Substitutability-Based Version Propagation to Manage the Evolution of Three-Level Component-Based Architectures

Alexandre Le Borgne[1], David Delahaye[2], Marianne Huchard[2], Christelle Urtado[1], and Sylvain Vauttier[1]

[1]LGI2P / Ecole des Mines d'Alès, Nîmes, France {Alexandre.Le-Borgne, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

[2]LIRMM / CNRS & Montpellier University, France {David.Delahaye, Marianne.Huchard}@lirmm.fr

Abstract

An important issue of software architecture evolution is the capability for architects to keep a trace of the evolution of their work. This paper states that existing research on versioning does not cope well with software architectures. Indeed, it does not propose any adapted solutions to manage the co-evolution of the different architecture representations produced during the development process. We base our work on a three-level architecture description language (ADL) named Dedal, which represents architectures at three abstraction levels. Moreover, Dedal provides a formal base for managing version propagation. It is based on component substitutability that generalizes Liskov's substitutability principle. We propose a set of rules to support the prediction of version compatibility in terms of impact on the different architecture levels.

Keywords: Component-Based Software Engineering, Architecture evolution, Architecture versioning, Component substitutability, Version propagation.

1 Introduction

One of the main issues of software evolution management is the versioning activity [11]. One needs to keep track of changes in software not only during software programming but all along its complete life-cycle, including early development stages such as specification or post-deployment phases such as maintenance. Moreover, maintaining the history of changes is not sufficient. Changes, and their side-effects, are to be managed in a very fine-grained manner in order to keep track of valid software configurations. Moreover, collaborative work accentuate the need for versioning systems.

With the growing complexity of software systems, versioning becomes a very essential activity either for developers who need to be able to reuse adequate versions of components, packages and libraries [20], or for users that need to maintain up-to-date versions of their applications.

Many version control mechanisms have been proposed over the past years. They track changes in either source code, objects or models [8]. However, despite the fact that software development gives an increasing importance to approaches based on software architectures [7], little works cope with architectural versioning issues such as keeping track of architectural decisions all along the life-cycle of software but also being capable of predicting compatibility of versioned architectural artifacts for reuse purpose and guiding architects and developers.

The remainder of this paper is organized as follows. Section 2 presents the background and motivations for this paper. Section 3 develops our contribution: a study on version propagation prediction based on component substitutability. Section 4 presents an overview of state-of-the-art work on the problematic of versioning. Finally, Section 5 concludes the paper with several perspectives.

2 Background and Motivations

In this section, we present a three-level architecture description language named Dedal [21, 15], which aims at supporting the main steps of software development. By nature, Dedal is an ideal candidate ADL to keep a record of software evolution during its whole life-cycle using versions. To do so properly, the impact of versioning a description on other levels is to be taken care of.

DOI reference number: 10.18293/SEKE2017-118

2.1 Dedal: A Three-Level Architecture Description Language

Dedal models architectures at three levels that correspond to the specification, implementation and deployment stages: (a) The **specification level** is composed of abstract component types. These types are called roles and describe the functionalities of the future software. (b) The **configuration level** is composed of concrete component classes that realize the roles. The relation between roles and component classes is a n to m relation. (c) The **assembly level** is a deployment model composed of component instances that instantiate the configuration. The syntax of Dedal is not detailed in this paper as the mechanisms that are presented here rely on general concepts. However, more information is accessible in one of our team's previous paper [22].

In Dedal, two properties guaranty the integrity of the three levels: (a) Intra-level consistency guarantees that all components are properly connected to each other. (b) Inter-level coherence guarantees that the configuration realizes all the roles that are defined in the specification and all the component classes from the configuration are instantiated in the assembly. However, those properties may be violated after changes and need to be recovered through a propagation mechanism within and / or between architecture levels. One of the major contributions of Dedal is its evolution manager which maintains the three architecture description levels coherent with one another. Dedal has been formalized [15] thanks to the B language [1]. This makes it possible to automatically calculate an evolution plan that, when it exists, restores the overall architecture coherence after any of its levels has been subject to change.

