Specification-Based Component Substitutability and Revision Identification
PhD Thesis Abstract

Přemysl Brada
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic
TBD skolitel, oponenti

August 2003

Contents

1. Introduction

After years of research and industrial development, software component technology [McI68, Szy98, OMG02a] has become well established as an important approach to engineering complex and flexible software systems.

Yet the functionality provided by current component frameworks is limited to the basic wiring of components (setting up their interconnections) and supporting key system-level aspects. They lack easy component composition [LvdH02], high level of re-use, and seamless evolution [Ore98] and version management support.

The thesis shows how current component technology can be enhanced to answer some of these concerns, by presenting a novel approach to meta-modelling, version management and controlled substitution of software components. In this abstract we provide an overview of the open issues which motivate our work, present its goals, describe the achieved results and list the contributions made to the component research field.

1.1 Motivation for the Work

The motivation for our work is given by the inadequacies in component modelling, versioning and substitutability checking.

Component Models and Meta-Models A software component is a coarse grained black-box software element with contractually specified interface syntax and semantics [Szy98, MT00], and a component model defines "a standard to which a set of components must adhere in order to be composable into applications" [LvdH02]. In many component frameworks, it is defined implicitly or informally (for instance in UniCon [S+95], SOFA [PBJ98], or Enterprise JavaBeans [Sun01]).

At a more abstract level, a *meta-model* captures the common aspects of a set of models in the above meaning. Good meta-models are important because they define the standard level of practice and technology in their subject areas – the terminology, structural and semantic features, element relationships, modelling possibilities etc. The UML Enterprise Distribut-

ed Object Computing (EDOC) Profile [OMG02b] is a key industrial component meta-model which maps well to current component frameworks. In the research area, several meta-models with interesting features have emerged (e.g. by Seyler and Anoirte [SA02], the Fractal framework [C+02], or Rastofer [Ras02]).

The problem is that these meta-models are mostly straightforward abstractions of the present technology and offer few forward-thinking ideas to handle future developments — configuration management issues (versioning, compatibility as a key to configuration consistency), aspects (distribution, location transparency, concurrency, persistence), or the quality of service properties. Thus they will need to be modified to accommodate upcoming developments (mobility, emphasis on quality of service) which will have negative effects on the stability of the technology at large.

Component Versioning Any software component, as a software artefact, inevitably evolves and changes. Thus several versions of one component are created during its active life. Compared to versioning used during software development [CW98], component versioning faces several distinctive challenges:

- Component versioning is applied to software elements which are treated as black boxes, thus version information needs to be available separately or via standardised introspection interfaces [LC99].
- Different component versions should be easy to select and assemble, mainly in a (semi-)automated manner. They may be used in multiple configurations, some of them unforeseen at the time of its creation.
- Component providers cannot govern the use of components after release to market. Therefore, version information need to be understandable, precise, and preferably standardised.

Unfortunately, together with other researchers [BW98, Szy98, SV01] we find very little support for these component version management functions in current industrial as well as research component systems. Even in systems with this support, version identification provides at most a tag to distinguish versions (this is the case of e.g. CORBA, Java product versioning, as well as some software packaging tools [OMG02a, Rig02, J+03]). They mostly use a two- to four-number revision identification scheme, which we call here the "$M.m.\mu$" (major, minor, micro) scheme, with fuzzy definition of its parts. Major enhancements (visible to clients) are mostly indicated by changing the major version number and/or giving the software a different marketing name, internal enhancements and bug fixes usually lead to some modifications of minor and/or micro revision number(s).

The situation may well become serious in the near future if components are used at a large scale and for a longer period of time. Suitable approaches and tools are therefore required to handle the proliferation of versions of successful components.

Substitutability and Compatibility Substitution and in particular upgrade of components is a vital mechanism for maintaining installed applications up-to-date. The key requirement is that the substitution must not introduce new problems [WZ88], but rather fix the old ones or enhance the application.

Currently there are two main classes of solutions dealing with this issue in similar systems. In the first one, meta-data is provided with each application package or component [J+03, Des98, LC00, vdHW02] with its version identification, information about compatibility with previous versions and the components depended upon. The problem is that in practice, the meta-data is usually created manually based on the developer's knowledge of the implementation. This is an error-prone process and leads to data that is either insufficiently informative or time consuming to create.

Several research systems use on the other hand various forms of subtyping relation [Car97, LW94] to check substitutability, determined by comparing (semi-)formal component descriptions [ZW97, VHT00, Nie93]. The negative aspect of these systems is that the algorithms used in determining the subtyping relation may have high computational complexity [PV02], resulting in potential delays. Also, some research systems define relaxed compatibility levels to increase the chances on substitution but these relaxations are designed for a different purpose (e.g. searching [ZW97]) and their application would reduce the reliablility of the substitution.

1.2 Goals of the Thesis

Motivated by these findings, our work described in the thesis pursues these primary goals:

Meta-model for Components Develop an open derived component meta-model that would serve as a common denominator in understanding component specifications and would make it possible to define at least some of the "penetrating" advanced technological features (flexible visual representation, substitutability, version identification) on the the meta-level.

Component versioning Design a scheme for component versioning suitable for automated processing and supporting component distribution and retrieval, while providing all of the traditional functions.

Component substitutability Define a notion of substitutability suitable for black-box components, and devise a method for checking whether a prospective component substitution will not break configuration consistency. Compatibility shall be considered as a special case of substitutability.

Link between versioning and compatibility In software configuration management, there is a close link between versioning and configuration consistency. This work should give an answer how this link can be established in the case of black-box components.

1.2.1 Addressing the Goals: The Constraints

Solutions relevant to practice need to offer end-user simplicity, reliability and standardisation. In the work towards the primary goals we therefore need to carefully choose suitable approaches and methods. The following constraints formulate the guidelines for their selection or design that our work should follow.

