Architecture-centric development and evolution processes for component-based software

Huaxi (Yulin) Zhang, Christelle Urtado, Sylvain Vauttier
LGI2P / Ecole des Mines d'Alès – Nîmes – France
{Huaxi.Zhang,Christelle.Urtado,Sylvain.Vauttier}@mines-ales.fr

***Abstract*—Component-based development focuses on component reuse and composition: abstract components (as wished) must be searched for and matched to existing component (as found). This search and reuse activity greatly impacts software development and evolution processes. Unfortunately, very few works propose adaptations of traditional software engineering processes and no existing ADL yet permits to describe the resulting development artifacts. This paper proposes architecture-centric processes for the development and evolution of component-based software. Architecture-centric development produces descriptions for architecture specification, architecture configuration and component assembly. The paper shows how Dedal, a three-level ADL, can be used to support the consistent description of these three artifacts. The paper then shows how these descriptions can be used during a controlled architecture-centric evolution process that helps build, test and record versions of component-based software. This tackles the well-known issues of architecture erosion and drift that denote mismatches between the different architecture definitions.**

I. INTRODUCTION

Component-based software engineering (CBSE) 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. It promotes a reuse-based approach to define, implement and compose loosely coupled independent software components into whole software systems [1], [2]. Software architectures can be described at three development stages: architecture specification, architecture configuration and instantiated component assembly. Conformance between these descriptions must be guaranteed top-down from an abstract description level to the next, more concrete one. Such descriptions could also be used to control software evolution by propagating changes bottom-up. Surprisingly, no architecture description language (ADL) proposes such a detailed description for architectures that covers the artifacts produced during the component-based development cycle. Most ADLs, such as Wright [3], [4], C2SADEL [5], [6], Darwin [7], focus on modeling one or two architecture descriptions levels. Furthermore, no ADL is rich enough to serve as the support of a complete evolution process for component-based software. This paper first presents an architecture-centric development process, based on intensive component reuse. It defines how architectures can be gradually defined, thanks to three-leveled architecture definitions which capture the design decisions at each step of the process. More specifically, explicit architecture specifications provide a means to effectively integrate the reuse of components as part of the development process. A controlled architecture-centric evolution process is then presented. The aforementioned three-leveled architecture definitions are used to check the consistency of changes and to manage the propagation of changes when architecture definitions are versioned to prevent architecture drift and erosion [8]. The features of Dedal – our proposed ADL – are specifically designed to support these two processes.

The remaining of this paper is organized as follows. Section II defines the proposed architecture-centric development process and its associated architecture descriptions. Section III presents how the Dedal ADL supports the three proposed architecture descriptions. Section IV depicts the context of software evolution in CBSD before Sect. V defines a controlled architecture-centric evolution process and its support with Dedal. Section VI concludes and draws future work directions.

II. ARCHITECTURE-CENTRIC DEVELOPMENT FOR CBS

A. An architecture-centric development process

Component-based software development is characterized by its implementation of the "reuse in the large" principle. Reusing existing (off-the-shelf) software components therefore becomes the central concern during development. In the context of component-based software development, traditional software development life-cycles have to be adapted [2]. Figure 1 illustrates our vision of such a development life-cycle which is classically divided in two: the component development life-cycle (sometimes referred to as component development *for* reuse), which is not detailed here, and the component-based software development cycle (referred to as component-based software development *by* reuse) that describes how previously developed software components can be used for new software development (and how this reuse process impacts the way software is built).

Our component-based software development life-cycle deliberately is an architecture-centric development process focusing on the produced architecture artifacts (architecture descriptions as models of the software) for each development step. In this component-based software development life-cycle, software is considered to be produced by the reuse of components that have previously been stored and indexed in a component repository. After a classical requirement analysis step, architects establish the abstract architecture specification.

