Specification-Based Component Substitutability and Revision Identification

Doctoral Thesis

Přemysl Brada

Charles University in Prague
Faculty of Mathematics and Physics
Department of Software Engineering

August 2003

Specification-Based Component Substitutability and Revision Identification

Doctoral Thesis

The author can be contacted at:

Department of Computer Science and Engineering
University of West Bohemia in Pilsen
Univerzitni 8
30614 Pilsen
Czech Republic
email: brada@kiv.zcu.cz

Electronic version of this thesis is available at
http://www.kiv.zcu.cz/~brada/research/thesis/

Specification-Based Component Substitutability and Revision Identification

Doctoral Thesis

Přemysl Brada

Charles University in Prague
Faculty of Mathematics and Physics
Department of Software Engineering

August 2003

— Thank you, Jana —

Contents

1. Introduction 1
- 1.1 State of the Art in Component Modelling, Versioning and Compatibility 2
  - 1.1.1 Components, Component Models and Meta-Models 2
  - 1.1.2 Component Versioning 4
  - 1.1.3 Substitutability and Compatibility 6
- 1.2 Goals of the Thesis 7
  - 1.2.1 Primary Goals 7
  - 1.2.2 Addressing the Goals: The Constraints 8
- 1.3 Structure of the Thesis 10
- 1.4 Contributions of the Work 10
  - 1.4.1 List of Published Articles 11
- 1.5 Conventions Used in the Text 12

2. Component Interface Representation: The ENT Meta-model . 13
- 2.1 Elements of Component Interface Specification 14
  - 2.1.1 Comparison of Component and Modular Systems . . 14
  - 2.1.2 Component Specification and its Elements 16
- 2.2 Classifying Specification Element Properties 18
  - 2.2.1 Common Properties of Elements 19
  - 2.2.2 The ENT Faceted Classification System 20
- 2.3 The Model: Elements, Traits and Categories 21
  - 2.3.1 Specification Elements 22
  - 2.3.2 Traits of Elements in Component Specification . . . 24
  - 2.3.3 Categories of Traits 25
  - 2.3.4 The *E*, *N*, *T* Category Set 27
  - 2.3.5 Restricted Elements and Categories 29
- 2.4 ENT Model of SOFA Components 30
  - 2.4.1 Mapping of SOFA CDL Constructs to ENT Elements 30

2.4.2 Trait Definitions 32
2.4.3 How to Find Traits 33
2.5 Applications of the Model 33
2.5.1 Applicability to Frameworks and Technologies 34
2.5.2 ENT-based Component Visual Representation . . . 35
2.5.3 Assistance in Component Search and Retrieval 36
2.5.4 Other Applications 37
2.6 Discussion 37
2.6.1 Advantages of the Model 37
2.6.2 Disadvantages and Open Issues 39
2.6.3 A Note on Specification Languages 39
2.7 Summary 40

3. Analysing and Classifying Specification Differences 43
3.1 Motivations and Approaches 44
3.1.1 Desired Properties of Component Comparison . . . 45
3.1.2 The Approach Taken 46
3.2 Differences Between Specifications 47
3.2.1 Comparing Specification Parts 48
3.2.2 The Differences and Their Classification 51
3.2.3 Difference Propagation 52
3.3 Specification and Carrier Language Issues 55
3.3.1 ENT-based Comparison and Type Rules 55
3.3.2 Type Systems for Specification Languages 56
3.3.3 Carrier Language Issues 57
3.4 Comparison of SOFA Component Specifications 58
3.4.1 Subtyping Rules 58
3.4.2 Examples of Frame Comparison 59
3.5 Discussion 66
3.5.1 Advantages 66
3.5.2 Disadvantages and Open Issues 67
3.6 Summary 68

4. Revision Identification Scheme for Components 71
4.1 Issues in Component Versioning 73
4.2 Specification-Based Revisions 75
4.2.1 Types of Changes Between Revisions 75
4.2.2 Relating Changes and Revision Identification 76

4.2.3 Our Approach: Specification-Based Revisions 76
4.3 The ENT Revision Identification Scheme 77
4.3.1 Detailed Revision Identification 77
4.3.2 Component Revision Identification 79
4.3.3 Primitive Revision Identification 80
4.3.4 Cascaded Derivation of Revision Markers 81
4.4 Properties of ENT Revision Identification 82
4.5 Application of the Scheme in the SOFA Framework 83
4.5.1 All Types Have Revision IDs 84
4.5.2 How Revision Identification is Derived 84
4.5.3 Versioning Complete: Handling Branches and Variants 85
4.5.4 Version Data in CDL 86
4.5.5 Identification of Versioned Types 86
4.6 Discussion 88
4.6.1 Advantages 88
4.6.2 Disadvantages and Issues 89
4.7 Summary 89

5. Component Substitutability and Compatibility 91
5.1 Issues in Component Substitution 92
5.2 Substitutability of Components 94
5.2.1 Strict (Subtype) Substitutability 95
5.2.2 Deployment Context of a Component 96
5.2.3 Contextual Substitutability 98
5.2.4 Partial Substitutability 99
5.3 Backward Compatibility of Components 100
5.3.1 Redefinition of Larsson's Compatibility Levels 101
5.4 Examples for SOFA Components 102
5.4.1 Compatibility of Frames 102
5.4.2 Determining Context 104
5.4.3 Role in Component Updates 107
5.5 Discussion 107
5.5.1 Advantages of our Method of Substitutability Checking 108
5.5.2 Limitations of Our Method 108
5.5.3 Compatibility and Real Life Development 110
5.6 Summary 111

6. Compatibility and Versioning Related . . . 113
6.1 Relating Versioning and Compatibility . . . 114
6.1.1 Motivation . . . 114
6.1.2 Generic Mechanism of Upgrades . . . 115
6.2 Meta-data: The Integrating Element . . . 116
6.2.1 What the Meta-Data Should Contain . . . 116
6.3 Use in the SOFA Framework . . . 118
6.3.1 Metadata Formats . . . 119
6.3.2 Repository for Versioned Components . . . 120
6.3.3 Component Updates with Versioning . . . 121
6.4 Summary and Discussion . . . 122
6.4.1 Advantages . . . 122
6.4.2 Issues . . . 123

7. Overall Evaluation and Related Work . . . 125
7.1 Component and Interface Meta-Models . . . 125
7.1.1 Distilling Commonalities from Component Models . 126
7.1.2 Meta-Models Defined as Such . . . 127
7.1.3 Summary . . . 130
7.2 Component Comparison and Substitutability . . . 130
7.2.1 Specification Comparison and Matching . . . 130
7.2.2 Subtyping-based Substitutability . . . 131
7.2.3 Component Substitutability and Compatibility . . . 133
7.2.4 Summary . . . 135
7.3 Component Versioning . . . 135
7.3.1 Industrial Frameworks . . . 136
7.3.2 Research in Component Versioning . . . 137
7.3.3 Syntactical Analysis and Meta-Data in Versioning . . 137
7.3.4 Summary . . . 138

8. Conclusion . . . 139
8.1 Summary of Our Work . . . 139
8.2 Lessons Learned . . . 140
8.3 Open Issues . . . 141
8.4 Future Work . . . 142

A. ENT Model Definitions for Primary Component Frameworks . 145
A.1 The ENT Model for SOFA Components . . . 145

A.2 The ENT Model for CORBA Components 146
A.2.1 Trait Definitions 146
A.2.2 Example: The `Parking` Component Source 147
A.2.3 Example: The `Parking` Component in ENT 147
A.3 The ENT Model for JavaBeans 147
A.3.1 Trait Definitions 148
A.3.2 Example: The `MyJuggler` JavaBean Source 149
A.3.3 Example: The `MyJuggler` JavaBean in ENT 150

B. SOFA CDL Subtyping Rules 153
B.1 The Rules for the SOFA CDL 153

C. The Specifications of SOFA Component Meta-Data 159
C.1 CDL Meta-Data Section Grammar 159
C.2 Grammar of the URI form of SOFA identifiers 160
C.3 XML Meta-Data Document Type Definition 160

Bibliography 167

Chapter 1

Introduction

After years of research and industrial development, software component technology [Szy98] has become well established as an important approach to engineering complex and flexible software systems. Building on extensive research in the field (systems like Darwin [M+95], UniCon [S+95], SOFA [PBJ98], ArchJava [ACN02b] or Fractal [C+02]), commercial component frameworks (Enterprise JavaBeans [Sun01a], CORBA Component Model [OMG02f], and Microsoft's DCOM and .NET technologies [Mic95, Rog97, Cor02]) have gained commercial success.

From the outset, the main aim of the component technology has been to create pluggable "software integrated circuits" [McI68] that can be easily traded and assembled, even by knowledgeable end-users. This is believed to increase productivity (via the reuse of general as well as domain-specific components) in building software applications at large.

But components do not seem to live up to these "silver bullet" expectations [Bro95]. Their current use concentrates on user interface widgets (e.g. Borland Delphi components [Sof01], Sun's JavaBeans [Sun97]), server-side data-centric components (Enterprise JavaBeans [Sun01a], CORBA Components [OMG02f]) or specialised applications in communications or embedded devices (the Koala model [vO01] or the Robocop project [Lav02]). The functionality provided by commercial component frameworks is limited to the basic wiring of components (setting up their interconnections) and supporting key system-level aspects (deployment to target hosts, location transparency, communication and transaction management).

We believe that much more is actually needed. Components would be in a more widespread use if they provided the intuitively expected sophisticated functionality combined with easy application composition [LR01b, LvdH02], high level of re-use, and seamless evolution found e.g. in personal computer expansion cards. In this respect, component frameworks currently lack suitable support for smooth and safe component upgrading [OMG01, Ore98], version management in relation to component naming and search/trading [vdH01], component modelling (based on robust

meta-models) and visual development with components [YAM99, LR01a, MAV02], etc.

This 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 introductory chapter we describe the open issues which motivate our work, present its goals and approach, and list the contributions made to the component research field.

1.1 State of the Art in Component Modelling, Versioning and Compatibility

Let us start by surveying briefly the current component technology and research landscape in view of the above needs. We will focus on three areas of the ones noted above: component modelling, versioning and substitutability checking.

The subsections below list the needs of component technology with respect to these areas, and discuss why their support in current systems is inadequate. (Chapter 7, Related Work, contains a more detailed analysis of the related research.) Motivated by these findings, the next section follows with a formulation of the goals of this thesis.

1.1.1 Components, Component Models and Meta-Models

The term *software component* was coined by McIlroy [McI68] when software engineering was still in its infancy. Because many definitions of this term have emerged since, we will clarify the situation and set our position before treating the issues of component modelling.

In this work we use the term in the following meaning: a software component is a coarse grained black-box software element with contractually specified interface syntax and semantics. This understanding has its roots in the research on software architecture description languages (ADLs, see [MT00] for a survey). The definition of the term component that is closest to our use is given by Szyperski in [Szy98]. Several other definitions can be found (e.g. in [BW98, PDH99]) which follow the same idea.

On the other hand, some authors [CCF00, Lav02, S$^+$95] take a component as a set of closely related artefacts that can be composed into more complex applications. While this corresponds to current practice (e.g. Enterprise JavaBeans, application packages), such components are difficult to reason about formally. We therefore do not consider such models in our work.

A component *model* defines “a standard to which a set of components

must adhere in order to be composable into applications" [LvdH02], i.e. what components consists of and how they can be bound together. In many component-based systems and ADLs, it is defined implicitly or informally (for instance in UniCon [S+95], SOFA [PBJ98], ACME [GMW97], or on the industrial side in EJB [Sun01a], Microsoft .NET [Cor02]).

At a more abstract level, a meta-model (the M3 level of the generic framework described in the Meta Object Facility [OMG02g]) captures the common aspects of a set of models in the above meaning. It is thus "an abstract language for some kind of metadata" [OMG02g] — in this case component specifications — which defines the common denominator for its terminology, structural and semantic features, element relationships, modelling possibilities etc.

The primary use of a component meta-model is usually to be the source from which concrete component models are instantiated (we call these "a-priori meta-models"), i.e. to establish the capabilities of a technology. The UML Enterprise Distributed Object Computing (EDOC) Profile [OMG02h] is a key meta-model of this kind which maps well to current industrial component frameworks. In the component research area, the meta-models by Seyler and Anoirte [SA02], Han [Han98] or from the Fractal framework [C+02] belong to this category.

On the other hand, meta-models can be created by distilling the commonalities of existing component models ("derived meta-models") for analysis or technology conversion purposes. Examples are the Vienna Component Framework [OGJ02] or Rastofer's meta-model [Ras02]; the developers of the UML EDOC Profile have also used this approach.

Good meta-models are important because they define the standard level of practice and technology in their subject areas. In the current state of art, component meta-models are used to define the component interface structures and relations, and in part also their visual representation [OMG02h]. However, the structures and relations they define are mostly straightforward abstractions of the present state of the technology. Except for [SA02], current meta-models offer few forward-thinking ideas and provisions to handle future developments.

We think that even some of the current issues that penetrate the technology (i.e. are dealt with in several concrete component models or their implementations) could be handled at the meta-level. Examples are configuration management issues (versioning, compatibility as a key to configuration consistency), aspects (distribution, location transparency, concurrency, persistence), or the "illities" [Han99a] (reliability, quality of service, performance). While specialised standards and technologies exist for some [IEE98, Cor02, OMG02d], none of the solutions have been made part of any meta-model.

