005.

[Bray08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler & F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation, nov 2008. http ://www.w3.org/TR/xml/.

[Bruneton06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma & J.-B. Stefani. *The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems.* Softw. Pract. Exper., vol. 36, no. 11-12, pages 1257–1284, 2006.

[Buckley05] J. Buckley, T. Mens, M. Zenger, A. Rashid & G. Kniesel. *Towards a taxonomy of software change : Research articles.* Journal of Software Maintenance and Evolution : Research and Practice, vol. 17, no. 5, pages 309–332, September 2005.

[Bures06] T. Bures, P. Hnetynka & F. Plasil. *Sofa 2.0 : Balancing advanced features in a hierarchical component model.* In SERA '06 : Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications, pages 40–48, Seattle, USA, 2006. IEEE Computer Society.

[Chapin01] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil & W.-G. Tan. *Types of software evolution and software maintenance.* Journal of Software Maintenance, vol. 13, no. 1, pages 3–30, 2001.

[Chaudron08] M. R. V. Chaudron & I. Crnkovic. Software engineering ; principles and practice, chapitre Component-based Software Engineering, pages 605–628. John Wiley & Sons, 2008.

[Chen03] P. Chen, M. Critchlow, A. Garg, C. v. d. Westhuizen & A. v. d. Hoek. *Differencing and merging within an evolving product line architecture.* In Proceedings of the Fifth International Workshop on Product Family Engineering, pages 269–281, 2003.

[Cheng02] S. W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel & P. Steenkiste. *Using architectural style as a basis for system self-repair.* In The 3rd Working IEEE/IFIP Conference on Software Architecture, pages 45–59, Montreal, Canada, August 2002.

[Choi06] Y.-H. Choi, S. Yoon, G.-S. Shin & Y. Yang. *An extension of uml 2.0 for representing variability on product line architecture.* In IASTED Conf. on Software Engineering, pages 119–124, 2006.

[Chung01] L. Chung & N. Subramanian. *Process-oriented metrics for software architecture adaptability.* In RE '01 : Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, page 310, Toronto, Canada, 2001.

[Conradi98] R. Conradi & B. Westfechtel. *Version models for software configuration management.* ACM Comput. Surv., vol. 30, no. 2, pages 232–282, 1998.

[Cook99] J. E. Cook & J. A. Dage. *Highly reliable upgrading of components.* In Proceedings of the 21st international conference on Software engineering(ICSE '99), pages 203–212, Los Angeles, California, May 1999.

[Crnkovic02] I. Crnkovic & M. Larsson, editeurs. Building reliable component-based software systems. Artech House Publishers, 2002.

[Crnkovic06] I. Crnkovic, M. Chaudron & S. Larsson. *Component-based development process and component lifecycle.* In ICSEA '06 : Proceedings of the International Conference on Software Engineering Advances, page 44, Papeete, French Polynesia, october 2006.

[Cross02] J. K. Cross & D. C. Schmidt. *Meta-programming techniques for distributed real-time and embedded systems.* In WORDS '02 : Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[Dashofy01] E. M. Dashofy, A. V. d. Hoek & R. N. Taylor. *A highly-extensible, xml-based architecture description language.* In WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), pages 103–112, Washington, DC, USA, 2001.

[Dashofy02a] E. M. Dashofy, A. v. d. Hoek & R. N. Taylor. *An infrastructure for the rapid development of xml-based architecture description languages.* In ICSE '02 : Proceedings of the 24th International Conference on Software Engineering, pages 266–276, Orlando, Florida, 2002. ACM Press.

[Dashofy02b] E. M. Dashofy, A. v. d. Hoek & R. N. Taylor. *Towards architecture-based self-healing systems*. In WOSS '02 : Proceedings of the first workshop on Self-healing systems, pages 21–26, Charleston, South Carolina, USA, 2002.

[Dashofy05] E. M. Dashofy, A. v. d. Hoek & R. N. Taylor. *A comprehensive approach for the development of modular software architecture description languages*. ACM Trans. Softw. Eng. Methodol., vol. 14, no. 2, pages 199–245, 2005.

[Dashofy07] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas & R. Taylor. *Archstudio 4 : An architecture-based meta-modeling environment*. In ICSE COMPANION '07 : Companion to the proceedings of the 29th International Conference on Software Engineering, pages 67–68, Minneapolis, MN, USA, 2007.

[Desnos06] N. Desnos, S. Vauttier, C. Urtado & M. Huchard. *Automating the building of software component architectures*. In V. Gruhn & F. Oquendo, editeurs, Software Architecture : 3^{rd} European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA), Revised selected papers, volume 4344 of *LNCS*, pages 228–235, Nantes, France, September 2006. Springer.

[Desnos08] N. Desnos, M. Huchard, G. Tremblay, C. Urtado & S. Vauttier. *Search-based many-to-one component substitution*. Journal of Software Maintenance and Evolution : Research and Practice, Special issue on Search-Based Software Engineering, vol. 20, no. 5, pages 321–344, September / October 2008.

[D'Souza99] D. F. D'Souza & A. C. Wills. Objects, components, and frameworks with uml : the catalysis approach. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Dubois04] E. Dubois & X. Franch. *Models and processes for the evaluation of cots components*. In ICSE '04 : Proceedings of the 26th International Conference on Software Engineering, pages 759–760, EICC, Scotland, UK, 2004.

[Englander97] R. Englander. Developing java beans. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[Erlikh00] L. Erlikh. *Leveraging legacy system dollars for e-business*. IT Professional, vol. 2, no. 3, pages 17–23, 2000.

[Fallside04] D. C. Fallside & P. Walmsley. *XML Schema Part 0 : Primer Second Edition*. W3C Recommendation, October 2004. http ://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[Feijs98] L. M. G. Feijs & R. d. Jong. *3d visualization of software architectures*. Communications of the ACM, vol. 41, no. 12, pages 72–78, 1998.

[Fialli06] J. Fialli & S. Vajjhala. *JSR 222 : Java Architecture for XML Binding (JAXB) 2.0*. , Sun Microsystems, Inc., 2006. http ://-jcp.org/aboutJava/communityprocess/final/jsr222/index.html.

[Fogel99] K. Fogel. Open source development with cvs. CoriolisOpen Press, Scottsdale, Arizona, 1999.

[Fra] Fractal. *http ://fractal.ow2.org/*.

[Gao03] J. Z. Gao, J. Tsao, Y. Wu & T. H.-S. Jacob. Testing and quality assurance for component-based software. Artech House, Inc., Norwood, MA, USA, 2003.

[Garlan00] D. Garlan. *Software architecture : a roadmap*. In A. Finkelstein, editeur, The Future of Software Engineering. ACM Press, 2000.

[Garlan01] D. Garlan, B. Schmerl & J. Chang. *Using gauges for architecture-based monitoring and adaptation*. In Proc. Working Conf. on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

[Garlan02] D. Garlan, S.-W. Cheng & A. J. Kompanek. *Reconciling the needs of architectural description with object-modeling notations*. Science of Computer Programming, vol. 44, no. 1, pages 23–49, 2002.

[Georgas06] J. C. Georgas, E. M. Dashofy & R. N. Taylor. *Architecture-centric development : a different approach to software engineering*. ACM Crossroads, vol. 12, no. 4, pages 6–6, 2006.

[Haake96] A. Haake & D. Hicks. *Verse : towards hypertext versioning styles*. In HYPERTEXT '96 : Proceedings of the seventh ACM conference on Hypertext, pages 224–234, Washington, DC, USA, March 1996. ACM.

[Hne06] *Dynamic reconfiguration and access to services in hierarchical component models*. In CBSE, pages 352–359, 2006.

[Hnetynka05] P. Hnetynka, F. Plasil, T. Bures, V. Mencl & L. Kapova. *Sofa 2.0 metamodel*. , Dep. of SW Engineering, Charles University, December 2005.

[Hoare78] C. A. R. Hoare. *Communicating sequential processes*. Commun. ACM, vol. 21, no. 8, pages 666–677, 1978.

[Inverardi05] P. Inverardi, H. Muccini & P. Pelliccione. *Dually : Putting in synergy uml 2.0 and adls*. In WICSA '05 : Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pages 251–252, Washington, DC, USA, 2005. IEEE Computer Society.

[ISO99] ISO. Standard 14764 on software engineering - software maintenance. ISO/IEC, 1999.

[Jul] Julia. *http ://fractal.ow2.org/julia/index.html*.

[Kazman98] R. Kazman, S. G. Woods & S. J. Carrière. *Requirements for integrating software architecture and reengineering models : Corum ii*. In WCRE '98 : Proceedings of the Working Conference on Reverse Engineering (WCRE'98), page 154, Honolulu, Hawai, USA, 1998. IEEE Computer Society.

[Khare01] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic & R. N. Taylor. *xadl : Enabling architecture-centric tool integration with xml*. In Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001.

[Kniesel05] G. E. Kniesel & R. E. Filman. *Unanticipated software evolution : Issue overviews.* J. Softw. Maint. Evol., vol. 17, no. 5, pages 307–308, 2005.

[Kramer90] J. Kramer & J. Magee. *The evolving philosophers problem : Dynamic change management.* IEEE Trans. Softw. Eng., vol. 16, no. 11, pages 1293–1306, 1990.

[Kramer98] J. Kramer & J. Magee. *Analysing dynamic change in software architectures : A case study.* In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems, pages 91–100, Annapolis, Maryland, USA, 1998.

[Lassing99a] N. Lassing, D. Rijsenbrij & H. v. Vliet. *Flexibility of the combad architectures.* In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pages 341–355, San Antonio, Texas, USA, February 1999. Kluwer Academic Publishers.

[Lassing99b] N. Lassing, D. Rijsenbrij & H. v. Vliet. *Towards a broader view on software architecture analysis of flexibility.* In APSEC '99 : Proceedings of the Sixth Asia Pacific Software Engineering Conference, page 238, Takamatsu, Japan, December 1999.

[Leclercq07] M. Leclercq, A. E. Ozcan, V. Quema & J.-B. Stefani. *Supporting heterogeneous architecture descriptions in an extensible toolset.* Software Engineering, International Conference on, vol. 0, pages 209–219, 2007.

[Lehman00a] M. M. Lehman & J. C. Fernandez-Ramil. *Towards a theory of software evolution - and its practical impact.* In Proc. International Symposium on Principles of Software Evolution, pages 2–11, 2000.

[Lehman00b] M. M. Lehman, J. C. Fernandez-Ramil & G. Kahen. *Evolution as a noun and evolution as a verb.* SOCE 2000 Workshop on Software and Organisation Co-evolution, July 2000.

[Lientz80] B. P. Lientz & E. B. Swanson. Software maintenance management. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[Madhavji06] N. H. Madhavji, J. Fernandez-Ramil & D. Perry. Software evolution and feedback : Theory and practice. John Wiley & Sons, 2006.

[Magee95] J. Magee, N. Dulay, S. Eisenbach & J. Kramer. *Specifying distributed software architectures.* In Proceedings of the 5th European Software Engineering Conference, pages 137–153, Sitges, Spain, September 1995.

[Magee96] J. Magee & J. Kramer. *Dynamic structure in software architectures.* SIGSOFT Softw. Eng. Notes, vol. 21, no. 6, pages 3–14, 1996.

[Medvidovic96] N. Medvidovic. *Adls and dynamic architecture changes.* In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pages 24–27, San Francisco, California, United States, 1996.

[Medvidovic98] N. Medvidovic, D. S. Rosenblum, & R. N. Taylor. *A type theory for software architectures.* UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, 1998.

[Medvidovic99a] N. Medvidovic. *Architecture-based specification-time software evolution.* PhD thesis, University of California, Irvine, 1999. Chair-Richard N. Taylor.

[Medvidovic99b] N. Medvidovic, D. S. Rosenblum & R. N. Taylor. *A language and environment for architecture-based software development and evolution.* In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pages 44–53, Los Angeles, CA, May 1999.