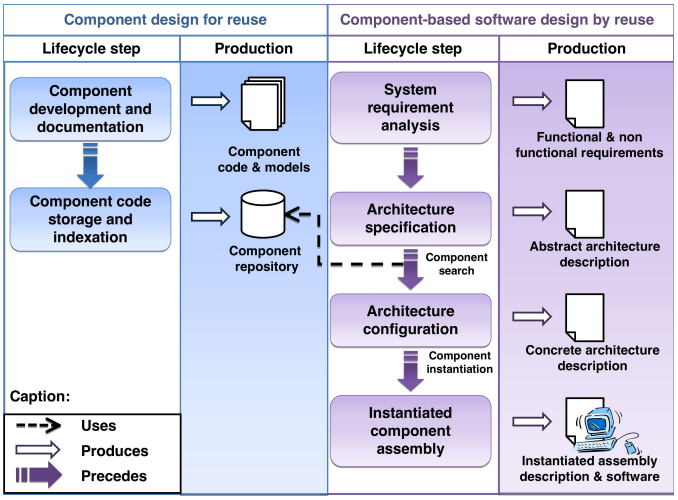

Fig. 1. Component-based software development process

They define which functionalities should be supplied by components, which interfaces should be exported by components, and how interfaces should connect to build a software system that 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.

The claim of this paper is that **an architectural description should correspond to each of the three steps (specification, configuration and assembly)** of the component-based software development process. These three point of views on architectures are necessary to reflect the architect's design decisions and should be expressed using an adequate ADL.

B. Development cycle coverage by existing ADLs

Software system architectures [9] gather design decisions on system. They are expressed using an ADL. In the syntax of most ADLs, architectures are usually described by two complementary views: an architecture specification where a class of systems is described as composed of component classes and connector classes, and an architecture configuration where a system instance is described as composed of component instances and connector instances.

Systems can also be described at various steps of their life-cycles. To our knowledge, no ADL really includes this time dimension. Some works such as UML [10] or Taylor *et al.* [9] implement or describe close notions. UML makes it possible to describe object-oriented software at various life-cycle steps but this capability is not transposed in their component model. Taylor *et al.* distinguish two description levels for architectures at design and programming time, respectively called perspective and descriptive architectures.

Garlan *et al.* [11] points out the importance of three levels (called task, model and runtime layers) for dynamic software evolution management but, as far as we know, do not propose any ADL or metamodel to concretely implement them. Other existing ADLs such as C2SADEL, Wright, Darwin, Unicon [12], SOFA2.0 [13], Fractal ADL [14] and xADL2.0 [15] do not either (see Table I).

As a conclusion, we found no research work that enables to model the three levels of software systems as architectural descriptions which correspond the development stages.

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

TABLE I
EXPRESSIVENESS OF EXISTING ADLS

C. Example of a Bicycle Rental System

Figure 2 shows the example used throughout the paper: the architecture specification of a bicycle rental system (BRS). A *BikerGUI* component manages a user interface. It cooperates with a *Session* component which handles user commands. The *Session* component cooperates with the *Account* and *Bike&Course* components to identify the user, check the balance of its account, assign him an available bike and then calculate the price of the trip when the rented bike is returned. In the following, we will use a part of this system (*BikeCourse* and *BikeCourseDB* component roles and their connection) to illustrate our concepts and ADL syntax.

III. OVERVIEW OF THE DEDAL ADL

In this section, we present elements of the syntax of the Dedal ADL [16] we propose to describe the artifacts produced at each of the three stages of component-based software development.

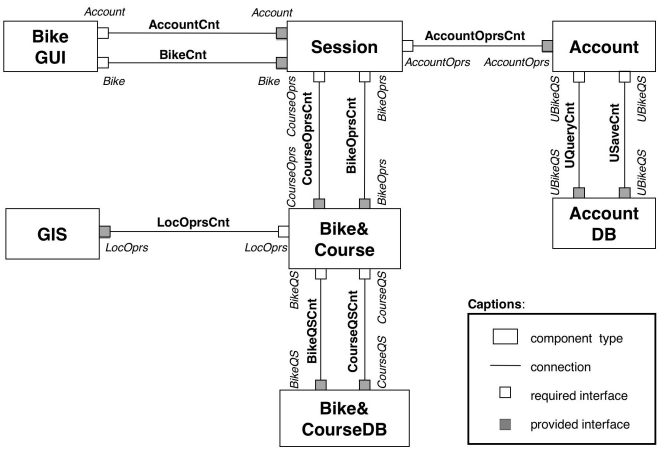

Fig. 2. BRS abstract architecture specification

A. Abstract Architecture Specifications

Abstract architecture specifications (AASs) are the first level of software architecture descriptions. They provide a generic definition of the global structure and behavior of software systems according to previously identified functional requirements. They are specified during the design step of software and serve as a basis to create concrete architecture configurations. These architecture specifications are abstract: they do not identify concrete component types that are going to be instantiated in the software system. They only describe "ideal" component types from the application point of view. Abstract architecture specifications can be compared to perspective architectures as denoted by Taylor *et al.* [9].