The result is that these issues are not handled consistently in concrete models. In practice this leads to a duplication of effort and ad-hoc handling of component interoperability [OGJ02], [OMG02e, Chapter 18]. In addition, current meta-models will need to be modified to accommodate these and some upcoming developments (mobility, emphasis on quality of service) or else they quickly become obsolete.

1.1.2 Component Versioning

Versioning has been a standard part of software configuration management for a long time [Bab86, Tic94, CW98], serving the need to distinguish different shapes of the same software artefact. Any software component, as a software artefact, inevitably evolves and changes. Thus several versions of one component are created, be it branches (results of parallel development), revisions (sequences of historical versions) or variants (different implementations of the same revision).

The need for version support suitable specifically for components is well articulated by Brown and Wallnau in [BW98]. They argue that "traditional configuration management and version control techniques provide an important starting point ... [but] new methods and tools are essential." Compared to versioning used during software development, 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].
- Components should be easy to assemble, mainly in a (semi-)automated manner [BW98, LR01b, Ore98]. A component may therefore be used in multiple configurations, some of them unforeseen at the time of its creation. In addition, many component-based applications will require maintenance (reconfiguration, bug fixes and upgrades) performed with no or minimal human intervention [Lav02]. Version support is needed to easily select versions suitable for such composition and maintenance operations [Szy98, Section 5.1.2].
- Components may exhibit unpredictable evolution, namely when evolved by other than the original producer. The result will be the emergence of many versions of the same component (defined by its specification) with complications in change management and version graph control [Szy98, LC00].
- Component providers cannot govern the use of components after release to market. Therefore, version information need to be understandable, precise, and preferably standardised [LC99].

These challenges require component versioning to provide several distinctively novel features. First, it must support automated creation and processing of version data (including querying, matching and searching) lest it puts additional burden on the developers and tools. Automated creation also increases the fidelity of the data, by eliminating errors caused by manual creation.

Second, component versioning should use information-rich data with a well-defined structure and format suitable for machine processing. Such version data should be understandable and modifiable — without impairing its consistency and meaning — by any player on the component market, not just the original developer [LC99]. The potential scale of component market calls for some standardisation in this area.

Third, component version data should have a well-defined meaning (semantics) so that automated agents can use it in their reasoning about components. In particular, it should be possible to search for particular component versions based on current configuration information (e.g. for upgrades) and prevent binding of incompatible component versions. The semantics of data includes a correspondence between the structure of a component, the changes to its parts, and its version identification.

As discussed in detail in the Related work (Chapter 7), we find very little support for these component version management functions in current industrial as well as research component systems. In some of them, notably Microsoft COM [Mic95], versioning is omitted altogether and thus component evolution is forbidden *de iure*. But since change cannot be avoided, version identification appears in the disguise of naming conventions (e.g. the `IClassFactory` and `IClassFactory2` COM interfaces [Mic03]) which makes component use complicated for developers. Worsely, the technology may fail to deliver its purpose. This is the case of un-versioned Windows shared libraries, where the developers prefer to bundle with the distributed applications the required versions of DLLs, contrary to their purpose.

Even in systems with versioning 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 [OMG02f, Rig02, Des98, Bai97]). The desirable advanced uses of versioning thus cannot be achieved. A representative example is Java product versioning, where two versions of a Java package can have version identifiers “3.2.5” and “3.6.1”. Because of the vague definition of the scheme, we cannot conclude where and how the second version differs, whether it can substitute the first one, etc. The only thing we can determine is that they should be different (“equals” relation) and that the second one is probably newer than the first one (“precedes/follows” relation).

The unsatisfactory state of component versioning had already been noted by the research community [BW98, Szy98], and is also evident from the

fact that the OMG activities in the versioning area [OMG96] have been abandoned [OMG02b]. 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.

1.1.3 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 upgrade must not introduce new problems, but rather fix the old ones or enhance the application. In configuration management terms, it must preserve (or improve) the configuration consistency of the application.

Currently there are two main classes of solutions dealing with this issue in similar (though not necessarily component-oriented) systems. In the first one, meta-data is provided with each application package or component [J$^+$03, Hes03, Bai97, Des98, LC00]. It contains its version identification, information about compatibility with previous versions and about the components that must be present in the deployment environment for its correct functionality.

These systems benefit if the meta-information is structured, rich and precise. This is important for querying repositories for available components and upon installation/upgrade of new ones to prevent compromising application or system consistency.

However, in practice the meta-data is usually created by the component producer manually based on their knowledge of its implementation. In our opinion this has two drawbacks. First, it either leads to very simple (and therefore insufficiently rich) data or its time consuming creation contradicts the need for short development time. Secondly, manual work is error prone due to the possibility of omissions, oversights or simple typos.

Approaching this issue from a more rigorous direction, some (mainly research) systems use various forms of subtyping relation to check substitutability. This relation is determined by comparing formal or semi-formal component descriptions. Examples of this approach are function- and module-matching relations [ZW97, HL99], compatibility definitions by Perry [Per87] or behavioural subtyping [VHT00, Nie93].

Being based on data (code and specifications) that is available as a result of normal software development, these approaches provide higher fidelity than the above discussed meta-data. They also enable full automation of its derivation and therefore reduce the risk of using incorrect data in the compatibility checks.

On the other hand, the algorithms used in determining the subtyping relation may have high computational complexity, resulting in potential delays. Also, some research systems define relaxed compatibility levels to increase the chances on substitution. But these relaxations are designed for slightly different purposes (e.g. searching) and do not always fit with the desire to make substitution as reliable as possible. (A standard feature is to allow a change in the number or in the order of method parameters, which is acceptable for human developers but may mean an insurmountable problem for automated component substitution.)

1.2 Goals of the Thesis

This thesis aims at providing answers to the open issues in component modelling, versioning and substitutability described above. Towards this end, it pursues the primary goals listed here.

1.2.1 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.

The existing component models and solutions to the penetrating issues have much in common but only some of these features have been captured in available meta-models. By doing so, we can achieve higher flexibility, interoperability and re-use of components. The challenge is to serve both the human and technology-related needs equally well, and to encompass current technologies as well as future developments.

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.

We would like to create a scheme which fulfills the requirements listed in the previous section, mainly the well-defined structure and semantics of version data. The main challenge is that its artefacts (the version data) should provide relevant information for both human users and component management tools, and at the same time reduce the need for manual version data creation.

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.

The challenges we face are twofold. First, the source code may not (most often will not) be available when component substitutability needs to be checked; the only information available may be component interface specification plus possible meta-data created during development. Second, due to the intimate connection of components and architectures the notion of component substitutability has to consider the overall architecture and environment configuration.

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.

The challenge (and the motivation for setting this goal) is to formally describe the oft-observed fact that an upgrade to an downstream revision is effectively a special case of substitution. In practical terms we would like to use the version data of the components as an aid in the compatibility assessment.

1.2.2 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.

From the perspective of software composition from independent parts, the interface represents the most important part of a component. Its description is often the only information about the component available to the tools (compilers, linkers, assembly etc.) as well as — more often than desirable — to the humans who control the particular activity.

This fact motivates the key viewpoint of our work: that we have a real need to base formal reasoning about software components on the already available description of their interfaces, possibly augmented by stand-alone data derived from the source code. Only then will the methods and tools resulting from the work be useful for the area of black-box components, in which their users have no access to the source code. Additionally, such

already existing data is directly used in component implementation which leads to cost effective (no added work) and more reliable methods.

Use automated methods There should be as much automation, and as little additional human effort involved in developing and using components as possible.

Components should make software, its development and use simpler, not more complicated. In our opinion this means (among other things) that we should strive for automated derivation of information and automated reasoning based on such data. Some effort in their development is understandable as the technology is complex and feature-rich (e.g. the need to describe the deployment options, ensure security, provide trading information).

But many tasks like creating technical description of components, their selection, deployment, and upgrading should have a reasonable fully automated default implementation (the UML EDOC Profile, Section 3.3 of [OMG02h], provides a similar argument). Only in exceptional cases, a manual control override should be used, and even this should be supported by tools.

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.

This constraint expresses our position that the software industry should create systems where people and their knowledge can take precedence over algorithms (hidden in opaque executables and binary data formats) and which allow manual control if needed.

We therefore favour solutions which use declarative specifications (like IDL languages), structured data in textual format (like XML), and uncomplicated syntactical structures.

Relate to real systems The methods should be readily applicable in practice, mainly in mainstream component frameworks and the associated development tools.

Software engineering research should provide answers to practical questions, and component technology still needs a lot of help in this respect. The work described in this thesis uses prototype implementations for the SOFA research component system. We nevertheless believe that the results of this work will be applicable in current practice, notably in the CORBA Component Model implementations.

1.3 Structure of the Thesis

The text of this thesis is broadly divided into three parts: foundational definitions and models, core chapters on component versioning and compatibility, and final summations of the work.

The first part forms a general foundation for the work. Chapter 2 opens the thesis with an in-depth discussion of the role of component specifications and the structures they contain. Considering the fact that software components are fairly complex and that the existing component models have many similarities, we present a general meta-model of component interface which allows its system-independent description. The chapter describes this meta-model, introducing the key abstractions (component features and qualities, and their grouping into traits and categories) used in subsequent chapters.

The core part starts by an analysis and classification of differences in component specifications (Chapter 3) that express the changes made during evolution. Its results are then used in two related areas. Chapter 4 presents a novel scheme for identifying component revisions, based on the component meta-model and classification of interface differences. Next, these abstractions are applied in the definitions of component substitutability in Chapter 5. The key result described in this chapter is the novel contextual compatibility which considers the architecture in which the component is bound.

Besides being based on the same model, these two key results of our work are closely related in consistency-preserving component upgrades. Chapter 6 describes this correspondence in detail, emphasising the use of the same data and algorithms for the two different purposes.

The work is done in an area with abundant research and strong industrial interests. Chapter 7 therefore provides a survey of the current research achievements and related efforts. Finally, the Conclusion (Chapter 8) summarises the achieved results and discusses how they match the objectives set in this Introduction. It also describes our ideas on how to apply the presented methods to other, non-component systems. The issues that remain open and the possibilities for further research complete the concluding remarks. At the end of the thesis, several appendices contain detailed descriptions of the results presented in the main text.

1.4 Contributions of the Work

The work described in this thesis and in related published articles contributes 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.

1.4.1 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.

[BR00a] 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.

1.5 Conventions Used in the Text

The following typographical and notational conventions are used throughout the text of the dissertation.

- In normal body text, *term definitions* are in italics.
- Samples of `source code` use fixed-width font.
- When identifiers are used in definitions and mathematical expressions, clarity is preferred to brevity, i.e. we often use names rather than single-letter symbols.
- The notation $tuple.element$ denotes a single element of an n-tuple defined as $tuple = (..., element, ...)$.
- The set *Identifiers* contains identifiers as defined in standard programming languages, i.e. string starting with a letter or underscore and containing letters, numbers and underscores.
- The $<:$ symbol denotes the subtyping relation (e.g. $short <: integer$).
- Concerning spelling, the text of the thesis is written in British English.

Chapter 2

Component Interface Representation: The ENT Meta-model

Software modules and components play an important role in software engineering, primarily as key abstractions for software decomposition. Despite their substantial differences in purpose and language of expression, we can on an abstract level observe many similarities — separation of interface and implementation, declaration of exported and imported elements, etc. As already noted in the Introduction, the same holds for the component models used in research and industry.

To capture these common aspects is the main motivation for creating meta-models above the component models. Several such a-priori and derived meta-models already exist [OMG02f, OMG02h, OGJ02, Ras02]. They place their emphasis on the technical aspect of components, mainly in order to develop frameworks for component interoperability.

The ENT meta-model which we have developed and describe here is different and unique — its motivation is to capture the common component characteristics from the user's point of view. In order to do so, it embodies selection and structuring mechanisms that reflect the way people 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; see Section 2.3.3 below. For brevity, it is referenced as "the ENT model" in this thesis.

The primary purpose of our meta-model is to enable analyses of component properties (understanding component purpose and usage, comparison for determining type differences), visualisation that helps in these analyses, and synthesis of component specifications (development of new features in specification languages). This reflects our opinion (presented in Section 1.1.1) that meta-models should encompass foreseeable future developments.

Moreover, the model's design allows our method of component comparison (described in the next chapter) to be defined on the meta-model level, rather than just of a single component model. Its subsequent use in component substitutability checking and revision identification (chapters 5 and 4) then provides a meta-level approach to these issues.

In the text of this chapter, we first discuss the reasoning that backs the ENT model's design as a result of analysing important component models and modular programming languages (sections 2.1 and 2.2). Section 2.3 presents the model itself by defining its constituent parts. Section 2.5 gives some hints on the possible uses of the model, and Section 2.4 shows its implementation for the SOFA component framework.

2.1 Elements of Component Interface Specification

When we study the the purpose of components and modules[1], and their realizations in various languages and systems, we note both similarities and differences. Let us review several comparative studies, component models and modular programming languages in order to analyse these similarities.

Note: We do not include connectors in our analysis, even though they play key roles in some architectural descriptions [S+95, MT00] and component models [BP01]. The reason is that, in correspondence with many researchers, we view them simply as special-purpose components. There is thus no need to treat connectors in a special way for our purposes.

