Metadata Support for Safe Component Upgrades [*]

Přemysl Brada
University of West Bohemia in Pilsen
Czech Republic
brada@kiv.zcu.cz

Abstract

Component platforms play a major role in current distributed information systems. As these systems evolve, components need to be upgraded without breaking the consistency of component interconnections. In this paper we present a method for the support of automated safe upgrades of black-box components. It relates component versioning with indication of changes between revisions of the same component and uses pre-computed information stored in component metadata. We show how this information is derived from component specifications and used to speed up pre-upgrade checks.

1. Introduction

Component platforms like ACME [2], SOFA [12], Enterprise JavaBeans [1], or CORBA components [9] play a major role in the research, development and operation of current distributed industrial information systems. As these systems evolve, their parts need to be replaced or upgraded safely – i.e. in a compatibility-preserving way.

In this paper we present a method supporting automated safe upgrades of black-box components [15] which uses pre-computed information stored in component metadata to speed up the necessary compatibility checks. It links component versioning with indication of changes that affect compatibility between revisions of the same component. This link is motivated by the observation that in most cases, the upgrade entails replacing a current (old) version by a replacement (new) one.

The structure of the paper is as follows. First we mention the related research in the area and give a brief overview of the ENT model used in our work. Section 3 presents the key contribution of this paper – the component change and revision data based on the ENT model and their use in compatibility checks. Then we shortly describe the implementation of this approach, and the paper ends with a conclusion and statement of future work.

1.1. Related Research

Analyses of software changes based on its specification are substantial for our work. Although this area is quite well researched [11, 16] the known approaches mostly work with software structured at a granularity of method signatures which is too fine for coarse-grained components.

There exist many industrial approaches to safe upgrades. However, they are either tightly bound to the given language or environment [7], or use only manually provided compatibility data [6]. Even the latest developments in the area [10] do not deal explicitly with this issue. The research goes in two directions. Interface adaptation [14] is flexible but there are situations which preclude its use, namely unattended upgrades. Type-safe replacement [4] is easier to automate but may be overly restrictive due to its use of contravariant subtyping.

On the side of versioning research, there have been works which suggest the requirements on component versioning support [8] and which include the possibility to reflect architecture changes in the version identification [5]. What we are missing in these approaches are hints on practical implementation.

2. The ENT Interface Model

In this section we briefly describe the ENT model of component interface upon which our work is based (see [3] for details). In this model, the interface specification of a component is at the lowest level dissected into the declarations of individual *elements* – features (attributes, event sources/sinks, etc.) and semantic properties (run-time protocol, invariant, etc.).

Each element is then assigned a *metatype* and classified using a simple faceted system with dimensions that reflect various usage-related characteristics: *contents, role, kind,*

[*]This work is partly supported by the Research Plan number MSM235200005 funded by the Ministry of Education of the Czech Republic.

lifecycle. In particular, the *role* dimension with the classifier terms {*provided,required*} is important as the provided and required elements play different roles in component interconnections. They consequently need different treatment during the pre-upgrade compatibility checks.

provisions	*metatype=interface* ∧ *role=provided*
dependencies	*metatype=interface* ∧ *role=required*
properties	*metatype=property* ∧ *role=required*
protocol	*metatype=protocol* ∧ *role* ∈ {*provided,required*}

Figure 1. Trait definitions for SOFA components (abbreviated)

Element classification lets us split the interface into sets (called *traits*) of element declarations which have the same metatype and classification values (see Figure 1). Traits have the notable characteristic that they group interface elements in a manner and on a granularity suitable to the needs of human users and developers (cf. the common request to "see which new interfaces this version provides").

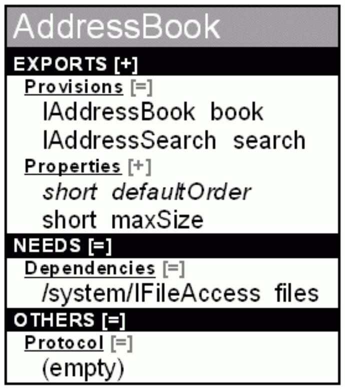

Figure 2. SOFA component in the ENT model

To enable coarse-grained views of the component useful for first-cut analyses, we use the notion of *categories* which are sets of traits that share some classification values. A prominent set of categories called *E-N-T* (*E*xports, *N*eeds, o*T*her; see Figure 2) is used to formalize the split of the component interface along the provides-requires line.