Use existing data We should (re)use already existing data including source code as much as possible; in particular, we should try not to introduce new human-entered data in our methods.

Use automated methods There should be as much automation, and as little additional human effort involved in developing and using components as possible. We should strive for automated derivation of information and automated reasoning based on such data.

Strive for simplicity and readability Aim at creating methods and systems that are simple, produce or require data that can be read and written by humans, and that fit well within current frameworks and tools. We believe people and their knowledge should take precedence over algorithms even in sophisticated systems.

1.3 List of Published Articles

Reviewed Articles

[Bra99] P. Brada. **Component Change and Version Identification in SOFA.** In *Proceedings of SOFSEM'99*, LNCS 1725, Springer-Verlag 1999. SOFSEM'99, Milovy, Czech Republic.

[BR00] P. Brada, J. Rovner. **Methods of SOFA Component Behavior Description.** In *Proceedings of ISM'2000*. Information Systems Modeling, Rožnov, Czech Republic.

[Bra01b] P. Brada. **Towards automated component compatibility assessment.** Position paper. *WCOP'2001 — Workshop on Component-Oriented Programming*. ECOOP 2001, Budapest, Hungary. Available at http://research.microsoft.com/~cszypers/events/WCOP2001/ .

[Bra01a] P. Brada. **Component Revision Identification Based on IDL/ADL Component Specification.** Poster. In *Proceedings of ESEC/FSE'01, European Conference on Software Engineering*. IEEE Computer Society Press 2001. Vienna, Austria.

[Bra02b] P. Brada. **Metadata Support for Safe Component Upgrades.** In *Proceedings of the 26th Computer Software and Applications Conference (COMPSAC'2002)*. Oxford, England. IEEE Computer Society Press, August 2002.

Unreviewed Articles

[ABV00] S.-A. Andréasson, P. Brada, J. Valdman. **Component-Based Software Decomposition of Flexible Manufacturing Systems.** In *Proceedings of International Carpathian Control Conference*. Podbanské, Slovak Republic, 2000.

[Bra00] P. Brada. **SOFA Component Revision Identification.** Technical report No. 2000/9, Department of Software Engineering, Charles University, Prague 2000.

[Bra02a] P. Brada. **The ENT model: A general model for software interface structuring.** Technical Report DCSE/TR-2002-10, Department of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic. 2002.

[Bra02c] P. Brada. **Parametrized Visual Representation of Software Components.** In *Proceedings of the 7th Objekty conference*, Prague, Czech Republic. November, 2002.

2. Contributions of the Thesis

2.1 The ENT Component Meta-model

Despite the differences between different component models, they share many similarities — separation of interface and implementation, declaration of exported and imported elements, etc. We conducted a study of several frameworks — SOFA [PBJ98], CORBA Component Model (CCM) [OMG02a], UniCon [S+95], Han's model [Han98], and Fractal [C+02] plus a survey of comparative studies by Shaw [S+95], and Medvidovic and Taylor [MT00]. It shows that current meta-models which describe this common denominator emphasise the technical aspect of components but provide little help in specifying and analysing high-level component properties.

We therefore designed the ENT meta-model which captures the common component characteristics from the user's point of view, using approaches that people use when they reason about component interfaces. The name of our meta-model technically comes from the abbreviation of a key set of structures — Exports-Needs-Ties — it defines; for brevity, it is referenced as "the ENT model" below.

In ENT, the specification of a given component is seen as a set of elements (functional *features* like IDL interfaces, and non-functional quality attributes like SOFA protocols) which define its capabilities. The characteristics that we as humans are interested in when observing the component specification are easy to formulate. We formalise them by a faceted classification system which uses seven facets called *dimensions*; the term space of each facet consists of pre-defined identifiers[1].

Definition 2.1.1 (ENT classification) *The* ENT classification system *is a system for faceted classification of component specification elements which uses an ontology* $Dimensions_{ENT} = \{Nature, Kind, Role, Construct, Presence, Arity, Lifecycle\}$ *where the dimensions (facets) are*

- $Nature = \{feature, quality\} \cup Id^{spec}$ *is a basic dimension used to describe the primary meaning of an element,*

[1] We use the set $Id^{spec} = \{nil, na, nk, all\} \subset Identifiers$ to handle special cases.

- $Kind = \{operational, data\} \cup Id^{spec}$ *is a dimension describing the nature of an element with respect to computational characteristics,*
- $Role = \{provided, required, neutral\} \cup Id^{spec}$ *describes the "orientation" of an element in component interactions and type relations,*
- $Construct = \{constant, instance, type\} \cup Id^{spec}$ *describes how an element is to be interpreted in terms of the specification language syntax,*
- $Presence = \{mandatory, optional\} \cup Id^{spec}$ *denotes whether the component interface must contain an element at run-time.*
- $Arity = \{single, multiple\} \cup Id^{spec}$ *denotes how many connections an element can accept/provide,*
- $Lifecycle = \{development, assembly, deployment, runtime\} \cup Id^{spec}$ *is a dimension describing the possible phases in component's lifecycle in which an element can be meaningfully accessed or used.*

The ENT classifier *is an ordered tuple* $(nature, kind, role, construct, presence, arity, lifecycle) = (d_1, d_2, \ldots, d_D)$ *such that* $d_i \subseteq dim_i$, *and* $dim_i \in Dimensions_{ENT}$.

■

2.1.1 Constituents of the Meta-model

The structures which form the ENT model as such start at the lowest level of specification elements. Its key structures however are aggregate constructs — traits and categories — which cluster elements according to their human-based classification.