2.1.1 Comparison of Component and Modular Systems

The high-level similarities of various systems, as studied e.g. by Shaw [S+95] or Medvidovic and Taylor [MT00], concentrate around the principal purpose of both modules and components which is information hiding. Both abstractions employ the key concept of *interface* to declare the features made available for use in inter-component communication, and sometimes also the non-functional properties governing its correct usage. Further, different parts of the interface play different roles in their interactions. This fact is exemplified by the separation of provided and required features in software components [Szy98].

The SOFA [PBJ98] and CORBA [OMG02f] component models are the primary component models targeted by our work. As such they are described in various places of this thesis, and we will mention here only their key characteristics. The SOFA model offers clearly defined component fea-

[1] In this section, we will consider both "components" as defined by Szyperski [Szy98] and "modules" as understood by Parnas [Par72] and module interconnection languages (MILs) [PDN86, SEI97].

tures — provided and required interfaces, configuration properties, and behaviour protocol (expresses the correct ordering of interface methods). The CORBA component model adds the notion of events for asynchronous component communication, plus keywords for specifying arity of interfaces and events. It is a key model because of its industrial importance. Both frameworks use simple to analyse IDL language which makes it easy to analyse component properties statically.

The JavaBeans [Sun97] and Enterprise JavaBeans (EJB) [Sun01a] component frameworks are also industrially important. In principle the components provide Java interfaces, can communicate by sending/receiving events, and be configured by properties. However, there is a difficulty in finding these features in a bean's interface because the models use naming conventions and library interfaces to designate them in Java source of the components. (The problems of modeling JavaBeans are treated in detail in Appendix A.3).

Rapide [LV95] and UniCon [S+95] are ADLs whose purpose are specifications of architectural structures, in the former case executable ones. Components are specified by an `interface` type which contains signatures of provided and required functions, observed and generated events, interface instances ("service" in Rapide, predefined "player" types in UniCon) provided or required by the component, behavioural and non-functional specification (Rapide allows to describe event and function ordering in a protocol-like notation, state transitions description, and declarative constraints). The component models are quite rich, UniCon in player types, Rapide in semantic descriptions. However, they exhibit lower abstraction level with functions as first-class component interface elements.

Han [Han98, Han99b] has developed a fairly rich component model aimed at easier configuration of components for different usage scenarios. A component interface in this model consists of a signature (sets of operations, events, and properties) plus description of semantic constraints (both per-element like pre/post conditions or property ranges, and per-relation like behaviour protocol). Lastly, "illities" (security, performance, reliability, ...) can also be specified for a component. A unique feature is the specification of required features on a per-role basis (rather than for the whole component interface). The model is also interesting in its explicit definition of semantic and quality-of-service properties, some of which express the link between provided and required services. The only drawback from our point of view is that the interface parts are too fine-grained, thus unsuitable for large components.

The Fractal component model specification [C+02, BCS02] includes a definition of its foundational meta-model. Components called *kell*s have interfaces (access points for sending/receiving signals) and a specification of behaviour as a set of transitions where $transition = (kell^{original}, signals_{in},$

$signals_{out}$, $\{kell_i^{resulting}\}$). The meta-model does not include properties (for kell configuration) although the related ADL for the Fractal concrete model [Fra03] uses them. The meta-model allows hierarchical composition of kells. The concrete model (implemented by the Fractal framework) differs from this source meta-model in several aspects: the component specification does not include the behaviour, the signals are implemented as Java methods, and each interface has a set of associated *tags* which specify its role, necessity of presence at run-time, and binding model.

One of the latest works in the field is the ArchJava language [ACN02b, ACN02a]. Its aim is to bridge high-level architectural descriptions with the actual executable code of its components, placing emphasis on enforcing integrity of the system. The relatively simple component model of ArchJava is unusual in that each port (the basic interface element) contains both provided and required items, i.e. method signatures. While this may reduce the number of interfaces, in our opinion it makes it harder to understand (and analyse) the behaviour of a component and much complicates the desired decoupling of exported and needed parts of component interface.

Modules (e.g. Ada packages [Ada95]) also provide a separate interface, usually consisting of a set of exported functions, variables and data types. Apart from the granularity of interface parts, they differ from components in several ways. Their specification is tightly bound to a particular programming language while components tend to use language-independent IDLs. Modules are primarily units of source code partitioning (components are often binary units), and thus units of compilation and linking, of otherwise monolithic applications. Components are additionally independent units of application composition, distribution and deployment.

The result of this brief analysis of several component models is that overall, each uses a slightly different terms for the same or very similar concepts. We can therefore distill the most basic concepts of their component interface specifications into high-level terms, similarly to other existing meta-models [GMW97, Ras02, OMG02h, C$^+$02, SA02]. When refined by additional properties, they can be mapped to the individual parts used by the different models. The following section defines these high-level terms in the scope of component specification.

2.1.2 Component Specification and its Elements

The *specification* of a component interface is a formal or semi-formal definition of the capabilities that cross the encapsulation barrier of the component. In this thesis we always assume that the specification uses a language with formal grammar (regular or context-free). Such specification can be utilised in many different ways, from documentation and evaluation purposes (application developers) through comparison (done by tools) for the

purpose of linking or interconnecting components, to the automated generation of (skeletons of) implementation source code.

System	Features	Quality attributes
Ada	object, number, subprogram, package, exception	*none*
ArchJava	port, method, field, extends, implements	*none*
COM	interface	*none*
CORBA (CCM)	facet, receptacle, sink, emitter, publisher, attribute	*none*
Fractal	server interface, client interface	transition set
Han	property, operation, event	interface constraint, role constraint, configuration constraint
JavaBeans	method, property, event fired, event received, extends, implements, import	*none*
Rapide	function, action, interface	behaviour, contraint, transitions
SOFA	provided interface, required interface, property	protocol
UniCon	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile, ReadNext, WriteNext, StreamIn, StreamOut, RPCDef, RPCCall, RTLoad	RecordFormat, Library, EntryPoint, Priority, Processor, SegmentDef, TriggerDef, RPCTypesIn, RPCTypedef

Tab. 2.1: Component features and qualities in several systems

The resulting correspondence between the specification and the final component is very important for our work. It means we can base our reasoning about the component (in the black-box form used for distribution), its features and properties on the information contained in its specification.

The specification of a given component can be modelled as a set of elements which define its capabilities. From the survey analysis of the previous subsection, summarised in table 2.1, we can distill common denominator abstractions of these elements. In doing this, we abstract away not only from the particular interface specification languages with their syntax and

type systems, but also from the individual characteristics of individual component models.

The following informal definitions describe the "functional" and "non-functional" kinds of elements.

Definition 2.1.1 (Feature) *A* component feature *is a minimal complete element of the component specification which (1a) is needed to establish its language type or (1b) is involved in the interactions between the component and its environment, and (2) is treated as an atomic unit when the component type information is constructed or its associations used for the interactions are established.* ■

This definition should correspond to the intuitive notion of a feature as "something sticking out of the component interface". Ideally, features are named and referenced by name. Examples of features are an IDL interface of a COM component, an event sink of a CORBA component, a log file created and written to by a web server module, etc. See Table 2.1 on the page before for examples of elements in several specification languages.

Definition 2.1.2 (Quality attribute) *A* component quality attribute *is a minimal complete element of the component specification which declares a single complete non-functional property of either the whole component or of a subset of its features.* ■

Again, the definition should correspond to the natural understanding that qualities often provide information about the implementation of features. Typical instances of quality attributes are semantic descriptions, for example invariant expressions as in Eiffel classes, frame protocols in SOFA, state transition descriptions in Rapide [LV95], the "illities" described in [Han99a], or quality of service indications [FK98]. Note that we do not require that qualities be named, in accordance with common usage.

Using the terminology of this section, features and qualities will be collectively called specification *elements* in the rest of the thesis. The following sections defines the fine-grain properties of elements that we can use to distinguish them further.

2.2 Classifying Specification Element Properties

The classification of specification elements into features and qualities is rather coarse. However, it is quite easy to formulate the characteristics that we as humans see when observing the various elements of the specification. Figure 2.1 on the facing page shows some of these characteristics schematically. From these characteristics we can create a formalised classification of element properties. Consequently, the features and qualities can on a finer level be distinguished in several orthogonal ways.

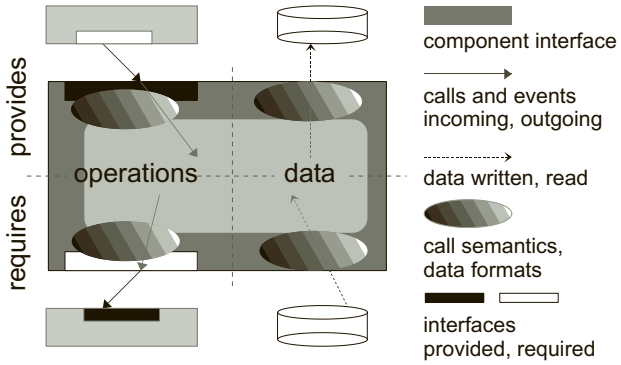

Fig. 2.1: Parts of software component interface

2.2.1 Common Properties of Elements

A fundamental distinction is by what we call the *kind* of the element. The *operational* features and qualities describe or are used to invoke functionality. The *data* features describe (sets of) data which the component exchanges with its environment. There can also be features and qualities which contain a mix of these two characteristics.

An orthogonal classification attribute is the *role* in component interactions. Each component *provides* features which its clients can use to invoke its functionality and which thus represent the purpose of the component. On the other hand, the component may *require* the connection to or existence of some features in its environment for correct linking or execution. Some kinds of elements (e.g. the behaviour protocol in SOFA components) provide *ties* between these two parts of component interface, i.e. exhibit both provided and required roles. This distinction of element roles is explicit in component-based systems [PBJ98, OMG02f, M$^+$95] and in many modular programming languages [Ada95, Sof01].

From the point of view of the specification language, it is sometimes important to distinguish the language *construct* of the element declaration. In most cases, the element will define an *instance* of a type; in rare circumstances (e.g. properties in UniCon) also a *constant* value. Sometimes however, the element will contain just type information in the form of *type definition* or *type reference*. Then its contents is not accessible via an identifier within the scope of the component declaration – as, for example, the `supports` interfaces of CORBA components.

In some systems, element's necessity of *presence* can be designated. Ordinarily an element is *mandatory* which means that it must be present on the component interface at run-time. On the other hand, *optional* elements may be missing and still the component conforms to its specification. An

example of a component model which uses this distinction is the Fractal framework [C+02].

Next, each feature may have different *arity* with respect to the bindings on that feature. We differentiate two cases — *single* arity for 1:1 bindings, and *multiple* for 1:many links. An example of using arity are CORBA Component Model's event publishers (which allow multiple sinks) and emitters (for one-to-one communication).

Lastly, we can differentiate features and qualities according to their usage during or applicability to different stages in component *lifecycle*. Current practice and research [Sun01a, PBJ98, LC99, LR01b] distinguishes several such stages: *development* for correct compilation, static or dynamic linking, and packaging (when e.g. component assemblies are created from individual pieces), *assembly* (or design) for the integration stage of creating component interconnections in a visual tool and configuring the composed application, *deployment* which covers the phase of (re)configuring the application in the actual deployment environment, and *run-time* stage which exercises interface elements during application execution for inter-component communication. Again, some elements may be relevant in more phases of the lifecycle – for example provided interfaces of a CORBA component are useful in compile-time, design-time as well as run-time stages.

2.2.2 The ENT Faceted Classification System

We now formalise these findings in a classification system which uses the faceted classification approach [PDF87]. The system has seven facets — called *dimensions* — suitable for the classification of component specification features and quality attributes from the human perspective, as described above. However, the number of dimensions is not fixed and the classification is open to future developments.

The term space of each facet is represented as a set of *Identifiers* described by the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`. We use the set $Id^{spec} = \{nil, na, nk, all\} \subset Identifiers$ of special identifiers: the nil value denotes an empty identifier, the na value is used in the cases when the given dimension is not applicable to the given feature or quality; the nk value (not known) is used when the class cannot be clearly determined; the all value is used as a substitute for a conjunction of all the user-defined terms of the dimension.

Definition 2.2.1 (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,

- $Kind = \{operational, data\} \cup Id^{spec}$ *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* $dim_i \in Dimensions_{ENT}$.

■

We should note that there may be systems which can unambiguously distinguish interface elements using a subset of the ENT classification. For example, the $\{Nature, Kind, Role\}$ facet collection would be sufficient for the current SOFA component model. However, the presented facet collection provides for a generality and makes the classification applicable to a wide range of modular and component specification languages.

2.3 The Model: Elements, Traits and Categories

This section defines the structures which form the ENT model as such. It starts with the lowest level, describing the specification elements. Their definition uses the ENT classification system to capture the human-driven understanding of elements. After that, the aggregation constructs for building more abstract specification parts are defined.