[Medvidovic00] N. Medvidovic & R. N. Taylor. *A classification and comparison framework for software architecture description languages.* IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70–93, 2000.

[Medvidovic07] N. Medvidovic, E. M. Dashofy & R. N. Taylor. *Moving architectural description from under the technology lamppost.* Information and Software Technology, vol. 49, no. 1, pages 12–31, 2007.

[Mehta00] N. R. Mehta, N. Medvidovic & S. Phadke. *Towards a taxonomy of software connectors.* In ICSE '00 : Proceedings of the 22nd international conference on Software engineering, pages 178–187, New York, NY, USA, 2000. ACM Press.

[Mens08a] T. Mens. *Introduction and roadmap : History and challenges of software evolution.* In Software Evolution, pages 1–11. Springer Berlin Heidelberg, 2008.

[Mens08b] T. Mens & S. Demeyer, editeurs. Software evolution. Springer, 2008.

[Mic95] Microsoft Corporation and Digital Equipment Corporation. *The component object model specification : Draft version 0.9*, October 1995.

[Ning97] J. Q. Ning. *Component-based software engineering (cbse).* Assessment of Software Tools, International Symposium on, vol. 0, page 0034, 1997.

[Obj02] Object Management Group. *Corba components, omg document formal/2002-06-65 edition*, June 2002.

[Oquendo06] F. Oquendo. *Formally modelling software architectures with the uml 2.0 profile for adl.* SIGSOFT Softw. Eng. Notes, vol. 31, no. 1, pages 1–13, 2006.

[Oreizy98a] P. Oreizy & R. Taylor. *On the role of software architectures in runtime system reconfiguration.* In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems, pages 61–70, Annapolis, Maryland, May 1998.

[Oreizy98b] P. Oreizy, N. Medvidovic & R. N. Taylor. *Architecture-based runtime software evolution.* In Proceedings of the 20th international conference on Software engineering(ICSE '98), pages 177–186, Kyoto, Japan, April 1998.

[Oreizy99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum & A. L. Wolf. *An architecture-based approach to self-adaptive software.* IEEE Intelligent Systems, vol. 14, no. 3, pages 54–62, 1999.

[Oussalah04] M. Oussalah, A. Smeda & T. Khammaci. *An explicit definition of connectors for component-based software architecture.* In Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), page 44, Washington, DC, USA, 2004.

[Palsberg92] J. Palsberg & M. I. Schwartzbach. *Three discussions on object-oriented typing.* SIGPLAN OOPS Mess., vol. 3, no. 2, pages 31–38, 1992.

[Perry92] D. E. Perry & A. L. Wolf. *Foundations for the study of software architecture.* ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pages 40–52, 1992.

[Pilato08] C. M. Pilato, B. Collins-Sussman & B. W. Fitzpatrick. Version control with subversion, 2nd Edition. O'Reilly Media, Inc., 2008.

[Plásil98] F. Plásil, D. Bálek & R. Janecek. *Sofa/dcup : Architecture for component trading and dynamic updating.* In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems, pages 43–51, Annapolis, MA, 1998.

[Plasil02] F. Plasil & S. Visnovsky. *Behavior protocols for software components.* IEEE Trans. Softw. Eng., vol. 28, no. 11, pages 1056–1076, 2002.

[Rakic01] M. Rakic & N. Medvidovic. *Increasing the confidence in off-the-shelf components : A software connector-based approach.* In Proceedings of the 2001 symposium on Software reusability(SSR '01), pages 11–18, Toronto, Ontario, Canada, 2001.

[Roh04] S. Roh, K. Kim & T. Jeon. *Architecture modeling language based on uml2.0.* In APSEC '04 : Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), pages 663–669, Washington, DC, USA, 2004. IEEE Computer Society.

[Roshandel04] R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic & N. Medvidovic. *Mae–a system model and environment for managing architectural evolution.* ACM Transactions on Software Engineering and Methodology, vol. 13, no. 2, pages 240–276, 2004.

[Rozanski05] N. Rozanski & E. Woods. Software systems architecture : Working with stakeholders using viewpoints and perspectives. Addison-Wesley Professional, 2005.

[Rumbaugh88] J. Rumbaugh. *Controlling propagation of operations using attributes on relations.* In Proc. of the Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, pages 285–296, 1988.

[Seiwald96] C. Seiwald. *Inter-file branching - a practical method for representing variants.* In ICSE '96 : Proceedings of the SCM-6 Workshop on System Configuration Management, pages 67–75, Berlin, Germany, 1996.

[Shaw95a] M. Shaw. *Making choices : A comparison of styles for software architecture.* IEEE Software, special issue on software architecture, vol. 12, no. 6, pages pp. 27–41, November 1995.

[Shaw95b] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young & G. Zelesnik. *Abstractions for software architecture and tools to support them*. IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 314–335, 1995.

[Shaw96a] M. Shaw, R. DeLine & G. Zelesnik. *Abstractions and implementations for architectural connections*. In Proceedings of the 3rd International Conference on Configurable Distributed Systems(ICCDS '96), pages 2–10, Annapolis, Maryland, 1996.

[Shaw96b] M. Shaw & D. Garlan. Software architecture : perspectives on an emerging discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Sommerville06] I. Sommerville. Software engineering. Addison Wesley, 8th edition, 2006.

[Spencer00] J. Spencer. *Architecture Description Markup Language (ADML) : Creating an open market for IT architecture tools*. Open Group White Paper, September 26 2000.

[Szyperski97] C. Szyperski. Component software : Beyond object-oriented programming. ACM Press and Addison-Wesley, second edition, 1997.

[Taylor96] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. James Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy & D. L. Dubrow. *A component- and message-based architectural style for gui software*. IEEE Trans. Softw. Eng., vol. 22, no. 6, pages 390–406, 1996.

[Taylor09] R. N. Taylor, N. Medvidovic & E. M. Dashofy. Software architecture : Foundations, theory, and practice. John Wiley & Sons, January 2009.

[Urtado98] C. Urtado & C. Oussalah. *Complex entity versioning at two granularity levels*. Information Systems, vol. 23, no. 2/3, pages 197–216, 1998.

[van der Hoek98a] A. v. d. Hoek, D. Heimbigner & A. L. Wolf. *Software architecture, configuration management, and configurable distributed systems : A menage a trois*. Computer Science Technical Report CU-CS-849-98, University of Colorado, 1998.

[van der Hoek98b] A. v. d. Hoek, D. Heimbigner & A. L. Wolf. *Versioned software architecture*. In ISAW '98 : Proceedings of the third international workshop on Software architecture, pages 73–76, New York, NY, USA, 1998. ACM Press.

[van Ommering02] R. v. Ommering. *Building product populations with software components*. In ICSE '02 : Proceedings of the 24th International Conference on Software Engineering, pages 255–265. ACM Press, 2002.

[Westhuizen02] C. v. d. Westhuizen & A. v. d. Hoek. *Understanding and propagating architectural changes*. In WICSA 3 : Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, pages 95–109, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[Yau93] S. S. Yau, J. S. Collofello & T. M. MacGregor. *Ripple effect analysis of software maintenance.* In Software engineering metrics I : measures and validations, pages 71–82. McGraw-Hill, Inc., 1993.

[Zhang08] H. Y. Zhang, C. Urtado & S. Vauttier. *Connector-driven gradual and dynamic software assembly evolution.* In Proceedings of the International Conference on Innovation in Software Engineering (ISE08), Vienna, Austria, December 2008.

[Zhang09] H. Y. Zhang, C. Urtado & S. Vauttier. *Connector-driven process for the gradual evolution of component-based software.* In Proceedings of the 20th Australian Software Engineering Conference (ASWEC2009), Gold Coast, Australia, April 2009.

[Zhang10] H. Y. Zhang, C. Urtado & S. Vauttier. *Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants.* In Actes de la 4ème Conférence francophone sur les Architectures Logicielles, Pau, France, Mars 2010. Cépaduès.

ANNEXE A

SYNTAX OF DEDAL

A.1 DEDAL ADL ARCHITECTURE SYNTAX DEFINITION

A.1.1 Specification

```
specification : :=
  specification specification_name
    component_roles component_role_list
    ( connections connection_list )?
    architecture_behavior architecture_behavior
    versionID revision_numb
    ( pre_version pre_version )?
    ( by change )?
component_role_list : :=
  component_role_name ( ; component_role_name )*
connection_list : :=
  connection ( connection )*
```

```
component_role : :=
  component_role component_role_name
    ( required_interfaces interface_list )?
    ( provided_interfaces interface_list )?
    ( role_behavior component_behavior )?
    ( MinInstanceNbr PositiveInteger )?
    ( MaxInstanceNbr PositiveInteger )?
interface_list : :=
   interface_name ( ; interface_name )*
```

```
interface : :=
  interface interface_name
    interface_direction direction
    implementation implementation_class
interface_direction : := provided | required
```

```
connection : :=
  connection connection_name
    client component_role_name . interface_name
    server component_role_name . interface_name
```

A.1.2 Configuration

```
configuration : :=
  configuration configuration_name
    implements specification_identifier
    component_classes component_class_list
    ( connector_classes connector_class_list )?
    versionID revision_numb
    ( pre_version pre_version )?
    ( by change )?
specification_identifier : :=
  specification_name ( revision_numb )
component_class_list : :=
  component_class_name ( revision_numb ) as component_role_name
  (; component_class_name ( revision_numb ) as component_role_name )*
connector_class_list : :=
  connector_class_name as connection_name
  (; connector_class_name as connection_name )*
```

```
component_type : :=
  component_type component_type_name
    provided_interfaces interface_list
    required_interfaces interface_list
    component_behavior component_behavior
```

```
primitive_component_class : :=
  primitive_component_class component_class_name
    implements component_type_name
    content implementation_class
    ( attributes attribute_list )?
    versionID revision_numb
    ( pre_version pre_version )?
    ( motivation motivation )?
    ( condition condition )?
attribute_list : := attribute (; attribute )*
attribute : := type attribute_name
type :=
  boolean | byte | char | short |
  int | float | long | double | string |
  class_name | interface_name
```

```
composite_component_class : :=
    composite_component_class component_class_name
      implements component_type_name
      content configuration_identifier
      delegated_interfaces delegated_interface_list
      ( attributes attribute_list )?
      versionID revision_numb
      ( pre_version pre_version )
      ( motivation motivation )?
      ( condition condition )?
```

```
configuration_identifier : :=
  configuration_name ( revision_numb )
delegated_interface_list : :=
  provided | required inner_interface as outer_interface
  ( ; provided | required inner_interface as outer_interface )*
inner_interface : :=
  component_class_identifier [ component_role_name ] . interface_name
outer_interface : :=
  component_type_name . interface_name
component_class_identifier : :=
  component_class_name ( revision_numb )
```

```
connector_type : :=
  connector_type connector_type_name
    provided_interfaces interface_list
    required_interfaces interface_list
    connector_protocol connector_protocol
```

```
connector_class : :=
  connector_class connector_class_name
    implements connector_type
    content implementation_class
```

A.1.3 Assembly

```
assembly : :=
  assembly assembly_name
    instance_of configuration_identifier
    component_instances component_instance_list
    ( assembly_constraints assembly_constraint_list )?
    versionID revision_numb
    ( pre_version version )?
    ( by change )?
component_instance_list : :=
  component_instance_name as component_role_name
  ( ; component_instance_name as component_role_name )*
assembly_constraint_list : :=
  assembly_constraint ( ; assembly_constraint )*
```

```
component_instance : :=
  component_instance component_instance_name
    instance_of component_class_identifier
    initiation_state attribute_value_list
    current_state attribute_value_list
attribute_value_list : :=
  attribute_name = attribute_value
  ( ; attribute_name = attribute_value )*
attribute_value : :=
  "an attribute value of the correct type"
```

```
assembly_constraint : :=
  logical_constraint | relational_constraint
logical_constraint : :=
  ( ! assembly_constraint ) |
    ( assembly_constraint ( || | && ) assembly_constraint )
relational_constraint : :=
  ( instance_attribute ( == | != | > | < | >= | <= )
    ( instance_attribute | attribute_value ) )
instance_constraint : :=
  ( ( MinInstanceNbr | MaxInstanceNbr | InstanceNbr )
    ( Component_role_name ) == PositiveInteger )
instance_attribute : :=
  component_instance_name . attribute_name
```