In Dedal, an AAS is composed of a set of component roles and a set of connections. The architecture version number and versioning information are also part of the abstract architecture specification description (see Sect. V). Figure 3 provides an example of the AAS for the BRS. For readability reasons, this description represents only a small part of the BRS AAS depicted in Fig. 2.

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB
 connections
  connection BikeQSCnt
   client BikeCourse.BikeQS
   server BikeCourseDB.BikeQS
  connection CourseQSCnt
   client BikeCourse.CourseQS
   server BikeCourseDB.CourseQS
version 2.0;
pre_version 1.0;
by additionGISList;
```

Fig. 3: AAS of the BRS (partial)

Component roles model abstract component types in that they describe the roles components should play in the system. A component role lists the interfaces (both required and provided) the component should expose[1]. As component roles are abstract component specifications, Dedal describes them outside architecture specifications, so as they can be reused from a specification to another (which would not be possible if they were embedded). Figure 4 shows the description of the *BikeCourse* component role.

B. Concrete Architecture Configurations

Concrete architecture configurations (CACs) are the second level of system architecture descriptions. They result from the search and selection of real component types (component classes) from a component repository. These component classes must match abstract component descriptions from the architecture but need not be identical; compatibility is sufficient. CACs describe the architecture from an implementation viewpoint (by assigning component roles to existing component types). CACs correspond to descriptive architectures as denoted by Taylor *et al.* [9].

[1]Dedal further includes the description of architecture dynamics using behavior protocols that are not presented here due to space limitations [16].

```
component_role BikeCourse
 required_interfaces BikeQS; CourseQS; LocOprs
 provided_interfaces BikeOprs; CourseOprs
```

Fig. 4: BikeCourse component role

CACs list the concrete component and connector classes which compose a specific version of a software application. The architecture of a given software is thus defined by a unique AAS and possibly several CACs (each of which must be conform to the AAS). This means that each component or connector class used in an architecture configuration must be a legal implementation of the corresponding component role or connection in the architecture specification. The configuration version number and versioning information are also part of the concrete architecture configuration description (see Sect. V). An excerpt of the BRS configuration derived from the specification presented on Fig. 2 and 3 is described in Fig. 5.

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

Fig. 5: A possible CAC for the BRS

Component classes used in the CAC are listed and each of them is associated to the component role it implements. Component classes can either be primitive or composite. *Primitive component classes* are implemented by a single class. *Composite component classes* are coarser grained components which implementation is a configuration. Component classes can have (externally visible) attributes so as to be able to express constraints on component instance states.

Conformance between an AAS and a CAC is a matter of conformance between component roles and the component classes that supposedly implement them. Many conformance relations could be defined, from stricter to very loose ones. On the one hand, we defend that reused components need not be exactly identical to specifications 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 it enables verifications that guarantees good chances that the thought component combination will execute. The rule of the thumb that can be used is that concrete components must provide at least what the specification declares it provides and require less than what the specification already requires.

C. Instantiated Software Component Assemblies

Instantiated software component assemblies (ISCAs) are the third level of architectures. They result from the instan-

tiation of the component classes from a configuration. They provide a description of runtime software systems and gather information on their internal states thus enabling the record of state-dependent design decisions [17].

ISCAs list the component and connector instances that compose a runtime software system, the attributes of this software system, and the assembly constraints the component instances are constrained by. The assembly version number and versioning information are also part of the instantiated software component assembly description (see Sect. V). Figure 6 gives the description of a software assembly that instantiates the BRS architecture configuration of Fig. 5.