We will illustrate the newly introduced concepts on an example of a SOFA component with specification given in Figure 2.2. The example uses a slightly modified CDL grammar which allows the *readonly* modifier for properties; see Section 2.4 for the reasons.

```
frame FAddressBook {
  readonly property long maxSize;
  requires:
    ::sys::IFileAccess files;
  provides:
    IAddressBook book;
  property short defaultSortOrder;
  provides: IAddressSearch search;
  protocol: // abbreviated
    (?book.addPerson … )*
};
```

Fig. 2.2: An example SOFA component specification

2.3.1 Specification Elements

To be able to analyse and manipulate the specification of a component interface, we need to handle the parts of the specification which correspond to the features and qualities as defined in the previous chapter. They describe the smallest elements of interest in our model.

Definition 2.3.1 (Specification element) *A* specification element *e found in the specification of a component M written in language L is a tuple $e = (name, type, tags, inh, metatype, classifier)$ where $name \in Identifiers \cup \{nil\}$, $type \in L$ is a language phrase, $tags = \{(name_i, value_i)\}, value_i \in L$ is a (possibly empty) ordered set of named language phrases, $inh = (i_1, i_2, \ldots, i_n); n \geq 0, i \in Identifier$ is an ordered tuple of identifiers, $metatype \in Identifiers$ and $classifier = (ce_1, ce_2, ..., ce_D); ce_i \subseteq dim_i$ is an* ENT *classifier.*

The (polymorphic) function $Elements : L \rightarrow \{e\}$ returns the set of all specification elements contained in the specification of a component M. ■

A specification element represents a complete information about one feature identified by language *name* and *type* or of one component-wide quality attribute; the *name* may be empty (*nil*).

The *tags* item contains a set of phrases with additional declarations pertaining to the particular element (not to its type). Tags often describe the semantics of the element. Contents of *tags* is ordered alphabetically by the $name_i$ part of the pairs. These names can be chosen arbitrarily but their relation to the corresponding phrases must be consistent for the given language L. We use the "associative array" notation for access to tags: $e.tags[name]$ denotes the set of values for a tag with name *name* in an element e.

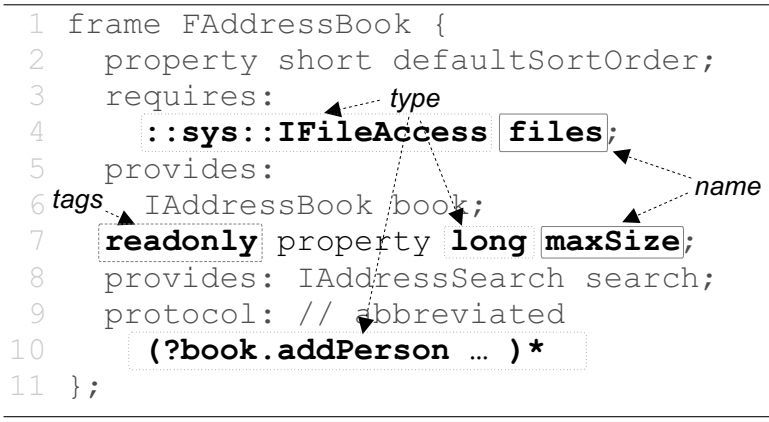

Fig. 2.3: Example elements in component specification

Tags serve as an aid if one needs to e.g. precisely compare two elements[2] or re-generate valid source code for the element. A correct ENT parser of the language shall provide the appropriate default value for each tag that is not explicitly declared in the specification (e.g. for SOFA CDL properties, the *access* tag values are $\{readonly, readwrite\}$ with the second being the default value).

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. For example, an element inherited from a component `::core::foo::Bar` will have $inh = (core, foo, Bar)$. If the element is not inherited, the tuple is empty. This inheritance indication is necessary for distinguishing directly specified elements from the inherited ones, e.g. when the ENT data is used for component visualisation.

These four parts (name, type, tags and inheritance indication) of the specification element can be derived directly from the specification source code. Operations on them are subject to the syntax and typing rules of the language L used for component specification – in other words, this model is parametrised by the specification language for which its concrete application is sought.

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

[2] For example, in a comparison of `final static int x = 5` against `int x`, the `final static` keywords stored in the *tags* part represent an important semantic information.

human-perceived meaning of its phrases. The purpose of such effort is to create a complete but minimal set of meta-types and classifier combinations which the specification elements in L can have, to reliably distinguish them. Once this analysis is done, it is relatively easy to create the appropriate supplementary code in a suitable parser/analyser of the specification language L. Such code fills the element data structures and possibly creates the aggregated structures described further below.

Completeness of the element means that it includes all the information about the feature or quality attribute contained in the specification (with respect to both the language declarations and the classification dimensions) even if this information is not available in a single language phrase. For example, in SOFA CDL an interface (a component's element) is contained in either the `provides` or `requires` section but these keywords are not part of the element declaration itself.

2.3.2 Traits of Elements in Component Specification

As was said at the beginning of this chapter, we would like our model to handle the declarations in the component interface specification in a manner natural to our human perception. In particular this involves grouping the specification elements into more abstract concepts — characteristic traits of the component.

Definition 2.3.2 (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 $name \in Identifiers$, $metatype \in Identifiers$, and $E \subseteq Elements(M)$ 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.*)*

Function $Traits : L \rightarrow \{t\}$ returns a complete set of traits in the specification of component M such that $\forall t_i, t_j \in Traits(M) : t_i.name \neq t_j.name$ and $\forall e \in Elements(M)\ \exists t_k \in Traits(M) : e \in t_k.E$.

Function $Elements : trait \rightarrow \{e\}$ returns the elements contained in a trait; that is $Elements(t) = t.E$. ■

A trait in a concrete component model can then be seen as a special kind of *element type* — 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. Figure 2.4 on the next page shows examples of traits in the SOFA component model together with concrete contents for a particular component.

The consequence of this definition is that, for a given concrete component model the set of traits is fixed because the model defines a finite

Name	Elements
Properties	{ (defaultSortOrder, short, . . .), (maxSize, long, . . .) }
Protocol	{ (*nil*, ?book.addPerson . . . , . . .) }
Provisions	{ (book, IAddressBook, . . .), (search, IAddressSearch, . . .) }
Requirements	{ (files, ::sys::IFileAccess, . . .) }

```
frame FAddressBook {
  property short defaultSortOrder;
  readonly property long maxSize;
  protocol: // abbreviated
    (?book.addPerson … )*
  provides:
    IAddressBook book;
  provides: IAddressSearch search;
  requires:
    ::sys::IFileAccess files;
};
```

Trait colour coding:

properties

protocol

provisions

requirements

Fig. 2.4: Traits of the `FAddressBook` SOFA component

set of the metatypes and classification properties of specification elements. This in practice means the number of traits and thus the complexity of the ENT representation is small. For example, the SOFA system provides the component specifier with just four traits of elements (see Section 2.4 later in this chapter).

For the purposes of analyses described in later chapters, we may assume that an order is defined on traits for the given specification language. By default, traits are ordered lexicographically by their names. An advanced option would be to define an order on the classification dimensions, which would incur an ordering of traits. We should also note that not all combinations of element classification dimension values need to be used (or make sense) in trait definitions for the given component model. In such cases we can use partial classification like $(quality, na, nk, runtime)$, providing as much information as practical.

2.3.3 Categories of Traits

Although traits provide a useful grouping of specification declarations, for an architectural level view of a component their granularity is still too small. In high-level analyses of software we often come into situations where would

like to handle for example "all provided features" as a single group. Such groups are called categories in our model.

Definition 2.3.3 (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* $name \in Identifiers$ *and* $T \subseteq Traits(M)$ *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.*

Function $Traits : category \rightarrow \{t\}$ *returns the set of traits contained in a category; that is* $Traits(K) = K.T$. *Function* $Elements : category \rightarrow \{e\}$ *returns the set of specification elements contained in the traits of a category; that is* $Elements(K) = \bigcup_{i=1}^{n} t_i.E$ *where* $t_i \in K.T$ *and* $n = |K.T|$. ■

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. The category set provides category definitions such that each trait from the component trait set belongs to at most one category of the set. Note that the category set need not cover all specification elements of a component.

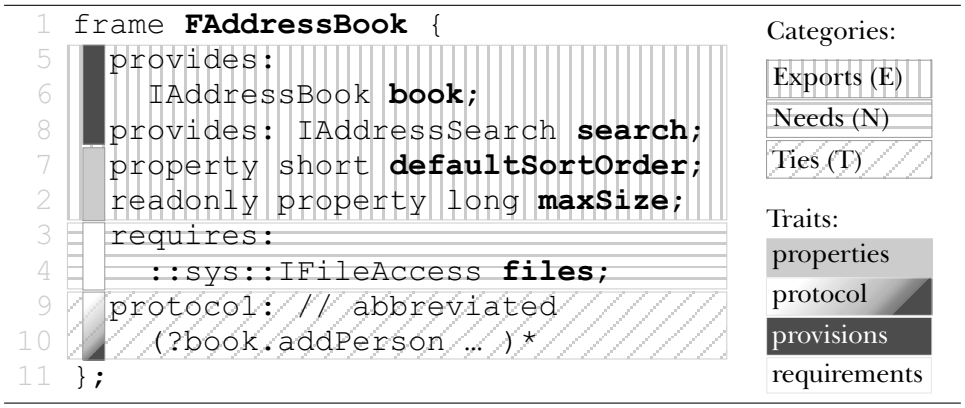

Fig. 2.5: Example categories in the `FAddressBook` SOFA component

For the purpose of analyses described in later chapters, we may assume that an order is defined on categories for the given specification language. By default, categories are ordered lexicographically by their names.

From the definition above it follows that, 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

F-D (Functionality-Data)

$f^F = \lambda C.\ (C.kind = \{operational\})$
$f^D = \lambda C.\ (C.kind = \{data\})$

Fe-Q (Features-Qualities)

$f^{Fe} = \lambda C.\ (C.nature = \{feature\})$
$f^Q = \lambda C.\ (C.nature = \{quality\})$

S-Q (Services-Qualities, or the Server view)

$f^S = \lambda C.\ (C.nature = \{feature\} \wedge\ C.kind = \{operational\} \wedge$
$C.role = \{provided\}\ \wedge\ C.lifecycle = \{runtime\})$
$f^Q = \lambda C.\ (C.nature = \{quality\} \wedge\ C.role = \{provided\} \wedge$
$C.lifecycle = \{runtime\})$

Fig. 2.6: Example category sets

traits, then give us completely different views of the component. They thus can match different needs of component users or developers. A key category set, the E, N, T set, is defined below. Other category sets that can be useful in the ENT model applications are shown in Figure 2.6.

It is worthwhile to note the subtle but important difference between trait and category definitions. Traits require that the element classifier be *complete and equal* for all trait's elements; this effectively defines the trait classifier. Categories may on the other hand group elements based on a *subset* of the ENT classifier.

Another notable characteristic of categories is that they group elements of different meta-types. They therefore allow operations on the component specification based on its human understanding (represented by the classification system described above) rather than on the syntax or the typing system of the language.

2.3.4 The *E*, *N*, *T* Category Set

The set of categories most useful for our work is obtained by using the $role$ dimension. It is based on a view of the component interface which developers (and some languages as well) use very often — that of elements provided for others to use, of those required from the environment to ensure proper functionality, and those which express the bindings of these two sets together (see Figure 2.7 on the next page).

This way we get three categories, "Exports", "Needs" and "Ties", which also give the name to our model of component interface:

$E = (exports, f^E, T^E)$ where
$f^E = \lambda C.\ (C.nature \in \{feature, quality\}\ \wedge\ C.role = \{provided\})$;

$N = (needs, f^N, T^N)$ where
$f^N = \lambda C.\ (C.nature \in \{feature, quality\}\ \wedge\ C.role = \{required\})$;

$T = (ties, f^T, T^T)$ where
$f^T = \lambda C.\ (C.nature \in \{feature, quality\}\ \wedge\ C.role = \{provided, required\})$.

Functions $Exports : L \rightarrow \{e\}$, $Needs : L \rightarrow \{e\}$ and $Ties : L \rightarrow \{e\}$ return the sets of elements contained in the respective categories, that is for $M = \{E, N, T\}$ they return the $Elements(E)$, $Elements(N)$ and $Elements(T)$ sets.

In subsequent work we use the following natural ordering of these categories: $E > N > T$. It is derived from the importance of elements from the developers point of view, in which the exported elements are the most interesting ones (they tell us how the component can be used). The dependencies come next (what do we have to give it so that it can provide its services), and the ties becomes important only after we have the previous two categories settled.

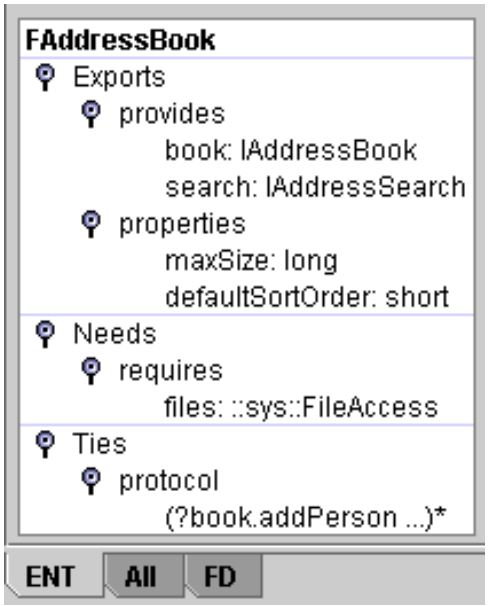

Fig. 2.7: A SOFA component structured by the E-N-T categories

In other words, the speciality of this category set is that it captures the different aspects which each part of the interface (and consequently the

corresponding specification trait) has from the point of view of the component interconnections. It is thus a formalisation of the general idea of what can be found on a component as presented in Figure 2.1. Moreover, it is a crucial structure for the definitions of compatibility presented in [Bra01b] and in Chapter 5 of this thesis.