A.2 EVOLUTION EXPRESSION SYNTAX DEFINITION

A.2.1 Version

```
version : :=
   element_name ( revision_numb )
pre_version : :=
   ( element_name )? ( revision_numb )
element_name : :=
  specification_name | configuration_name | assembly_name
  |component_class_name
revision_numb : :=
  nonZero ( digit )* . ( digit )+
```

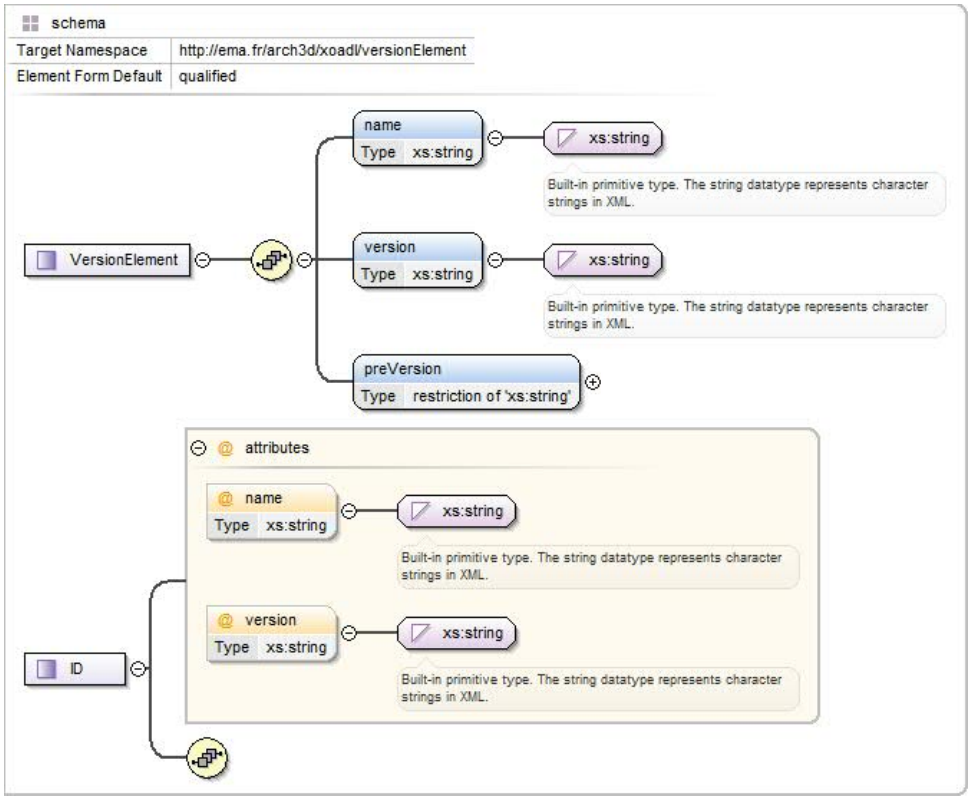
A.2.2 Change

```
1 change : :=
2  change change_name
3    time change_time
4    level initial_level
5    operation change_operation
6    artifact architecture_element is element_name
7    purpose change_purpose
8    origin change_origin ( from change )?
9 time : := static | dynamic
10 level : := specification | configuration | assembly
11 operation : := addition | removal |substitution | modification
12 architecture_element : := specification_element| configuration_element
   | assembly_element
13 specification_element : := component_role | connection |
   architecture_behavior
14 configuration_element : := component_class | connector_class
15 assembly_element : := component_instance | connector_instance |
   assembly_constraint
16 purpose : := corrective | perfective | adaptive
17 origin : := given | generated | propagated
```