Definition 2.1.2 (Specification element) *A* specification element *e found in the specification of a component M written in language L is a tuple* $e = (name, type, tags, inh, metatype, classifier)$. ■

A specification element represents a complete information about one feature or of one component-wide quality attribute. The element is identified by its $name$ and $type$; however, in some cases the $name$ may be empty.

The $tags$ item contains a set of phrases with additional declarations pertaining to the particular element (not to its type). For example, the designation of an element as *readonly* or *readwrite* is captured in the tag $access$. The inh item contains, as an ordered n-tuple of identifiers, the fully qualified type name of a component from which the element is inherited. The name, type, tags and inheritance indication of an element can be obtained directly by analysing the specification source code.

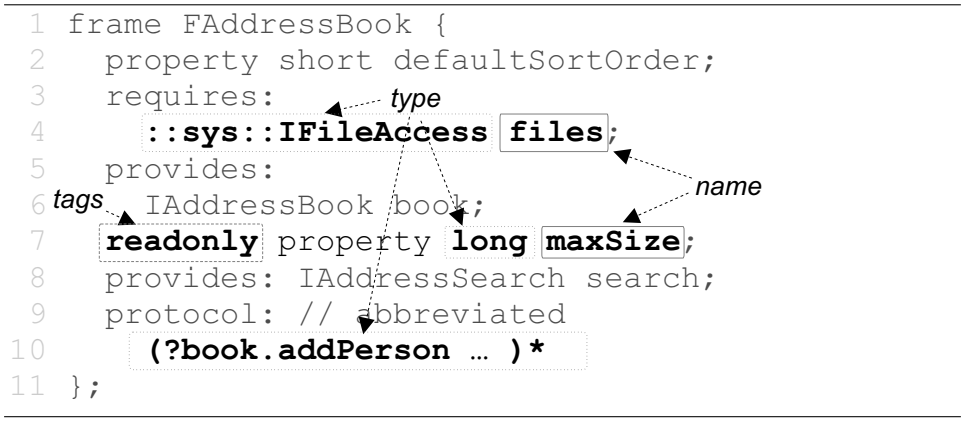

Figure 2.1: Example elements in component specification

The *metatype* element is a name describing the general type of feature or quality, such as "interface" or "event". It is often related to or derived from the name of the corresponding non-terminal symbol in the grammar of L. The *classifier* contains the classification of the element according to the ENT classification system. The information about meta-type and classification of an element has to be based on an manual analysis of the specification language L and the human-perceived meaning of its phrases.

Definition 2.1.3 (Trait) *Let* $C^T = (ct_1, ct_2, ..., ct_D)$ *be an* ENT *classifier. A* specification trait *(or just* trait *in short) of a component* M *is a tuple* $t = (name, metatype, C^T, E)$ *where* $E \subseteq Elements(M)$ *is a set of specification elements such that* $\forall e_i \in E : metatype = e_i.metatype \wedge C^T = e_i.classifier$. *(*$C^T$ *is called* trait classifier.*)* ■

The notion of trait is a key idea of our meta-model — it is a named set of specification elements which are equal in terms of their classification and metatype, i.e. have the same meaning from user's point of view. (This differs from language types which group elements with the same structure.) The consequence is that in ENT , a given concrete component model is defined by the (fixed) set of its traits.

Definition 2.1.4 (Category) *Let* $f^K : (d_1, \ldots, d_D) \rightarrow Boolean$ *be a boolean function on* ENT *classifiers, called the* selection function.

A specification trait category *(shortly* category*) of a component* M *is a tuple* $K = (name, f^K, T)$ *in which* $T \subseteq Traits(M)$ *such that* $\forall t \in T : f^K(t.C^T) = true$. *A* category set *is a set of categories* $\{K_1, K_2, \ldots, K_n\}$ *such that* $\forall t_1 \in K_i.T$, $t_2 \in K_j.T : t_1 \neq t_2$. *The expression* $S = \{K_1, K_2, \ldots, K_n\}$ *means a component specification* S *structured into* n *categories.* ■

Categories group traits which are similar in some high-level aspect(s), expressed in our model by sharing the values in some of their classification dimensions as specified by the category's selection function f^K. For a defined category set, each trait from the component trait set belongs to at most one category.

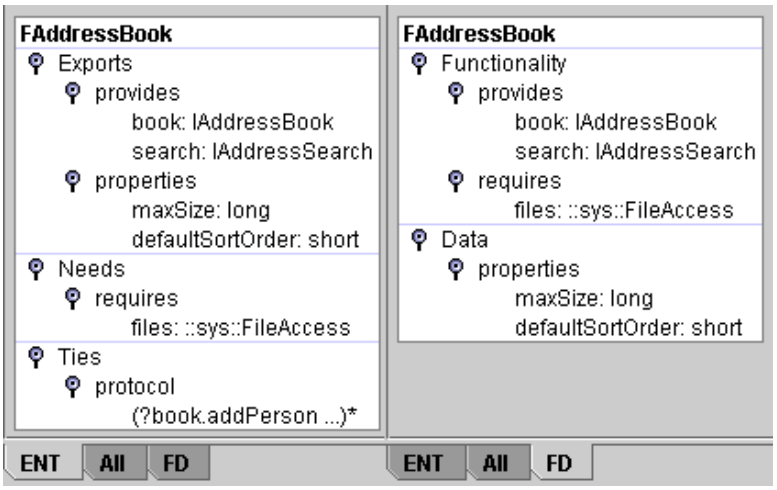

Figure 2.2: Example categories in the `FAddressBook` SOFA component

For a given specification language, we can define an arbitrary number of different category sets. These sets, superimposed on a particular component specification expressed in traits, then give us completely different user-defined views of the component (see Figure 2.2).