2.3.5 Restricted Elements and Categories

For the purpose of comparison of ENT structures, we need to define the concept of restriction on elements and categories. The motivation are operations on elements in the T category which often reference other elements from the component interface (by their names). This is the case of e.g. SOFA behaviour protocols or Eiffel invariants.

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 (see Chapters 3 and 5).

Definition 2.3.4 (Element restriction) *The element $e' = e/A$, e* restricted by alphabet *A, is an element for which it holds*

1. *the declaration of e' uses only identifiers from the set A, and*

2. $e <: e'$.

A category restriction *by alphabet A (denoted K/A) is a category such that elements $e' \in Elements(K/A)$ are obtained as elements of K restricted by A.* ■

Element restriction is based on language restriction: the declaration of e (and consequently the $type$ and $tags$ parts) is a sentence in a language $L_{e.metatype}$ generated by the subset of the specification language grammar which corresponds to the metatype of e. The set A restricts the alphabet of the language, resulting in $L' = L_{e'.metatype} \subset L_{e.metatype}$. Thus the declaration of e' is "contained in" the original sentence e, expressed by the subtyping relation.

An example of restriction on elements is the protocol restriction operator defined for SOFA behaviour protocols in [PV02] or trace projection in [FW00]. In the rest of this thesis, we assume that the specification language to which the ENT model is applied supports element restriction. For languages that do not fulfil this condition (albeit in just some traits) the notions of substitutability defined in Chapter 5 cannot be applied.

2.4 ENT Model of SOFA Components

In this section, we provide the technical details of the ENT representation of SOFA CDL constructs. We start with the mapping of these constructs to elements and traits, and then present experiences gained from the implementation of a trait-separating parser.

Note: For explanatory purposes, we took the liberty to slightly modify the SOFA CDL (component description language) used in the examples throughout the thesis: the `readonly` modifier can be added to the *property* component element. By doing so we do not lose the applicability of the presented results to the SOFA framework; in fact, since the original CDL is simpler, the comparison of its component specifications (Chapter 3) as well as all other methods are easier to implement.

2.4.1 Mapping of SOFA CDL Constructs to ENT Elements

The following paragraphs define the elements that can be found within a SOFA component specification, and how the parts of the ENT element structure are obtained from the specification source. Their examples shown in Figure 2.8 refer to the component CDL specification source given previously (see Figure 2.2 on page 22).

frame The component as a whole is represented by the `frame` construct. Thus the specification element set of a concrete component will always be a representation of a container named after the frame.

Abstract syntax:

frame ::= ***frame*** *name '{' (provisions requirements property)* protocol '}'*

property These elements represent data structures that can be used to configure the component.

Abstract syntax: *property ::= [ro]* ***property*** *type name ';'*

ro ::= ε | ***readonly***

The *name* and *type* of these elements are based on the syntax as shown in the grammar rule. The *metatype* of these elements is *property*. The *tags* structure is defined as follows: $tags = \{access\}$ where $access = (\text{“}access\text{”}, \{readonly, readwrite\})$. The value of the *access* tag is *readonly* if *ro* = ***readonly***, and *readwrite* if $ro = \varepsilon$.

The *classifier* of the property elements is $(\{feature\}, \{data\}, \{provided\}, \{instance\}, \{mandatory\}, \{multiple\}, \{all\})$. Properties can be used in any lifecycle phase: in development for generating code, in assembly to connect components and/or to set globally applicable component properties, in deployment to access/set component properties applicable to the particular

maxSize
$name$ = maxSize,
$type$ = long,
$tags$ = ($access$,readonly),
inh = $\emptyset$,
$metatype$ = property,
$classifier$ = ($\{feature\}, \{data\}, \{provided\}, \{instance\}, \{mandatory\}, \{multiple\}, \{all\}$)

files
$name$ = files,
$type$ = ::sys::IFileAccess,
$tags$ = $\emptyset$,
inh = $\emptyset$,
$metatype$ = interface,
$classifier$ = ($\{feature\}, \{operational\}, \{required\}, \{instance\}, \{mandatory\}, \{multiple\}, \{development, assembly, runtime\}$)

protocol
$name$ = nil,
$type$ = (?book.addPerson ...)*,
$tags$ = $\emptyset$,
inh = $\emptyset$,
$metatype$ = protocol,
$classifier$ = ($\{quality\}, \{operational\}, \{provided, required\}, \{type\}, \{mandatory\}, \{na\}, \{development, assembly, runtime\}$)

Fig. 2.8: ENT representation of the selected elements in the specification of the `FAddressBook` SOFA component

application, at runtime to access and manipulate properties.

protocol This element represents the run-time semantic of the component in terms of valid sequences of interface method calls.

Syntactically a protocol is a regular expression over interface method names, with operators for specifying sequential and concurrent invocation. Details are listed in [PV02].

The *name* part of the element is empty, the *type* is the declaration of the protocol itself. The *metatype* of these elements is *protocol*, and the *tags* structure is empty.

The *classifier* of the protocol element is ($\{quality\}, \{operational\}, \{provided, required\}, \{type\}, \{mandatory\}, \{na\}, \{development, assembly, run-$

$time\}$). Protocol is not useful for configuring component in the deployment phase as it describes run-time semantics; during development control code can be generated from it, during assembly it can be checked when setting up interconnections.

provisions These elements represent the provided interfaces of the component.

Abstract syntax:

provisions ::= ***provides****: (interface)**
interface ::= type name ';'

The $name$ and $type$ of these elements are based on the syntax as shown in the grammar rule. The $metatype$ of these elements is $interface$, and the $tags$ structure is empty (CDL provides no constructs to modify the semantic information about provided interfaces).

The $classifier$ of provision elements is ($\{feature\}$, $\{operational\}$, $\{provided\}$, $\{instance\}$, $\{mandatory\}$, $\{multiple\}$, $\{development, assembly, runtime\}$) — interfaces are not used during deployment time, as interconnections have been fixed during the assembly phase.

requirements These elements represent the interfaces that the component needs to use during its execution.

Abstract syntax:

requirements ::= ***requires****: (interface)**
interface ::= type name ';'

The $name$ and $type$ of these elements are based on the syntax as shown in the grammar rule. The $metatype$ of these elements is $interface$, and the $tags$ structure is empty (CDL provides no constructs to modify the semantic information about required interfaces).

The $classifier$ of requirement elements is ($\{feature\}$, $\{operational\}$, $\{required\}$, $\{instance\}$, $\{mandatory\}$, $\{multiple\}$, $\{development, assembly, runtime\}$) — interfaces are not used during deployment time as interconnections have been fixed during the assembly phase.

2.4.2 Trait Definitions

The SOFA CDL in its current shape has four traits of elements in component frame specification: "provides", "requires", "properties" and "protocol". Their full specification is given in Appendix A.1.

The elements of the `frame` declaration as specified by the language syntax and analysed above actually map one-to-one to the traits. This is merely a coincidence due to the design of the SOFA CDL; other languages may not have such clear mapping.

2.4.3 How to Find Traits

As can be seen, the definitions of specification item and trait are not so exact that they would allow us to build an algorithm for automatic trait separation in the given specification language grammar. This is caused by the fact that traits are to a large extent a semantic concept that needs human interpretation. However, the definitions help to create a parser that can build trait contents for a given component specification.

One approach when creating such a parser is to find the individual elements in the specification, assign classification terms to them, and name the traits that result from the unique combination of classification terms and meta-types.

An alternative approach is to take the specification grammar, and assign traits directly to non-terminals. The guide is intuition in the first place (separate the different kinds of elements that can be found in the specification). In the next step, the classification values are assigned to the traits.

While the first approach is closer to the definitions found in this chapter, the second one feels more natural. Our experience from building an ENT parser for SOFA CDL shows that it is actually very straightforward. Once we became familiar with the classification system and the concept of traits, it was very easy to find the grammar nonterminals and rules that correspond to them. The following sample of the CDL grammar rules shows this case:

```
frame_dcl ->
 "frame" identifier [ uri_attributes ] "{"
 (   ( frame_provides   [ frame_requires ] )
   | ( frame_requires   [ frame_provides ] )
   | ( frame_properties [ frame_properties ] )
 )+ "}"
```

In many cases the prospective trait name or element metatype can be derived from a counterpart nonterminal symbol in the grammar. Sometimes in may also be advantageous to modify the grammar so that the elements can be easily parsed out. The element name, type and/or type declaration, and tags can be then easily found in the rules.

2.5 Applications of the Model

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. In other words, the model gives hints

about what is useful and possible rather than merely about what is currently implemented.

The ENT model is however general enough to serve other different purposes, making it interesting for the component developers (mainly in component understanding) as well as their tools (automated component comparisons).

In this section we briefly survey several potential uses of the model. The applications that our work is directly concerned with — component comparison, versioning and substitutability — are covered in detail in Chapters 3, 4 and 5.

2.5.1 Applicability to Frameworks and Technologies

The SOFA [PBJ98] and CORBA [OMG02f] component models were the primary sources of inspiration and verification during the development of the ENT model. The particular features of SOFA which make it appealing to ENT modelling are the simple and readable CDL component specification language, plus the fact that it offers elements in all three key ENT categories and exercises several classification dimensions.

In a straightforward extension, the model is well applicable to the CORBA Component Model (CCM [OMG02f]). While CCM does not use any quality attributes in component specification, its has constructs for several kinds of features, adds the notion of events (which belong to the operational features) and element arity Its IDL is simple to analyse and the model has the advantage of high industrial importance. The example shown in Figure 3.6 on page 54 shows a visual representation of a CORBA component using the ENT model. Appendix A provides the definitions of traits applicable to CCM, as well as further examples.

In a similar manner, other component frameworks or modular systems that use an IDL-like language for the specification of component interface [Mic95, T$^+$96] can utilise the ENT model.

The generality of the model makes it possible to apply it also to components specified in other kinds of languages. In particular, the JavaBeans and EJB [Sun97, Sun01a] component models would benefit for such modelling. The application of the ENT model here is however seriously hindered by the primary deficiency of these platforms — the non-existence of a language for external formal specification of JavaBean and EJB components. To reconstruct the component specification and the classification properties of its elements for JavaBean and EJB components, a sophisticated analysis of method body code has to used (see Appendix A.3 for detailed explanations). From this case study we conclude that it is difficult to formally model components in frameworks which use source code and naming/design conventions as the primary means of their specification.

Based on our experiences, we expect the ENT model should be applicable also to some well-designed modular programming languages, such as to Delphi units, Ada packages. Again, finding and classifying elements of modules written in programming languages can be complicated to a different degree. Especially the case of languages with preprocessing constructs and looser grammar or type systems (e.g. the C language) would require a modified grammar and use of heuristics (e.g. to represent module dependencies with $role = required$ by the `#include` directive).

2.5.2 ENT-based Component Visual Representation

The ENT representation of software modules and components can be helpful in human understanding of the software. This is a direct result of the design of the model achieved mainly via its classification system. The three levels of interface structuring – elements, traits and categories – can present the interface in different levels of detail and in various views oriented towards different aspects. For an example of how this can look like, see Figure 2.9.

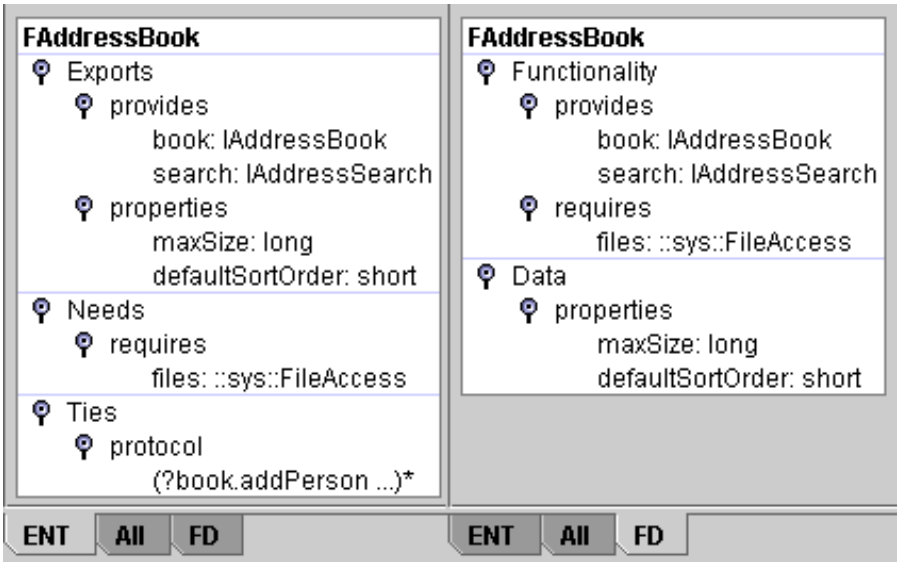

Fig. 2.9: An ENT-based visual representation of a SOFA component

The aim of this use of the model is to provide for easier and less error prone software evaluation, thus facilitating tasks such as visual design, reengineering and maintenance. The primary envisaged application in this respect is in visual design tools. The use of the ENT model in this context would result in software presentation in user terms rather than (as common now) in language terms.