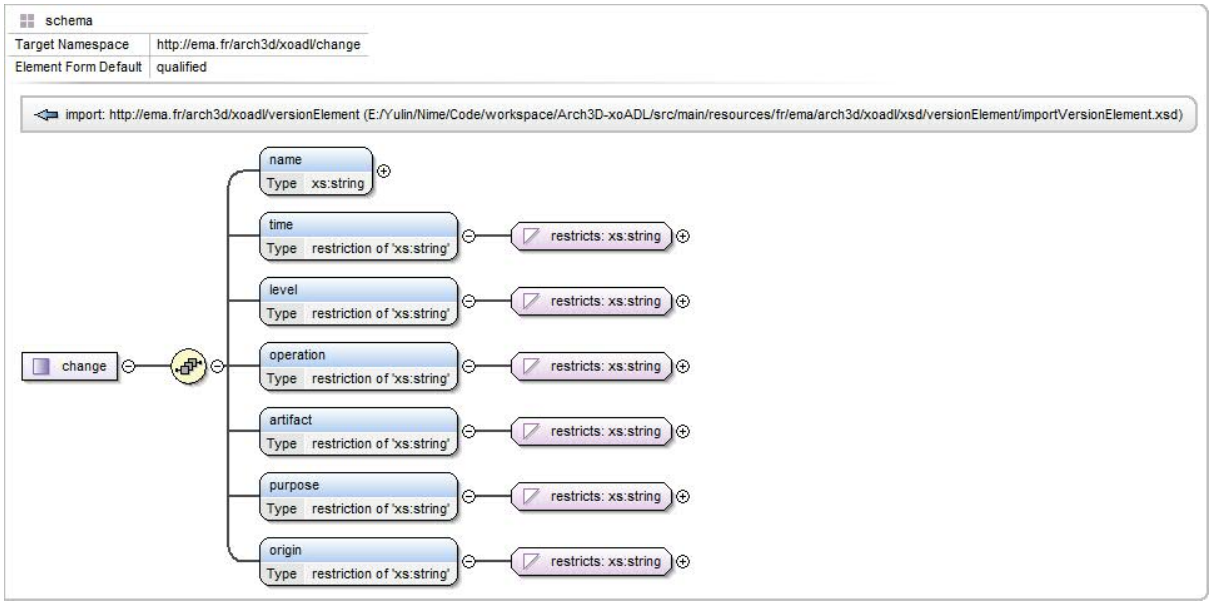
ANNEXE B

XML SCHEMA DEFINITION OF DEDAL

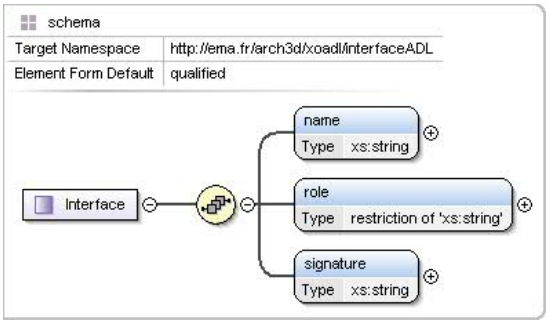
B.1 VERSION ELEMENT



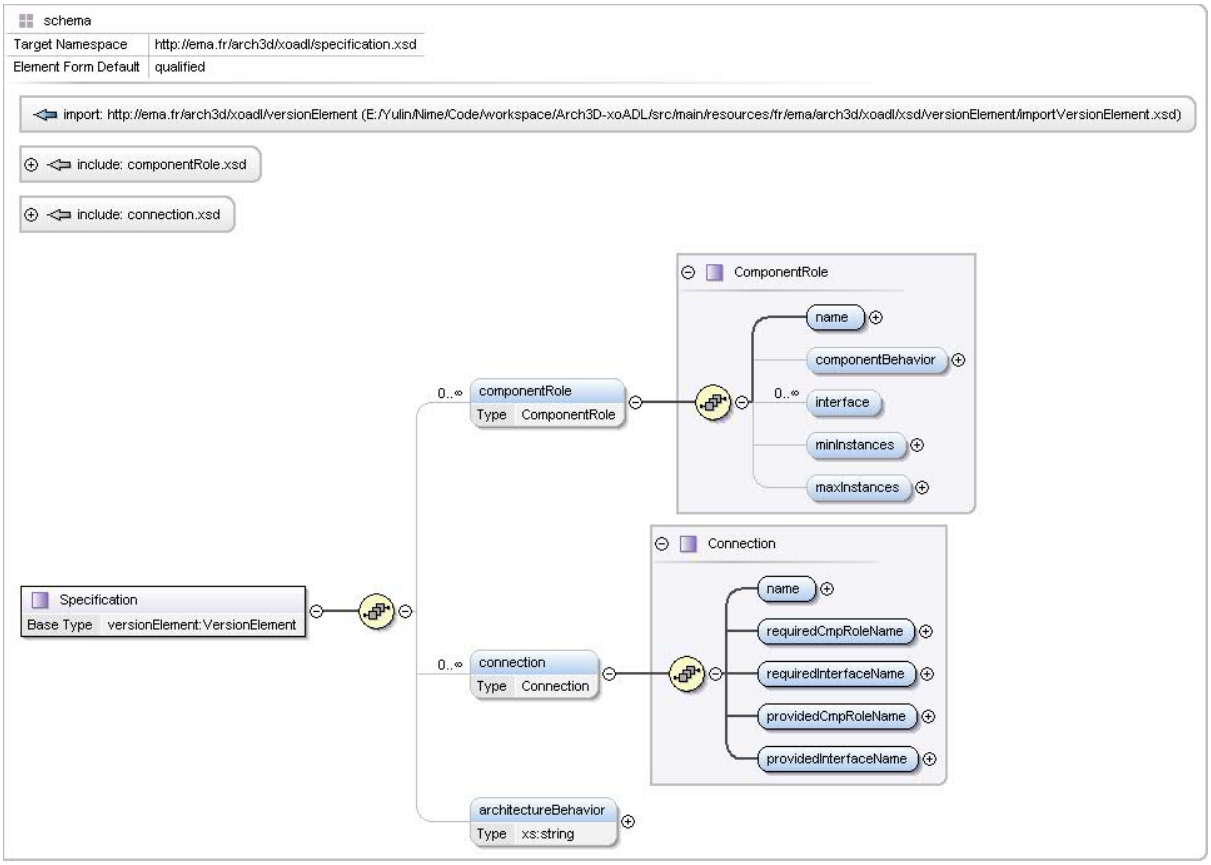
B.2 CHANGE



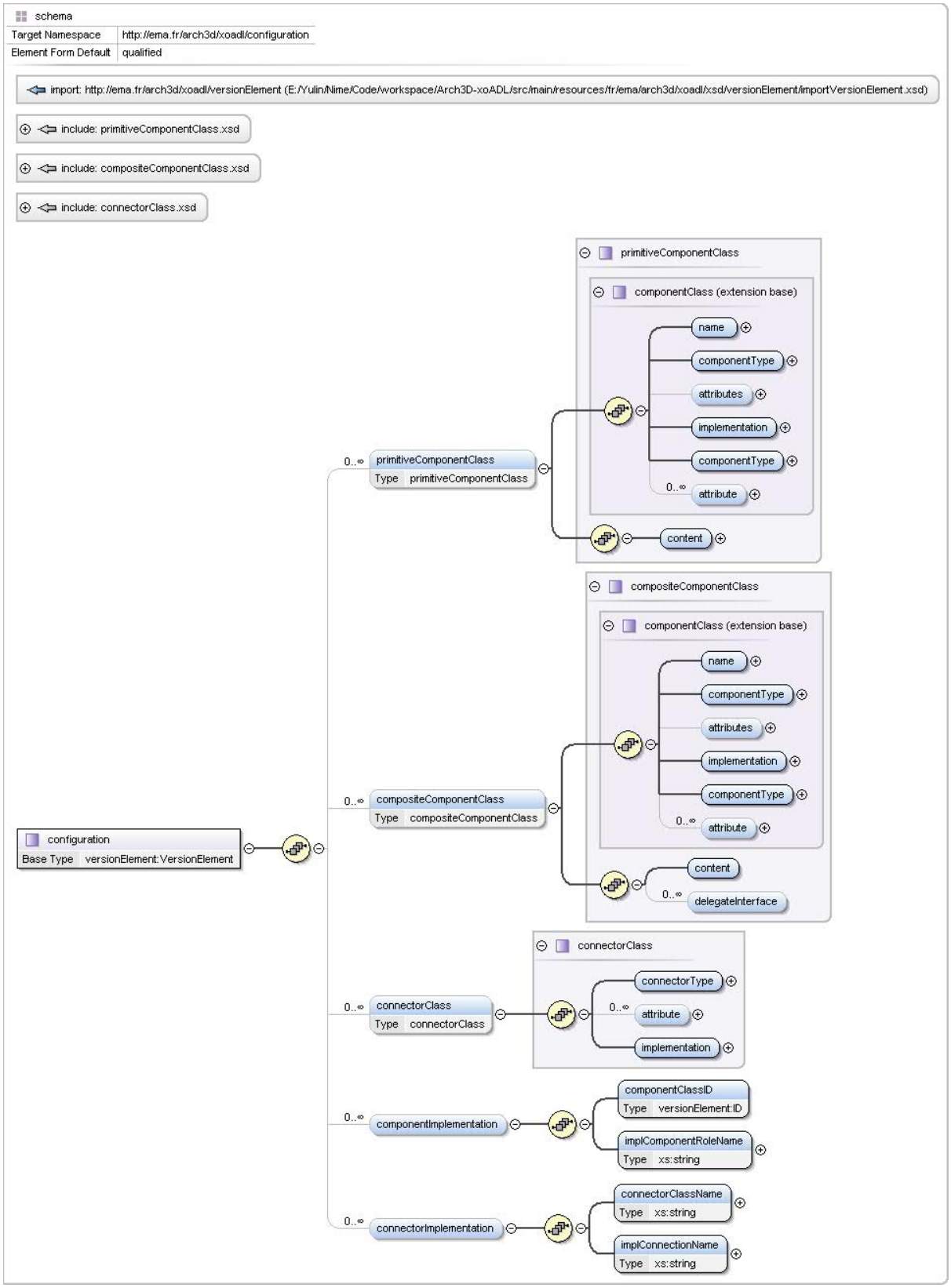
B.3 INTERFACE



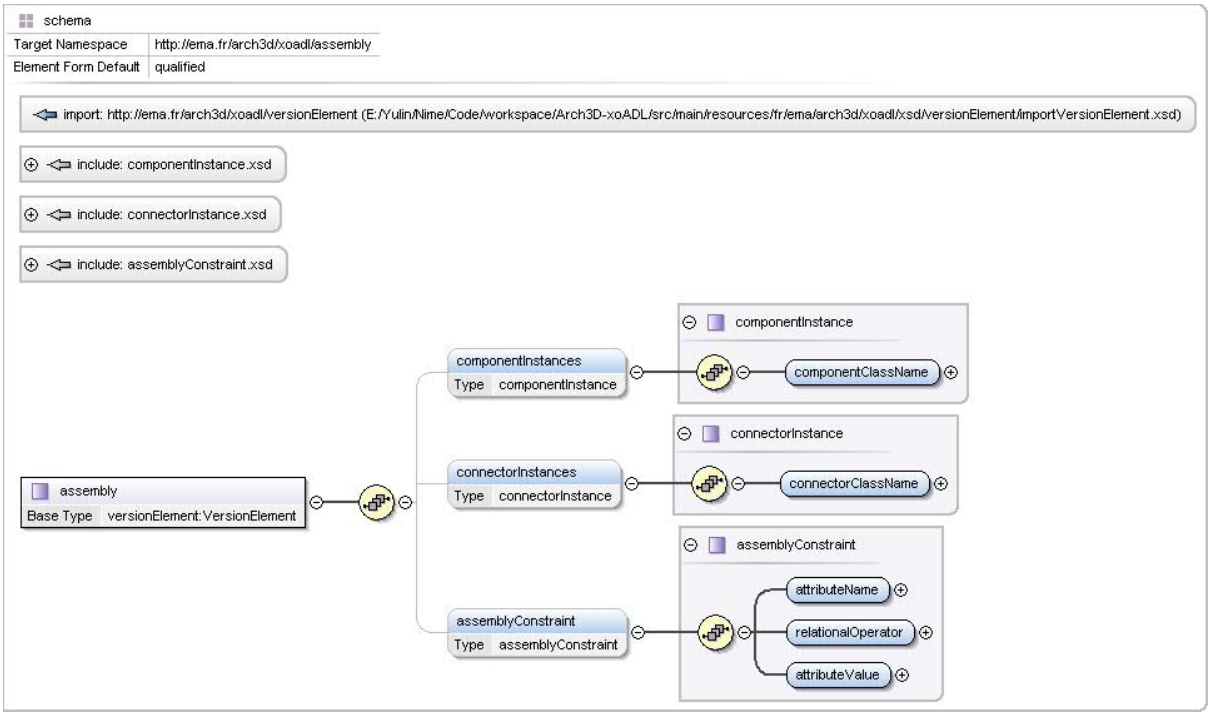
B.4 ABSTRACT ARCHITECTURE SPECIFICATION



B.5 CONCRETE ARCHITECTURE CONFIGURATION



B.6 INSTANTIATED SOFTWARE COMPONENT ASSEMBLY



Annexe C

Résumé en Français

C.1 INTRODUCTION

Le développement à base de composants est la principale réponse proposée à la problématique de l'accroissement de la complexité des logiciels, afin de diminuer les temps et les coûts de développement sans sacrifier la qualité des logiciels produits. Il propose une démarche de construction des logiciels à large maille, par assemblage de blocs de programmation pré-existants – les composants – définis de manière suffisamment indépendante et découplée afin d'être réutilisables dans de multiples contextes [Sommerville06, Crnkovic06]. Le logiciel ainsi produit est décrit sous la forme d'une architecture logicielle, listant les composants utilisés et les connections devant les relier. Pour assurer la gestion d'un logiciel tout au long de son cycle de vie, depuis sa spécification jusqu'à son déploiement puis son exploitation, la définition d'une architecture logicielle doit rassembler des méta-informations correspondant à différents niveaux d'abstraction. La cohérence de la définition d'une architecture repose sur la conformité de sa description à un niveau d'abstraction donné avec sa description au niveau d'abstraction immédiatement supérieur. Ce principe permet de proposer des processus contrôlés de développement et d'évolution des architectures. Malheureusement, aucun Adl ne propose un modèle de définition d'architectures couvrant le cycle de vie d'une architecture. La plupart des Adl, tels que Wright [Allen97b], C2SADEL [Medvidovic99b] et Darwin [Magee96], ne couvrent que les étapes de spécification et d'implantation, négligeant la description du déploiement des architectures (assemblages d'instances des composants). Par ailleurs, les différents niveaux d'abstraction composant la définition d'une architecture ne sont pas explicitement identifiés et séparés. Il est alors difficile d'utiliser les définitions des architectures comme un support efficace aux processus de développement et d'évolution des applications.

Une des contributions principales de cette thèse est Dedal, un Adl permettant de représenter explicitement trois niveaux d'abstraction dans la définition d'une architecture. Ces trois niveaux – la spécification, la configuration et l'assemblage d'une architecture – correspondent aux étapes de conception, d'implémentation et de déploiement d'une architecture. Ainsi, Dedal permet de conserver la trace des décisions des architectes dans un processus de développement (forward engineering). Dedal peut également servir de support à un processus d'évolution contrôlée (reverse engineering), en identifiant clairement la portée d'une modification et les mécanismes de propagation de ses conséquences à l'ensemble de la définition d'une architecture. Ces mécanismes ont pour objectif d'éviter

les problèmes d'érosion et de dérive affectant les définitions des architectures [Perry92].

La suite de ce résumé suit le plan du manuscrit. La section C.3 compare l'expressivité des ADL existants. La section C.4 décrit le modèle de définition d'architectures proposé dans Dedal. La section C.6 définit le processus d'évolution contrôlée pouvant être mis en œuvre à l'aide de Dedal. La section C.8 introduit des perspectives à ce travail.

C.2 DÉVELOPPEMENT À BASE DE COMPOSANTS

La réutilisation est une technique permettant de produire, rapidement et au meilleur coût, du logiciel de qualité. Le développement à base de composants est une approche de développement centré réutilisation. Le développement à base de composant est ainsi spécifiquement organisée autour de l'activité de réutilisation, comme le montre la Figure C.1.

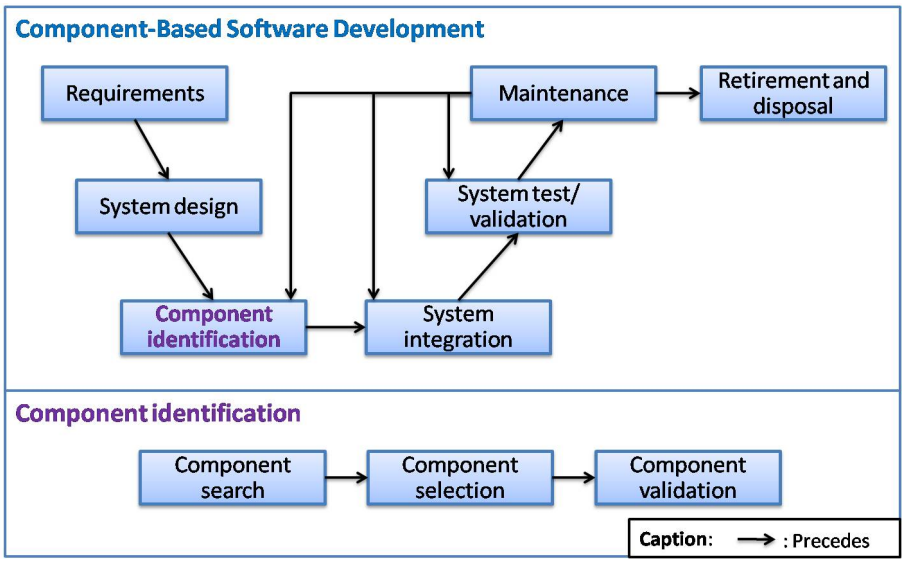


FIGURE C.1 – Le processus de développement à base de composants

1. *Ingénierie des besoins.* Lorsque les besoins du système sont identifiés, les services et les contraintes qui définissent la spécification du système sont raffinés en un ensemble de descriptions de composants, structurés en une architecture. Ces descriptions de composants identifient les types abstraits de composants qui sont nécessaires à l'accomplissement des besoins identifiés par les utilisateurs. En adoptant une approche hiérarchique, les spécifications peuvent être associées à un composant unique de granularité élevée qui encapsule une architecture interne (elle-même à base de composants) qui sera définie à l'étape de conception du système.
2. *Conception du système.*À cette étape, une première architecture conceptuelle a été produite et sert à établir des requêtes (à destination d'un référentiel de composants) pour la recherche de composants existants à réutiliser. A cette étape, l'architecture complète du système doit pouvoir être définie avec des types de composants concrets. L'ensemble des composants tels que spécifiés est étendu et les types des composants sont raffinés de telle façon à correspondre à des types de composants concrets (implantés).

