Preliminary study on predicting version propagation in three-level component-based architectures

Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

Preliminary study on predicting version propagation in 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

Keeping a trace of the evolution of software architectures is an important issue for architects. This paper is a summary of a submitted paper which states that the versioning activity does not propose solutions that fit software architectures, especially when dealing with co-evolution of different architecture representations that may be produced during the development process. This work is based on a three-level architecture description language (ADL) named Dedal, which gives a representation of architectures at the main stages of a software life-cycle. Dedal also performs co-evolution through change propagation within these representations. Another advantage of Dedal is that it has been formalized and thus provides a formal ground for studying version propagation. We based our study on substitutability relations that exist when a component is replaced at any of the three architecture levels. We aim at predicting compatibility of versioned artifacts in terms of impact on the different architecture levels.

1 Introduction and context

The versioning activity is one of the main issues of software evolution management [ELVDH$^+$05]. Indeed, during its life-cycle, a software may be subject to many changes. Developers and software architects deeply need to keep track of changes that occur on a software all along its life-cycle, including early development phases like specification or even post-production activities such as software maintenance. However, changes may have side-effects that have to be managed in a very fine-grained manner for keeping track of valid software configurations and collaborative work makes versioning systems necessary. As a consequence of the growing complexity of software systems, versioning becomes a very necessary activity for either developers or users that respectively need to be able to reuse specific versions of components, packages or libraries [UO98], and need to maintain up-to-date versions of their applications. Many version control approaches have been proposed. They can track changes within source code, objects or models [CW98]. However, little works deal with architectural versioning issues. Some of those issues are keeping track of architectural decisions that occur during the whole life-cycle of the software and also predicting versioned architectural artifacts compatibility for reuse purposes and also for guiding architects and developers.

This work is based on a three-level architecture description language (ADL) named Dedal [ZUV10, MHU$^+$14, MUV$^+$16] which aims at representing the whole life-cycle of a software by providing three levels of architecture descriptions. Those levels correspond to the specification, implementation and deployment development stages : (a) the specification level is composed of abstract component roles that describe the functionalities of the software, (b) the configuration level is composed of component classes that realize component roles (n to m relation), and (c) the assembly level is a deployment model composed of all the component instances that instantiates the component classes.

The Dedal ADL ensures integrity of its three levels thanks to two properties: (a) intra-level consistency ensures

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

that all component of an architecture level are connected to each other, and (b) inter-level coherence guarantees that the configuration realizes all the roles that are described in the specification level and that the instances of the assembly instantiate all the component classes that are described in the configuration. However, a change may break this integrity that needs to be recovered through a propagation mechanism within and / or between architecture levels. To do so, Dedal has been formalized [MUV+16] thanks to the B language [Abr96] and provides an evolution manager which makes it possible to automatically calculate an evolution plan that, if it exists, restores the overall architecture coherence after any of its levels has been subject to change.

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.

In first place, versioning aimed at keeping an hisory of changes to represent and retrieve past states of a file. Most of the time, this activity relies on text-based mechanisms [CCL12]. This is the case in version control systems like Git [TH10] or CVS [Mor96]. However, models and architectures cannot be versioned as text. Then literature describes the versioning activity on model as a sequence of three phases [ABK+09]: (i) the change detection phase records the delta between the previous version and the new one either as an history of operations that lead to the new version or as an history of states [PMR16], (ii) the conflict detection phase, since parallel changes may be overlapping or contradicting, and (iii) the inconsistency detection phase which happens when merging concurrent versions of a model artifact.

Many approaches exist to manage model co-evolution: (a) inference approaches rely on meta models for deriving strategies of evolution of models, (b) operator approaches are based on patterns and defined by a set of predetermined strategies that can handle a step-by-step co-evolution of meta-models and models, and finally (c) manual approaches to manually migrate models so they correspond to an updated meta-model.

Surprisingly, little work deals with versioning architectures. Among those few approaches we can cite some ADLs such as SOFA [BHP06], Mae [RVDHMRM04] and xADL [DVDHT05]. However, those ADLs are not sufficient for keeping trace of changes during the whole life-cycle of a development. Indeed, none of them provides a representation of the entire life-cycle of a software since they only provide two abstraction levels. Moreover, the SOFA 2.0 ADL does not cope with a fine enough grained typing for predicting version propagation as discussed in next section.

2 Predicting version propagation

As discussed earlier, Dedal is a three-level ADL, which implies that changes may occur at any of its architecture levels. The component substitutability relation between two functionalities has already been formalized in Dedal. This concept conditions the impact a change may have on adjacent architecture descriptions.