2.2 Motivation

Having multiple description levels makes versioning of component-based architectures at several abstraction levels necessary. Indeed, when one of the three architecture-description level evolves, it may have serious impacts on the other levels.

This issue has been discussed in a position paper [14] which uses a three-level versioning graph inspired from Conradi's taxonomy [8]. Authors differentiate two version types and also propose three strategies in order to manage three-level architecture versioning.

Version types are as follows: (a) **Revisions** are intended to replace their predecessor. This means that a revision should preferably be able to substitute to its previous version. A revision aims at improving an existing artifact. (b) **Variants** may coexist with other versions. A variant aims at adding new functionalities to an existing artifact.

Figure 1 illustrates versioning relations between three-level architecture descriptions. First of all, this example represents two versioning branches and gives an overview of how related description levels may be affected by single-level-versioning. The first three-level architecture version of this example is the A.1.0 version. Then we create two different versions that will fork the versioning graph into two branches. A.1.1 is a revision derivated from the assembly level revision itself. A.2.0 creates a new branch by creating a variant in specification level. Finally, either a variant or a revision of a single level may affect the other levels of the architecture. This is what it is shown in A.1.2 that propagates the revision of its configuration level *Config.1.1* to *Assembly.1.2* and A.2.0 where the variant of the specification level is propagated to the configuration and then to the assembly level.

A very interesting use of variant and revision concepts could be to predict the impact of changes in terms of retro-compatibility of versionned entities (*e.g.*, architectures, components etc.). Indeed, four types of derivation are identified: (i) **Compatible revision**, which means that the new revision does not raise compatibility issues (*e.g.*, bug fixes, refactoring) and does not imply to propagate changes. (ii) **Incompatible revision**, which means that the new revision raises compatibility issues (*e.g.*, new technology) and implies change propagation. (iii) **Compatible variant** provides an alternative version that does not imply to propagate any change in the architecture. (iv) **Incompatible variant** provides an alternative version that will require to propagate changes in the architecture.

Next section presents a study on version propagation in order to identify scenarios dealing with predicting the impact component substitution may have at any architecture level.

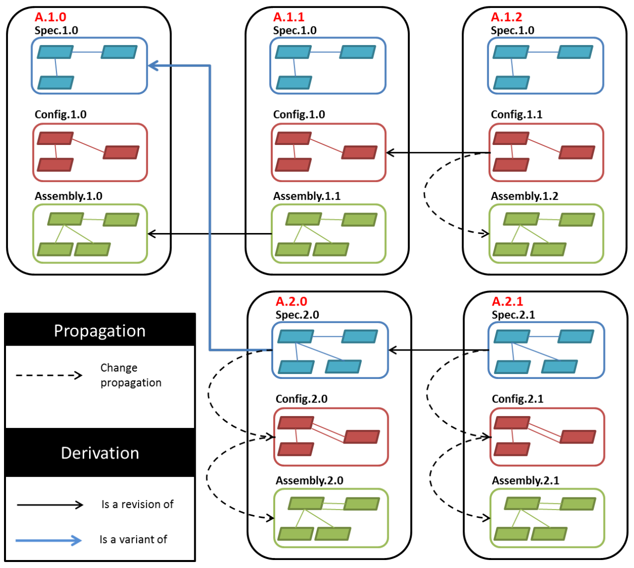

Figure 1. Versioning Architectures

3 Predicting Version Propagation

This section discusses a typology of the architecture evolutions managed by Dedal that aims at bringing semantics to versioning concepts of Dedal.

3.1 Typology of Architecture Evolution

The component substitutability relation has already been formalized in Dedal. This concept embodies the impact a change may have on architecture descriptions. This is the reason why this section introduces a typology of architecture evolution based on substitutable artifacts replacements instead of listing every single change operation on architecture artifacts. The aim of such a typology is to be able to identify which change is compatible or not with existing architectures.

