An evolution management model for multi-level component-based software architectures

Abderrahman Mokni[1], Marianne Huchard[2], Christelle Urtado[1], Sylvain Vauttier[1], and Yulin Zhang[3]

[1]LGI2P / Ecole des Mines d'Alès, Nîmes, France, {Abderrahman.Mokni, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr
[2]LIRMM / CNRS & Montpellier University, France, huchard@lirmm.fr
[3]Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France, yulin.zhang@u-picardie.fr

Abstract

Handling evolution in component-based software architectures is a non trivial task. Indeed, a series of changes applied on software may alter its architecture leading to several inconsistencies. In turn, architecture inconsistencies lead to software erosion and shorten its lifetime. To avoid architectural inconsistencies and increase software reliability, architecture evolution must be handled at all steps of the software lifecycle. Moreover, changes must be treated as first class entities. In this paper, we propose an evolution management model that takes these criteria into account. The model is a support for our three-level Dedal architectural model. It captures and handles change at any of the Dedal abstraction levels: specification, implementation and deployment. It generates evolution plans using evolution rules proposed in previous work. The generation process is implemented using the ProB model checker and evaluated through three evolution scenarios of a Home Automation Software.

***Keywords**: software architecture evolution, component reuse, evolution rules, evolution management, abstraction level, change propagation, consistency checking.*

1 Introduction

Software evolution [1] is becoming more and more challenging due to the increasing complexity of software systems and their importance in everyday life. While component reuse has become crucial to shorten large-scale software systems development time, handling evolution in such systems is a serious issue. As witnessed by Garlan *et al.* in their recent study [2], the difficulty of reuse lies essentially on architectural mismatches that arise due to several changes that affect software. A famous problem is software architecture erosion [3, 4]. It arises when modifications of the implementation of a software violate the design principles captured by its specification architecture. Such erosion leads to software degradation and shortens its lifetime. Increasing confidence in reuse-centered, component-based software systems lies on resorting out multiple issues.

First, software architectures must support change at any step of component-based development to meet new user needs, improve component quality, or cope with component failure. Second, the impact of change must be handled locally (at the same abstraction level) to avoid architecture inconsistencies and propagated to the other abstraction levels to avoid incoherence between the architecture descriptions, notably erosion. Third, the evolution activity must be tracked to enable monitoring, commitment and/or rollback and versioning. In this paper, we propose an evolution management model that deals with all these issues. It is based on our Dedal architectural model [5, 6] that covers the three main steps of component-based software development: specification, implementation and deployment. The model uses architecture properties and evolution rules proposed in previous work [7, 8] to generate evolution plans that preserve the consistency of all architecture descriptions as well as coherence between them. The remainder of this paper is outlined as follow: Section 2 gives the background of this work. Section 3 presents the evolution management model. Section 4 presents the evolution plan generation process, an implementation and evaluation. Section 5 discusses

DOI reference number: 10.18293/SEKE2015-172

related work before Section 6 concludes the paper and gives future work directions.

2 Background

This article is concerned with evolution management in multi-level component-based architectures. Specifically, we address software architectures described by Dedal [6], a three-level architectural model. First, we introduce Dedal and then we give a brief overview of its formalization.

2.1 Dedal a three-level architectural model

Dedal is a novel architectural model that covers the three main steps of component-based development by reuse: specification, implementation and deployment. The idea of Dedal is to build a concrete software architecture (called configuration) from suitable software components stored in indexed repositories. Candidate components are selected according to an intended architecture (called specification) that represents an abstract and ideal view of the software. The implemented architecture can then be instantiated (the instantiation is called assembly) and deployed in multiple contexts.

The Dedal model is then constituted of three descriptions that correspond to three architecture abstraction levels.

The architecture specification corresponds to the highest abstraction level. It is composed of component roles and their connections. Component roles encapsulate the required functionalities of the future software.

The architecture configuration corresponds to the second abstraction level. It is composed of concrete component classes selected from repositories and realize the identified component roles in the architecture specification.

The architecture assembly corresponds to the lowest abstraction level. It is composed of component instances that instantiate the component classes of the architecture configuration. While the specification and configuration descriptions represent the software at design-time, the assembly description represents the software at runtime.

Figure 1 illustrates the three abstraction levels of Dedal on an example of a Home Automation Software. The software enables to manage the light of buildings in function of the time through an orchestrator (component role *HomeOrchestrator*). The specified functionalities are turning on/off the light (component role *Light*), controlling its intensity (component role *Intensity*) and getting information about the time (component role *Time*).

