A Three-Level Versioning Model for Component-Based Software Architectures

Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

A three-level versioning model for component-based software architectures

Abderrahman Mokni*, Marianne Huchard†, Christelle Urtado* and Sylvain Vauttier*
*LGI2P, Ecole Nationale Supérieure des Mînes Alès, Nîmes - France
Email: {abderrahman.mokni, christelle.urtado, sylvain.vauttier}@mines-ales.fr
†LIRMM, CNRS and Université de Montpellier, Montpellier, France
Email: huchard@lirmm.fr

***Abstract*—Software versioning is intrinsic to software evolution. It keeps history of previous software states (versions) and updates the software to the latest stable version. In the last twenty years, a lot of work was dedicated to software versioning. Many version control mechanisms were proposed to store and track software versions for different software forms (code, objects, models, ...). This paper addresses in particular software architecture versioning considering three abstraction levels: specification, implementation and deployment. In previous work, we proposed an approach that generates evolution plans for three-level component-based software architectures. The generated plans deal with a change initiated from one of the three abstraction levels and propagate it to the other levels in order to keep software descriptions consistent and coherent all along the software lifecycle. On this basis, we propose a versioning model that stores information about evolution and also keeps track of the right version of architecture descriptions at a given abstraction level.**

Keywords–architecture evolution, abstraction levels, versioning, component reuse.

I. Introduction

Versioning is central to software evolution management [1]. In order to ensure the continuity of a software product, it is necessary to keep track of its changes and previous versions after each evolution. Versioning is both essential for users and developers. For users, versioning helps to maintain their installed software up-to-date or at least warn them if their current software version becomes obsolete. For developers, versioning helps select / use the adequate versions of reusable software components, packages or libraries (considering, for instance, compatibility issues) and contributes to collaborative work by developing several versions in parallel or merging them [2].

During the last years, many version control mechanisms were proposed to store and track software versions for different software forms (code, models, objects, ...) [3].

While software architectures have become central to software development [4], little work was dedicated to architectural versioning. Existing work on architectural versioning [5], [6], [7] proposes basic versioning mechanisms that do not take into account the whole software lifecycle. Evolving a software architecture should not only focus on distinguishing the different versions of software system as a whole. Instead, it should state which versions of a description resulting from the different software development steps (*e.g.*, documentation, implementation model, deployment models, ...) is compatible to which other versions of other descriptions of the same system. Indeed, this information is crucial for requirement traceability. For instance, when evolving a software architecture, the architect needs mechanisms to know the latest version of its specification and also all the related implementations that will be affected by this evolution.

In this work, we address such versioning issues by proposing a version model that considers the three main steps of component-based software lifecycle: specification, implementation and deployment. The remainder of this paper is outlined as follows: Section II presents the background of this work namely the Dedal three-level architectural model [8] and its evolution management process. Section III presents the contribution of this paper consisting in the three-level versioning model of software architectures and the different versioning strategies. Section IV discusses related work and finally Section V concludes the paper and presents future work directions.

II. Background and motivation

In this work we address the versioning of component-based software architectures at three abstraction levels. First, we introduce the three-level architectural model Dedal and then we briefly explain how architecture evolution is managed in Dedal.

A. Dedal: the three-level architectural model

Reuse is central to component-based software development (CBSD) [9]. In CBSD, the software is constructed by assembling pre-existing (developed) entities called components. Dedal [8] proposes a novel approach to foster the reuse of software components in CBSD and cover all the three main steps of software development: specification, implementation and deployment. The idea 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 architecture specification) that represents an abstract and ideal view of the software. The implemented architecture can then be instantiated (the instantiation is called architecture assembly) and deployed in multiple contexts.

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, that realize the identified component roles in the architecture specification.

The architecture assembly corresponds to the third and lowest abstraction level. It is composed of component instances

that instantiate the component classes of the architecture configuration. An architecture assembly description represents a deployment model of the software.

Figure 1 illustrates the three architecture levels of Dedal and represents the running example of this paper. It consists of a variant of a Home Automation Software that controls the building's light during specific hours through an orchestrator (*HomeOrchestrator* component role). The specified functionalities – turning on / off the light (*Light* component role), controlling its intensity (*Intensity* component role) and getting information about the time (*Time* component role) – are respectively realized through the *AdjustableLamp* and *Clock* component classes. Two instances of *AdjustableLamp* are deployed to control the lighting of a Sitting room (*SittingLamp*) and a Desk (*DeskLamp*).