Several change operations are relevant: (a) *Adding* new artifacts. (b) *Removing* artifacts. (c) *Replacing* artifacts with other that may be or not substitutable with the previous ones.

At architecture description level, a component may be replaced: (a) by a component that is *substitutable* for the replaced component. (b) by a component that is *not substitutable* for the replaced component.

Besides in a component-based architecture, several kinds of artifacts are subject to change: (a) components themselves, this is the most coarse-grained change, (b) a finer–grained change regards the interfaces of a component, (c) finally, the finest-grained change is performed on signatures.

When studying architecture versioning in a Dedal development, the initial level of change is especially relevant. Indeed, a very important aspect of having a three-level ADL is to be able to perform co-evolution of those levels according to the origin of the perturbation. This is discussed in next section.

3.2 Change Impact Analysis

As mentioned in the beginning of this paper, Dedal is a three-level ADL, which means that an initial change may occur at any of its architecture levels. This study is based on substitution of provided / required functionality signature. The notion of type is derived from Liskov *et al.* [13].

Notations. In order to avoid any kind of ambiguities, the used notation is described here.
$T_1 \prec T_2$: T_1 is a subtype of T_2.
$T_1 \preceq T_2$: T_1 is a subtype of T_2 or equal to T_2.
$T_1 \succ T_2$: T_1 is a supertype of T_2.
$T_1 \succeq T_2$: T_1 is a supertype of T_2 or equal to T_2.
$T_1 \parallel T_2$: T_1 is not comparable to T_2.
$(T_1 \npreceq T_2) \Leftrightarrow \neg(T_1 \preceq T_2) \Leftrightarrow ((T_1 \succ T_2) \vee (T_1 \parallel T_2))$: T_1 is either a supertype of T_2 or not comparable to T_2.

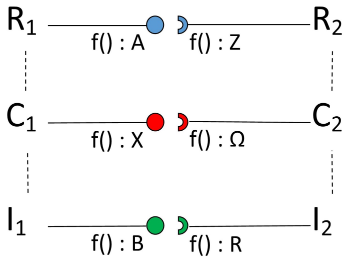

Figure 2. Base-Case: Functionality Connection Within a Three-Level Component-Based Architecture

$(T_1 \nsucceq T_2) \Leftrightarrow \neg(T_1 \succeq T_2) \Leftrightarrow ((T_1 \prec T_2) \vee (T_1 \parallel T_2))$: T_1 is either a subtype of T_2 or not comparable to T_2.
$T_2 \looparrowright T_1$: T_2 replaces T_1.

Functionality substitutable for another. For a provided functionality sp$_{new}$, being substitutable for another functionality sp$_{old}$ means that (1) the return type of sp$_{new}$ is equal or subtype [13] of the return type of sp$_{old}$ and (2) that the input parameters of sp$_{old}$ are subtypes of the ones of sp$_{new}$ [3]. Conversely, for a required functionality sr$_{new}$, being substitutable for another functionality sr$_{old}$ means that (1) the return type of sr$_{new}$ is equal or a supertype of the return type of sr$_{old}$,and (2) that the input parameters of sr$_{old}$ are supertypes of the ones of sr$_{new}$.

Figure 2 is a base case that represents a specification composed of two component roles R_1 and R_2 that are realized respectively by the component classes C_1 and C_2, which are in turn instantiated by respectively I_1 and I_2.