2.2 Dedal formalization

Dedal formalization is crucial to enable the verification and validation of the derived architectural models as well as evolution management. In [7], we propose a formalization of Dedal that comprises two kinds of typing rules. The first kind is about intra-level rules. These rules define the relations between components of the same abstraction level such as compatibility and substitutability. The second kind is about inter-level rules. These rules define the relations between components of different abstraction levels (*cf.* Figure 1 for inter-level relations). For instance, the realization rule checks whether a (or a set of) component class(es) realizes a (or a set of) component role(s). This rule is also used to search for candidate concrete components in repositories to implement a specified software architecture. Inter-level and intra-level rules are generalized to support the architectural level. First, using intra-level rules, we can check architecture consistency at any abstraction level. Second, using inter-level rules, it is possible to check coherence between architecture descriptions at different abstraction levels. For instance, we can check whether a configuration implements all the desired functionalities documented in its specification. In [8], we propose a set of evolution rules that enable to evolve Dedal models. The proposed rules allow the manipulation (addition, deletion and substitution) of architectural elements (*e.g.*, components, connections) at the three Dedal abstraction levels.

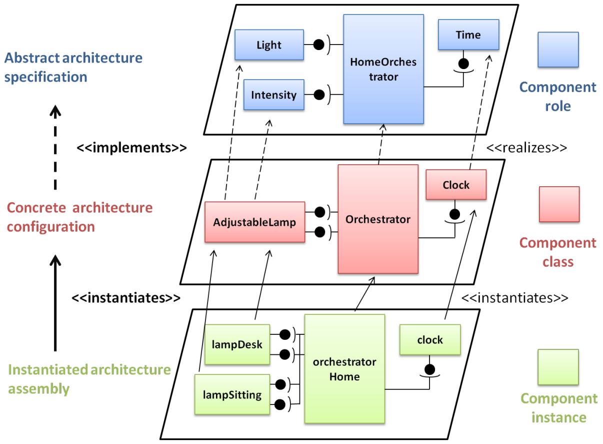


Figure 1. Example of a Dedal model: Home Automation Software

In this work, we present an evolution management model for architecture models derived from Dedal. We show how evolution rules can be used to generate evolution plans that preserve the consistency of architecture descriptions and coherence between them.

3 The evolution management model

Figure 2 presents our evolution management model. It is composed of three parts: the architectural Model (meta-classes of group 1), the changes that affect the architectural model (meta-classes of group 2) and the evolution manager (meta-classes of group 3).

3.1 The architectural model

The architectural model is the target of change. It is derived from the Dedal meta-model and hence includes three architecture descriptions, each represents the component-based software at a different abstraction level.

Architecture properties represent the set of rules that decide about the well-formedness of the architectural model. These properties have to be preserved after change and hence are part of the analysis goals that must be enforced during the software evolution. In our work we focus on two properties,

Architecture consistency states whether the elements of the architecture are correctly typed and well connected (each interface is connected to a compatible one). Additionally, consistency involves the internal completeness of the architecture, *i.e.*, an architecture is said complete if all its required properties are met. From a structural viewpoint, an architecture is complete if all its required interfaces are connected to compatible provided ones.

Architecture coherence states whether an architecture description at an abstraction level is in conformance with an architecture description at an adjacent abstraction level. Verifying architecture coherence keeps all architecture descriptions up-to-date and avoids the problems of drift and erosion.

Model manipulation operations are elementary change operations that manipulate the artifacts of the architectural model (*e.g.*, component roles, component classes, or connections). They are classified into three types: addition, deletion and substitution. Manipulation operations are composed of four parts. A signature defines the operation name and states its arguments. Preconditions are related to the architectural model (*e.g.*, a precondition checks if substitutability between two components is possible). Actions are applied on the architectural model by updating the set of its artifacts. Post-conditions must be verified by the new state of the architectural model after applying the actions of the operation.

3.2 The architectural change

Architectural changes characterize all the modifications that alter the software architecture. They meet new requirements to keep software up-to-date or arise due to an environmental change (*e.g.*, lack of resources, or faults). Software architectures are subject to change at any abstraction level of component-based development. The impact of change may affect the other abstraction levels. All of these facts about software change make its handling a non trivial task. In order to tackle this complexity, we represent change as a first class entity that interferes with the architectural model. Furthermore, we identify three main change characteristics necessary for the evolution management process: origin, level and subject.