```
assembly BRSAss
 instance_of BRSConfig (2.0)
 component_instances
  BikeTripC1 as BikeCourse;
  BikeCourseDBClassC1 as BikeCourseDB
 assembly_constraints
  BikeTripC1.currency=="Euro.";
  BikeCourseDBClassC1.company==
   BikeTripC1.company;
version 2.0;
pre_version 1.0;
by additionStationDataInsList;
```

Fig. 6: ISCA for the BRS

Component instances document the instances of components that constitute the runtime software system. They are instantiated from the corresponding component classes of the architecture configuration. They might contain constraints on components' attributes that reflect design decision that impact component states (attribute values).

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. Assembly constrains are illustrated on the example of Fig. 6 where the value of the *currency* attribute of component *BikeTripC1* is fixed to the *"Euro"* value. These constraints are very simple and do not yet enable the expression of alternatives, negation, nor the resolution of possible conflicts.

Conformance between a CAC and an ISCA is quite straightforward. All component instances of the assembly must be an instance of a corresponding component class from its source configuration (and reciprocally). 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. 6 entails that the *BikeTrip* component class (from which *BikeTripC1* is instantiated) must possess a *currency* attribute.

IV. CONTEXT OF ARCHITECTURE-CENTRIC EVOLUTION

A. Requirements of architecture-centirc evolution

Component-based software development has been widely studied during recent years, while there has been less effort to study component-based software evolution. *Software evolution* is defined as the collection of all programming activities intended to generate a new version of some software from an older operational version [18]. In the context of component-based development, a software evolution process must be re-invented as claimed by [19], [20].

To our opinion, software evolution should possibly be triggered from any architecture level of component-based software (specification, configuration or assembly). Indeed, depending on contexts, evolution can occur during specification or design (*e.g.* if the evolution management process is integrated into some development environment), or directly at runtime (*e.g.* if the evolution management process is to be used by some autonomic software) [21]. For example, the specification of some software architecture can be required to evolve to meet new requirements, thus adding a new component role to provide new functionalities. The architect might also want the configuration to evolve at design time, for example to replace some component class by another compatible one that has better qualities (*e.g.* cheaper, more efficient, better maintained or safer). Changes might also occur at runtime as for example when some component fails and must be replaced by another one that provides similar (or sometimes only acceptable replacement) functionality. Changes, either they be triggered from the specification, the configuration or the assembly description levels, must nonetheless be managed adequately for all three description levels to be kept consistent with one another. This entails architecture re-engineering or architecture re-factoring.

B. Evolution support in existing ADLs

Evolution in existing ADLs typically characterizes by five facts. Firstly, few ADLs enable to *describe changes*, except C2SADEL and Darwin 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. Secondly, evolution can only be *triggered from the configuration level* (*e.g.* C2SADEL, Darwin, Wright). As these ADLs do not describe running assemblies, they cannot enable changes to be triggered from this level. Thirdly, some *consistency checks* are performed (*e.g.* C2SADEL, Wright), but none of them checks all consistencies of architectures: name, behavior, interface, interaction and refinement. Then, *change propagation* is limited. Evolution processes should handle all three description levels of software architectures. Some change at one of the three levels should be propagated to the other two in order to maintain all descriptions consistent and prevent architecture drift and erosion. This propagation should possibly be directed top-down (from specification to assembly) or bottom-up (from assembly to specification). To our knowledge, the few existing ADLs that support change propagation (*e.g.* xADL2.0) only direct it in a top-down manner. This is adapted to traditional software development and evolution but could be improved in the case of open and dynamic component software (as autonomous systems). At last, *versioning* components and architectures is not frequent except SOFA2.0 and MAE [22].

ADL	Consistency checking	Evolution test	Change propagation	Versioning