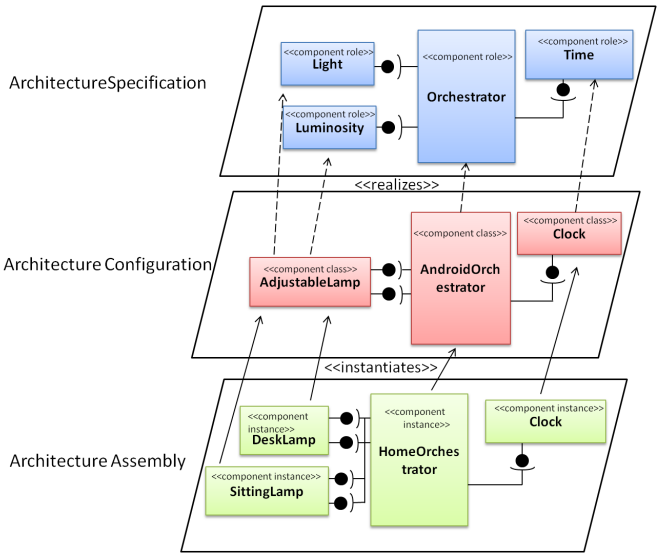

Figure 1. Running example

B. Evolution management in Dedal

Software architectures are subject to change at any abstraction level to meet new requirements, improve software quality, or cope with component failure. In previous work [10], [11], we proposed an evolution management process that deals with architectural change based on Dedal and the B formal language [12]. Using a customized solver, the evolution manager captures change at any abstraction level, controls its impact on the affected architecture and propagates it to the other abstraction levels to keep all the three architecture descriptions coherent. This results in generating sequences of change operations that evolve the affected architecture to a new consistent state. The generated sequences (called evolution plans) represent the delta between two software architecture versions in an operation-based manner.

C. Motivation

Versioning component-based software architectures at multiple abstraction levels is an important issue. Indeed, evolving an architecture description at one abstraction level may impact its other descriptions at the other abstraction levels. For instance, evolving a software specification may require evolving all its implementations and evolving an implementation may entail evolving all its instantiations. In the remainder, we set up a version model for three-level software architectures inspired by Conradi's taxonomy [3] and propose three strategies to manage multi-level versioning. The interest of this version model is twofold: (1) To capture information about evolution by storing the operations list that transformed the old architecture into the new architecture version and (2) to ensure that every architecture description version is related to its "right" homologue version at the other abstraction levels.

III. Versioning Component-Based Software Architectures

To set our version model, we take inspiration from Conradi's taxonomy [3] that distinguishes between two graphs representing two dimensions of software: the product space where each node is a part of the product and edges represent composition relationships and, the version space where nodes represent versions and edges derivations between them. Depending on the versioning model, the version space can be a linear, arborescent or direct acyclic graph. A version is called a *revision* when it is intended to replace its predecessors and is called a *variant* when it can coexist with other versions. In our model, we distinguish the architectural space that represents the architecture descriptions of the software at three abstraction levels (*i.e.*, specification, configuration and assembly) from the version space that represents the versions of an architecture at a given abstraction level. In the remainder, we give the representation of each space: a three-level graph for the architectural space and the version graph for the version space.

A. Three-level graph

The three-level graph (see example on Figure 2) is a representation of the software architecture without considering versioning. Nodes represent the architecture descriptions of the software while edges denote the implementation relations between nodes at different abstraction levels. The root node is the abstract architecture specification of the software (*e.g.*, Home Automation Software). All nodes at the second level (configuration level) represent the various implementations of that specification (*e.g.*, Android OS, Windows system). Finally, nodes at the third level (assembly level) represent the different deployment contexts related to a specific implementation (*e.g.*, Office, Sitting room, ...).