Especially for visual development with components, the developer can then have the component appearance affected by category selection. Such view parametrisation can have several applications:

1. In assembly (binding) of components into applications, e.g. in solving the tasks "now I want to see just the links between the provided and required ifaces" in CORBA components, or "let's see how events propagate" by showing just event sinks/sources with event names.

2. In search/evaluation, the model can provide a tree view of a single component in which the user can expand category, trait, and specification item contents to trace down a particular feature. For example, in searching for the `animationRate` property of the `Juggler` JavaBean, the developer would use the Operational-Data categories and unfold, in sequence, the "Data" category and the "Properties" trait, to find the specification.

3. For component deployment and application assembly, it would be of advantage to use a visual representation of the difference of two components as shown in Figure 3.6 on page 54. This would help in the evaluation of module or component substitutability.

4. Similarly in maintenance or servicing, the maintainer can test change propagation in "what-if" scenarios using the *role* of features — change is allowed if the proposed modification is an extension of the provided or a reduction of the required features.

2.5.3 Assistance in Component Search and Retrieval

Another use of the ENT model is in library search and retrieval, which will become of greater importance with the growing component market and consequently the number of components available. There are two primary problems in this area: the options in formulating the search query, and the precision and extent of the result.

The ENT model can assist in both cases so that the queries are more precise and the result set is more precise (narrower). This can be achieved by augmenting the search methods (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.

The user can, for example, restrict a signature-based search for a SOFA component to match just signatures of the provided operational features (within interfaces in the *provides* trait). This will eliminate the false hits caused by components that require the same interface, which clearly is not the intent of such query.

2.5.4 Other Applications

Similarly to the use of UML in CASE tools, the ENT model visualisation can form a base for component design and development tools that produce component specification code. From a wider perspective, the low-level ENT data could also be used e.g. for translation of component interfaces between systems. For example, after extracting ENT data for a JavaBean component we could use this data to generate a CORBA IDL skeleton of a corresponding CCM component.

We also envisage that the ENT model could be beneficial for component testing, as identification of traits/categories allows separate testing of independent aspects (functionality vs. quality of service tests).

On an abstract level, the definition of the meta-model provides a guidance for specification language creators. It shows the range of component interface elements that can be specified, and provides hints of their role in component development. Thus a new language can be created by instantiating the ENT model for a particular problem or technology domain. First, a suitable set of element traits would be designed (by selecting relevant classification properties and meta-types). Then the appropriate grammar rules can be defined to create concrete syntax for the resulting features and quality attributes.

2.6 Discussion

The purpose of creating models is to abstract away details of the subject which are not interesting from the particular point of view. Therefore, care must be taken to balance simplicity and precision in the model definition. The subject of our work, software module and component systems, exhibit a great degree of variation. In short, the goal of our work may be noble but is not easy to attain.

In this section, we would therefore like to discuss in more detail the model, its advantages and weaknesses. This opens the way for further work on applications and improvements of the model, as well as in related areas.

2.6.1 Advantages of the Model

The primary objectives of the model are conceptual simplicity and close correspondence to human (primarily developer's) view on software components. These aspects can be directly counted as the model's advantages.

The simplicity lies primarily in the use of a restricted set of classification facets and other meta-data items attached to specification elements, and in straightforward rules for their grouping into traits and categories. The

model should thus be easy to understand and implement in code.

The definition of categories (via the selection functions) is independent of a particular component model or specification language. On the other hand, each component model will have its own set of traits. The ENT model therefore provides both standardisation and customisation possibilities.

The application of the model to a given module-based programming language or component framework results in a representation of components or components that is easy to visualise and comprehend. This representation can be used for forward as well as reverse engineering of component specifications. The model thus contributes to the area of program understanding.

The aggregation of specification elements, using the classification terms, into traits and categories enables purpose-specific views. For instance, an application composer can highlight operational features that are essential for run-time interactions, while an administrator who wants to upgrade a component can check compatibility by comparing the groups of exported and imported features (regardless of whether they are operational interfaces, data elements, or semantic properties).

Traits and categories group specification elements of a component even if in the source these may be written in various places (as shown in Figure 2.3 on page 23 and Figure 2.4 on page 25). Thus the interface specification can be analysed and manipulated by the meaning of its parts rather than by their place of occurrence or language type. This approach is similar to *connection protocols* described in [aJP00].

The model was designed to be very general and independent of any particular technology or specification language. It is thus applicable to many research and industrial platforms — among others to SOFA [PBJ98], C2 [T+96], CORBA [OMG02f], JavaBeans [Sun97] and Enterprise JavaBeans [Sun01a]. (Examples of ENT model definitions of CORBA and JavaBean components are given in Appendix A.) Furthermore, the model hints possible improvements in specification languages by showing the classifiers which are not paired with concrete syntactical structures.

Finally, the model is open for extensions. It was noted in Section 2.3 that the facet collection used in ENT classification is not a closed one. Should the analysis of platforms, frameworks and languages not covered by our research reveal new classification dimensions, they can be added without directly affecting the model itself. Similarly, the ordering relations for specification elements as defined in Chapter 3 can be changed, e.g. using the approach to relaxed signature matching presented by Zaremski and Wing [ZW97].

2.6.2 Disadvantages and Open Issues

The ENT model presented here has however several shortcomings that need the attention of future research. The primary problem as we see it is the difficulty of manual classification of specification elements in the given language. This problem arises because automated classification is in general a difficult problem [Bör95, ZW97], in this case further complicated by the lack of expressiveness of some specification and programming languages. Manual classification opens room to different interpretations and thus ambiguity of features and properties (e.g. along the *Lifecycle* dimension).

The second problem concerns the fact that elements are taken as atomic units without considering the details of their internal structure. This calls for a more accurate handling of the $tags$ part of the specification element. The desired effect would be achieved by defining this part as a set of pairs $tags = \{(declarations, classifier)\}$, i.e. tagging individual parts of the declaration with classifiers. This would make the model closer to reality but at the expense of readability and simplicity. In this work we opted for the simpler approach and consider declarations as monolithic, classified by its overall proximity to the classification facet terms. The use of internal structuring of declarations is reserved for future work.

Handling of component inheritance (supported by e.g. JavaBeans or CORBA Component Model) is a little complicated in the ENT model. Each element carries information about its origin in the inheritance hierarchy, but this information is imprecise should we need to reconstruct the identifiers of the component's parent(s) as specified in its type declaration. This issue is relevant in the context of visualisation using the ENT model, and would deserve a cleaner solution.

Last but not least, the implementation of the ENT model for some languages requires non-trivial amount of work. In some cases it is necessary to redesign the language grammar so that elements and traits are easier to separate. In any case its use depends on the creation of suitable parsers which extract the relevant data from the specification source. These two tasks combined pose a challenge mainly in the case of syntactically rich programming languages like C/C++ or Java.

2.6.3 A Note on Specification Languages

There are however a few problems outside of the model, in specification languages themselves, which may hinder the full use of our approach to specification modelling. The most unfortunate one is the lack of expressiveness of current specification languages. For example, while the support for the `provides` role is common, only several research and a few industrial languages allow to specify required features [PBJ98, OMG02f].

Similarly, the languages provide only a limited repertoire for specifying features of the *data* kind. The only common one are properties (named data structures used for pre-runtime configuration), but in reality software components often also depend on or create various data files and streams [SA02]. No component framework in widespread use provides support for file or stream specifications that would capture this important aspect of their functionality. Similarly, event-based communication is supported by just some, and only a handful of component models and their languages support data elements, semantic or behavioural specifications.

The result is that the model presented in this paper can easily accommodate today's specifications but is not used to its full potential. Thus our reasoning about features and properties provides hints on what can (and should) be done in terms of improving component specifications.

2.7 Summary

In this chapter we presented a novel meta-model of software components. It defines a component as a set of user-defined categories, each of which consists of a set of characteristic traits of the component's specification language. The traits contain specification elements that represent individual features and quality attributes of the component.

The structuring into categories and traits is based on a faceted classification system derived from the human-perceived characteristics of elements. Traits group elements with the same classifier and meta-type, categories group traits which conform to a user-defined classifier (called selection function) that determine each category. Thus traits are specific to a given component model whereas categories are system-independent.

Each element of the specification belongs to exactly one trait, and each trait to at most one category. This means that traits and categories are not recursive, i.e. the model creates a three-level view of the component specification. The most useful categories are obtained by grouping by the $role$ classifier which results in categories named E (exports), N (needs), and T (ties) which give the model its name.

Depending on the needs of developers, a software specification structured by the ENT model can be manipulated at various levels of abstractions and from various viewpoints. The next two chapters use such techniques to provide foundations for component versioning and compatibility assessment.

The ENT model representation has been implemented in prototype applications for the SOFA and CORBA Component models. The implementations create the data structures of traits and categories for given CDL resp. IDL3 sources, and make it possible to display the component in the

form shown in Figure 3.6 above. The CCM prototype also allows the component specification to be edited in this graphical form.

The analyses performed while developing the ENT model also show that more mature systems (e.g. CORBA Component Model, or the Rapide ADL) provide means to declaratively specify various characteristics of features and qualities which simpler approaches hide in source code. A prime example is the distinction of functions and events, which platforms like JavaBeans represent uniformly as methods.

The meta-model is applicable to a wide range of platforms, from IDL-like languages used by CORBA, COM+ or SOFA systems, through modular programming languages such as Ada or even the C language, and ending with some formal specifications (e.g. as used by the Wright architecture description language [ADG98]). The only requirement that the model places on its subject is the ability to determine primitive elements in the component/module specification and the ability to compare these elements.

Chapter 3

Analysing and Classifying Specification Differences

The possibility to see and analyse differences between similar software artefacts is important in all stages of software development cycle. During analysis and design, the developers need to evaluate existing libraries or components for possible reuse, in implementation and maintenance the programmers often want to see the changes from previous versions or need to adapt to existing interfaces when incorporating reused code.

In this chapter we present a method for describing software differences based on the comparison of the ENT data structures. The primary target of the method are black-box components or similar coarse-grained software modules. We therefore assume nothing about the availability of their source code and rely in this comparison solely on the component (interface) specifications, expressed in terms of the ENT model. In this reasoning, we assume the correspondence of the specification and implementation as discussed in the previous chapter.

The goal of our method is to provide such comparison results that are readily understandable by human users and at the same time suitable to further automated processing and analyses. The method is therefore based on finding the subset relation between corresponding parts of the specifications. The comparison results are represented using a simple yet powerful classification scheme which provides human readability and sufficient information to reason about compatibility of the components.

The text of this chapter first presents the motivation for the overall approach taken during the design of the method. Section 3.2 is the core of the chapter — it describes our method for hierarchical comparison of ENT data structures and the classification scheme used to represent comparison results. The complete specification of comparison rules for SOFA CDL can be found in Section 3.4 plus in Appendix B. Subsequent sections discuss the role of specification language type systems in our method, and compares it with similar approaches used in practice.

The results presented in this chapter are then used in Chapters 5 and 4 in two different (yet related) situations — in component substitutability assessment and in deriving revision identifications.

3.1 Motivations and Approaches

While the structuring of interface into traits and categories described in the previous chapter is the basis for component understanding, the methods which fulfill the goals of this thesis need to *compare* the components. Such comparison can be done for different purposes — highlighting different coding styles, showing the changes made during development, selecting a component most appropriate for the given architectural environment, finding a suitable interface to match a set of criteria, etc.

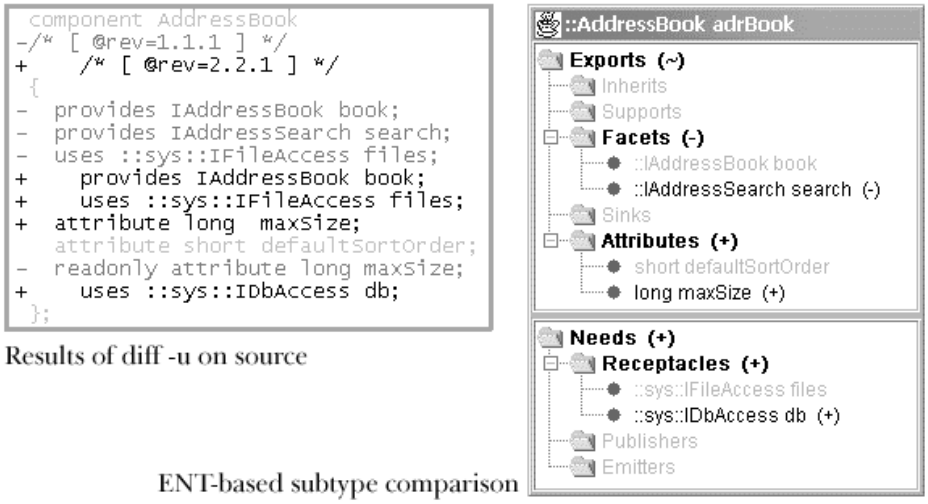

Fig. 3.1: Text- vs. grammar-based component comparison (specifications in the CORBA Component model IDL3 language)

Our component comparison is driven by the need to detect differences between components primarily in order to decide about their substitutability. In the design of the comparison method we were motivated by the needs of human developers and users. Particularly, we wanted the method to behave similarly to humans with respect to the assumptions, emphases, data structures, algorithms used for comparing the specifications, and the resulting conclusions.