3. *Identification des composants.* Cette étape, typique du développement à base de composants, se découpe en trois sous activités : la recherche de composants, la sélection de composants et la validation, comme montré dans la Fig. C.1. En effet, un type de composant peut être implanté par différentes classes de composants. L'implémentation (la classe de composants) choisie doit répondre à des contraintes telles que définies à l'étape de spécification. Ce sont donc ces qualités non fonctionnelles qui guideront le choix d'une classe de composants parmi celles implantant le type recherché. Cette étape correspond (et se substitue à) l'étape d'implantation dans les cycles de développement traditionnels.
   (a) *Recherche de composants.* La première étape consiste à rechercher des composants disponibles localement ou auprès de fournisseurs de confiance. La recherche de composants consiste à parcourir ou à émettre une requête à destination d'un (ou de) référentiel(s) de composants afin de trouver les composants qui satisfont un critère de recherche. Typiquement, les types sont utilisés pour rechercher les composants qui correspondent aux composants conceptuels définis lors de la phase de conception. Les types de composants existants peuvent être extraits des référentiels et utilisés pour compléter la conception d'une architecture. Ainsi, la réutilisation de composants est mise en œuvre dès la phase de conception grâce à la réutilisation des descriptions de composants conceptuels.
   (b) *Sélection de composants.* Après que le processus de recherche ait identifié les composants candidats, les composants concrets qui vont être effectivement mis en œuvre dans l'architecture doivent être choisis parmi les candidats. Ce choix peut se baser sur des critères non fonctionnels (coût, confiance dans le fournisseur, rapidité, etc.).
   (c) *Validation des composants.* Une fois que les architectes ont sélectionné les composants qu'ils souhaitent inclure dans le système, ces composants peuvent être vérifiés, de façon à vérifier qu'ils se comportent comme annoncé par le fournisseur. Cette étape correspond à l'étape de tests unitaires des processus de développement classiques. La validation peut être omise si le fournisseur a la confiance de l'architecte.
4. *Intégration.* L'étape d'intégration vise à produire un assemblage exécutable de composants. Elle consiste à récupérer les composants dans les référentiels, les déployer dans un conteneur de composants et les connecter les uns aux autres grâce à des connecteurs et en accord avec la définition de l'architecture du système.
5. *Test / validation.* Cette étape correspond à l'étape de test dans les processus de développement classiques. Le comportement du système complet est comparé aux spécifications pour vérifier que les exigences des utilisateurs sont satisfaites.
6. *Maintenance.* Tant que les exigences restent inchangées, la maintenance consiste principalement à itérer l'étape d'identification de composants pour rechercher de nouveaux composants ou de nouvelles versions de composants existants. Un des objectifs de la maintenance peut être de faire évoluer l'architecture ou de corriger ou améliorer le comportement global du système (par exemple, soit en choisissant des composants dans une autre technologie, soit des composants dont des erreurs ont été corrigées ou qui tirent mieux partie des ressources du système).

C.3 ÉTAT DE L'ART DE L'ARCHITECTURE LOGICIEL

La définition d'une architecture synthétise les décisions prises lors du processus de conception d'un logiciel [Taylor09]. Elle est exprimée à l'aide d'un ADL qui permet de décrire la structure d'un logiciel en listant les composants qui le constituent et les connexions qui relient leurs interfaces. Des attributs de qualité sont parfois proposés pour décrire les aspects non fonctionnels du logiciel (*e.g.,* xADL [Dashofy02a]). Le comportement dynamique est souvent également décrit (*e.g.,* C2SADEL [Medvidovic99b], Wright [Allen97b], SOFA [Plasil02]) à l'aide de différents moyens (*e.g.,* message-based communication, CSPs, expressions régulières). Les ADL décrivent en général de manière distincte la spécification et la configuration d'une architecture. La spécification définit l'ensemble des classes de composants et de connecteurs qui composent une architecture (un type de logiciel). Parfois, les types des connecteurs ne sont pas explicitement définis mais déduits des types des interfaces connectées (*e.g.,* Darwin [Magee96]). Les types et les classes de composants utilisés sont définis dans une même définition (*e.g.,* Wright [Allen97b]) ou dans des définitions séparées (*e.g.,* C2SADEL [Medvidovic99b]), ce qui facilite leur réutilisation et leur évolution. La configuration d'une architecture (une instance exécutable d'un logiciel) définit les instances de composants et de connecteurs qui doivent être créées lors du déploiement du logiciel.

Quand un logiciel est trop complexe pour être facilement décrit, deux mécanismes sont utilisés pour diviser la définition de son architecture en modèles plus simples. La composition hiérarchique permet de représenter le logiciel à différents niveaux de granularité (*e.g.,* Darwin [Magee96], SOFA [Plasil02] ou Fractal ADL [Bruneton06]). L'architecture d'un logiciel est définie par ensemble de composants abstraits, de forte granularité, représentant chacun un sous-système du logiciel. La définition de chaque composant peut être détaillée sous la forme d'une architecture qui définit la structure du sous-système qu'il représente. Cette décomposition hiérarchique est itérée jusqu'à la définition de composants élémentaires. La définition de l'architecture du logiciel est obtenue par la composition des architectures de ses sous-systèmes. La décomposition en différents points de vue (*e.g.,* vue statique et dynamique en UML [Booch05]) fait coexister plusieurs définitions d'une architecture, afin d'isoler et détailler des descriptions spécifiques. La définition globale de l'architecture est obtenue par une synthèse de ces points de vue juxtaposés.

Les architectures devraient pouvoir être décrites à différents stades de leur conception. Certains travaux, UML [Booch05] ou [Taylor09], décrivent des concepts proches. UML fournit une large palette de modèles (de structure, de collaboration, de comportement, de déploiement, ...) couvrant l'ensemble du cycle de développement d'un logiciel mais ces modèles, orientés objets, n'ont pas encore été transposés au modèle orienté composants proposé dans UML2.0. [Taylor09] distinguent deux niveaux de description, utilisés lors de la conception et du codage, appelés respectivement l'architecture perspective (as-intended) et l'architecture descriptive (as-realized). [Garlan01] définissent une infrastructure à trois niveaux (tâche, modèle et exécution) et mettent en évidence l'utilité d'une telle infrastructure dans la gestion de l'évolution dynamique des logiciels. Cependant, aucun de ces travaux ne propose d'ADL permettant de décrire ces différents niveaux de description adaptés au support des différentes étapes du cycle de vie d'une architecture. Dedal remédie à ce problème.

C.3.1 Support des trois niveaux de description des architectures

Dans cette section, nous proposons une comparaison du support, par les langages présentés précédemment, des trois niveaux de description des architectures. Les résultats de cette comparaison sont synthétisés dans Tab. C.1.

ADL	Spécification	Configuration	Assemblage
C2SADEL	✓	✓	×
Wright	×	✓	×
Darwin	×	✓	×
Unicon	×	✓	×
SOFA 2.0	×	✓	×
Fractal ADL	×	✓	×
xADL 2.0	×	✓	✓

TABLE C.1 – Les trois niveaux de description dans les ADLs existants

C.3.2 Notre vision du processus de développement à base de composants

La plupart des ADL existants ne sont pas adaptés au développement de logiciel à base de composants. L'activité de réutilisation (*cf.* Fig. C.1) ne peut pas ne pas avoir d'impact sur l' ADL lui-même. La figure C.2 illustre notre vision d'un tel cycle de développement, qui est comparable à celle proposée par Crnkovic, Chaudron *et al.* [Crnkovic06, Chaudron08]. Cette proposition met l'architecture au centre du processus de développement et se concentre ainsi sur les descriptions architecturales produites à chaque étape du développement. Afin de ne pas compliquer notre propos, ce processus ne décrit que les activités liées à la réutilisation de composants issus d'un référentiel (dans lequel ils sont stockés et indexés) et ne décrit pas comment les composants sont :

- adaptés s'il n'existe aucun composant qui corresponde exactement aux spécifications,
- développés entièrement si aucun composant satisfaisant ou presque satisfaisant est trouvé,
- testés et intégrés,
- physiquement déployés.

Après une étape classique d'analyse du besoin, les architectes conçoivent une première définition de l'architecture du logiciel, appelé spécification, définissant de manière abstraite les différents types de composants qui permettent de répondre aux exigences du cahier des charges. Une deuxième étape, appelée configuration, consiste à rechercher et à sélectionner dans le répertoire de composants les classes de composants qui vont constituer l'implémentation concrète de cette architecture. Une troisième étape, appelée assemblage, consiste à décrire comment doivent être instanciés et initialisés les composants lors du déploiement du logiciel.

Nous proposons de découper la définition d'une architecture en différents niveaux d'abstraction reflétant les étapes du processus de développement. Ils permettent de disposer de modèles adaptés aux objectifs de chaque étape ; ils permettent de garder une trace des décisions de conception prises par les architectes au fil du processus ; ils

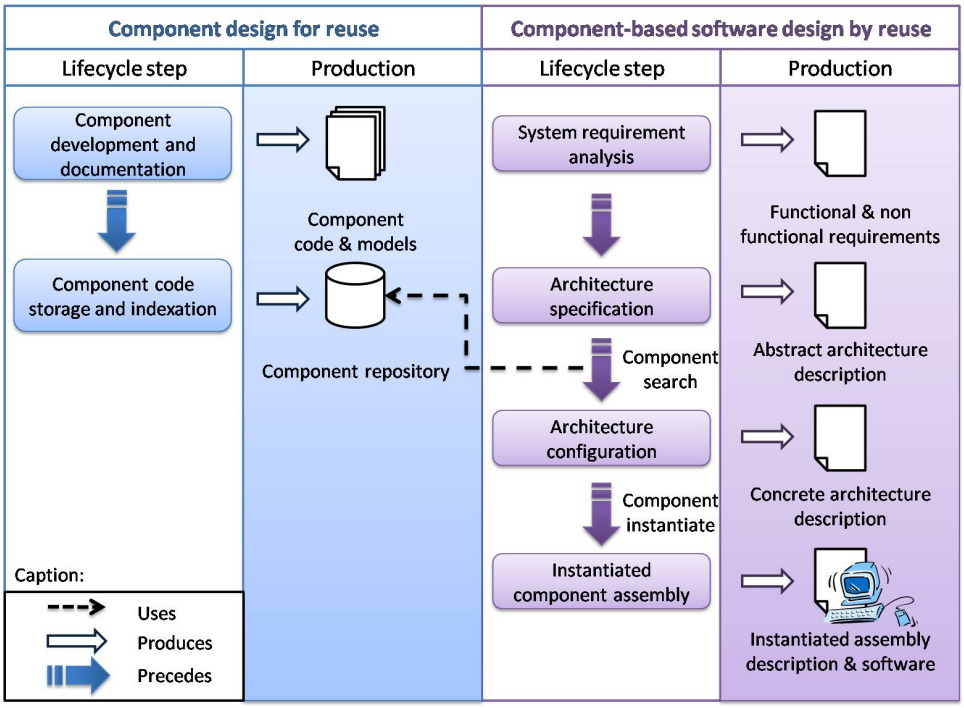


FIGURE C.2 – Component-based software development process

permettent de réutiliser les modèles génériques des architectures pour en dériver des versions spécifiques (gestion de ligne de produits) ; ils permettent enfin de contrôler les évolutions des architectures en imposant le maintien de la cohérence entre les différents niveaux de description d'une architecture. Ces différentes représentations des architectures doivent être représentées par un ADL adéquat.

C.4 DEDAL, UN ADL À TROIS DIMENSIONS

Cette section présente la première partie de notre proposition : Dedal, un langage de description d'architecture qui permet la définition d'une architecture en trois modèles distincts ou dimensions. La figure C.3 décrit l'architecture d'un système de location de vélos (Bike Rental System ou BRS) qui servira d'exemple dans cet article. Le composant *BikerGUI* gère l'interface utilisateur. Il collabore avec le composant *Session* qui gère l'exécution des commandes de l'utilisateur. Il collabore avec les composants *Account* et *BikeCourse* qui sont chargés d'identifier l'utilisateur, de vérifier le solde de son compte, de lui affecter un vélo disponible et de calculer le prix du trajet effectué lorsque le vélo est rendu. Les composants *BikeCourse* et *BikeCourseDB* serviront d'exemple, dans la section suivante, pour présenter les concepts et la syntaxe de Dedal.