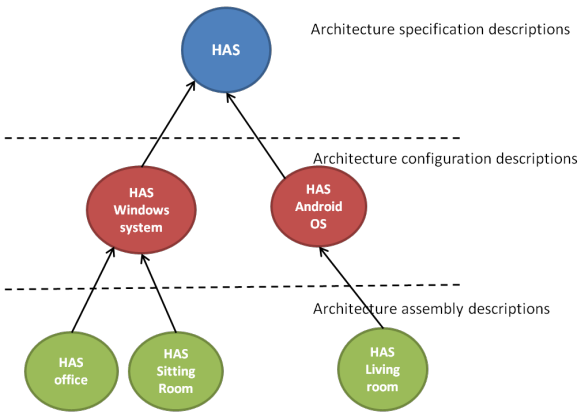

Figure 2. The three-level graph

The three-level graph supports multiple granularity levels. Indeed, each node points to another graph representing the architecture structure in terms of components and their connections. Composite components embed an inner architecture as well.
This paper focuses on architectural versioning and its impact at the other abstraction levels. Therefore, component versioning is out of the scope of this paper.
To summarize, the architectural space is a three dimensional space. It includes the specification dimension with a single node (that represents the architecture specification), the configuration dimension (that represents the different implementations of the software) and the assembly dimension (that represents the different deployments of the software).

B. Architecture version graph

Our version model covers all three architecture levels. Versioned entities may thus be an architecture specification, an architecture configuration or an architecture assembly. The version graph (Figure 3) is a representation of the V version set related to a given architecture. Each node describes a unique version of the architecture (identified by a unique version identifier) while edges represent derivations between versions.

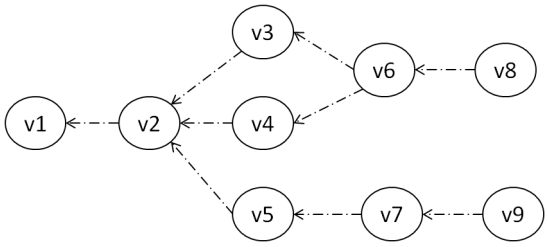

Figure 3. The version graph

The version model is change-based since the delta between two versions is expressed in term of change operations rather than states. A derivation is the change sequence enabling to construct a version v_2 from its predecessor v_1. Formally, a derivation is a function of type $d : V \rightarrow V$ where V is the version space and $d = op_1 \circ op2 \circ \ldots \circ op_n$ where op_i is an elementary change operation. If v_1 is a version of the software architecture, then successors of v_1 are the set of all the versions resulting from the derivations applied on v_1: $succ(v_1) = \{v | v = d(v_1)\}$.

The architecture version identifier contains information according to the abstraction level and the operation list that lead to the current version. At specification level, recorded information consists of a version ID and the operation list. At configuration level, these information are a version ID, the ID of the implemented specification and the operation list. Finally, at assembly level, the recorded information are the following: a version identifier, the instantiated configuration identifier and the operations list. We note that the operation list may be empty when the architecture description is newly created (for instance a new implementation variant of the software or a new deployment context).

C. Interaction between the three-level graph and the version graph

The version graph is orthogonal to the three-level graph. Indeed, every node in the three-level graph represents a point in the version graph and vice versa. Given a three-level graph G and a new derivation d of an architecture a in G, we aim to find the resulting three-level graph G' related to $a' = d(a)$. Therefore, we need to evaluate the impact of d on the whole graph G. Indeed, d may trigger a change propagation to the other nodes linked to a which in turn may recursively require to derive other nodes.

In most cases, this task requires human assistance to decide which derivations are really necessary (*e.g.*, correcting bugs, security faults, ...) and which are optional (*e.g.*, functional extensions, improvements, ...). Since we aim to automate the versioning process, we need to set some strategies so that the user can select the one to be used by default.

D. Versioning strategies

We propose three versioning strategies:

a) Minimum derivation strategy.: The minimum derivation strategy aims to limit change propagation and minimize the number of derivations to be applied on the three-level graph. The principle of this strategy is to version only the active impacted nodes without considering the propagation to all the other nodes. Active nodes consist in a tuple of three nodes (s, c, a) where s, c and a respectively denote an architecture specification, an architecture configuration and an architecture assembly. For instance, let us consider the three-level graph shown in Figure 4-a. The active node is $(s.v_1, c_1.v_1, a_{12}.v_1)$.

