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 traces all the changes that updates a software to its latest stable version. A lot of work has been dedicated to software versioning and many version control mechanisms are proposed to store and track software versions for different software artifacts (code, objects, models, etc.). This paper addresses in particular component-based software architecture versioning, considering three abstraction levels: specification, implementation and deployment. In previous work, we proposed an approach that generates evolution plans for such software architecture models. The generated plans deal with changes initiated on one of the three abstraction levels and propagate them to the other levels in order to keep architecture descriptions consistent and coherent. As an extension to these mechanisms, a versioning model is proposed in this paper to keep history of architecture definition versions. This versioning model soundly handles the co-evolution of the three abstraction levels by tracking both versions of each abstraction levels and versions of global architecture definitions.**

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].

Many version control mechanisms are currently proposed to store and track software versions for different software forms (code, models, objects, etc.) [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 tracking the different versions of software system as a whole. Indeed, the different steps of the software development process generates many artifacts (*e.g.*, documentation, implementation model, deployment models, etc.). It is valuable to keep separate version histories for each artifact and to build a global version history for the whole software from them. It fosters the reuse of artifacts in forward engineering processes (*e.g.*, the implementation of a given specification on different technical platforms or the deployment of a given implementation in different execution contexts). It also enables to trace every design decisions and their impacts (required co-evolution). 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 and its evolution management process [8]. Section III presents the contribution of this paper consisting in a three-level versioning model for software architectures and its different versioning strategies to support co-evolution on these three levels. Section IV discusses related work and finally Section V concludes the paper and presents future work directions.

II. BACKGROUND AND MOTIVATION

This work addresses 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, 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 execution contexts.

A Dedal architecture model is then constituted of three descriptions that correspond to the three abstraction levels:

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

 ISBN: 978-1-61208-498-5

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 (customized component instances fitting given execution contexts).

Fig. 1 illustrates the three architecture levels of Dedal and represents the running example of this paper. It consists of a Home Automation Software that controls the building's light during specific hours. Its architecture specification is composed of an orchestrator (*Orchestrator* component role) linked to device control functionalities – turning on / off the light (*Light* component role), controlling its intensity (*Luminosity* component role) and getting information about the time (*Time* component role). These component roles are respectively implemented by the *AndroidOrchestrator*, *AdjustableLamp* and *Clock* component classes. This architecture implementation can then be instantiated to describe specific architecture deployments. For instance, the architecture assembly in Fig. 1 is composed of two *AdjustableLamp* component instances that control the lighting of a Sitting room (*SittingLamp*) and a Desk (*DeskLamp*).

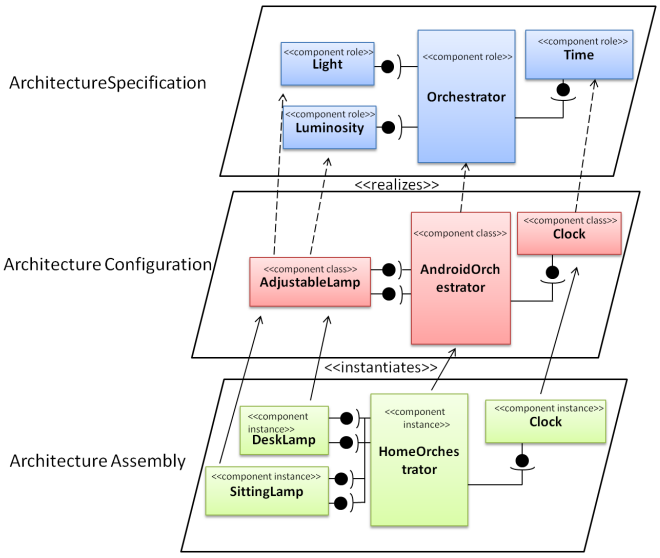

Figure 1. Running example

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 B 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 architecture descriptions coherent, both locally (each architecture description level separately) and globally (the whole architecture definition). 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. The same way, evolving an implementation may entail evolving not only all its instantiations but also its corresponding specification (to prevent inter-level definition mismatches known as drift and erosion [13]). 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 history by storing the change operation lists that transform architecture definitions into new versions and more importantly (2) to capture information about the co-evolution history by maintaining links between corresponding versions on the different abstraction levels to define versions of the whole architecture definition.