3.2.1 Versioning at Specification Level

Table 1a summarizes what effect, replacing R_1 by its new version $R_1^{'}$ which provides a functionality $f() : Y$, may have on the configuration. Several outcomes are observable:

- **The version is not propagated.** This happens when the new version of the role still is compatible with other roles within specification, and the component class that realized the replaced role still is a subtype of the new role. The condition of non-propagation is summarized by $X \preceq Y \preceq Z$ for any replacement type. Y can either be substitutable for A or not.
- **The version is propagated.** This happens when information is lost during the replacement operation. According to the information that is lost, propagation may take different aspects: (i) **Inter-level propagation**, which occurs if Y is a subtype of X or if they are not comparable. (ii) **Intra-level propagation**, which is the result of breaking connections within the specification. This is possible if Y is a supertype of Z or

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

(a) Specification Level

Hypothesis on types	
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$	
$Y \looparrowright X$	
Non-propagation	
$B \preceq Y \preceq A$	
Propagation	
Inter-level	Intra-level
$(Y \npreceq A \Rightarrow\uparrow)$ $\vee (Y \nsucceq B \Rightarrow\downarrow)$	$Y \npreceq \Omega$
$[(Y \npreceq A) \vee (Y \nsucceq B)] \wedge (Y \npreceq \Omega)$	

(b) Configuration Level

Hypothesis on types	
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$	
$Y \looparrowright B$	
Non-propagation	
$Y \preceq X$	
Propagation	
Inter-level	Intra-level
$Y \npreceq X$	$Y \npreceq R$
$Y \npreceq R$	

(c) Assembly Level

Table 1. Replacing Components: Providing a Functionality

if they are not comparable. (iii) **Inter and intra-level propagation**, this is a combination of both propagation conditions. However the only reachable condition is that Y is not comparable neither to X nor Z.

3.2.2 Versioning at Configuration Level

Table 1b summarizes what effect, replacing C_1 by a third component class C'_1, which provides a functionality $f() : Y$, may have on specification or/and assembly. Then several outcomes are observable:

- **The version is not propagated.** The condition of non-propagation is summarized by $B \preceq Y \preceq A$ for any type of replacement. Y can either be substitutable to X or not. $(Y \preceq A)$ ensures C'_1 realizes R_1 and $(Y \succeq B)$ ensures I_1 can be used as an instance of C_3.
- **The version is propagated.** As configuration is the intermediate architecture level, then change may be propagated: (i) **To the specification ($\uparrow$)** if Y is not a subtype of A. (ii) **To the assembly ($\downarrow$)** if Y is not a supertype of B. (iii) **Within the configuration** if Y is not a subtype of Ω. This condition also implies at least a propagation to the specification since $(A \prec \Omega) \vdash (Y \npreceq \Omega) \Rightarrow (Y \npreceq A)$ (iv) **In every directions** with any combination of the previously expressed conditions. The change may be propagated in one, two or three directions at a time.

3.2.3 Versioning at Assembly Level

Table 1c is a summary of the impact that replacing I_1 by a third component instance I'_1, which provides a functionality $f() : Y$, may have on the configuration. Two cases are possible:

- **The version is not propagated.** The condition of non-propagation is expressed by $Y \preceq X$ for any type of replacement (*substitutable* or *not-substitutable*). This condition ensures that I'_1 instantiates C_1 and is compatible with I_2.
- **The version is propagated.** As previously, there exist several ways to propagate change: (i) **Inter-level propagation** if Y is not a subtype of X. (ii) **Intra-level propagation** if Y is not a subtype of R. This is also a sufficient condition for an inter-level propagation.

Tables 2a, 2b and 2c summarize symetric change impact analysis that corresponds to required functionality replacement at the three architecture levels (R_2, C_2 and I_2 are replaced by a component that requires a functionality $f() : Y$).

3.2.4 Generalization

1 **to** n **replacement.** The only cases that have yet been discussed are 1 to 1 replacement operations. However, this is sufficient to describe the propagation problem. Indeed, when a single role is realized by n component classes, then we can see those component classes as only one single composite component class that realizes a role. This is exactly the same situation when a single component realizes n component roles: we can see those component roles as a single one that exposes the interfaces which describe the n roles.

Multiple connections. A component interface may be connected to several interfaces in an architecture. A solution to generalize to such cases is to separately study each connection.

Thus the result of this study is as follows: substitutability is a good criterion for predicting impact on intra-level consistency. However a more detailed approach is needed for studying impact on inter-level coherence as it has been presented in the previously discussed tables.