There are two types of **change origin**: *initiated change* and *triggered change*. Initiated change has an external source. It may be originated from user action or from the execution environment. Triggered change is internal and is induced by the evolution manager to reestablish the architecture consistency at the same abstraction level (*local change*) and/or preserve conformance between all architecture descriptions at other abstraction levels (*propagated change*).

Change level designates the level where a change is currently performed. It allows to identify which properties must be checked to ensure a successful evolution.

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

3.3 The evolution manager

The evolution manager captures software change and controls its impact on the architectural model. It uses *evolution rules* to generate an *evolution plan* that satisfies a given *evolution goal*. The evolution manager ensures also the co-evolution of the descriptions of the software architectures at the other abstraction levels.

Evolution rules are specific operations that are composed of model manipulation operations. They manage and control access to these operations using preconditions related to the change, according its origin, level, or subject.

The **evolution goal** sets all the conditions that must be satisfied by the architectural model after the change. Basically, it is composed of two parts: architecture properties (*i.e.*, consistency and coherence) and the post-condition of the initiated change.

4 The generation process

In this section, we state the generation process problem. Then, we test and evaluate two search strategies implemented by the ProB [9] model-checker.

4.1 Problem formalization

Notations: Let M be an instance of the Dedal architectural model and S be the state space of M. Then, let L be the enumeration of Dedal abstraction levels. $L = \{specLevel, configLevel, asmLevel\}$. Let E be the set of all evolution rules and E_l the subset of E related to an abstraction level l, $l \in L$. For each $e_i \in E$, let $pre(e_i)$ be

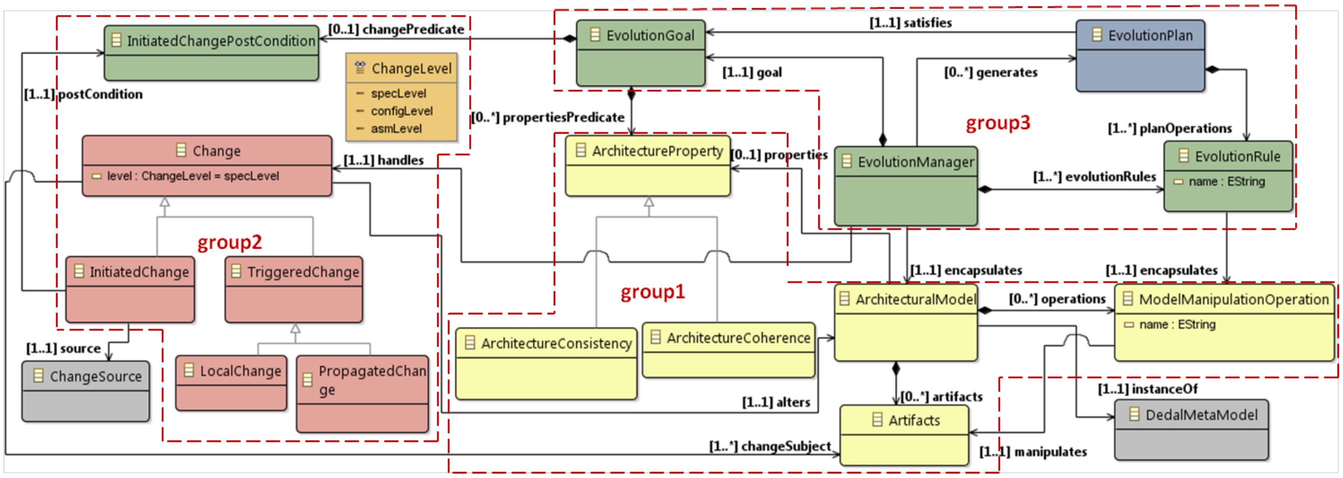


Figure 2. The evolution managment model (ecore)

the precondition of e_i. Let Ci_l be the change initiated at an abstraction level l and $post(Ci_l)$ the post-condition satisfied by $s \in S$ after applying Ci_l on M. Let $P_{consistency}(l)$ be the consistency property related to the abstraction level l and $P_{coherence}(l,k)$ be the coherence property between the two adjacent abstraction levels l and k. Finally, let G_l be an evolution goal related to an abstraction level l.