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

Substitutable functionalities [LBDH+17].
For a provided f^{prov}_{new} functionality, being substitutable for a f^{prov}_{old} functionality means that: (1) f^{prov}_{new} and f^{prov}_{old} have the same name, (2) the return type of f^{prov}_{new} is equal to or a subtype of [LW94] the return type of f^{prov}_{old} and (3) the input parameters of f^{prov}_{old} are subtypes of the ones of f^{prov}_{new} [ADH+07].
Conversely, for a required f^{req}_{new} functionality, being substitutable for a f^{req}_{old} functionality means that: (1) f^{req}_{new} and f^{req}_{old} have the same name, (2) the return type of f^{req}_{new} is equal to or a supertype of the return type of f^{req}_{old},and (3) the input parameters of f^{req}_{old} are supertypes of the ones of f^{req}_{new}.

This definition is used to recursively define interface substitutability and, furthermore, component substitutability [ADH+07].

Figure 1 represents the three-level description of a connection as the simplest base-case for illustrating the described substitutability mecanisms. The specification is composed of two component roles (R_1 and R_2) that are realized in the configuration by two component classes (respectively C_1 and C_2), which are instantiated in the assembly by two instances (respectively I_1 and I_2). A, B, X, Z, Ω

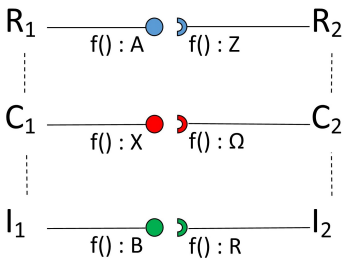

Figure 1: Connecting functionalities within a three-level component-based architecture

Hypothesis on types $B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$					
Specification level		Configuration level		Assembly level	
$Y \nrightarrow A$		$Y \nrightarrow X$		$Y \nrightarrow B$	
Non-propagation					
$X \preceq Y \preceq Z$		$B \preceq Y \preceq A$		$Y \preceq X$	
Propagation					
Inter-level	Intra-level	Inter-level	Intra-level	Inter-level	Intra-level
$(Y \parallel X) \vee (Y \prec X)$	$(Y \parallel Z) \vee (Y \succ Z)$	$(Y \npreceq A \Rightarrow\uparrow) \vee (Y \not\succeq B \Rightarrow\downarrow)$	$Y \npreceq \Omega$	$Y \npreceq X$	$Y \npreceq R$
$(Y \parallel X) \wedge (Y \parallel Z)$		$[(Y \npreceq A) \vee (Y \not\succeq B)] \wedge (Y \npreceq \Omega)$		$Y \npreceq R$	

Table 1: Versioning a component at any of the three abstraction levels: Providing a service

and R are return types of f functionality which is expressed at three levels of abstraction and that is either provided or required We can notice that there are two possible ways of propagating versions, by versioning either a provided or a required service.

2.1 Versioning a component that provides a service

Table 1 summarizes what effect replacing R_1 by its version R'_1 (at specification level) or C_1 by its version C'_1 or I_1 by I'_1 may have on the different architecture levels. In each case, the new version provides a service $f() : Y$. At any of the abstraction levels, two outcomes are observable:

- **The version is not propagated.** Conditions of non propagation are identifiable at the three levels:
  (a) At **specification level**, the version is not propagated if $X \preceq Y \preceq Z$, Y may either be substitutable to A or not.
  (b) At **configuration level**, the non-propagation condition is summarized by $B \preceq Y \preceq A$ and Y can either be substitutable or not to X. Moreover, the realization relation between R_1 and C'_1 is preserved if $(Y \preceq A)$ and the instantiability of C'_1 by I_1 is ensured by $(Y \succeq B)$.
  (c) At **assembly level**, the version does not need to be propagated if $(Y \preceq X)$, Y may be substitutable to B or not.
- **The version is propagated.** This is the case when information is lost during the version replacement. At each of the three levels, conditions for propagating version are as follows:
  (a) At **specification level**, we identify three kinds of propagation: (i) Inter-level propagation that corresponds to a propagation to the configuration level when Y is a subtype of X or if they are not comparable. (ii) Intra-level propagation if the connection is broken in the specification, which can occur if Y is a supertype of Z. (iii) Inter and intra-level propagation that corresponds to the combination of (i) and (ii).
  (b) At **configuration level**, we also identify three kinds of propagation: (i) Inter-level propagation that corresponds either to a propagation to the specification level ($\uparrow$) when Y is not a subtype of A or to a propagation to the assembly level ($\downarrow$) when Y is not a supertype of B. (ii) Intra-level propagation occures when Y is not a subtype of Ω. (iii) Propagation to any directions that corresponds to the combination of (i) and (ii).
  (c) At **assembly level**, there are, as previously, several ways of propagating versions: (i) Inter-level propagation when Y is not a subtype of X. (ii) Intra-level propagation if Y is not a subtype of R. We can notice that it is a sufficient condition for an inter-level propagation as well.