A key category set most useful for our work, "Exports", "Needs" and "Ties" (or "ENT"), captures the view of the component interface which developers (and some languages as well) use very often — that of elements provided for others to use ($Role = \{provided\}$), of those required from the environment to ensure proper functionality ($Role = \{required\}$), and those which express the bindings of these two sets together ($Role = \{provided, required\}$).

Restricted Elements and Categories When there is a need to compare elements in two components, the sets of names contained in the components — and referenced in the elements — may differ. It is therefore necessary to compare only those parts of the element declaration that correspond to a relevant intersection of the sets of names.

We therefore define restricted element $e' = e/A$ as an element whose declaration uses only identifiers from the set A and $e <: e'$; category restriction K/A is then an aggregate of such elements. An example of re-

striction on elements is the protocol restriction operator defined for SOFA behaviour protocols in [PV02].

2.1.2 Applications

The primary application of the ENT model is the description of current, and design of new, components and component models. Its novel approach to meta-modelling allows the designers to reason about the desired usage properties of components, rather than restricting them to the low-level problems of component wiring.

The ENT-based visual representation of software modules and components (see Figure 2.2 on the preceding page) can be helpful in understanding of the software by presenting the interface at various levels of detail and from different viewpoints in user terms. In search and retrieval, the ENT model can assist by narrowing the search (e.g. full-text search in descriptions, signature matching, and so on) using the classifiers and other meta-data associated with elements, traits and categories.

Concerning platform applicability, we provide detailed ENT model trait definitions for the SOFA, CORBA (CCM) and JavaBeans component frameworks. The SOFA CDL for example has four traits of elements in component frame specification: “provides”, “requires”, “properties” and “protocol”. In a similar manner, other component frameworks with IDL-like specifications or modular languages can utilise the ENT model.

2.2 Component Substitutability and Compatibility

A frequent kind of software modification is component substitution and its special case, upgrade, that is the replacement of an out-dated version of a component by a more current one. It is natural to require that after such substitution, the whole application must function correctly and its behaviour must be consistent with that before the change — the new component must be *substitutable* for the old one.

In the thesis we present a formal underpinning of methods which test two components for substitutability a-priori using subtyping-based specification comparison. Our approach is applicable in scenarios where the component substitution is fully automated and provides flexibility over pure subtyping.

2.2.1 Comparing Specifications

The possibility to see and analyse differences between components is a basis for determining their substitutability. In our work we use a method of com-

paring the ENT data structures, which gives it the property that the results are easy to interpret for humans. This contrasts with some "implementation dependent" approaches to comparing specification, e.g. the "diff " tool or DCE interface change rules.

The comparison of component specifications starts at the level of elements and is expressed using a novel *subsumes* relation. We say that element e_i subsumes element e_j (denoted $e_i \succ e_j$) if $e_i.type <:_L e_j.type \wedge \forall u \in e_j.tags\ \exists t \in e_i.tags : t.name = u.name \wedge t.value <:_L u.value$.

For traits and categories, the definition of the subsumes relation uses the subsumption on their contents — name-equivalent trait's elements must subsume or be equal, and categorie's traits likewise.

Definition 2.2.1 (Component subsumption) *Assume two components, C_1 and C_2, and their specifications structured by the* ENT *category sets $\{K_{1,i}\} = \{E_1, N_1, T_1\}$ and $\{K_{2,j}\} = \{E_2, N_2, T_2\}$. Let $A = Names(C_1) \cap Names(C_2)$ where $Names(C)$ denote the set of all identifiers (the $e.name$ parts of elements) that occur in the specification of component C.*

We say that component C_1 subsumes component C_2 (denoted $C_1 \succ C_2$) if $E_1 \succeq E_2 \ \wedge\ N_2 \succeq N_1 \ \wedge\ T_1/A \succeq T_2/A$. ■

Element comparison deliberately pays no attention to the element's classification, in particular to its role (occurrence on the provided or required side of component interface). This is accounted for in the component comparison (see below) where it results in the application of covariant or contravariant rules to the *provided* and *required* elements, respectively.

For easy representation of comparison results we use a the classification set $\mathit{Differences} = \{init, none, specialization, generalization, mutation\}$. Its values are generated by a polymorphic specification matching function $\mathit{diff} : \_ \times \_ \rightarrow \mathit{Differences}$ and correspond to the following situations in $\mathit{diff}(\sigma_1, \sigma_2)$: $init$ if σ_1 is not defined, $none$ for equality, $specialisation$ for $\sigma_2 \succ \sigma_1$, $generalisation$ for the reverse, and $mutation$ for incomparability. This straightforward classification makes it easy to visualise the differences, as in Figure 2.3.

2.2.2 Substitutability of Components

Our component substitutability is based on the principle coined by [WZ88]: *a subtype (replacement) component should be usable whenever a supertype (the current one) was expected, without the client noticing it.* Considering *compatibility*, we view it as a special case of substitutability applied to subsequent revisions of a software component.

The principle of substitutability shows that this property does not concern just the two components in question. It tells us that we additionally

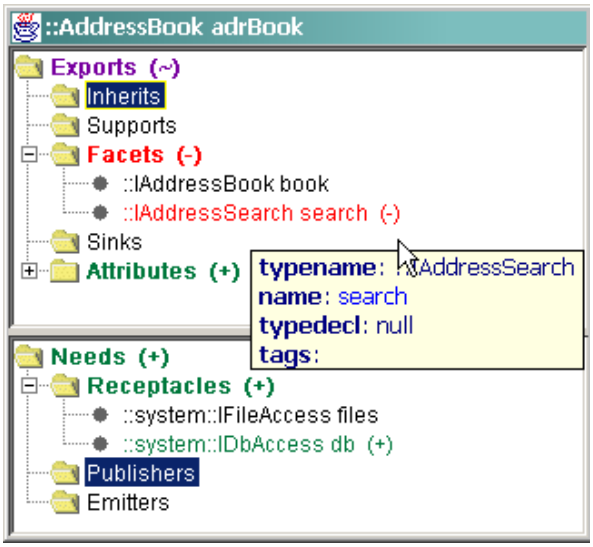

Figure 2.3: ENT-based difference highlighting for CORBA components

need to take into account their use by clients, and from the usage point of view changes in the provided and required parts of component interface do not affect substitutability in a uniform way. The situation is unlike most programming languages where the type of the "replacement" object must be an exact subtype of the current type.