3. Upgrade Support: Versioning and Compatibility Related

To achieve successful component upgrades, we need to compare the specifications of the replacement component version C^r and the current one C^c with respect to compatibility. We use the fact that their differences show up as changes in the corresponding interface traits.

These changes can be of four types: *none*, *specialization*, *generalization*, and *mutation* (specialization and generalization mean extending and reducing the interface trait, respectively; mutation means a mix of changes). For example, the *properties* trait in Figure 2 was specialized by adding the `defaultOrder` property.

Algorithms appropriate for the elements contained in each trait (based on element metatype) must be used in determining these changes – subtype checks for elements which are declared as typed variables, boolean expression comparison for pre- and post-condition elements, etc.

This method creates *change data* at the trait level. Since the provides/requires role of elements fundamentally affects compatibility, for basic checks it is useful to aggregate such data using the *E-N-T* categories. This is done as follows: the change of category K is $none$ if no trait in the K category has changed, $spec$ (gen) if the differences in K's traits are of the $specialization$ ($generalization$) or $none$ type, and mut if some of K's traits have mutated, or some became specialized while others generalized, or both.

3.1. From Changes to Compatibility

The change data obtained by the approach described above are suitable for compatibility checking in which we distinguish two cases.

Strict compatibility requires that the replacement component exhibits "contravariant" changes in its interface. This ensures upgrade in any context. In terms of category change data, the replacement component $C^r = (E^r, N^r, T^r)$ is strictly compatible with the current one $C^c = (E^c, N^c, T^c)$ iff $change(E^c, E^r) \in \{none, spec\} \wedge change(N^c, N^r) \in \{none, gen\} \wedge change(T^c, T^r) \in \{none, gen\}$.

In concrete situations however, upgrade may be possible even in the presence of covariant or type-incompatible changes – for example when the C^r specification omits some provided elements of C^c that are not being used in its deployment context.

The *contextual compatibility* is defined to capture this case. It uses an ENT representation of the deployment context $Cx = (E'^c, \bar{N}^c, T^c/E'^c)$ where E'^c are traits of provided elements that are actually bound to (used by) other components, $\bar{N}^c$ are traits of elements that other components provide and that can satisfy the requirements of C^r, and T^c/E'^c are the semantic declarations related to the bound provided features. The context categories are used in the compatibility definition in place of E^c, N^r, and T^c.

3.2. From Changes to Revision IDs

In software configuration management, the revision numbers of a software item express the time order of its revisions and implicitly their ability to be backward compatible. We now formalize this intuitive usage via ENT-based structure and semantics of revision numbers.

Specification traits represent the smallest interface structures to be meaningfully versioned separately (individual elements are too small for this). On the first release of a component, each trait is therefore assigned revision number $revnum(t_i) = 1$. The number is incremented on subsequent releases if there is a change in the trait.

However, the number of component traits (e.g. 4 for SOFA, 8 for CORBA components) is usually such that trait data would hardly lead to simple version identifiers. We therefore use the *E-N-T* category set to create a revision numbering scheme that is concise enough for practical purposes.

This *component revision data* is a triple (r_E, r_N, r_T) where $r_\alpha \in \mathbb{N}$. Its parts are derived using the category change data: for category K, if $change(K^c, K^r) \neq none$ then $r_K^r = r_K^c + 1$. The component revision ID is a "$r_E.r_N.r_T$" string form of this data.

These component revision identifiers have the interesting property that their differences indicate in which aspects the component revisions vary. This is very useful for quick visual checks as well as in avoiding compatibility checking for categories which have not changed. They of course also express the time ordering of component revisions.

As this section shows, the ENT model provides a convenient vehicle for a formalization of the intuitive link between revision identification and compatibility indication of software components. The following section describes briefly our implementation of this approach.

4. Implementation in Component Metadata

In many software deployment systems the application packages contain metadata which describes their purpose, version, dependencies and compatibility information. In a similar way we propose to include the change and revision data with each component. This should free the target environment of the computationally intensive task of specification comparison that may introduce long delays with semantic specifications like protocols [13] or CSP [2].