2.2 Versioning a component that requires a service

Table 2 summarizes the impact of replacing a component that requires a service, in each of the three architecture levels. Then R_2, C_2 and I_2 are replaced by their respective version R'_2, C'_2 and I'_2 that requires the $f() : Y$ service.

2.3 Generalization

1 to n replacement.
The only cases that have been discussed are 1 to 1 replacement operations. However, this is sufficient to describe the

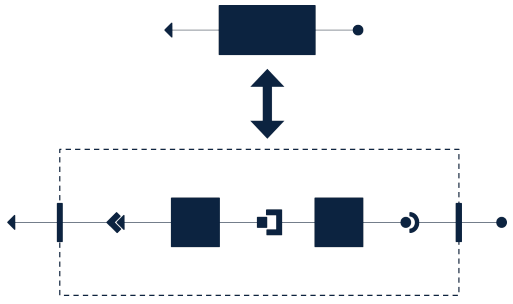

Figure 2: Multiple component seen as one composite component for 1 to n replacement

Hypothesis on types $B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R$					
Specification level		Configuration level		Assembly level	
$Y \nrightarrow Z$		$Y \nrightarrow \Omega$		$Y \nrightarrow R$	
Non-propagation					
$A \preceq Y \preceq \Omega$		$Z \preceq Y \preceq R$		$Y \succeq \Omega$	
Propagation					
Inter-level	Intra-level	Inter-level	Intra-level	Inter-level	Intra-level
$Y \not\preceq \Omega$	$Y \not\succeq A$	$(Y \not\succeq Z \Rightarrow\uparrow) \vee (Y \not\preceq R \Rightarrow\downarrow)$	$Y \not\succeq X$	$Y \not\succeq \Omega$	$Y \not\succeq B$
$(Y \parallel \Omega) \wedge (Y \parallel A)$		$[(Y \not\succeq Z) \vee (Y \not\preceq R)] \wedge (Y \not\succeq X)$		$(Y \not\succeq \Omega) \wedge (Y \not\succeq B)$	

Table 2: Versioning a component at any of the three abstraction levels: Requiring a service

propagation problem. Indeed, when a single role is realized by n component classes then we can see those component classes as a 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. This is what Figure 2 illustrates.

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

3 Conclusions and future work

As a result of this study, we found out that component subtitutability is a good criteria for predicting the impact of change on intra-level consistency. However we could also identify multiple additional criteria for guarantying the integrity of the three architecture levels that have been discussed in the previous tables. A consequence of identifying non-propagation conditions is that we could also identify conditions for inter-level and / or intra-level version propagation. Thanks to this study, we are now able to predict the impact that a versioned architectural artifact may have on its own level as well as on the adjacent levels by only knowing its type. An essential work for the future is to predict version propagation in an exhaustive manner by studying version propagation when adding or removing versioned artifacts.

References

[ABK+09] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why model versioning research is needed!? an experience report. In *Proceedings of the MoDSE-MCCM Workshop@ MoDELS*, volume 9, 2009.

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[ADH+07] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel 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.

[BHP06] Tomas Bures, Petr Hnětynka, and František 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.

[CCL12] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. A solution for concurrent versioning of metamodels and models. *Journal of Object Technology*, 11(3):1–32, 2012.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[DVDHT05] Eric M. Dashofy, André Van Der Hoek, and Richard 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, 2005.

[ELVDH+05] Jacky Estublier, David Leblang, André Van Der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy 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.

[LBDH+17] Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Substitutability-Based Version Propagation to Manage the Evolution of Three-level Component-Based Architectures. In *Proceedings of the* 29^{th} *International Conference on Software Engineering & Knowledge Engineering*, 2017. (to appear).

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[MHU+14] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. A three-level formal model for software architecture evolution. In *Proceedings of the* 7^{th} *Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE 2014)*, L'Aquila, Italy, July 2014.

[Mor96] Tom Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[MUV+16] Abderrahman Mokni, Christelle Urtado, Sylvain Vauttier, Marianne Huchard, and Huaxi Yulin Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming*, 127:24–49, 2016.

[PMR16] Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272 – 280, 2016.

[RVDHMRM04] Roshanak Roshandel, André Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

[TH10] Linus Torvalds and Junio Hamano. Git: Fast version control system. *URL http://git-scm. com*, 2010. last visited: 03.05.2017.

[UO98] Christelle Urtado and Chabane Oussalah. Complex entity versioning at two granularity levels. *Information systems*, 23(3-4):197–216, 1998.

[ZUV10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In Muhammad Ali Babar and Ian Gorton, editors, *Proceedings of the* 4^{th} *European Conference on Software Architecture*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, Aug. 2010. Springer.