III. Versioning Component-Based Software Architectures

The design of our version model is inspired 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*, which represents software architecture descriptions at the three abstraction levels we consider (*i.e.*, specification, configuration and assembly), from the *version space*, which represents the versions of an architecture at a given abstraction level. In the remainder, we detail the representation of each space.

A. The architectural space

The architectural space consists in a set of trees that provide software architecture definitions at three abstraction levels (*cf.* Fig. 2). Nodes represent architecture definitions while edges denote realization relations between nodes at different abstraction levels. The root node of each tree corresponds thus to the specification of an architecture (*e.g.*, Home Automation Software architecture specification). The second level nodes represent all the variant implementations of that specification (*e.g.*, Android OS, Windows system architecture configurations). Finally, the third level nodes represent the variant instantiations that are used to deploy architectures configurations in different execution contexts (*e.g.*, HAS Office architecture assembly, Sitting room architecture assembly, etc.).

 ISBN: 978-1-61208-498-5

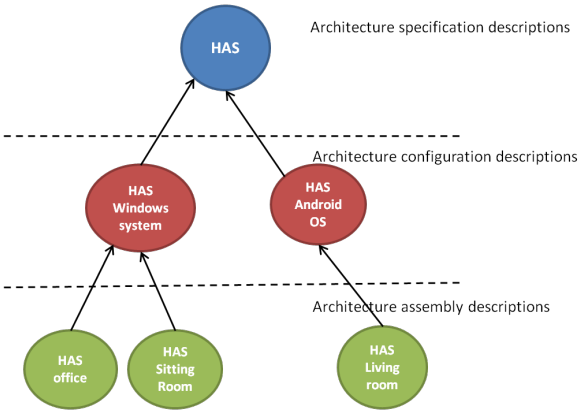

Figure 2. The three-level graph

The architectural space graph supports multiple granularity levels. Indeed, each node points to another graph representing the architecture structure in terms of components and their connections (*cf.* Fig. 1). Composite components embed an inner architecture as well.

The architectural space provides thus a comprehensive set of architecture definitions, including all their existing variants. It can be used to structure and then browse architecture model repositories, as part of a Model-Based Software Engineering environment. Its point of view is intentionally static (the historic derivation relations between architecture definition elements are omitted), in order to separate evolution concerns in the version space.

B. The version space

The architecture version space is composed of a set of version graphs. Each version graph (Fig. 3) is a representation of the version set, called V, related to a given architecture. Each node defines a unique version of the architecture (identified by a unique version identifier) while edges represent derivation relations between versions (the source version is obtained by an evolution of the target version). Our version model covers all the three architecture definition levels. Versioned entities may thus be architecture specifications, architecture configurations or architecture assemblies.

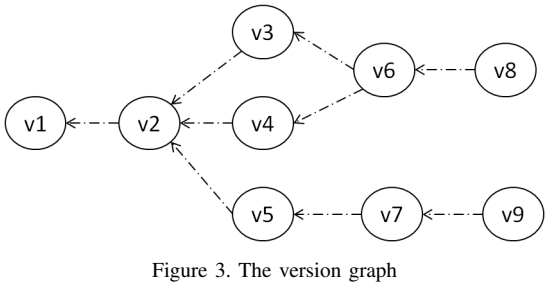

Figure 3. The version graph

The version model is change-based since the delta between two versions is expressed in terms 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 corresponding 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 change operation list. At configuration level, these information are a version ID, the ID of the implemented specification and the change operation list. Finally, at assembly level, the recorded information are accordingly a version identifier, the instantiated configuration identifier and the change operations list. The change operation list may be empty when the architecture description is created from scratch (for instance the specification of a new architecture or an implementation variant for a new platform).

C. Relations between the architectural and the version spaces

As aforementioned, the version space is intended to record all the versions of all the architecture definitions that are created by development and evolution processes. It provides a comprehensive and historic vision of architecture definitions, that is suitable to design architectures by the reuse and the evolution of exiting ones. However, as it does not distinguish revisions from variants, the version space does not provides a synthetic vision of the actual architectures definitions, *i.e.*, the up-to-date architecture definitions (based on the latest revisions), with its possible variants. This is the purpose of the architectural space, which can be extracted from the version space to provide architects with a clear view of the usable architecture definitions.

