Preliminary study on predicting version propagation in three-level component-based architectures
Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

To cite this version:
Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, Sylvain Vauttier. Preliminary study on predicting version propagation in three-level component-based architectures. SAT-ToSE: Seminar on Advanced Techniques and Tools for Software Evolution, Universidad Rey Juan Carlos, Spain, Jun 2017, Madrid, Spain. 5 p. lirmm-01580899

HAL Id: lirmm-01580899
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01580899
Submitted on 3 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Preliminary study on predicting version propagation in three-level component-based architectures

Alexandre Le Borgne¹, David Delahaye², Marianne Huchard², Christelle Urtado¹, and Sylvain Vauttier¹

¹LGI2P / Ecole des Mines d’Alès, Nîmes, France {Alexandre.Le-Borgne, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr
²LIRMM / 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
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 [Abl96] 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 history 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 a 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 [DVHT05]. 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.

\[ T_1 \prec T_2: T_1 \text{ is a subtype of } T_2. \]
\[ T_1 \preceq T_2: T_1 \text{ is a subtype of } T_2 \text{ or equal to } T_2. \]
\[ T_1 \succ T_2: T_1 \text{ is a supertype of } T_2. \]
\[ T_1 \succeq T_2: T_1 \text{ is a supertype of } T_2 \text{ or equal to } T_2. \]
\[ T_1 \parallel T_2: T_1 \text{ is not comparable to } T_2. \]
\[ (T_1 \nleq T_2) \iff ((T_1 \prec T_2) \lor (T_1 \parallel T_2)): \]
\[ T_1 \text{ is either a subtype of } T_2 \text{ or not comparable to } T_2. \]
\[ (T_1 \ngeq T_2) \iff ((T_1 \succ T_2) \lor (T_1 \parallel T_2)): \]
\[ T_1 \text{ is either a supertype of } T_2 \text{ or not comparable to } T_2. \]
\[ T_2 \implies T_1: T_2 \text{ replaces } T_1. \]

Substitutable functionalities [LBDH+17].

For a provided \( f_{new}^{prov} \) functionality, being substitutable for a \( f_{old}^{prov} \) functionality means that: (1) \( f_{new}^{prov} \) and \( f_{old}^{prov} \) have the same name, (2) the return type of \( f_{new}^{prov} \) is equal to or a subtype of \( f_{old}^{prov} \) and (3) the input parameters of \( f_{old}^{prov} \) are supertypes of the ones of \( f_{new}^{prov} \) [ADH+07].

Conversely, for a required \( f_{new}^{req} \) functionality, being substitutable for a \( f_{old}^{req} \) functionality means that: (1) \( f_{new}^{req} \) and \( f_{old}^{req} \) have the same name, (2) the return type of \( f_{new}^{req} \) is equal to or a supertype of the return type of \( f_{old}^{req} \) and (3) the input parameters of \( f_{old}^{req} \) are subtypes of the ones of \( f_{new}^{req} \).

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 mechanisms. 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 \( I_2 \)).

\[
\begin{align*}
R_1 & : f() : A \\
C_1 & : f() : X \\
I_1 & : f() : B \\
R_2 & : f() : Z \\
C_2 & : f() : \Omega \\
I_2 & : f() : R
\end{align*}
\]

Figure 1: Connecting functionalities within a three-level component-based architecture
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 R1 by its version R1′ at specification level or C1 by its version C1′ or I1 by I1′ 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 ≤ Y ≤ Z, Y may either be substitutable to A or not.
  
  (b) At configuration level, the non-propagation condition is summarized by B ≤ Y ≤ A and Y can either be substitutable or not to X. Moreover, the realization relation between R1 and C1′ is preserved if (Y ≤ A) and the instantiability of C1′ by I1 is ensured by (Y ≥ B).
  
  (c) At assembly level, the version does not need to be propagated if (Y ≤ 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 (†) when Y is not a subtype of A or to a propagation to the assembly level (↓) when Y is not a supertype of B. (ii) Intra-level propagation occurs 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 R2, C2, and I2 are replaced by their respective version R2′, C2′ and I2′ 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

---

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

<table>
<thead>
<tr>
<th>Propagation</th>
<th>Inter-level</th>
<th>Intra-level</th>
<th>Inter-level</th>
<th>Intra-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Y ⊑ X) ∨ (Y ⊑ X)</td>
<td>(Y ⊑ Z) ∨ (Y ⊑ Z)</td>
<td>[(Y ⊑ A) ⊑ ↑] ∨ (Y ⊑ B)</td>
<td>Y ⊑ Ω</td>
<td>Y ⊑ X</td>
</tr>
<tr>
<td>(Y ⊑ X) ∧ (Y ⊑ Z)</td>
<td>(Y ⊑ X) ∨ (Y ⊑ B) ∧ (Y ⊑ Ω)</td>
<td>Y ⊑ R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2: Multiple component seen as one composite component for 1 to n replacement**
Hypothesis on types

\[ B \preceq X \preceq A \preceq Z \preceq \Omega \preceq R \]

<table>
<thead>
<tr>
<th>Specification level</th>
<th>Configuration level</th>
<th>Assembly level</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y \rightarrow Z )</td>
<td>( Y \rightarrow \Omega )</td>
<td>( Y \rightarrow R )</td>
</tr>
</tbody>
</table>

Non-propagation

\[ A \preceq Y \preceq \Omega \preceq Z \preceq Y \preceq R \]

Propagation

\[
\begin{array}{cccccc}
| & Inter-level & Intra-level & Inter-level & Intra-level & Inter-level & Intra-level |
|---|---|---|---|---|---|---|
| \( Y \not\in \Omega \) & \( Y \not\in A \) & \( (Y \not\in Z \Rightarrow \top) \lor (Y \not\in R \Rightarrow \bot) \) & \( Y \not\in X \) & \( Y \not\in \Omega \) & \( Y \not\in B \) & \( (Y \not\in \Omega) \land (Y \not\in B) \) |
\end{array}
\]

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 substitutability 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