C.4.1 Spécification abstraite d'une architecture

La spécification (abstraite) d'une architecture est la première dimension de la description d'une architecture. Elle permet à un architecte de définir une architecture idéale capable de satisfaire les exigences du cahier des charges. Une spécification ne décrit pas les composants et les connexions de l'architecture en termes de types concrets, ni même

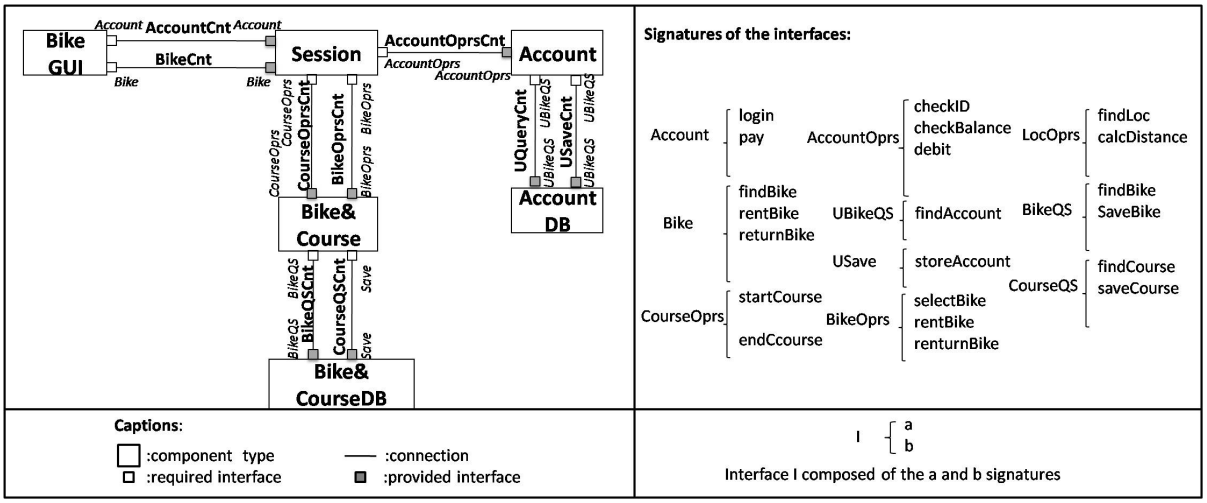


FIGURE C.3 – Architecture du système BRS

de définitions complètes de types abstraits. A l'instar du concept de rôle utilisé dans la définition des collaborations en UML [Booch05], nous proposons de définir la spécification d'une architecture par la description de l'ensemble des rôles joués par les composants qui participent à la réalisation de l'architecture. La définition concrète d'une architecture peut alors être dérivée de sa spécification en sélectionnant des classes de composants capables de remplir les rôles définis. La spécification d'une architecture peut ainsi être comparée aux architectures perspectives introduites par [Taylor09]. La spécification d'une architecture contient la définition de sa structure (ensemble de rôles, de connections) et de son comportement (protocoles). Elle contient également des informations sur le versionnement des différents éléments qui la composent (voir la section C.6). La figure C.4 fournit un extrait d'une spécification d'architecture avec Dedal.

Les rôles. Dans une spécification d'architecture, un rôle définit les responsabilités d'un composant, c'est-à-dire le comportement individuel qui est attendu du composant pour participer à une exécution correcte du comportement d'ensemble de l'architecture. Un rôle est décrit comme un type abstrait. Il définit la liste des interfaces et le protocole de comportement qu'un composant doit posséder pour pouvoir remplir le rôle attendu dans l'architecture. Dedal utilise la syntaxe proposée dans SOFA [Plasil02] pour décrire les comportements sous la forme d'expressions régulières [1]. Le protocole de comportement décrit les appels de méthodes que le composant doit recevoir et envoyer au travers de ses interfaces. Le type abstrait défini par un rôle n'est pas la spécification complète d'un composant mais décrit l'une de ses utilisations particulières (un ensemble de collaborations). Toutes les classes de composants conformes à la définition d'un rôle peuvent être utilisées dans l'implantation concrète de l'architecture. Un rôle peut correspondre à des fonctionnalités récurrentes (login, panier d'achats, etc.) et être utilisable dans la définition de différentes architectures. Les rôles sont ainsi définis indépendamment des spécifications d'architectures. Les figures C.5 et C.6 présentent respectivement les rôles *BikeCourse* et

1. `!i.m` et `?i.m` décrivent l'envoi et la réception d'un appel de méthode, respectivement. `A+B` signifie `A` ou `B` (alternative) et `A;B` signifie `A` suivi de `B` (séquence).

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB
  ...
 connections
  connection connection1
   client BikeCourse.BikeQS
   server BikeCourseDB.BikeQS
  connection connection2
   client BikeCourse.CourseQS
   server BikeCourseDB.CourseQS
  ...
 architecture_behavior
  (?BikeCourse.BikeOprs.selectBike;
  !BikeCourse.BikeQS.findBike;
  ?BikeCourseDB.BikeQS.findBike;)
  +
  (?BikeCourse.CourseOprs.startC;
  !BikeCourse.CourseQS.findCourse;
  ?BikeCourseDB.CourseQS.saveCourse;)
  ...
 version 1.0
```

FIGURE C.4 – *Extrait de la spécification de l'architecture du système BRS*

BikeCourseDB.

Les connections. Les connections définissent comment les interfaces des composants sont reliées pour assurer le fonctionnement de l'architecture. La figure C.4 définit ainsi les connections nécessaires à l'utilisation des rôles *BikeCourse* et *BikeCourseDB* dans l'architecture du BRS.

Le comportement d'architecture. Le comportement d'architecture définit, sous la forme d'un protocole de comportement, comment sont combinés les comportements des composants qui la composent. Le protocole de comportement de l'architecture doit être conforme aux connections spécifiées et aux comportements individuels des composants définis dans les rôles utilisés. Cette conformité peut être vérifiée par un calcul d'inclusion de traces d'exécution comme proposé dans SOFA [Plasil02]. La figure C.4 propose une définition du comportement de l'architecture BRS. Il est intuitivement possible de vérifier sur cet exemple simple que le comportement défini pour l'architecture combine de manière cohérente le comportement défini dans les rôles *BikeCourse* et *BikeCourseDB*.

C.4.2 Configuration concrète d'une architecture

La configuration (concrète) d'une architecture est la deuxième dimension de sa définition. Elle a pour objectif de définir les classes de composants et de connecteurs utilisées pour implanter l'architecture. Elle correspond aux architectures descriptives proposées par [Taylor09]. La configuration d'une architecture est dérivée de sa spécification en

```
1 component_role BikeCourse
2  required_interfaces BikeQS; CourseQS
3  provided_interfaces BikeOprs; CourseOprs
4  component_behavior
5   (?BikeOprs.selectBike, !BikeQS.findBike;)
6   +
7   (?CourseOprs.startC, !CourseQS.findCourse;)
```

FIGURE C.5 – *Spécification du rôle de composant BikeCourse*

```
1 component_role BikeCourseDB
2  provided_interfaces BikeQS; CourseQS
3  component_behavior
4   ?BikeCourseDB.BikeQS.findBike; + ?BikeCourseDB.BikeOprs.findCourse;
```

FIGURE C.6 – *Spécification du rôle de composant BikeCourseDB*

utilisant la description des rôles comme critère de recherche et de sélection dans une bibliothèque de composants [Arévalo09]. Les types des classes de composants sélectionnées doivent correspondre aux types de leur(s) rôle(s) dans l'architecture. Cette correspondance n'est pas exacte ou stricte mais repose sur une notion de compatibilité entre types de composants : le type d'une classe de composants peut se substituer au type de son rôle dans la spécification [Desnos08]. La configuration d'une architecture, par la sélection d'un ensemble de classes de composants, définit une implantation particulière de l'architecture. Différentes configurations, correspondant à différentes implantations d'une architecture, peuvent être définies pour une même spécification, afin d'obtenir des qualités non fonctionnelles spécifiques (ligne de produit) ou pour tracer les évolutions des classes de composants (versionnement). La définition d'une configuration contient des informations permettant de tracer la définition et les évolutions des différentes versions d'une configuration. La figure C.7 donne un exemple de configuration correspondant à la spécification présentée figure C.4. Elle définit l'utilisation des classes de composants *BikeTrip* et *BikeTripDB* dans les rôles *BikeCourse* et *BikeCourseDB*.

```
1 configuration BRSConfig
2  implements BRSSpec (1.0)
3  component_classes
4   BikeTrip (1.0) as BikeCourse;
5   BikeTripDBClass (1.0) as BikeCourseDB
6  version 1.0
```

FIGURE C.7 – *Exemple de configuration d'architecture pour le système BRS*

Les classes de composants. Dedal permet de décrire les classes de composants qui sont utilisées dans les configurations d'architecture. La description d'une classe de composants est définie comme un type abstrait (méta-modèle) qui documente ses fonctionnalités et son comportement. On distingue les classes de composants primitifs et composites. Une

classe de composants primitifs (*e.g., BikeTrip* présentée dans la figure C.8) est définie par l'ensemble de ses interfaces, son protocole de comportement, sa version et la classe qui contient son implémentation (code exécutable stocké dans la bibliothèque de composants). Les classes de composants composites se distinguent des classes de composants primitifs en étant implantées par un ensemble de composants internes. L'organisation des composants internes est définie par une configuration d'architecture décrivant la structure interne du composant composite. Il est possible de définir, tant pour les classes de composants primitifs que composites, des attributs permettant de paramétrer et de gérer leur instanciation lors du déploiement des architectures. De même, la version d'une classe peut être définie par son numéro de révision et la motivation de sa création (évolution corrective ou perfective). Les motivations sont utilisées pour gérer l'évolution graduelle des architectures par substitution de composants versionnés [Zhang09]. Ainsi, contrairement aux ADL existants, il est possible de représenter, sous la forme de versions multiples, les différentes implantations existantes d'une classe de composants.

```
1  component_class BikeTrip
2   implements BikeCourse
3   using fr.ema.BikeTripImpl
4   required_interfaces BikeQS; CourseQS; LocOprs
5   provided_interfaces BikeOprs; CourseOprs
6   component_behavior
7    (?BikeOprs.selectBike;
8  !LocOprs.findStation;
9  !BikeQS.findBike;)
10   +
11   (?CourseOprs.startC,
12 !CourseQS.findCourse;)
13  version 1.0
14  attributes company; currency
```

FIGURE C.8 – *Définition de la classe du composant primitif BikeTrip*

Les classes de connecteurs. Il est possible de définir explicitement le type de connecteur utilisé pour réaliser une connexion dans l'architecture. Par défaut, les connecteurs sont considérés comme des entités génériques administrées et fournies par les conteneurs de composants (environnements d'exécution) dans lesquels les architectures sont déployées. La définition explicite d'une classe de composants permet à l'architecte de préconiser l'utilisation de mécanismes de communication, de coordination et de médiation spécifiques, pour répondre par exemple à des besoins d'adaptation (de types ou de protocoles).

C.4.3 Assemblage d'une architecture

L'assemblage d'une architecture décrit l'ensemble des instances, composants et connecteurs, qui la composent. C'est la troisième dimension de la description d'une architecture, correspondant à son déploiement et à son exécution. Elle permet de décrire des paramétrages spécifiques des états des composants (attributs), correspondant à différentes utilisations de l'architecture. Ces paramétrages sont exprimés par des affectations de

valeurs ou des contraintes. Différents assemblages peuvent être définis pour une même configuration, permettant de répertorier les décisions de conception correspondant à ces différents contextes d'utilisation [Shaw96b]. Des informations de versionnement permettent de représenter les multiples versions d'un assemblage et de tracer leur évolution. La figure C.9 propose la définition d'un assemblage qui instancie la configuration présentée sur la figure C.7.