C2SADEL	Refinement	×	×	×
Wright	Name, interaction, deadlock	×	×	×
Darwin	State	×	×	×
SOFA2.0	Behavior	×	×	State-based
xADL2.0	×	×	Horizontal (topdown)	×
MAE	Subtyping	Perfective	✓	Change-based

TABLE II
EVOLUTION SUPPORT IN EXISTING ADLS

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

Fig. 7: Change description example

MAE uses configuration management to control component versions, but provides no version support for whole architecture configuration or specifications. In conclusion, most ADLs provide a limited support for evolution as shown in Table II.

V. ARCHITECTURE-CENTRIC EVOLUTION FOR CBS

In this section, we present our vision of a component-based software evolution process based on Dedal. Architecture evolution can be triggered by changes at any of the three representation levels. Moreover, both top-down (re-factoring) and bottom-up (re-engineering) change propagation are supported. Classically, evolution operations at a representation level are controlled in order to enforce their conformance with upper (more abstract) representation levels. Conversely, changes are propagated to lower representation levels in order to update them and maintain their consistency with upper representation levels. Our approach allows bottom-up evolution too, in which transitional non-conform architectures can be created to experiment new solutions [16]. Combined top-down and bottom-up evolution prevents architecture erosion and drift [8].

A. First class change expression

In order to be able to trace the evolution of software architectures, changes should be explicitly described in the same language as for architectures. Furthermore, having them described modularly makes them reusable. This can be summed up saying that **changes need to be first class entities**. Dedal implements this principle. Changes from a version to the next (delta) are described as a list of change operations. Figure 7 gives the example of the *additionStationData* change description.

The effects of changes on architectural elements (component roles, component classes, component instances, architectures, configurations, assemblies) might result in their versioning. Dedal records two pieces of information for each version: the *versionID* that identifies versions and the *pre_version* link that constitute the structure of the version tree. Figure 5 gives an example of such versioning information for the *BRSConfig* architecture configuration.

B. Architecture-centric evolution process

The three levels of architecture descriptions constitute the underlying logical foundation of software evolution. Evolution can be initiated at any of the three levels of software architecture. The **evolution process** decomposes into three stages (see Fig. 8). Evolution planning analyzes the impact of change and checks its consistency in each architecture description. Evolution implementation prepares, tests the change and implements it in the implementation environment. Evolution re-engineering propagates changes to other levels and versions software if necessary. This evolution process is controlled by both an architecture evolution management module and an implementation evolution management module.

Consistency is an internal property of an architecture description, which intends to ensure that its elements do not contradict one another [9]. The aim of **consistency checking** is to predict whether changes induce inconsistencies inside and among the three levels of a given architecture. We define *intra-level consistency* that checks name inconsistency, interface inconsistency, attribute inconsistency, interaction inconsistency and *inter-level consistency* that checks mapping inconsistency. 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.

If inconsistencies are detected, the evolution process either forbids the thought evolution or consider the thought evolution as being part of a new architecture version. This step is called **change propagation** [23]. The new architecture version will be derived and its content inferred so as to be consistent with

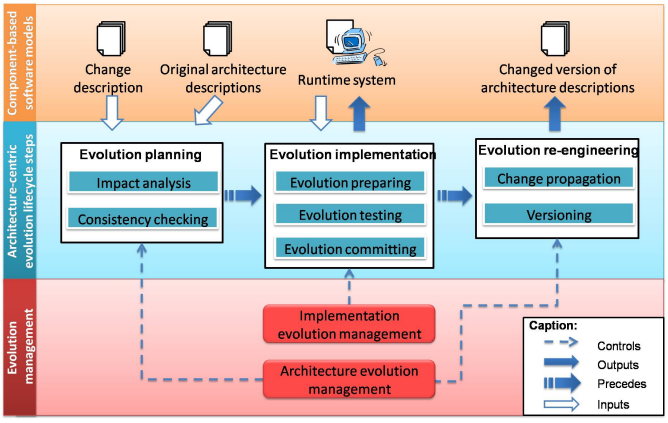

Fig. 8. Architecture evolution process for component-based software

the thought change. The information necessary to derive new versions on each of its levels are extracted from lower to higher levels.

Versioning in Dedal relies on a changed-based version model which maintains version trees for the three descriptions of components (roles, classes and instances) and for whole specifications, configurations and assemblies. In each level, versions record three types of information: version ID, previous version ID and change operations. Fig. 9 shows an example of such a version tree for the BRS.

VI. CONCLUSION