We therefore define two kinds of component substitutability that deal with the extent to which the environment is considered: strict and contextual substitutability. Strict substitutability is in fact the usual subtyping-based one [VHT00], in which component C^r can substitute C^c if $C^r \succ C^c$. It is useful for comparing two components alone, i.e. without any information about their actual use.

Contextual Substitutability In the evaluation of substitutability, we can determine the actual run-time architectural environment of the application in which the component will be bound. This leads to a novel form of substitutability that takes into account

1. which of the current component's provided features actually have bindings to particular required features of other components, and
2. whether the environment provides features which the replacement component declares as required.

This idea is illustrated by Figure 2.4 on the next page, where e.g. the new version of `HTTPClient` requires an additional `XMLParser` interface that

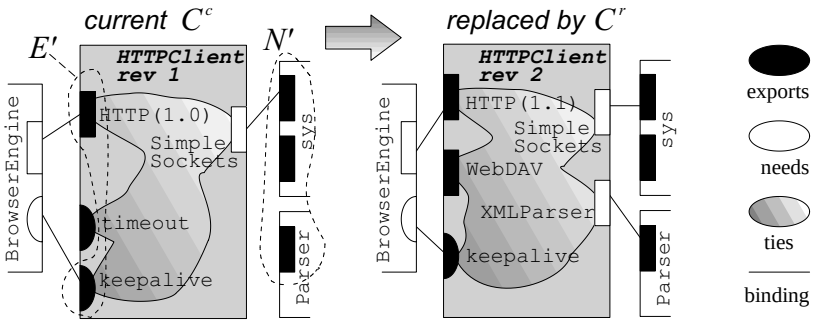

Figure 2.4: Context and Component Substitution

is being provided in the environment by an already present parser component.

The description of the deployment environment in terms of the ENT model is called deployment context. It is defined as a pseudo-component $Cx = E' \cup N' \cup T'$. The E' set represents the subset of provided elements of the current component that are actually bound to other components. The N' set represents the provided elements of other components that can satisfy the requirements of the replacement one. The T' are the current component's ties that are related to the bound exports of the component and the counterparts of its needs available in the context.

The novel kind of substitutability which uses this notion is therefore termed *contextual substitutability*.

Definition 2.2.2 () *Given a current component C^c and its deployment context $Cx = \{E', N', T'\}$, the replacement component $C^r = \{E^r, N^r, T^r\}$ is* contextually substitutable *for C^c (modulo renaming of elements $n' \in N'; n'.name = nil$) if $C^r \succ Cx$, that is $E^r \succ E' \ \wedge \ N' \succ N^r \ \wedge T^r/A \succ T'/A$ where $A = Names(C^r) \cap Names(Cx)$.* ■

In plain words, the replacement component provides at least the same features and qualities as are used of the current one in the context, requires at most what is available from other components, and its ties correspond to those of the current ones related to the replacement elements. Note that among other things the definition allows downgrading of provided features and extension in the required ones, through the definition of deployment context.

Intuitively, one would expect that strict substitutability implies contextual. This is proven in the following proposition.

Proposition 2.1 (Strict substitutability implies contextual) *Let us have two components C^c and C^r. If C^r is strictly substitutable for C^c, then it is contextually*

substitutable for C^c in any deployment context Cx.

Proof: *What we need to prove is that $C^c \succ Cx$. From the definition of context we can easily see that $E^c \succ E' \wedge N' \succ N^c$. Let us therefore consider the ties, for which we want to prove that $T^c/A \succ T'/A$ with the reduction set $A = Names(C^c) \cap Names(Cx)$.*

It follows from the definition of context that $Names(E') \subseteq Names(E^c)$ and $Names(T') = Names(T^c)$. Also, for comparing the $Ties$ categories we may safely lay $Names(N') = Names(N^c)$ because the "nil" name (added for the available context's provided elements not used by C^c) cannot be used by elements in T^c. From these assumptions we get $Names(Cx) \subseteq Names(C^c)$ and therefore $A = Names(Cx)$.

Consequently, we obtain $T^c/Names(Cx) \succ T'/Names(Cx)$ but this is equal to $T' \succ T'/Names(Cx)$. Because $Names(Cx)$ is neutral relative to reduction of T', we get $T' \succ T'$ which holds by Definition ?? *and therefore $T^c/A \succ T'/A$.*

Thus $C^c \succ Cx$ and, because $C^r \succ C^c$ was assumed, we prove the claim.

This fact can be useful in certain common cases, e.g. subsequent revisions of a component — we can easily prove strict substitutability at component release, store appropriate indication in its meta-data, and use it when upgrading the component.

Checking substitutability of black-box components using these definitions should bring clear advantages to substitution and upgrades. We consider the contextual substitutability to be especially useful in cases of big components with many interfaces, where some of them may be optional, and in systems consisting of a large number of components.

Since the substitutability definitions use the ENT meta-model structures, we have created a generic framework for component substitutability and compatibility, applicable to different technologies and extensible to future developments. Our evaluation of substitutability will continue to work if we modify the specification language by adding new parts of specification, or devise new classification dimensions.

Partial Substitutability Component's clients may be interested in the compatibility of only particular parts of the substituted component's interface, and/or of only selected aspects of these parts. Orthogonally to the two types defined above we therefore define a hierarchical system of partial substitutability levels. The levels were motivated by and clarify the work of Larsson [LC99].