```
1 assembly BRSAss
2  instance_of BRSConfig (1.0)
3  component_instances
4   BikeTripC1 as BikeCourse;
5   BikeTripDBC1 as BikeCourseDB;
6  assembly_constraints
7   BikeCourse.currency="Euro";
8   BikeCourseDB.company=BikeCourse.company
9  version 1.0
10  component_instance BikeTripC1
11   instance_of BikeTrip (1.0)
12  component_instance BikeTripDBC1
13   instance_of BikeTripDBClass (1.0)
```

FIGURE C.9 – *Assemblage de composants pour le BRS*

Les instances de composants. La description d'une instance de composant définit son nom, sa classe et les valeurs spécifiques affectées à ses attributs. Les noms des instances de composants sont utilisés dans la définition des assemblages pour identifier leurs rôles. La structure d'un assemblage n'est pas nécessairement isomorphe à celle de la configuration associée, une instance de composant pouvant y jouer plusieurs rôles (décision de conception). La définition des instance de composants est extérieure et indépendante de la définition des assemblages. Il est ainsi possible de définir des bibliothèques d'instances de composants réutilisables, correspondant par exemple à des préférences ou à des collections de données métier.

Les contraintes d'assemblage. Les contraintes d'assemblage définissent les conditions qui doivent être vérifiées par les attributs des composants pour respecter la cohérence d'une architecture. Les contraintes sont exprimées sur les rôles et sont appliquées sur les composants associés à ces rôles. Les contraintes définissent ainsi de manière explicite des conditions génériques qui sont respectées dans toutes les instanciations d'une architecture. Pour l'instant, les contraintes proposées en Dedal définissent les valeurs imposées à certains attributs, des conditions entre attributs et des contraintes de cardinalités pour les connexions sur les interfaces. Des exemples de contraintes sont proposées sur la figure C.9. La valeur de l'attribut *currency* du composant jouant le rôle *BikeCourse* est fixée à la valeur *Euro*. De même, les valeurs de l'attribut *company* des composants jouant les rôles *BikeCourse* et *BikeCourseDB* doivent être égales. Cette dernière contrainte impose que le composant *BikeCourse*, qui implante la logique métier, utilise un composant d'accès aux données métier *BikeCourseDB* de la même société. Les types de contraintes proposés

sont très simples et ne permettent pas l'expression de schéma complexes tels que des alternatives ou des négations, ni la propagation de contraintes ou la résolution d'éventuels conflits. La définition de nouveaux types de contraintes, en s'inspirant de l'expression de styles architecturaux à l'aide de contraintes [Allen97b, Cheng02], est l'une des perspectives de ce travail.

Cohérence entre la configuration et l'assemblage d'une architecture. La vérification de la cohérence entre la définition d'une architecture au niveau configuration et au niveau assemblage est directe. Un composant désigné dans l'assemblage pour jouer un rôle dans l'architecture doit être une instance de la classe de composants désignée dans la configuration pour implanter ce rôle. Les valeurs des attributs des composants doivent vérifier les contraintes d'assemblage définies.

C.5 ÉTAT DE L'ART DE L'ÉVOLUTION LOGICIEL

C.5.1 Taxonomie des changements

L'objectif de cette taxonomie est de mieux comprendre et de comparer la façon dont les ADLs existants modélisent le changement et implémentent le processus d'évolution. Les caractéristiques choisies ainsi que leurs valeurs possibles sont listées dans la Table C.2.

Characteristiques du changement	**Valeurs**
Moment du changement	▶statique, ▶au chargement, ▶dynamique
Anticipation	▶changement anticipé, ▶changement non anticipé
Type de changement	▶sémantique, ▶structurel
Objectif du changement	▶correctif, ▶perfectif, ▶adaptatif, ▶preventif
Artéfact sujet au changement	▶composant, ▶connecteur, ▶connexion
Initiation du changement	▶spécification, ▶configuration, ▶assemblage
Opérations de changement	▶addition, ▶suppression et ▶substitution
Activités de l'évolution d'architecture	**Valeurs**
Vérification de cohérence	▶comportement, ▶interaction, ▶cohérence du raffinement
Analyse d'impact	▶vertical, ▶horizontal
Test d'évolution	▶oui, ▶non
Propagation des changements	▶verticale, ▶horizontale
Versionement	▶fondé sur les états, ▶fondé sur les changements

TABLE C.2 – Caractéristiques et activités liées au changement avec leurs valeurs possibles

C.5.2 Support de l'évolution dans les ADL

La table C.3 propose une comparaison des ADL principaux selon la taxonomie proposée.

Caractéristiques du changement	C2SADEL	Darwin	Dynamic Wright	SOFA2.0	xADL2.0	MAE
Moment du changement	dynamique	dynamique	dynamique	dynamique	dynamique	dynamique
Anticipation	changement non anticipé	changement anticipé, changement non anticipé	changement anticipé	changement anticipé	changement non anticipé	changement non anticipé
Type de changement	structurel	structurel	structurel	structurel	structurel	sémantique
Objectif du changement	—	—	—	—	—	perfectif
Initiation du changement	configuration	configuration	configuration	configuration	configuration	configuration
Opérations de changement et Artéfact sujet au changement	addition et suppression de composants, connecteurs ou connexion	addition et suppression de composants ou connexion	addition et suppression de composants ou connexion	addition et suppression de composants, connexion ou interfaces de composant composite	addition et suppression de composants, connecteurs ou connexion	substitution de composant
Activités de l'évolution d'architecture	**C2SADEL**	**Darwin**	**Dynamic Wright**	**SOFA2.0**	**xADL2.0**	**MAE**
Vérification de cohérence	Refinement consistency checking	State consistency checking	Name, interaction and deadlock consistency checking	Behavior consistency checking	—	Sub-type consistency checking
Analyse d'impact	—	horizontal	—	—	—	—
Test d'évolution	—	–	—	—	—	Perfectif test pour substitution de composant
Propagation des changements	horizontal (top-down)	horizontal(top-down)	—	—	horizontal (top-down)	—
Versionnement	—	—	—	fondé sur les états	—	fondé sur les changements

TABLE C.3 – Comparaison des caractéristiques et des activités liées au changement dans les ADL existants

Dans les ADL existants, les évolutions sont généralement initiées à l'étape de configuration des architectures (*e.g.,* C2SADEL [Oreizy98b, Medvidovic96], Darwin [Magee96], Wright [Allen97c]). Ces ADL ne représentent pas les assemblages et ne gèrent donc pas l'évolution pendant l'exécution. La cohérence entre configurations et spécifications est assurée (*e.g.,* C2SADEL, Wright) mais la propagation des changements est limitée. A notre connaissance, les changements sont uniquement propagés dans cadre d'une ingénierie directe ou d'une re-conception (*e.g.,* C2SADEL, Darwin). Ce mécanisme s'applique aux évolutions réalisées par un architecte sur les niveaux conceptuels puis propagées pour réutiliser, par versionnement, les implantations existantes. Ainsi, les évolutions de composants ou d'architectures qui interviennent à l'exécution (logiciels autonomiques, environnements ouverts, etc.) sont une exception. MAE [Roshandel04] propose une gestion de configuration contrôlant les versions des composants à l'exécution, mais ne fournit aucun support pour conserver une trace de ces nouvelles architectures par le versionnement des configurations ou des spécifications d'architectures correspondantes. Les trois niveaux de représentation de Dedal, associés à un outillage adapté, en particulier à l'exécution [Zhang09], permettent de gérer l'évolution directes et la rétro-évolution, afin d'éviter l'apparition d'incohérences (érosion ou dérive).

C.6 PROCESSUS D'ÉVOLUTION CENTRÉ ARCHITECTURE

Le développement à base de composants est un domaine qui a été l'objet de nombreuses recherches. Peu de travaux, en comparaison, s'intéressent à définir des processus d'évolution adaptés aux applications développées à base de composants. L'*évolution logicielle* est définie comme étant la collection de toutes les activités de développement qui permettent de générer une nouvelle version d'un logiciel à partir d'une version opérationnelle plus ancienne de celui-ci [Lehman00a]. Dans le contexte du développement à base de composants, la définition ainsi que les processus d'évolution doivent être repensés [Yau93, Bennett00, Kazman98]. La figure C.10 illustre notre vision d'un tel processus d'évolution.

Nous pensons que les différentes descriptions architecturales (ou différentes dimensions) – spécification, conception et assemblage – peuvent chacune être à l'origine d'un changement. En effet, selon les contextes, l'évolution d'un logiciel peut intervenir pendant sa spécification ou sa conception (*e.g.,* si le mécanisme de gestion de l'évolution est intégré à un environnement de développement), ou pendant son exécution (*e.g.,* si le mécanisme de gestion de l'évolution est utilisé par un mécanisme autonomique) [Medvidovic99a, Oreizy98b]. Par exemple, la spécification d'une architecture logicielle peut avoir besoin d'évoluer afin de satisfaire de nouveaux besoins utilisateur qui entraînent l'ajout d'un rôle supplémentaire, permettant la fourniture de nouvelles fonctionalités. L'architecte peut aussi avoir besoin de faire évoluer une configuration au cours de la phase de conception, par exemple pour remplacer une classe par une classe compatible qui offre de meilleures qualités (*e.g.,* moins coûteuse, plus efficace, mieux maintenue ou plus sûre). Des changements peuvent aussi intervenir directement à l'exécution, par exemple lorsqu'un composant tombe en panne et qu'il doit donc être remplacé par un composant qui offre des fonctionnalités similaires (ou, parfois, des fonctionnalités dégradées mais considérées comme acceptables). Les évolutions, qu'elles soient initiées à n'importe laquelle des trois dimensions (spécification, configuration ou assemblage), peuvent avoir un impact sur l'ensemble des dimensions. Elle doivent donc être gérées de telle façon que les descriptions d'une architecture dans les trois

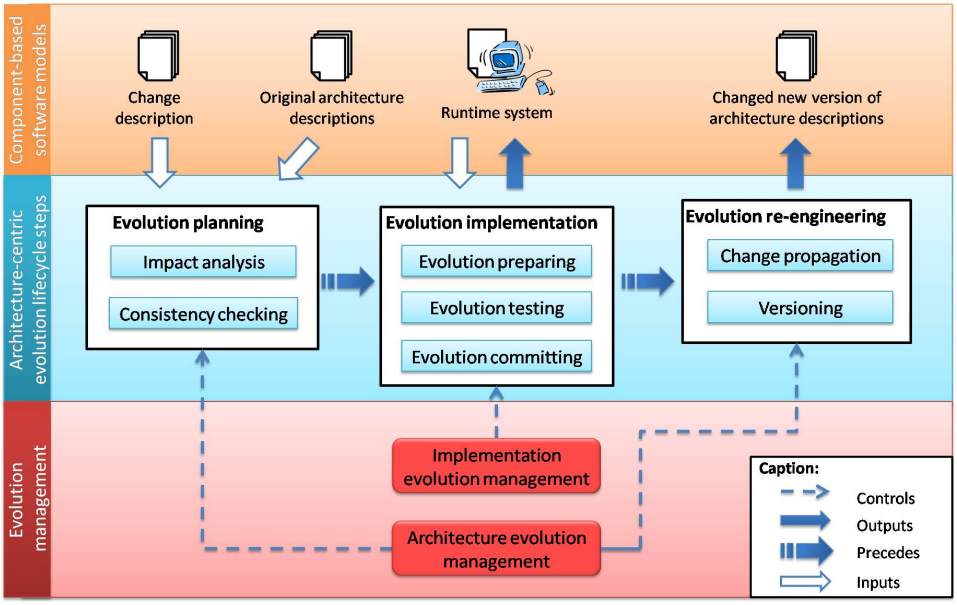


FIGURE C.10 – Processus d'évolution d'une architecture logicielle à base de composants

dimensions restent cohérentes entre elles. Cela implique de mettre en œuvre une certaine forme de rétro-ingénierie (propagation des changements des dimensions concrètes vers les dimensions abstraites) ou de reconception (propagation des changements des dimensions abstraites vers les dimensions concrètes).

C.6.1 Support de l'évolution dans les principaux ADL existants

C.6.2 Propagation des changements

La cohérence est une propriété interne des modèles architecturaux qui assure que les différentes vues (dans notre cas, les différentes dimensions) d'une même architecture ne se contredisent pas entre elles [Taylor09]. L'objectif de la vérification de cohérence est de prévenir que des changements induisent des incohérences intra ou inter-dimensionnelles. Lorsque des changements préservent la cohérence d'une architecture, ils sont autorisés et appliqués. Dans le cas contraire, soit les changements sont interdits, soit ils déclenchent le versionnement de l'architecture afin de préserver la cohérence de la version existante. Le versionnement peut être limité à l'une des représentations de l'architecture ou nécessiter, par propagation de l'évolution, un versionnement de ses différentes dimensions.

Le modèle de versions de Dedal permet de garder une trace des changements intervenus entre versions d'un même arbre de dérivation. Au sein de chaque dimension, les versions enregistrent un identifiant de version, l'identifiant de la version précédente et la liste des changements dans la version courante (deltas). Peu d' ADLs permettent d'exprimer les changements entre versions. C2SADEL [Oreizy98b, Medvidovic96] et Darwin [Magee96] gèrent des changements de configuration transactionnels sur le système en cours d'exécution mais ne conservent pas la trace de ces changements dans les représentations des architectures. Dedal représente les changements entre versions (deltas) comme des listes d'opérations de changement (ajouts, suppressions ou modifications). La figure C.12 pré-

sente la version *2.0* de la spécification du BRS. Cette architecture diffère de la précédente par l'addition du composant *GIS*.

C.6.3 Scénario d'évolution pour le BRS

Ce scénario d'évolution étudie l'ajout de la classe de composant (*StationData*) à la configuration du BRS(*cf.* figure C.12). Il illustre les mécanismes de propagation évoqués précédemment. Ce nouveau composant se connecte à *BikeTrip* par l'interface requise *LocOprs* (*cf.* figure C.11). La compatibilité syntaxique et sémantique de l'interface *LocOprs* du composant *BikeTrip* avec l'interface *LocOprs* du composant *StationData* est vérifiée. Le protocole de comportement du composant *BikeTrip* fait appel à la méthode *findStation* fournie par l'interface *LocOprs* du composant *StationData* (*cf.* figureC.8). La classe de composants *StationData_1.0* (*cf.* figureC.11) n'impose aucune contrainte supplémentaire. Cette nouvelle configuration est cohérente, mais sa définition n'est plus compatible avec sa spécification. La propagation des modifications déclenche la création d'une nouvelle version de la spécification par ajout du rôle *GIS*. De même, la propagation des modifications vers l'assemblage déclenche la création d'une nouvelle version, décrivant l'instanciation du composant *StationData* (*cf.* figure C.12).