Every node in the architectural space corresponds thus to a node in the version space. Given a three-level graph G and a new derivation d of an architecture definition a in G, we aim to find the resulting three-level graph G' related to $a' = d(a)$. To do so, 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, what may in turn recursively imply to derive other nodes.

In most cases, this task requires human assistance to decide which derivations are really necessary (*e.g.*, correcting bugs, security enforcement, etc.) and which are optional (*e.g.*, functional extensions, improvements, etc.). To automate this process, we propose versioning strategies that can be selected and activated as required by architects to manage architecture model repositories.

D. Versioning strategies

We propose three versioning strategies:

a) Minimum derivation strategy: The minimum derivation strategy aims to minimize the number of derivations to be applied on the architectural space 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 Fig. 4-a. $d(c_1)$ triggers a change on the specification s_1 and a change on a_{12}. The active nodes are then $(s.v_1, c_1.v_1, a_{12}.v_1)$.

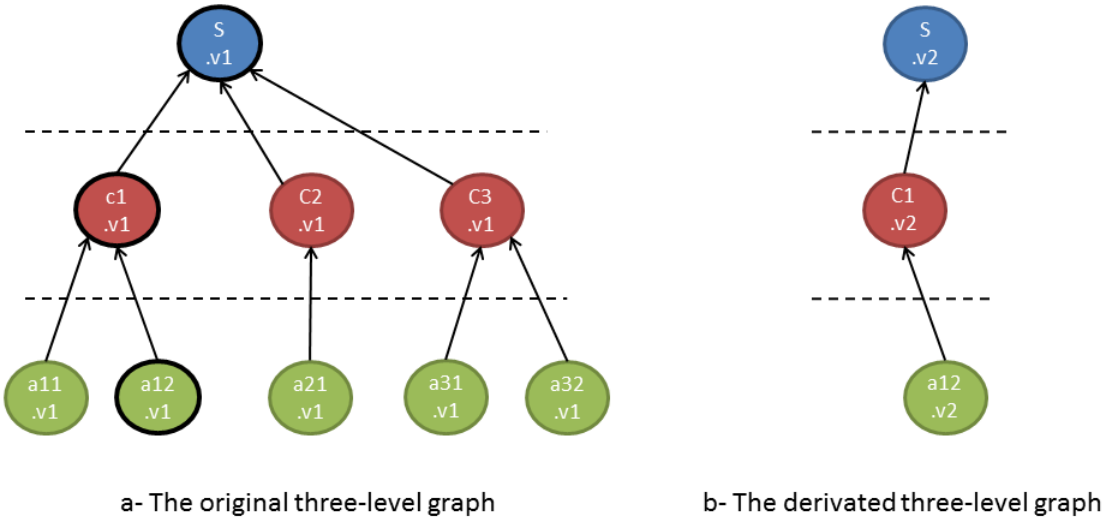

Figure 4. Example of minimum derivation

The minimum derivation strategy creates a new three-level graph with the new versions $(s.v_2, c_1.v_2, a_{12}.v_2)$ (*cf.* Fig. 4-b).

The minimum derivation strategy is suitable when the change purpose is not to correct some version of an architecture definition, resulting in the derivation of a revision, but to create a variant that can coexist with previous versions. Fig. 4 illustrates a special case where the derived architecture definition shares finally no element with its source architecture definition. These architectures definitions could be identified as variants belonging to a software product line [14]. This is a perspective of this work.

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

- 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 definitions are kept and which ones are replaced by new versions. Custom derivation strategy is used after a default application of the minimum derivation strategy so that minimum necessary versions, that ensure coherent global architecture definitions, are always created.

IV. Related work

Software versioning has been studied for many years with the the objective to provide a Software Configuration Management (SCM) systems [3], handling various kinds of entities and different granularities (source code lines, objects, libraries, etc.). Early work targeted mainly source code versioning. Several collaborative source code versioning systems were more recently proposed and have become industrial standards such as SVN [15], CVS [16] and Git [17].

To overcome the limitation of version management based on source code, [18] propose to generate from meta-models version control systems that are aware the corresponding modeling language concepts, in order to trace the evolution of significant logical units.