Dedal enables the explicit and separate representations of architecture specifications, configurations and assemblies. Architecture design decisions can thus be precisely captured and traced throughout the component-based development process. Evolution may be initiated at any description level. Consistency is checked and changes are propagated (top-down and bottom-up) to the other description levels so as to maintain the architecture descriptions up-to-date and consistent. This provides a controlled support for component-based software evolution that prevents architecture drift and erosion.

Dedal and the process presented in this paper have been implemented in a tool that validates the feasibility of the approach. It already allowed us to manually experiment evolution scenarios on small examples and gave us preliminary experimental feedback on the expressiveness and adequateness of the proposed ADL. Our implementation is an extension of Julia, an open-source java implementation of the Fractal[2] component model. Dedal-based component architectures can be described and visualized through multiple synchronized views: using a box-based architectural view, using an XML-based syntax for Dedal or using the BNF-based syntax for Dedal presented in this paper. Preliminary experiments have been run on the software architecture of customizable electronic music instruments. A perspective for this work is to experiment it to manage component-based software product lines.

[2] http://www.objectweb.org

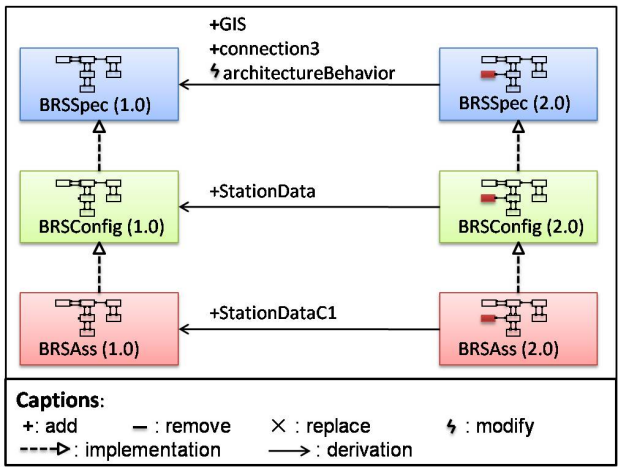

Fig. 9. Version of the BRS architecture

REFERENCES

[1] I. Sommerville, *Software Engineering*, 8th ed. Addison Wesley, 2006.

[2] M. R. V. Chaudron and I. Crnkovic, *Software Engineering; Principles and Practice*. John Wiley & Sons, 2008, ch. Component-based Software Engineering, pp. 605–628.

[3] R. Allen, D. Garlan, and R. Douence, "Specifying dynamism in software architectures," in *Proc. of the Wkshp on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.

[4] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proc. of the Conf. on Fund. Appr. to Soft. Engineering*, Lisbon, Portugal, March 1998, pp. 21–37.

[5] N. Medvidovic, "ADLs and dynamic architecture changes," in *Joint Proc. of the 2nd Int'l software architecture Wkshp and Int'l Wkshp on multiple perspectives in software development on SIGSOFT '96 Wkshps*, San Francisco, USA, 1996, pp. 24–27.

[6] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in *Proc. of the 21st Int'l Conf. on Software Engineering*, Los Angeles, USA, May 1999, pp. 44–53.

[7] J. Magee and J. Kramer, "Dynamic structure in software architectures," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 6, pp. 3–14, 1996.

[8] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[9] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[10] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, (2nd Ed.)*. Addison-Wesley, 2005.

[11] D. Garlan, B. Schmerl, and J. Chang, "Using gauges for architecture-based monitoring and adaptation," in *Proc. Working Conf. on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December 2001.

[12] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE TSE*, vol. 21, no. 4, pp. 314–335, 1995.

[13] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Proc. of the 4th Int'l Conf. on Software Engineering Research, Management and Applications*. Seattle, USA: IEEE, 2006, pp. 40–48.

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and 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, pp. 1257–1284, 2006.

[15] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM TOSEM*, vol. 14, no. 2, pp. 199–245, 2005.

[16] H. Y. Zhang, "A multi-dimensional architecture description language for forward and reverse evolution of component-based software," Ph.D. dissertation, Montpellier II University, France, April 2010.

[17] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, 1996.

[18] M. M. Lehman and J. C. Fernandez-Ramil, "Towards a theory of software evolution - and its practical impact," in *Proc. Int'l Symposium on Principles of Software Evolution*, 2000, pp. 2–11.

[19] S. S. Yau, J. S. Collofello, and T. M. MacGregor, "Ripple effect analysis of software maintenance," in *Software engineering metrics I: measures and validations*. McGraw-Hill, Inc., 1993, pp. 71–82.

[20] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proc. of the Int'l Conf. on Software Engineering – Future of SE track*, 2000, pp. 73–87.

[21] N. Medvidovic, "Architecture-based specification-time software evolution," Ph.D. dissertation, University of California, Irvine, 1999.

[22] R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic, and N. Medvidovic, "Mae–a system model and environment for managing architectural evolution," *ACM TOSEM*, vol. 13, no. 2, pp. 240–276, 2004.

[23] C. Urtado and C. Oussalah, "Complex entity versioning at two granularity levels," *Information Systems*, vol. 23, no. 2/3, pp. 197–216, 1998.