```
1 component_class StationData
2   implements GIS
3   using fr.ema.StationDataImpl
4   provided_interfaces LocOprs
5   version 1.0
```

FIGURE C.11 – *Description de la classe de composants StationData*

C.7 IMPLÉMENTATION

Le fait de disposer d'une suite logicielle facile d'utilisation est un aspect important qui permet tout à la fois de convaincre l'architecte de la faisabilité de nos idées et de l'aider au cours du processus de conception d'une architecture. De nombreux outils sont ainsi adossés à des ADLs. Citons par exemple ArchStudio [Arc], Fractal [Fra], ArgoUML [Arg].

Nous avons implanté le langage Dedal et le processus d'évolution proposé dans cette thèse dans la suite d'outils Arch3D. Arch3D est une suite logicielle pour modéliser, visualiser, analyser, implémenter et faire évoluer les architectures de logiciels. Les outils d'Arch3D peuvent être classés en trois catégories : modélisation d'architecture, cadre d'exécution et outils d'évolution.

1. *Modélisation d'architecture.* La syntaxe de Dedal est implanté de deux façons différentes : avec des schémas XML, et avec des classes Java.
   - *Dedal XML schema.* Les descriptions d'architectures en Dedal sont exprimées en utilsant XML (Extensible Markup Language) [Biron04]. La syntaxe de Dedal est définie grâce à des schémas XML.
   - *Arch3D-JDC.* Afin de facilitater l'édition des fichiers XML Dedal, Les descriptions XML en Dedal sont représentées en mémoire par des instances d'un modèle de

```
1  specification BRSSpec
2   component_roles
3    BikeCourse; BikeCourseDB; GIS
4   connections
5    connection connection1
6    ...
7    connection connection3
8     client BikeCourse.LocOprs
9     server GIS.LocOprs
10  architecture_behavior
11   (!BikeCourse.BikeOprs.selectBike;
12   ({?BikeCourse.LocOprs.findLoc;
13   !GIS.LocOprs.findLoc;
14   ?BikeCourse.BQuery.findBike;}
15   +
16   {?BikeCourse.BQuery.findBike;}))
17   +
18   ...
19  version 2.0;
20  pre_version 1.0;
21  by addition of GIS;
22 configuration BRSConfig
23  implements BRSSpec (2.0)
24  component_classes
25   BikeTrip (1.0) implements BikeCourse;
26   BikeTripDBClass (1.0) implements
27    BikeCourseDB;
28   StationData (1.0) implements GIS
29  version 2.0;
30  pre_version 1.0;
31  by addition of StationData; assembly BRSAss
32  instance_of BRSConfig (2.0)
33  component_instances
34   BikeTripC1 as BikeCourse;
35   BikeTripDBC1 as BikeCourseDB
36   StationDataC1 as GIS
37  assembly_constraints ...
38  version 2.0
39  pre_version 1.0;
40  by addition of StationDataC1;
41  component_instance StationData
42   instance_of StationData (1.0)
```

FIGURE C.12 – *Description de l'architecture après évolution*

classes de données Java, ou JDC, (mapping des schémas XML) grâce à l'utilisation de JAXB[Fialli06].
- *Arch3D-JModel.* Afin d'intégrer le support de l'évolution, les JDC sont étendues avec JModel, modèle Dedal en Java qui contient la gestion de l'évolution.

2. *Cadre d'exécution.* Le cadre d'exécution Arch3D-REF (Runtime Evolution Framework) gère le logiciel en cours d'exécution. Il comprend le module Arch3D-IEM (Implementation Evolution Management) qui permet le support de l'évolution dynamique et médiatise la communication entre les descriptions d'architecture et le système en cours d'exécution afin que ceux-ci restent synchronisés.
3. *Outils d'évolution.* Le développement et l'évolution d'architectures logicielles en Dedal est réalisé au travers d'outils dédiés.
   - *Arch3D-Editor.* Arch3D-Editor est un éditeur graphique pour visualiser et modifier des descriptions d'architectures logicielles. Il affiche les architectures en différents formats : arbres, graphes, texte sous la forme BNF et XML.
   - *Arch3D-AEC.* Arch3D-AEC (Architecture Evolution Center) permet de contrôler l'évolution et comprend la planification d'évolution, l'implémentation des changements et la rétro-évolution.
   - *Arch3D-Analyzer.* Arch3D-Analyzer est un outil qui permet d'analyser la correction d'une description architecturale. Les critères d'analyse comprennent la comparaison des comportement, nom, interface, interaction et la vérification du raffinement.

Ces outils et leurs relations sont représentés sur la Fig. C.13. La volumétrie du code de chaque module est donnée dans la Table C.4.

Arch3D tool	Total LOC
Dedal XML Schema	579
JDC	1809
JModel	712
Editor	7445
AEC	638
Analyzer	194
REF	4302

TABLE C.4 – Volumétrie du code (LOC) des outils d'Arch3D

C.8 CONCLUSION

Dedal permet la représentation explicite et distincte des spécifications, des configurations et des assemblages qui constituent les trois dimensions des architectures. Les décisions de conception des architectures peuvent ainsi être énoncées et leur impact suivi tout au long du cycle de développement. Les évolutions peuvent intervenir dans n'importe laquelle des trois dimensions. Leur cohérence est vérifiée et les changements nécessaires propagés dans les autres dimensions. Dedal propose ainsi un support de l'évolution original car il permet la propagation directe mais aussi la rétro-propagation des changements. Il offre ainsi une solution pour prévenir la dérive et l'érosion des architectures. L'ADL Dedal ainsi que le processus d'évolution présentés dans cette thèse ont été implémentés comme une

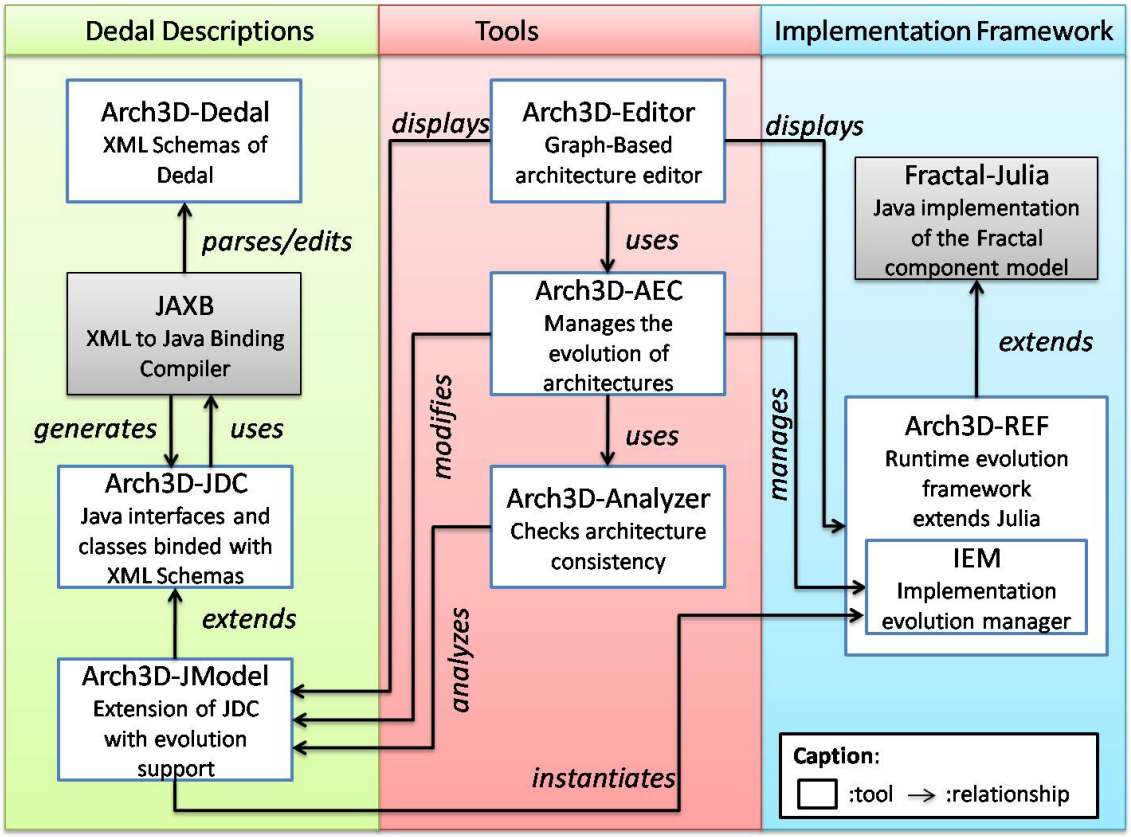


FIGURE C.13 – Arch3D : une suite logicielle pour la modélisation et l'évolution d'architectures

extension de Julia, l'implémentation open-source de référence du modèle de composants Fractal, et de ses outils associés [2]. La gestion de la compatibilité entre protocoles est réalisée grâce à SOFA, qui propose une implémentation de ces mécanismes dans le cadre de Fractal. Des expérimentations préliminaires ont été menées sur les architectures logicielles d'instruments de musique électronique configurables. Une des perspectives envisagée pour ces travaux est leur expérimentation dans le cadre de la gestion de lignes de produits logiciels à base de composants.

2. `http://www.objectweb.org`