4 Related Work

Initially, the versioning activity aimed at representing and retrieving the past states of a file during its evolution [10]. Versioning most often relies on text-based mech-

Hypothesis on types	
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$	
$Y \looparrowright Z$	
Non-propagation	
$A \preceq Y \preceq \Omega$	
Propagation	
Inter-level	Intra-level
$Y \npreceq \Omega$	$Y \nsucceq A$
$(Y \parallel \Omega) \wedge (Y \parallel A)$	

(a) Specification Level

Hypothesis on types	
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$	
$Y \looparrowright \Omega$	
Non-propagation	
$Z \preceq Y \preceq R$	
Propagation	
Inter-level	Intra-level
$(Y \nsucceq Z \Rightarrow\uparrow) \vee (Y \npreceq R \Rightarrow\downarrow)$	$Y \nsucceq X$
$[(Y \nsucceq Z) \vee (Y \npreceq R)] \wedge (Y \nsucceq X)$	

(b) Configuration Level

Hypothesis on types	
$B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$	
$Y \looparrowright R$	
Non-propagation	
$Y \succeq \Omega$	
Propagation	
Inter-level	Intra-level
$Y \npreceq \Omega$	$Y \nsucceq B$
$(Y \npreceq \Omega) \wedge (Y \nsucceq B)$	

(c) Assembly Level

Table 2. Replacing Components: Requiring a Functionality

anisms [5] as in version control systems like Git [19] or CVS [16]. However, models cannot be versioned as text.

4.1 Previous Work

In previous papers, Dedal has been introduced as an ADL for automatically managing evolution among multi-level component-based software architectures [15]. As changes may occur at any of the three architecture-levels of Dedal, the ADL has been formalized with the B language [1] to calculate evolution plans. An evolution plan aims at recovering consistence within architectural levels and coherence between those levels after a perturbation of any of the architecture levels. Also, the versioning activity has been considered in Dedal through a position paper, which presents a three-level versioning graph for managing Dedal architecture versioning [14].

4.2 Versioning Models

A model versioning process is composed of three steps [2]: (a) **The change detection phase.** Two types of approaches are identifiable for change detection [17]: (i) *State-based detection* only considers the final state of the modified versions. (ii) *Operation-based detection* relies on the model editor for keeping track of all the operations applied on the original version that lead to the final version. (b) **The conflict detection phase.** Conflicts may arise in case of parallel changes that are potentially overlapping or contradicting. For instance, several architects can modify a single version of a single model artifact simultaneously. (c) **The inconsistency detection phase.** Inconsistencies may happen while merging concurrent versions of a model artifact.

Models typically are entities linked to one another. Versioning one entity may thus have an impact on others. This makes co-evolution an interesting field of research. Research on model versioning has brought various approaches for managing co-evolution [17]: (a) **Inference approaches** [6] rely on meta-model comparison to generate a strategy for evolving models to conform to an updated meta-model. (b) **Operator approaches** [12] are based on patterns and are characterized by a set of predetermined strategies that can handle a step-by-step co-evolution of meta-models and models. (c) **Manual approaches** migrate models manually so they correspond to an updated meta–model.

Evolution in Dedal typically relies on inference mechanisms. The existing work on model co-evolution only copes with a top-down (meta-model to model) approach that corresponds to an historic use of meta-models in model-driven engineering (MDE). Indeed as a meta-model describes rules a model must respect in order to conform to it, there is no real need to adopt a bottom-up approach in MDE processes to evolve meta-models. Yet, multi-direction co-evolution is needed because an architect may need, for technical reasons, to adapt an implementation in such a way that it will not conform to the specification anymore. In such a case, change must be propagated from the implementation to the specification. This co-evolution need is well illustrated in Mokni *et al.* [15].

4.3 Versioning Architectures

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

SOFA [4] does not completely support the whole life-cycle of software since it only provides two abstraction levels (configuration and non-descriptive assembly). Moreover, the finest-grained type SOFA takes into account is the interface type which is not enough to predict version propagation.