3.1.1 Desired Properties of Component Comparison

This "human view" approach can be described by several key characteristics given below. They summarise the points that are important for humans in evaluating specification comparison results.

- The specification is structured rather than flat — the developers do not see it as a one-level list of declarations, but rather mix a birds-eye view, comparing whole chunks of declarations, with a detailed examination of differences in particular feature or quality; this structuring is reflected in the ENT model.
- The order of declarations is not important — the perceived semantic of a component, and consequently the comparison result, is the same regardless of the sequence of its declarations (cf. Figure 3.1).
- The formatting style is irrelevant for the semantics — indentation, white space, comments, etc. do not matter in understanding the workings of the component; of course good formatting helps in orientation in the specification but this issue is left aside here.
- Names (type and instance identifiers) matter in the comparison — they are key in expressing and understanding the semantics, for example `provides IFileAccess fileOps;` is more telling than `provides IFa fa;` and `property string NumberOfElements;` is a misleading code.
- The typing rules of the languages involved should be honoured — this is because changes in the declared and referenced types have a big effect on human as well as machine understanding of the component. Thus the method should detect changes in the contents of referenced types, e.g. records, and changes in the order of interface method parameters.

These properties contrast with some "implementation dependent" approaches to dealing with differences in specification (e.g. the "diff" tool or DCE interface change rules discussed in more detail in Section 7.2 on page 130 as part of the overview of related work). We consider such approaches rather inappropriate for the current state of software engineering, where the complexity of software systems is rapidly increasing. This complexity provides a strong incentive to move towards methods which put more weight to the human needs than to the computer implementation aspects, shielding the developers from low-level issues as much as possible.

The only exception is possibly the last property — typing rules are more of the "computer implementation aspect" sort. However, they still reflect

the relations between objects in the problem domain (certainly much more than, for example, DCE's rules for compatible interface evolution). Perhaps more importantly, they are necessary to warrant practical usefulness of the specification comparison methods. The components compared are in the end always implemented in some programming language and their code must obey type rules in order to be safe [Car97].

The method for detecting and classifying specification differences described in this chapter is designed to provide these properties. It uses the ENT model of interface structuring to aggregate the comparison results on several levels. This provides a means for quick overview of the changes as well as their detailed descriptions, regardless of the textual format of the component specification or its binary representation.

3.1.2 The Approach Taken

The primary question we ask in this chapter is: given two components, described by their specifications, where and how do they differ? In general, the answer will always be along the line that "one is a subset of the other", at least in some aspect(s). The precise meaning of this answer can however be approached from two directions.

One view on component differences is based on the client-server interactions in which components are ultimately engaged at run-time (this approach is schematically depicted on Figure 3.2). In this scenario we have a component C_1 employed in an application by binding its features to and from features of other components. If we want to use C_2 in its place we need to evaluate the differences between C_1 and C_2 from the point of view of the bindings, i.e. how the differences would affect their consistency.

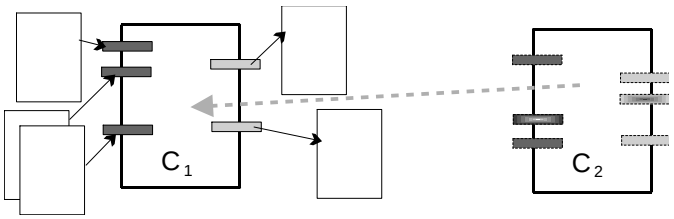

Fig. 3.2: Comparing components in client-server relationships

Thus the question is transformed into the question of component compatibility or substitutability. At the specification level it is usually resolved by checking the subtype relation between C_1 and C_2.

Another view on the changes is standalone component comparison, depicted in Figure 3.3 on the facing page: taking C_1 and C_2 just as they are, what can we say about their differences? In other words, the goal is to com-

pute a "diff" (as in comparing two versions of a program source) that is taken as a characteristic of component differences. This diff however must be based on the syntactic and semantic structure of the component specification — rather than on the bytes of its binary format or lines of its source text — otherwise it will be of no real utility.

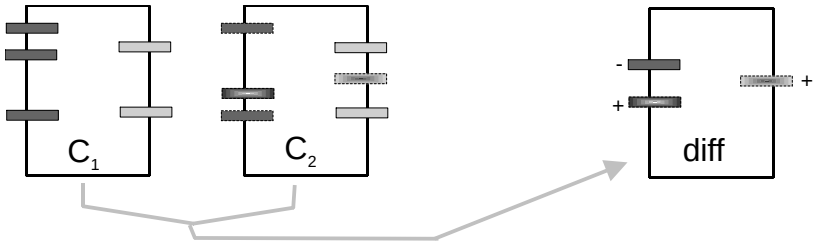

Fig. 3.3: Standalone component comparison

The reasoning about the effects of the differences is in this approach postponed until we know the context for the reasoning. By the "context" we mean, for example, that at some stages in component development or deployment, only some of its features and/or qualities are relevant for comparison (cf. configuration properties of CORBA components, or choice of connectors in the SOFA framework, that are not referenced at run-time). In this way we obtain a more general comparison method, hopefully a more useful one.

The price to pay is the need for a careful design of the data structures and algorithms used in computing and interpreting the diff. In particular, the comparison results must be expressed in a way which allows their efficient use in various circumstances — ideally there should be several levels of detail in the results and some flexibility in their interpretation.

The latter approach to determining the difference between components is the one taken in our work; the comparison from the client-server point of view is derived using its results. The rest of this chapter describes the way we compare the component specifications and the data in which the resulting diff is stored.

3.2 Differences Between Specifications

This section describes the core of our method for specification comparison. Its main idea is to structure the specification using the ENT model and derive the difference by comparing the resulting structures.

In what follows, we first define the comparison operations for the individual levels of the ENT model, using the specification language type rules

as a starting point. Using these operations we then create a straightforward classification system for specification differences, and discuss its properties.

3.2.1 Comparing Specification Parts

The motivation for comparing specifications pair-wise is to provide the information about how and where the specifications differ, in a format suitable for human interpretation. Such differences are usually expressed as the subset relation. Applied to the ENT model, we therefore want to define this relation for the elements, traits and categories found in the component specification source.

Definition 3.2.1 (Element comparison) *Assume two specification elements e_i and e_j written in specification language L; assume further that their $metatype$, inh and classifier items are equal. We say that*

- *the elements are equal (denoted $e_i = e_j$) if*
  $e_i.name = e_j.name$
  $\wedge\ \forall u \in e_j.tags\ \exists t \in e_i.tags :$
  $t.name = u.name \wedge t.value =_L u.value$
  $\wedge\ e_i.type =_L e_j.type$
- *element e_i* subsumes *element e_j (denoted $e_i \succ e_j$) if*
  $e_i.name = e_j.name$
  $\wedge\ \forall u \in e_j.tags\ \exists t \in e_i.tags :$
  $t.name = u.name \wedge t.value <:_L u.value$
  $\wedge\ e_i.type <:_L e_j.type$
- *element e_i is* incomparable *to e_j (denoted $e_i \triangle e_j$ in this work) if*
  $e_i.name = e_j.name$
  $\wedge\ e_i \not\succeq e_j \wedge e_j \not\succ e_i$

■

The definition assumes that an order is defined on the tag values which forms their subtyping relation. For example, for $access \in \{readonly, readwrite\}$ tag we expect $readwrite <: readonly$, using the principle of substitutability.

Note that the relations are based on the contents derived from the specification source only and are parametrised by the specification language L of the elements. That is, element comparison 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.

We require that the names of the elements be the same for a good reason: the names are used in the bindings of the containing component (e.g. by the `architecture` specification in SOFA CDL). The *tags* and *type* items are both compared for subtyping because frequently they are tightly interrelated — a prime example is the *readwrite* vs. *readonly* access modifier which has a profound impact on the typing rules.

An example of the element comparison for basic data types is given in Figure 3.4. The detailed specification of element comparison rules for SOFA CDL can be found in Section 3.4 and Appendix B.

Assume Java class attributes represented as ENT elements $e_1 = (count, short, \{(access, readonly)\}, \dots)$, $e_2 = (count, long, \{(access, readwrite)\}, \dots)$, and $e_3 = (count, long, \{(access, readonly)\}, \dots)$. Then, the following holds:

- $e_1 \succ e_3$, because $short <: long$ and the tags are equal,
- $e_2 \succ e_3$, because types are equal and $readwrite <: readonly$,
- $e_1 \triangle e_2$, because $short <: long$ but $readonly \not<: readwrite$,

---

Fig. 3.4: Examples of element comparison results

The direction of the subtyping relation used in the definition of element subsumption is the intended meaning "e_i can substitute e_j" (see Chapter 5). In the case of interface types used most often in component specifications, this very often manifests as "e_i has more declarations than e_j".

Definition 3.2.2 (Trait comparison) *Assume traits t_i and t_j such that $t_i.name = t_j.name$. We say that the two traits are*

- *equal (denoted $t_i = t_j$) if*
  $|t_i.E| = |t_j.E|$
  $\wedge\ \forall e_j \in t_j.E\ \exists e_i \in t_i.E : e_i = e_j$,
- *t_i subsumes trait t_j (denoted $t_i \succ t_j$) if*
  $|t_i.E| \geq |t_j.E|$
  $\wedge\ \forall e_j \in t_j.E\ \exists e_i \in t_i.E : e_i \succeq e_j$,
- *incomparable (denoted $t_i \triangle t_j$) if* $t_i \not\succeq t_j \wedge t_j \not\succ t_i$.

■

Note that, in line with the specification element equality and order, these relations are again based solely on the specification contents. The

human-added information is inaccurate for this purpose — namely, it is difficult to find any natural ordering of the classification facets and their terms.

Definition 3.2.3 (Category comparison) *Assume two categories K_i, K_j such that $K_i.name = K_j.name$. We say that*

- *these two categories are equal (denoted $K_i = K_j$) if*
  $|K_i.T| = |K_j.T| \ \wedge \ \forall t_i \in K_i.T \ \exists t_j \in K_j.T : t_i = t_j$,
- *category K_i subsumes category K_j (denoted $K_i \succ K_j$) if*
  $|K_i.T| \geq |K_j.T|$
  $\wedge \ \forall t_{1,j} \in K_j.T \ \exists t_{2,i} \in K_i.T :$
  $t_{1,i}.name = t_{2,j}.name \ \wedge \ t_{1,i} \succeq t_{2,j}$,
- *the two categories have incomparable contents (denoted $K_i \triangle K_j$) if*
  $K_i \not\succeq K_j \wedge K_j \not\succ K_i$.

■

To complete the comparison framework, the last definition states how complete component specifications are compared using the ENT structures. This definition differs from the above slightly in its pattern, as the element classification comes into play (mainly the *Role* dimension).

Definition 3.2.4 (Component comparison) *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*

- *these two components are equal (denoted $C_1 = C_2$) if*
  $\forall i \in \{1..|K_{1,i}|\} : K_{1,i} = K_{2,i}$,
- *component C_1 subsumes component C_2 (denoted $C_1 \succ C_2$) if*
  $E_1 \succeq E_2 \ \wedge \ N_2 \succeq N_1 \ \wedge \ T_1/A \succeq T_2/A$,
- *the two components have incomparable contents (denoted $C_1 \triangle C_2$) if*
  $C_1 \not\succeq C_2 \ \wedge \ C_2 \not\succ C_1$.

■

That is, the *provided* and *required* elements (and consequently the E and N categories) play covariant and contravariant role, respectively, in component comparison: for $C_1 < C_2$ to hold, the provided parts must be in subtype relation and the required ones in supertype relation. This is the

normal and expected behaviour; it is also fundamental to the understanding of component substitutability defined in Chapter 5.

The role of the *Ties* category is similar to that of the *Exports* as we want the C_2 to obey the same semantic contract(s) as the ones defined by C_1. However, we must use the restricted elements in the comparison in order to "align" the compared elements. The use of the name sets intersection to restrict the alphabet is the only option — it guarantees that the comparison will not include the elements added to C_2's exports and those removed from its needs. The component C_1 does not have to obey the specification of C_2 concerning these elements; if it had to, it would cause subtyping problems in the ties.

Note also that in the definition, we differentiate the contravariant role of the categories by the *role* classifier (using the ENT category set, i.e. separating $role = provided$, $role = required$, and $role = provided \wedge role = required$ respectively). Further investigations are needed to provide useful comparison rules for the cases of more complicated classifiers.

3.2.2 The Differences and Their Classification

We now describe the classification of specification differences, which forms the basis for the subsequent analyses of their effects. The system is an attempt to create a simple scheme based on natural observations. The main one is that specifications, seen as sets of declarations, can be compared using set operations.

As well as taking the specifications *per se*, we approach their comparison as if one has evolved from the other. Therefore, the subscripts used in the definitions that follow may be understood as "S_2 was created from S_1" or, rephrased, "S_1 and S_2 are two subsequent revisions". In general however this aspect is not important to the definition of the comparison method and interpretation of its results.