Problem: Considering an initiated change Ci_l on M, we would like to (1) find a sequence of $e_i \in E_l$ where $G_l = post(Ci_l) \wedge P_{consistency}(l)$ is satisfied and (2) $\forall\, l, k$ where $P_{coherence}(l,k) = false$, find a sequence of $e_k \in E_k$ where $G_k = P_{coherence}(l,k) \wedge P_{consistency}(k)$ is satisfied. The evolution plan Pl is thus the concatenation of all found sequences. $Pl = Pl_{local}; Pl_{propagated}$ where Pl_{local} is the plan related to the local change and $Pl_{propagated}$ is the concatenation of the plans related to the propagated change.

4.2 Implementation overview

The implementation is composed of two parts: the Dedal modeler and the ProB [9] model-checker. The Dedal modeler is an eclipse-based tool that enables the creation and edition of Dedal diagrams (*i.e.*, specification, configuration and assembly diagrams) and the automatic generation of B [10] formal models corresponding to those diagrams. ProB is a model-checker and animator for B models. It calculates and simulates state-transitions. In our case, it can calculate all the enabled evolution rules (defined as B operations) at each state of the model. Moreover, using forward chaining inferences, ProB can search for a sequence of evolution rules that reaches the evolution goal. It proposes depth-first (DF), breadth-first (BF) and mixed depth-first/breadth-first (DF/BF) search strategies to find invariant violations or defined goals. The most suitable strategy depends on the kind of checking the user wants to perform. In the case of searching for a specific state by exploring a large state space, depth first seems to be the most efficient strategy as stated in [11]. In the remainder, we use ProB to generate evolution plans and observe the resolution time using DF and mixed DF/BF. We compare the results to see which strategy is the most efficient.

4.3 Evaluation

To illustrate the generation process, we run three evolution scenarios on the example of HAS. The initial architectures are illustrated by Figure 3.

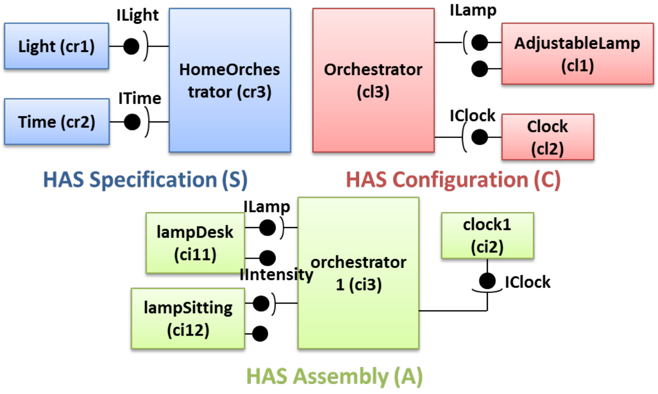


Figure 3. The initial state of HAS

Scenario 1 corresponds to a requirement change. The specification of HAS needs to be evolved to enable the control of the building luminosity. This consists in adding a new role ($cr1a$) that provides a luminosity functionality.

Scenario 2 corresponds to an implementation change.

The configuration needs to be evolved to turn under android OS. The initiated change consists in adding an android orchestrator (*cl3a*). This entails to delete the current orchestrator (*cl3*).

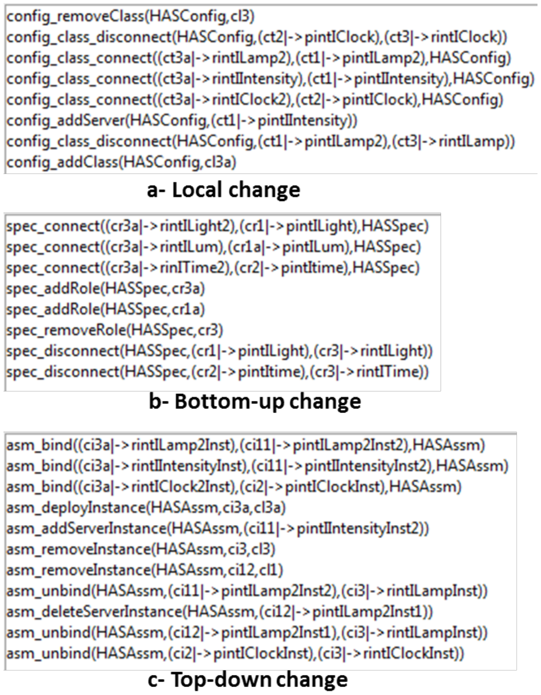
Scenario 3 corresponds to a runtime change. The *clock1* (*ci2*) component instance must be replaced due to a dry battery of the clock. That of the *embeddedClock1* (*ci2a*) mobile device is used instead.