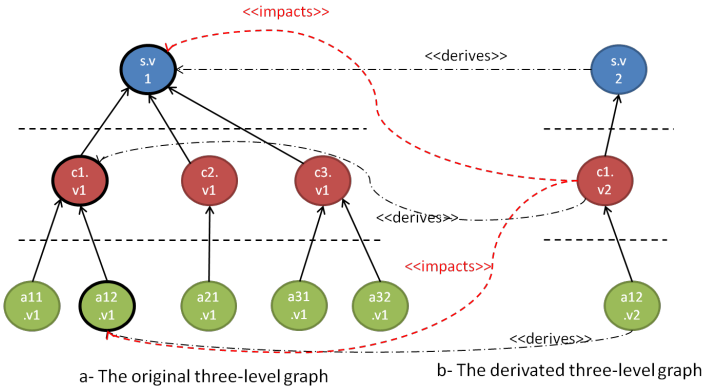

Figure 4. Example of minimum derivation

$d(c_1)$ triggers a change on the specification s_1 and a change on a_{12}. We have then to create a new three-level graph with the new versions $(s.v_2, c_1.v_2, a_{12}.v_2)$ (*cf.* Figure 4-b).

The minimum derivation strategy is suitable when the change purpose is not an emergency and when previous versions can coexist with new ones.

b) Full derivation strategy.: In contrast to the minimum derivation strategy, the full derivation strategy aims to version all the (directly and recursively) impacted architecture descriptions. It should be applied when the reason of evolution is important (for instance a security fault detected in a component used by all architecture implementations). Firstly, derivation is applied to the active node and then change is propagated recursively to the other nodes (*cf.* Figure 5). For instance, the revision of node $c1.v1$ (configuration level) is propagated to the other nodes as follows:

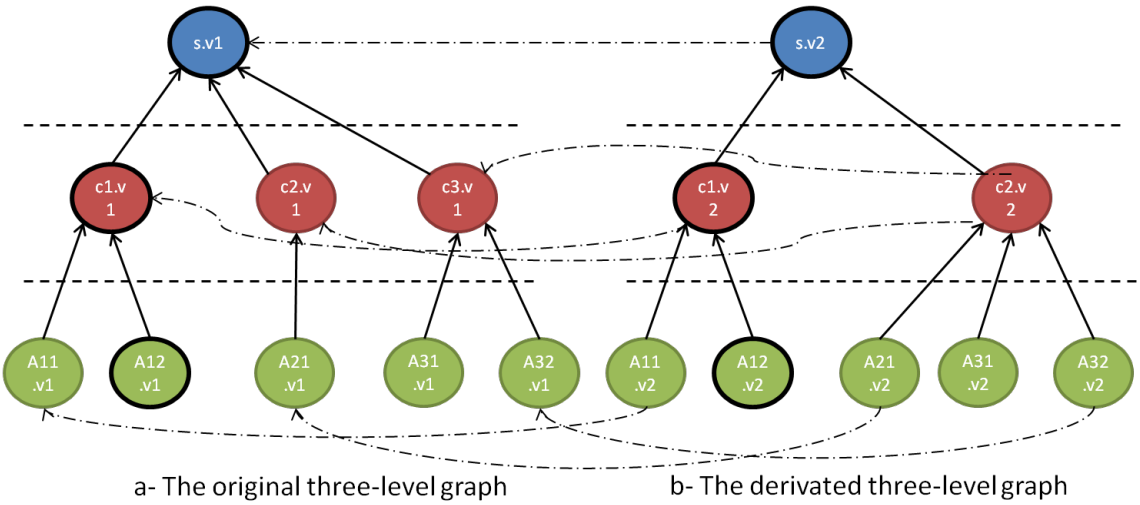

Figure 5. Example of full derivation

- derivation of a new specification revision $s.v2$ from $s.v1$,
- merging of $c2.v1$ and $c3.v1$ nodes into the new $c3.v2$ node (both evolution of $c2.v1$ and $c3.v1$ leads to $c3.v2$) and,
- revisions of all nodes at assembly levels, notably $a21.v2$ derived from $a21.v1$ becomes associated to $c3.v2$ configuration revision.

c) Custom derivation strategy.: This strategy is guided by the architect that has to specify which architecture descriptions to keep and which ones to replace by a new versions. The custom derivation strategy is used after a default application of the minimum derivation strategy so that necessary versions are always created.

IV. Related work

Software versioning has been studied for many years with the the objective to provide a SCM (Software Configuration Management) system [3]. Versioned entities has taken several forms and different granularities (source code lines, objects, libraries, ...). Early work targeted mainly source code versioning. Several versioning systems were proposed and widely used such as SVN [13], CVS [14] and Git [1].