Additionally, our metadata comprises the change and revision data for the whole revision history of the component (see Figure 3). This makes it easier to perform compatibility checks even when several intermediate revisions were not installed.

```
<compdata system="sofa">
 <provider>cz.zcu.kiv</provider>
 <name>OfficeApps/AddressBook</name>
 <revision>
  <parent>3.1.1</parent>
  <data level="component">
   <trait name="E"> <revnum>4</revnum>
     <change>spec</change>  </trait>
   <trait name="N"> <revnum>2</revnum>
     <change>spec</change>  </trait>
   <trait name="T"> <revnum>1</revnum>
     <change>none</change>  </trait>
  </data>
 </revision>
 <history>
 <!-- shortened for brevity -->
 <revision seq="2">
   <parent>2.1.1</parent>
   <data level="component">
    <trait name="E"><revnum>3</revnum>
      <change>spec</change> </trait>
    ...
   </data>
 </revision>
 </history>
</compdata>
```

Figure 3. Component metadata example

The implementation uses a XML representation of the data. When the new revision is downloaded for upgrade, several actions can be done. First, the revision IDs of C^c and C^r might be compared to see which parts need to be checked for compatibility. Then, the change data parts of the component metadata would be compared for strict compatibility. If they pass this check the upgrade is allowed.

Otherwise, the system may try to determine the context of C^c and check for contextual compatibility. At any stage, the interface specification can still be used for direct "manual" checking. (This may be useful if intermediate revisions were skipped – the change data based on pair-to-pair comparison may not give enough information in these cases).

5. Conclusion and Future Work

In this paper we have shown how safe component upgrade can be supported by appropriate metadata. The key advantage of the approach lies in the possibility to create structured revision identification and compatibility information and provide a well defined relation between them.

As this information can be determined once upon component release and stored in component metadata, its use can reduce the computational complexity of compatibility checks done prior to the upgrade. We consider this approach to be suitable for unattended upgrades in many component-based systems, among others CORBA [9] and EJB [1].

There are however some aspects that need further work. First, practical implementations of the approach require suitable parsers and interface element comparison algorithms which further increase system complexity. We will therefore investigate the possibilities for model support and automation in this area.

Second, the relation to assembly comparison [10] needs to be considered. Lastly, the approach does not support any means of interface adaptation which may be impractical in some cases. To this end, the ENT interface model and compatibility definitions will probably need to be redesigned.

The presented work is still in progress. The tools for generating the ENT-based representation of components and their metadata are partly implemented for the SOFA component framework. A working support of pre-upgrade checks as well as a CORBA implementation are a near future goal.

References

[1] Enterprise JavaBeans(TM) Specification, version 2.0. Technical report, Sun Microsystems Inc., August 2001.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[3] P. Brada. The ENT model: A general model for software interface structuring. Technical Report DCSE/TR-2002-03, Department of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic, 2002.

[4] M. Buchi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Department of Computer Science, Åbo Akademi University, Turku, Finland, 2000.

[5] H. Christensen. Experiences with architectural software configuration management in Ragnarok. In *Proceedings of SCM-8 Workshop, ECOOP 1998*. Springer-Verlag, 1998.

[6] Desktop Management Task Force. *Desktop Management Interface Specification, version 2.0*, 1998.

[7] J. Gosling, B. Joy, and G. Steele. *Java Language Specification*. Sun Microsystems, 1996. Chapter 13 (Java Binary Compatibility).

[8] M. Larsson and I. Crnkovic. New challenges for configuration management. In *Proceedings of the SCM-9 workshop, ECOOP 1999*, LNCS 1675, Toulouse, France, Sep 1999.

[9] P. Merle. CORBA 3.0 new components chapters. Technical Report ptc/2001-11-03, Object Management Group, November 2001.

[10] Object Management Group. Deployment and configuration of component-based distributed applications. request for proposals. Technical Report orbos/2002-01-19, OMG, January 2002.

[11] D. Perry. Version control in the Inscape environment. In *Proceedings of ICSE'87*, Monterey, CA, 1987.

[12] F. Plášil, D. Bálek, and R. Janeček. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

[13] F. Plášil, M. Bešta, and S. Višňovský. Bounding component behavior via protocols. In *In Proceedings of TOOLS USA '99*, Santa Barbara, USA, 1999.

[14] H. W. Schmidt and R. H. Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia, 2000.

[15] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1998.

[16] A. M. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), Oct. 1997.