We run two tests for each evolution scenario. The first test uses the DF strategy and the second one uses the DF/BF strategy. The results (*cf.* Table 1) show that DF is more efficient than BF/DF in all cases.

	Change level	**DF (seconds)**	**DF/BF (seconds)**
Scenario1 (top-down change)	specLevel (initial)	2.36	48
	configLevel	16.71	4.97
	asmLevel	9.12	23.83
	full process	**28.19**	**76.8**
Scenario2 (mixed)	configLevel (initial)	9	13.39
	specLevel	2.98	5.37
	asmLevel	12.01	65.41
	full process	**23.99**	**84.17**
Scenario3 (bottom-up change)	asmLevel (initial)	4.5	26.54
	configLevel	77.8	136.08
	specLevel (not affected)	0	0
	full process	**82.3**	**162.62**

Table 1. Evaluation results

Due to space limitation, we only show the generated plans corresponding to `scenario2`. Figure 4-a, Figure 4-b and Figure 4-c respectively show the ProB output for local change (configuration level), bottom-up change (specification level) and top-down change (assembly level). The sequence must be read from the bottom to the top of the output view. We use the following notation to explain the syntax of the generated rules: cr, cl, ct and ci correspond respectively to component role, class, type and instance. The $pint$ and $rint$ prefixes respectively denote a provided or a required interface. *HASSpec*, *HASConfig* and *HASAssm* respectively name the HAS specification, configuration and assembly.



```
config_removeClass(HASConfig,cl3)
config_class_disconnect(HASConfig,(ct2|->pintIClock),(ct3|->rintIClock))
config_class_connect((ct3a|->rintILamp2),(ct1|->pintILamp2),HASConfig)
config_class_connect((ct3a|->rintIIntensity),(ct1|->pintIIntensity),HASConfig)
config_class_connect((ct3a|->rintIClock2),(ct2|->pintIClock),HASConfig)
config_addServer(HASConfig,(ct1|->pintIIntensity))
config_class_disconnect(HASConfig,(ct1|->pintILamp2),(ct3|->rintILamp))
config_addClass(HASConfig,cl3a)
```

a- Local change

```
spec_connect((cr3a|->rintILight2),(cr1|->pintILight),HASSpec)
spec_connect((cr3a|->rintILum),(cr1a|->pintILum),HASSpec)
spec_connect((cr3a|->rinITime2),(cr2|->pintItime),HASSpec)
spec_addRole(HASSpec,cr3a)
spec_addRole(HASSpec,cr1a)
spec_removeRole(HASSpec,cr3)
spec_disconnect(HASSpec,(cr1|->pintILight),(cr3|->rintILight))
spec_disconnect(HASSpec,(cr2|->pintItime),(cr3|->rintITime))
```

b- Bottom-up change

```
asm_bind((ci3a|->rintILamp2Inst),(ci11|->pintILamp2Inst2),HASAssm)
asm_bind((ci3a|->rintIIntensityInst),(ci11|->pintIIntensityInst2),HASAssm)
asm_bind((ci3a|->rintIClock2Inst),(ci2|->pintIClockInst),HASAssm)
asm_deployInstance(HASAssm,ci3a,cl3a)
asm_addServerInstance(HASAssm,(ci11|->pintIIntensityInst2))
asm_removeInstance(HASAssm,ci3,cl3)
asm_removeInstance(HASAssm,ci12,cl1)
asm_unbind(HASAssm,(ci11|->pintILamp2Inst2),(ci3|->rintILampInst))
asm_deleteServerInstance(HASAssm,(ci12|->pintILamp2Inst1))
asm_unbind(HASAssm,(ci12|->pintILamp2Inst1),(ci3|->rintILampInst))
asm_unbind(HASAssm,(ci2|->pintIClockInst),(ci3|->rintIClockInst))
```