- *Full substitutability* All features and qualities are included in the assessment, implying that both syntax and semantics of component interactions is compared.

- *Feature substitutability* This medium level concentrates on the syntactical aspects of the component interface, disregarding the specification of qualities. For the assessment, a subset $F \subseteq Elements(S)$ of component element set is used such that $\forall e \in F : (\{feature\}) \in e.classifier$.

- *Data substitutability* The least strictness is achieved by considering only the data features to preserve, as the last resort, the usefulness of data created by the current component. For the assessment, a subset $D \subseteq Elements(S)$ of component element set is used such that $\forall e \in D : (\{feature\}, \{data\}) \in e.classifier$.

We can use combinations of substitutability kinds and levels, due to their orthogonality. Thus we can require e.g. *full strict substitutability* to ensure plug-in replacement, or just *contextual data substitutability* if we know there are only a few operational bindings that we can adapt.

2.2.3 Backward Compatibility of Components

As we noted in the introduction to this chapter, there is the common case of component upgrade, where the C^r is actually a downstream revision of the C^c (with a smaller revision number rev). This means that both of them will have the same name. Substitutability between such two components is commonly called backward compatibility.

We therefore say that component C^2 is strictly *backward compatible* with C^1 if the two components have the same name, $rev(C^1) < rev(C^2)$, and C^2 is strictly substitutable for C^1. Similarly, contextual backward compatibility means that the new revision C^2 is contextually substitutable for C^1. The contextual compatibility makes it explicit that newer versions may not be plug-in replacements for the old ones; in fact, real life sometimes requires such incompatible changes to happen.

2.3 Revision Identification Scheme for Components

As any piece of software, a component that is successfully used for a period of time evolves into many revisions which differ in their specifications. We therefore need a cheme for their identification better than that provided by common practice and current version control tools, as discussed in the Introduction and shown in Figure 2.5 on the following page.

Our scheme reconciles the sometimes conflicting needs of the human developers (readable data for version selection, indication of changes) and their automated tools (precise data for version graph traversal, compatibility checks).

```
/**
* $RCSFile: AdrBookExample.cdl,v $ $Revision: 1.2.2.2 $
*/
frame FAddressBook
{  ...
```

`Revision: 1.2.2.2` *means "this is the second change to this file on a branch created from the second revision on the trunk."*

Figure 2.5: Meaning of revision identifiers in RCS-based systems

In line with the fundamental positions of the thesis, we design our versioning scheme as *specification-based*. It uses component specification as the primary object of versioning, and at the same time the specification provides the source data used to compute the revision identification. Furthermore, its novel in that it links the component revision identification to the information about exactly which parts of the specification are affected by the change between revisions.

2.3.1 The ENT Revision Identification Scheme

The usage of the ENT model provides us with the opportunity to create revision identification at several levels of abstraction, rather than a single-level scheme. In this process, higher (abstract) levels — which are useful for human understanding — can be derived from lower (detailed) ones, which have a clear correspondence to individual parts of the specification.

At all levels, the revision of the level is given by its revision marker which is an ordered tuple of natural numbers in which each has a relation to a well-defined part of the specification. A (specification-based) revision identifier is a human readable form of revision marker.

The lowest level of revision markers has component trait set as its object. It is designed to provide the most detailed information about the evolution of the component specification with a fixed number of elements. A *detailed revision marker* of a component C^2 (an immediate ancestor of revision C^1) is a tuple $R_D = (r_1, \ldots, r_n)$, $r_i \in N$ such that $\forall t_j \in Traits(C^2)\ \exists r_j \in R_D$: $r_j = rev(t_j)$ and $\forall t_j : rev(t_j) = rev(t_i) + 1, t_i \in Traits(C^1)$ iff $diff(t_i, t_j) \notin \{init, none\}$.

Component Revision Identification Since component specifications usually contain a number of traits, detailed revision marker may consist of too many numbers to be practical for human reading and understanding. Using the E, N, T category set we obtain a suitable aggregate revision marker.

The component revision marker of component C^2 is therefore a triple $R_C(C^2) = (r_E, r_N, r_T)$, where $r_E = rev(E^r)$, $r_N = rev(N^r)$ and $r_T = rev(T^r)$. It has all the benefits of a specification-based marker, a balanced number of elements, a clear relation to the detailed marker data and in addition fits the pattern of the industry-standard "M.m.μ" revision identifiers.

In certain cases, namely if user-defined data types need to be versioned, the component revision markers are still overly elaborate. As the most abstract level we therefore define a *primitive revision marker* as a single revision number $R_P(C^2)$ that is incremented upon any ENT-relevant change.

Each level of our revision identification can be derived either directly from specification comparison or from the markers on the lower level. This "cascaded" derivation uses the following mechanism: if there is a change (increment) in any revision number on the lower level, then the corresponding revision number on the higher level according to the ENT model aggregation rules is incremented. For example, an increment in $rev(t_i)$ of one trait propagates to the revision number of the category containing t_i.

The behaviour of revision identifiers can be characterised by three properties: *idempotence* (the same set of changes in C^1 must always result in the same revision marker of C^2), *differentiation* (the derived revision of a component cannot have the same revision marker as its predecessor) and *monotonicity* (the identifiers preserve the time order of revision creation). In the thesis we prove these properties for our scheme.

2.3.2 Application in the SOFA Framework

In the SOFA framework prototype implementation, the ENT-based revision information is stored in two places: as part of the CDL specification of the component (see Figure 2.6) plus in a meta-data attached to its distribution form.

The structures subject to our versioning scheme are all user-defined types. The motivation for this rather far-reaching step is the need to handle the evolution of components at large. Once we allow one name to denote multiple versions, we have to uniquely identify also all the types it references in order to ensure the correctness of their interactions. Except for component types (`frames`, which use component revision markers), primitive revision identification is then sufficient.