With the emergence of component-based software development, more recent work addressed component versioning rather than source code [2]. Examples include JAVA [19], and COM .Net. More recent approaches treated as well the issue of component substitutability like the work of Brada *et al.* (SOFA) [20] and the issue of compatibility like the work of Stuckenholz *et al.* [21].

Regarding architectural versioning, only little work was dedicated. The SOFA 2.0 ADL [5] enables to version composite components and therefore entire architectures (which are used to define composite component implementations). Other existing ADLs like MAE [6] and xADL 2.0 [7] also enable architecture versioning. However, all these architectural versioning models neither handle detailed information about evolution (rationale, change operations list that result in new architecture versions) nor maintain the trace of architecture evolutions throughout the whole software lifecycle (what requires to handle the co-evolution of multiple definitions for every architecture).

Another closely related work addressed architectural versioning at multiple abstraction levels [22]. The proposed approach is based on the SAEV model [23] 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

 ISBN: 978-1-61208-498-5

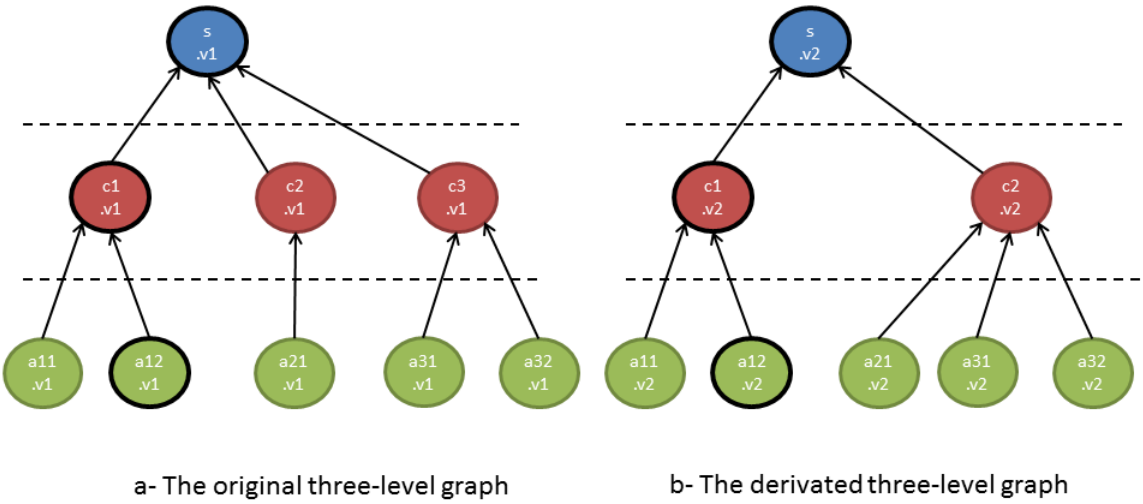

Figure 5. Example of full derivation

encompasses the definition of architectural concepts to be used at the lower level.

V. Conclusion and Future Work

This work proposes a version model for our software architecture description language Dedal. It considers version management at three abstraction levels in order to support the co-evolution of architecture definitions throughout the whole software lifecycle. It captures information about evolution (change operation list) and enables to distinguish its rationale (revisions and variants). Moreover, versioning strategies are proposed to automate the version derivation propagation that may or must result from the co-evolution of the different definition levels of architectures.

Future work consists in studying component versioning and its impact on architectural versioning considering compatibility issues, to detect automatically revisions and variants. Another perspective is to extend our version model in order to support product line engineering. From a practical perspective, ongoing work is to automate further versioning mechanisms and integrate them into DedalStudio, our eclipse-based tool that automatically manages the architecture evolution process [11].

References

[1] J. Estubier et al., "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] R. Taylor, N. Medvidovic, and E. Dashofy, Software architecture: Foundations, Theory, and Practice. Wiley, 2009.

[14] K. Pohl, G. Bckle, and F. van der Linden, Software Product Line Engineering. Springer, 2005.

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

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

[17] J. Loeliger and M. Matthew, Version Control with Git. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2012.

[18] T. Oda and M. Saeki, "Generative technique of version control systems for software diagrams," in Proceedings of the 21st IEEE International Conference on Software Maintenance, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 515–524. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2005.49

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

[20] 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

[21] 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

[22] 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.

[23] 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

 ISBN: 978-1-61208-498-5