With the emergence of component-based software development, more recent work addressed component versioning rather than source code [2]. Examples include JAVA [15], and COM .Net. More recent approaches treated as well the issue of component substitutability like the work of Brada *et al.* (SOFA) [16] and the issue of compatibility like the work of Stuckenholz *et al.* [17].

Regarding architectural versioning, only little work was dedicated. The SOFA 2.0 ADL [5] enables to version composite components and therefore an entire architecture (which is considered as a composite component). Other existing ADLs like MAE [6] and xADL 2.0 [7] also enable architecture versioning. However, all these architectural versioning models neitheir store enough information about evolution (operations list that results in the new architecture version) nor maintain the trace of the architecture throughout the whole software lifecycle. Another closely related work addressed architectural versioning at multiple abstraction levels [18]. The proposed approach is based on the SAEV model [19] that defines three abstraction levels of software architectures: the meta level, the architecture level and the application level. However, this taxonomy is different from Dedal since the meta level is a kind of language that encompasses the definition of architectural concepts to be used at the lower level.

[1] https://git-scm.com/about

V. Conclusion and future work

This work proposes a version model for software architectures. Based on Dedal, the model considers versioning at three architecture abstraction levels that covers the whole software lifecycle. It captures information about evolution (operation list) and enables to revise all or a part of the existing versions of the software architecture descriptions using three versioning strategies: full, minimal and custom. Future work consists in studying component versioning and its impact on architectural versioning considering compatibility issues. From a practical perspective, ongoing work is to automate this versioning mechanism and integrate it into DedalStudio, our eclipse-based tool that automatically manages the architecture evolution process [11].

References

[1] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of software engineering research on the practice of software configuration management," ACM Trans. Softw. Eng. Methodol., vol. 14, no. 4, Oct. 2005, pp. 383–430. [Online]. Available: http://doi.acm.org/10.1145/1101815.1101817

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

[3] R. Conradi and B. Westfechtel, "Version models for software configuration management," ACM Comput. Surv., vol. 30, no. 2, Jun. 1998, pp. 232–282. [Online]. Available: http://doi.acm.org/10.1145/280277.280280

[4] P. Clements and M. Shaw, ""the golden age of software architecture" revisited," IEEE Software, vol. 26, no. 4, July 2009, pp. 70–72.

[5] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in Software Engineering Research, Management and Applications, 2006. Fourth International Conference on, Aug 2006, pp. 40–48.

[6] R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic, and N. Medvidovic, "Mae—a system model and environment for managing architectural evolution," ACM Trans. Softw. Eng. Methodol., vol. 13, no. 2, Apr. 2004, pp. 240–276. [Online]. Available: http://doi.acm.org/10.1145/1018210.1018213

[7] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," ACM Trans. Softw. Eng. Methodol., vol. 14, no. 2, Apr. 2005, pp. 199–245. [Online]. Available: http://doi.acm.org/10.1145/1061254.1061258

[8] 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.

[9] I. Sommerville, Software engineering (9th edition). Addison-Wesley, 2010.

[10] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, and H. Y. Zhang, "An evolution management model for multi-level component-based software architectures," in The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015, 2015, pp. 674–679. [Online]. Available: http://dx.doi.org/10.18293/SEKE2015-172

[11] ——, "A formal approach for managing component-based architecture evolution," To appear in Science of Computer Programming, 2016.

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

[13] M. Pilato, Version Control With Subversion. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004.

[14] K. F. Fogel, Open Source Development with CVS. Scottsdale, AZ, USA: Coriolis Group Books, 1999.

[15] R. Englander, Developing Java Beans. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1997.

[16] P. Brada, "Component revision identification based on idl/adl component specification," SIGSOFT Software Engineering Notes, vol. 26, no. 5, Sep. 2001, pp. 297–298. [Online]. Available: http://doi.acm.org/10.1145/503271.503250

[17] A. Stuckenholz, "Component updates as a boolean optimization problem," Electronic Notes in Theoretical Computer Science, vol. 182, 2007, pp. 187 – 200, proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066107003945

[18] T. N. Nguyen, "Multi-level architectural evolution management," in System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, Jan 2007, pp. 258a–258a.

[19] M. Oussalah, N. Sadou, and D. Tamzalit, "A generic model for managing software architecture evolution," in Proceedings of the 9th WSEAS International Conference on Systems, ser. ICS'05. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2005, pp. 35:1–35:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1373716.1373751