To be able to reference these versioned types and components, we propose that type names in SOFA have the form of an URI (Uniform Resource Identifier, [BLFM98]). This enables us to create structured identifiers which carry a lot of information yet remain human readable, in contrast to e.g. UUIDs used in COM.

The greatest advantage of the ENT revision identification scheme over

```
frame FAddressBook
[ @rev = 3.2.1;      // automatically generated
  @diff= (spec,mut,none); ]
{
  provides:
    IAddressBook book; // default revision
    sofa://com.notscape/ab/IAddressSearch#rev=3  search;
  requires:
    OfficeApps/IPhoneBook#rev=2  phone;
};
```

Figure 2.6: Proposed revision identifiers in CDL source

the current state of the art is its blend of algorithmic predictability and comprehensibility of meaning. Firstly, the way revision markers and identifiers are derived ensures idempotence — the same set of changes will always result in the same marker, unlike RCS-based or manual schemes. Secondly, the ENT revision identifiers convey clear meaning to humans (developers, component users) with their correspondence between the place of change in terms of the ENT model and position in the identifier.

2.3.3 Relating Revision Identification and Compatibility

In many software application installation systems the distribution packages contain meta-data which describes their purpose, version, dependencies and compatibility information. It is used for application lookup, installation, and upgrades [2].

Since components should be easy to use and assemble, improvements in the automation and reliability of these activities are highly desirable. In particular, the reliability of upgrades is often checked using an intuitive rule, that "$rev(a) < rev(b)$ implies that b can substitute a but not vice versa." In a simple form (used e.g. in DCE) the rule leads to the plain comparison of revision numbers to determine substitutability. More complex systems rely on meta-data with information about the applications' compatibility, but even these do not formalise the rule sufficiently.

Our approach to solving upgrade reliability uses meta-data which combines revision data with indications of compatibility between component revisions. The meta-data of a given component version contain primarily these elements:

- identification of the component (name, description, ...);

[2] Upgrade is a special case of substitution, in which a revision N is substituted by the revision $N + 1$ (or $N + m$).

- version identification, i.e. the branch, revision and variant identifiers;
- difference indications of this revision against the previous one;
- optionally the revision history, i.e. the path from the first release revision to the current one;
- optionally the pairwise differences between the current revision and each revision listed in the history.

The difference indications represent the results of type-based ENT comparison of component specifications, using the *diff* classification values, as described above. The revision and difference data is contained in two tuples, $d_{ENT} = (R_C, D_C)$ and $d_T = (R_D, D_T)$. The first are coarse-grained data: component revision marker and comparison of the E, N, T categories. The second are the detailed revision marker and trait comparison results. The revision history is a set $H = \{d_{ENT,1}, d_{ENT,2}, \ldots, d_{ENT,v-1}\}$ with category-level data of all preceding revisions of the component.

The pairwise differences data is an ordered set $\{d_{1,v}, \ldots, d_{v-1,v}\}$ which contains category-level differences between each historical revision and the current one. It can be included to increase the speed and reliability of compatibility checks if an upgrade of an old revision is performed.

A key contribution of our work is that meta-data with this design explicitly state the relation between revisions and their compatibility, and that this information is obtained reliably and automatically. It can be pre-computed once (on component release; the *diff* algorithms often have even exponential complexity) and stored in a form which allows the checks to run (any number of times) in linear time. It is used prior to the upgrade to achieve reliable and at the same time fast substitutability checks.

Metadata Formats In SOFA, the meta-data is present in two places. The component revision identification and *diff* is made part of the CDL component specification by extending the grammar of the language and its compiler (see Figure 2.6 on the preceding page). Thus an important part of the meta-data is accessible in a readable form to the developers directly in the component specification.

The complete data in XML format is stored in the component distribution package. It contains all the meta-data elements described above in one place, and is useful for tools that manipulate the component. A prototype implementation of SOFA Template Repository uses the meta-data included with the components to store them in appropriate places and for user queries of the repository contents.

3. Summary

The work contained in the thesis is summarised by its key contributions and the lessons we have learned during the research.

3.1 Contributions of the Work

The work described in the thesis and in related published articles is based on the hypothesis that part of the lack of success of components is due to inadequate component versioning and compatibility evaluation methods.

Its results contribute to the current state-of-the-art in component research and development in the following:

1. It defines an abstract meta-model of component interface which makes it possible to model components in a wide range of current component frameworks [Bra02a, Bra02c]. Furthermore, it can easily accommodate future developments that will result in creating new kinds of component specifications.

2. It introduces the definitions of and algorithms for a novel notion of contextual substitutability and compatibility, specifically designed for black-box software components as parts of architectures [Bra99, Bra01b, Bra02b].

3. It describes a versioning scheme (revision/release identification) suitable for black-box components and providing a precise meaning of version data [Bra99, Bra01a]; the author is aware of no similar approach neither in component systems nor in other software development areas.

4. It clearly defines the (intuitively obvious) relation between component versioning and compatibility, and shows the advantage of using this relation in component upgrading [Bra02b].

5. The use of data resulting from normal development processes, namely IDL and source code, is a key aspect which other approaches tend to neglect or at least do not emphasise.

3.2 Lessons Learned

During the development of the ideas and methods covered in this thesis we have learned several lessons.

Component modelling It is fairly easy to extract model information from IDL or ADL sources, but doing so for component models implemented directly in programming languages is very difficult (an example is the JavaBeans component model). Component programming and modelling benefits from languages with a direct support of its key abstractions.

Specification language features Various component models provide interesting and/or useful features, but there is no single specification language which would support most (if not all) of them. These languages could (and should) be much richer in their repertoire to achieve better usability of components and improved reliability of their substitution.