Mae [18] is based on xADL 2.0 [9] that provides two abstraction levels by distinguishing design-time and run-time. They thus do not take into account the whole life-cycle of software. The finest-grained type xADL 2.0 takes care of is the interface type and so it does not deal with input parameters, which is not enough for predicting version propagation. Mae enhances xADL 2.0 interfaces by adding interface elements that correspond to signatures and their input parameters but still does not cope with the entire software life-cycle.

5 Conclusion and Future Work

This paper introduces a study on version propagation, based on substitutability principles that have been formalized in Dedal. Through this study we could identify component substitution scenarios at any architecture-level that do not imply propagation of changes. As a consequence, we could also identify different propagation conditions within and / or between architecture levels. Then this study brings the capability of predicting the impact of new artifact versions (*e.g.*, architecture, component) by knowing their types. As a result of this study we could determine that component substitution is not a fine-grained enough criterion for predicting propagation on adjacent architecture levels, so we need to deal with parameter types into signatures. However, this is a sufficient criterion to predict propagation within an architecture level.

In future work, it will be essential to handle propagation problematics when adding or removing versioned artifacts in architectures in order to exhaustively predict version propagation.

References

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

[2] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Why model versioning research is needed!? an experience report. In *Proceedings of the MoDSE-MCCM Workshop@ MoDELS*, volume 9, 2009.

[3] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier. Precalculating component interface compatibility using FCA. In J. Diatta, P. Eklund, and M. Liquière, editors, *Proceedings of the* 5^{th} *international conference on Concept Lattices and their Applications*, pages 241–252. CEUR Workshop Proceedings Vol. 331, Montpellier, France, Oct. 2007.

[4] T. Bures, P. Hnětynka, and F. Plášil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications.* 4^{th} *International Conference on*, pages 40–48. IEEE, 2006.

[5] A. Cicchetti, F. Ciccozzi, and T. Leveque. A solution for concurrent versioning of metamodels and models. *Journal of Object Technology*, 11(3):1–32, 2012.

[6] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In *Proceedings of the International Conference on Theory and Practice of Model Transformations*, pages 35–51. Springer, 2009.

[7] P. Clements and M. Shaw. " The golden age of software architecture" revisited. *IEEE software*, 26(4), 2009.

[8] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[9] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, Apr. 2005.

[10] J. Estublier and R. Casallas. Three dimensional versioning. *Software Configuration Management*, pages 118–135, 1995.

[11] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, 2005.

[12] M. Herrmannsdoerfer. Operation-based versioning of metamodels with COPE. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM 2009*, pages 49–54, 2009.

[13] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[14] A. Mokni, M. Huchard, C. Urtado, , and S. Vauttier. A three-level versioning model for component-based software architectures. In *Proceedings of the* 11^{th} *International Conference on Software Engineering Advances*, pages 178 – 183, Roma, Italy, Aug. 2016.

[15] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, and H. Y. Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming*, 127:24–49, 2016.

[16] T. Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[17] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272 – 280, 2016.

[18] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

[19] L. Torvalds and J. Hamano. Git: Fast version control system. *URL http://git-scm. com*, 2010. last visited: 03.05.2017.

[20] C. Urtado and C. Oussalah. Complex entity versioning at two granularity levels. *Information systems*, 23(3-4):197–216, 1998.

[21] H. Y. Zhang, C. Urtado, and S. Vauttier. Architecture-centric component-based development needs a three-level ADL. In M. A. Babar and I. Gorton, editors, *Proceedings of the* 4^{th} *European Conference on Software Architecture*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, Aug. 2010. Springer.

[22] H. Y. Zhang, L. Zhang, C. Urtado, S. Vauttier, and M. Huchard. A three-level component model in component based software development. In *Proceedings of ACM SIGPLAN Notices*, volume 48, pages 70–79. ACM, 2012.