c- Top-down change

Figure 4. generated plans

5 Related work

Managing software architecture evolution is still an open issue. For more than two decades, a lot of efforts have been dedicated to provide methods, tools and techniques for architecture modeling and analysis. Several ADLs (Architecture Description Languages) [12] have been proposed. Most of them provide textual notations for describing and analyzing software systems. They are usually supported by tools or integrated environments for edition, analysis and simulation. Examples include C2SADL [13], Wright [14], Darwin [15] and ArchJava [16]. C2SADL is an ADL for the design of concurrent systems. It includes a sub-architecture modification language (AML) to support evolution. Changes are first applied at the architectural level and then implemented using a runtime infrastructure. Wright is also a domain-specific ADL. It aids the design of distributed architectures. Wright focuses more on increasing the architect's confidence on the design of systems. It enables consistency checking and analysis using CSP (Communicating Sequential Processes) as a formal basis. However, it does not propose any language or notation to describe architectural changes. Darwin is relatively similar to Wright as it shares the same goal. Unlike Wright, it includes a declarative language that supports change description (including operations such as create, remove, link, or unlink). Beside the fact that they are domain-specific, C2SADL, Wright and Darwin do not support reverse (bottom-up) evolution. Moreover, they cover only the specification level of the software system and hardly support the implementation and deployment levels. This gap between architecture specification and its implementation generates several inconsistencies when applying changes and shortens the software lifetime. ArchJava is an ADL that unifies the architectural level and the implementation code in a single entity. It uses a type system to check conformance between both descriptions. Although, this unification enables co-evolution, it makes it harder to separate program implementation from its specification. Separating specification from implementation is central in our approach to foster component reuse. Moreover, as far as we know, all cited ADL do not handle changes as first class entities, neither do they propose a mechanism for generating evolution plans.

6 Conclusion and future work

This work proposes an evolution management model for component-based software architectures. It represents changes as first class entities as the architecture models derived from Dedal. The model enables to (1) capture change at any architecture abstraction level, (2) handle its impact at the same level and (3) propagate it to the lower and higher abstraction levels keeping then all the architecture descriptions coherent. Both top-down (forward) and bottom-up (reverse) evolution are then supported by the proposed model.

At this stage of work, evolution is not yet automated and is simulated using model-checking techniques on the generated formal models of Dedal [7]. Unfortunately, this is limited by combinatorial explosion. Using meta-heuristic techniques could be a solution to reduce the complexity of evolution plan generation. Ongoing work is the development of an eclipse-based environment of Dedal that automates the evolution plan generation process. Change requests can be expressed using a change description language such as Dedal-CDL [17]. Furthermore, we are considering to set a versioning mechanism for the architectural models derived from Dedal.

References

[1] T. Mens and S. Demeyer, *Software Evolution*. Springer, 2008.

[2] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is still so hard," *IEEE Software*, vol. 26, no. 4, pp. 66–69, July 2009.

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

[4] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *JSS*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[5] H. Y. Zhang, C. Urtado, and S. Vauttier, "Architecture-centric component-based development needs a three-level ADL," in *Proc. of the 4th ECSA conf.*, ser. LNCS, vol. 6285. Copenhagen, Denmark: Springer, August 2010, pp. 295–310.

[6] H. Y. Zhang, L. Zhang, C. Urtado, S. Vauttier, and M. Huchard, "A three-level component model in component-based software development," in *Proc. of the 11th GPCE Conf.* Dresden, Germany: ACM, Sept. 2012, pp. 70–79.

[7] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, and H. Y. Zhang, "Towards automating the coherence verification of multi-level architecture descriptions," in *Proc. of the 9th ICSEA*, Nice, France, Oct. 2014, pp. 416–421.

[8] ——, "Formal rules for reliable component-based architecture evolution," in *Formal Aspects of Component Software - 11th International FACS Symposium revised selected papers*, Bertinoro, Italy, Sept. 2014, pp. 127–142.

[9] M. Leuschel and M. Butler, "ProB: An Automated Analysis Toolset for the B Method," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 185–203, Feb. 2008.

[10] J.-R. Abrial, *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[11] M. Leuschel and J. Bendisposto, "Directed model checking for B: An evaluation and new techniques," in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, J. Davies, L. Silva, and A. Simao, Eds. Springer Berlin Heidelberg, 2011, vol. 6527, pp. 1–16.

[12] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE TSE*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[13] N. Medvidovic, "ADLs and dynamic architecture changes," in *Joint Proc. of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development on SIGSOFT '96 Workshops*. New York, USA: ACM, 1996, pp. 24–27.

[14] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM TOSEM*, vol. 6, no. 3, pp. 213–249, Jul. 1997.

[15] J. Magee and J. Kramer, "Dynamic structure in software architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 3–14, 1996.

[16] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proc. of the 24rd ICSE Conf.*, May 2002, pp. 187–197.

[17] H. Y. Zhang, C. Urtado, S. Vauttier, L. Zhang, M. Huchard, and B. Coulette, "Dedal-CDL: Modeling first-class architectural changes in Dedal," in *Proc. of the Joint 10th WICSA and 6th ECSA conf.*, Helsinki, Finland, August 2012.