Specification-based versioning is needed Until now, component versioning has been understood as simple technical or marketing tagging. However, our work reveals that the nature of components — at the same time design abstractions, language constructs and tradeable items — requires versioning to be integrated into the component (meta-)models and related languages.

Bibliography

[ABV00] Sven-Arne Andréasson, Přemysl Brada, and Jan Valdman. Component-based software decomposition of flexible manufacturing systems. In *Proceedings of International Carpathian Control Conference*, Podbanské, Slovak Republic, 2000. TU Košice.

[BLFM98] Tim Berners-Lee, Roy Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396, IETF, 1998.

[BR00] Přemysl Brada and Jan Rovner. Methods of SOFA component behavior description. In *Proceedings of Information Systems Modeling (ISM 2000)*, Rožnov pod Radhoštem, Czech Republic, 2000.

[Bra99] Přemysl Brada. Component change and version identification in SOFA. In Jan Pavelka and Gerald Tel, editors, *Proceedings of SOFSEM'99*, LNCS 1725, Milovy, Czech Republic, 1999. Springer-Verlag.

[Bra00] Přemysl Brada. SOFA component revision identification. Technical report 2000/9, Department of Software Engineering, Charles University, Prague, Nov 2000.

[Bra01a] Přemysl Brada. Component revision identification based on IDL/ADL component specification. In *Proceedings of the 10th European ACM Conference on Software Engineering (ESEC/FSE)*, Vienna, Austria, 2001. ACM Press. Poster presentation.

[Bra01b] Přemysl Brada. Towards automated component compatibility assessment. In *Workshop on Component-Oriented Programming (WCOP'2001)*, Budapest, Hungary, June 2001. Position Paper. Available at http://research.-microsoft.com/~cszypers/events/WCOP2001/.

[Bra02a] Přemysl Brada. The ENT model: a general model for software interface structuring. Technical Report DCSE/TR-2002-10, De-

partment of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic, 2002.

[Bra02b] Přemysl Brada. Metadata support for safe component upgrades. In *Proceedings of COMPSAC'02, the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE Computer Society Press.

[Bra02c] Přemysl Brada. Parametrized visual representation of software components. In *Proceedings of the 7th Objekty Conference*, Prague, Czech Republic, November 2002.

[BW98] Alan W. Brown and Curt Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[C+02] T. Coupaye et al. *The Fractal Composition Framework (Version 1.0)*. The ObjectWeb Consortium, July 2002.

[Car97] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

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

[Des98] Desktop Management Task Force. *Desktop Management Interface Specification, version 2.0*, 1998.

[Han98] Jun Han. A comprehensive interface definition framework for software components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference*, pages 110–117, Taipei, Taiwan, December 1998. IEEE Computer Society.

[J+03] Ian Jackson et al. *Debian Policy Manual*, 2003. Available at http://www.debian.org/doc/debian-policy/.

[LC99] Magnus Larsson and Ivica Crnkovic. New challenges for configuration management. In *Proceedings of the SCM-9 workshop, ECOOP 1999*, LNCS 1675, Toulouse, France, September 1999.

[LC00] Magnus Larsson and Ivica Crnkovic. Component configuration management. In *Proceedings of the ECOOP Conference, Workshop on Component Oriented Programming*, Nice, France, June 2000.

[LvdH02] Chris Lüer and André van der Hoek. Composition environments for deployable software components. Technical Report UCI-ICS-02-18, University of California Irvine, August 2002.

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

[McI68] Doug McIlroy. Mass-produced software components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, pages 138–155, Garmisch, Germany, October 1968.

[MT00] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of OOPSLA '93*, volume 28 (10) of *ACM SIGPLAN Notices*, October 1993.

[OMG02a] Object Management Group. *CORBA Components*, June 2002. Version 3.0. OMG Specification formal/02-06-65.

[OMG02b] Object Management Group. *UML Profile for Enterprise Distributed Object Computing Specification*, 2002. OMG Specification ptc/02-02-05.

[Ore98] Peyman Oreizy. Decentralized software evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE)*, Kyoto, Japan, April 1998.

[PBJ98] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

[PV02] František Plášil and Stano Višnovský. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(10), November 2002.

[Ras02] Uwe Rastofer. Modeling with components – towards a unified component meta-model. In *ECOOP Workshop on Model-based Software Reuse*, Malaga, Spain, 2002. Available at http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ECOOP2002/papers.shtml.

[Rig02] Roger Riggs. *The Java Product Versioning Specification*. JavaSoft, 2002. Available at http://java.sun.com/j2se/1.4.2/docs/guide/versioning/spec/versioning.html.

[S+95] Mary Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, March 1995.

[SA02] Frédérick Seyler and Philippe Aniorte. A component meta model for reused-based system engineering. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, Dresden, Germany, 2002. Available at http://www.metamodel.com/wisme-2002/.

[Sun01] Sun Microsystems, Inc. *Enterprise JavaBeans(TM) Specification (Version 2.0)*, August 2001. Available at http://java.sun.com/-products/ejb/docs.html.

[SV01] Douglas C. Schmidt and Steve Vinoski. Object interconnections: CORBA and XML, part 1: Versioning. *C/C++ Users Journal*, 19(5), May 2001.

[Szy98] Clemens Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1998.

[vdHW02] André van der Hoek and Alexander L. Wolf. Software release management for component-based software. *Software - Practice and Experience*, 2002.

[VHT00] Antonio Vallecillo, Juan Hernández, and José M. Troya. Component interoperability. Technical Report ITI-2000-37, Universidad de Málaga, Spain, July 2000.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 322, pages 55–77. Springer-Verlag, 1988.

[ZW97] Amy Moormann Zaremski and Jeanette Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.