We would like to emphasise here that, as straightforward reasoning shows, the differences between specifications can be uniformly classified regardless of the level of granularity. We therefore define the classification of differences formally for all levels of the ENT model hierarchy at once. This uniformity of specification differences is key for our approach to component compatibility assessment.

The following definition formalises the intuition used by developers, using the ENT structuring of specifications and the comparison rules (grounded in type theory) set above.

Definition 3.2.5 (Specification Differences) *Let S_1 and S_2 be the specifications of components C_1 and C_2. Assuming the specifications are structured according to the* ENT *model, let $\sigma_1 \subseteq S_1$ and $\sigma_2 \subseteq S_2$ such that $\sigma_1.name = \sigma_2.name$ de-*

note two parts of the specifications at the same level of abstraction (i.e. two elements, traits, categories or the complete specifications).

Let $Differences = \{init, none, specialization, generalization, mutation\}$ *be a set of classification terms.*

Then, specification difference is a value $d \in Differences$ *generated by a polymorphic specification matching function* $diff : \_ \times \_ \rightarrow Differences$ *defined as follows:*

- $diff\,(\sigma_1, \sigma_2) = init \quad if \quad \sigma_1$ *is not defined;*
- $diff\,(\sigma_1, \sigma_2) = none \quad if \quad \sigma_2 = \sigma_1$*;*
- $diff\,(\sigma_1, \sigma_2) = specialisation \quad if \quad \sigma_2 \succ \sigma_1$*;*
- $diff\,(\sigma_1, \sigma_2) = generalisation \quad if \quad \sigma_1 \succ \sigma_2$*;*
- $diff\,(\sigma_1, \sigma_2) = mutation \quad if \quad \sigma_1 \triangle \sigma_j$.

■

We may also rephrase the above definition as follows: for the *specialization* difference, only specialisation changes are allowed in σ contents (and required in at least one); for the *generalization* difference, only generalisation changes are allowed (required in at least one). The *mutation* occurs if some contents mutates or if two subsets of the contents undergo different changes (some specialisation, some generalisation) which make the σ specification parts incomparable.

That is, σ_2 can be a *specialization*, *generalization* or a *mutation* of σ_1. The *init* value represents a special case — on the first creation/release of a specification being compared, there is no previous revision and therefore no specification to make difference against.

3.2.3 Difference Propagation

Consider the specifications of two CORBA components shown in Figure 3.5 on the facing page. When we take the contents of the `component` declarations alone, we see several differences directly:

- the *facets* trait has been specialised by adding the `IBar` interface;
- the *attributes* trait has been generalised, because the developer removed the `logname` property; and
- the *receptacles* trait has mutated as the C_2 component requires a different interface (this is equivalent to removing the `ILog` interface and adding the `IAdvLog` one).

C_1

```
interface IFoo {
  long foo();
};

component Example {
  attribute
   String logname;
  provides
    IFoo foo;
  uses
    ILog log;
};
```

C_2

```
interface IFoo {
  short foo();
  short baz();
};

component Example {
  uses
    IAdvLog log;
  provides
    IFoo foo;
  provides
    IBar boo;
};
```

Fig. 3.5: Standalone comparison of CORBA components: IDL3 sources

This comparison is what the developer can and would do at the first approximation. However, it is obviously incomplete and inaccurate — there are also "hidden" changes in the specifications of the referenced types:

- the `baz()` method was added to the `IFoo` interface; and
- its `foo()` method was changed into a subtype of that in C_1.

These changes must be considered by the comparison method in the same way as the "direct" ones to provide adequate support for the developers. In the terms of the ENT model, the difference can occur at the level of categories, traits, or elements (the "direct" ones) but also within the elements (the "hidden" ones). This has the consequence that the comparison must propagate the differences in lower levels of the ENT model to the upper ones. In consequence, it must recursively compare the types referenced in the specification of the component type. The result for the `Example` component should be as shown in Figure 3.6 on the next page.

The method of comparing specification parts defined above provide this propagation. This can be informally proven as follows:

1. Assume two identical component specifications represented by component element sets, $E'_{M1} = E'_{M2}$; this means all elements are equal.
2. Now extend the specifications by elements e_1 and e_2: $E_{M1} = E'_{M1} \cup \{e_1\}$, $E_{M2} = E'_{M2} \cup \{e_2\}$. Let these elements have the property that
   - they belong to the same trait (and consequently to the same category): $e_1 \in t^i_{M1}.E$, $e_2 \in t^i_{M2}.E$ (thus $t^i_{M1} \in K^j_{M1}.T$, $t^i_{M2} \in^j_{M2} .T$);

- they are "visually" equal (their names and type identifiers are the same) but they reference a type which has evolved in a type-compatible way: $e_1.name = e_2.name \wedge e_2.type <: e_1.type$;
- for simplicity we can assume that other parts of the elements are empty, i.e. $e_1.tags = e_2.tags = \emptyset$ (this simplification does not affect the result of the comparison).

3. Using Definition 3.2.1 on page 48, the result of comparing these two elements is $e_2 \succ e_1$. This means that the "hidden" difference in the referenced type has propagated to the difference between the elements themselves.

4. By a successive application of the trait and category comparison rules (Definitions 3.2.2 and 3.2.3), it follows that $t^i_{M2} \succ t^i_{M1}$ and $K^j_{M2} \succ K^j_{M1}$. In plain words, the traits and categories differ because of a change in a referenced type in their element.

As a related issue we note that when differences propagate, they may have opposing effects. For example, if one trait in a category K is specialised and another one generalised, the overall effect is that we cannot find any subset relation at the level of the K category as a whole. The resulting difference at the level of the E category is therefore mutation.

This is a clear and desirable consequence of Definition 3.2.5 — it corresponds to natural understanding and fulfils the requirement set forth at the beginning of this chapter, that the comparison should honour typing rules.

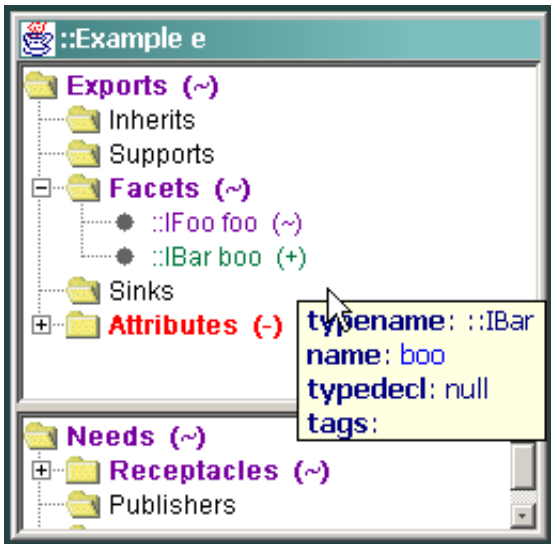

Fig. 3.6: ENT-based difference highlighting for CORBA components

On the other hand it shows the usefulness of the several levels of detail provided by the ENT model. If all we knew was that the *mutation* difference occurred at the category level it would not be sufficiently clear information. The trait differences explain why and how the mutation occurred.

Finally we want to note here that the practical consequences of a particular difference depend on the part of component in which it manifests, as expressed by its ENT classifier. For instance, specialisation in provided interfaces has clearly different semantics than the same difference in the required interfaces. It would therefore be wrong to draw conclusions about component substitutability based on the diff results only (e.g. requiring all traits to have the *specialization* difference), without considering the ENT classification values of specification parts.

3.3 Specification and Carrier Language Issues

In this section we want to discuss the relation of our specification comparison method to specification language type systems and the need for its correspondence with implementation language typing rules. The motivation for the analysis of these areas is the fact that we use comparison of specifications to express differences in the implementations of a component.

3.3.1 ENT-based Comparison and Type Rules

The definition of the element comparison (and consequently the trait, category and component comparisons) given in the previous section is parametrised by the component specification language, namely its typing rules. Because the comparison defines an inclusion relation between specification parts, it in effect creates subtyping rules for these parts.

The astute reader may have noticed that the *specialization* difference in fact constitutes a subtype relation, *generalization* a supertype, and *mutation* is an expression of type incompatibility. Thus the sentence "S_1 subsumes S_2" ($S_1 \succ S_2$, a result of comparison) can be read "S_2 defines a supertype of S_1".

There may already exist type rules governing the use of the component type in the given specification language — although the author is not aware of such rules for any of the widely used IDLs. So what is special about the approach we are presenting? The key novelty is a result of the application of the ENT model: it gives us the possibility to compare (i.e. define the subtyping rules for) only *parts* of the component type. We can therefore evaluate the effects of changes, in terms of the subtype relation, in just one or several well defined *particular aspects* (represented by the corresponding

trait or category) of the component type.

This differs from the traditional approach to typing in programming languages which treat even structured types as uniform wholes, without any internal structure (with respect to the type rules). Of course, the separate treatment of the particular aspects is enabled by their very existence — and often explicit declarative separation — in component types. However, the approach could be used even in some "traditional" programming languages. The only condition to be fulfilled is that it is possible to distinguish a type's aspects as traits — an example is a class declaration with attributes and methods, in the Eiffel language augmented by pre-conditions, post-conditions and class invariant.

The aggregation mechanism which groups elements into traits and categories has considerable impact on these subtyping rules. In particular, the *role* classification dimension distinguishes the contravariant role of the elements at the complete type level — the *provided* and *required* elements are similar to function parameters and return type, respectively. This is the fundamental reason behind the E, N, T categories; it also shows the possibilities that the ENT classification system gives in more flexible handling of component specifications. Chapter 5 gives a detailed treatment of the consequences of the differences in different *role*s to component substitutability.

3.3.2 Type Systems for Specification Languages

The practical implementation of our component comparison method has to handle the issues that stem from the properties of current component models. The method assumes that typing rules for the specification language exist, as the element comparison is based on them. The real situation is however, that for most languages the formal specification of their type systems is not available. For our work this would have the consequence that we may lack a formally sound foundation for the element comparison.

This is of course a substantial shortcoming that is not mitigated by the fact that its causes are mostly beyond our control. There are two possible solutions to this situation:

1. Define complete formal type system for the specification language in question, i.e. mainly SOFA CDL and CORBA IDL.

2. Create an implementation using informal rules that can be derived from textual specifications and similar languages.

The first option is certainly the right one as it provides the foundation for the necessary correctness of our method. However, the task is rather complex and is not in alignment with the focus of this thesis. (The fact that

the creation of type systems for programming languages is not a trivial task is indirectly supported e.g. by the Featherweight Java [IPW01] work.)

We therefore decided to go half way, defining only the *sub*typing rules (as a subset of the complete type system) for the SOFA CDL language. The rules are presented in Appendix B. Concerning the rest of the languages and the remaining parts of the type systems, we opt for the second solution and suggest the creation of typing rules as a possible future work.

Once the (sub)typing rules exist for a language, the ENT comparison of component types should produce the same results as their subtyping relation. We expect this is straightforward to achieve as the goals of these two comparison approaches are essentially the same. The designer and implementer of the comparison rules must nevertheless make sure that this condition is fulfilled.

3.3.3 Carrier Language Issues

The second language-related issue are the effects caused by language bindings, i.e. the translations from the specification language to the programming language in which the component is implemented. We use the term *carrier language* in this work to denote such programming languages. They come into play in any practically usable comparison system for components with declarative specification. In order to create the component implementation, the component specification defined in an interface description language must be translated into skeletal constructs in a programming language. The programmers then supply the actual business-logic code into this skeleton, and compile it to create the binary form of the component.

The problem lies in the fact that sometimes inconsistencies can be found between the (sub)typing rules of the given IDL and those of the carrier languages. Thus each language mapping would create different typing rules. To complicate matters even more, the design- and run-time linking of the component binaries via interface bindings may have its own restrictions on the linked elements, thus introducing additional rules.

The following examples illustrate these effects.

- In CORBA IDL and SOFA CDL mappings to Java, both `short` and `unsigned short` types are represented by the Java `short` type. Thus we could add the rule *unsigned short* $<:$ *short* to our system presented in the next section. However, this would violate the type system for the C language mapping in which the types are separate.

- The Java Binary compatibility specifications [GJS96] specifies in the paragraph 13.4.12 that "changing the name of a method, the type of a formal parameter to a method or constructor ... has the combined

effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature;" and similarly to the result type of a method.

This means that Java binary compatibility forbids subtyping changes in methods: for example `int foo(long)` $\not<:$ `int foo(int)`, contrary to standard type system rules. If we wanted to take this into account and retrofit our subtyping rules, we would lose an important part of their flexibility and create possible inconsistencies with other language mappings. On the other hand, implementation of components in Java will encounter the clash of this rule with our subtyping rules.

In effect, this example reveals that using Java as the implementation language for black-box components may create unforeseen issues.

This section has shown some of the problems encountered when creating type systems for declaratively specified black-box components. In our work we take the position that the specification and its rules have precedence over the implementation-related ones. In the following section we use this position to present the subtyping rules for SOFA CDL which are used in element comparison.

3.4 Comparison of SOFA Component Specifications

Based on the findings and definitions set in the previous section, we now show the basis for the implementation of SOFA component comparison. We use the nominal type system in this implementation, that is we require that types have the same name (as well as structure) to be considered equal [Pie02].

3.4.1 Subtyping Rules

The comparison of SOFA language constructs is based on their subtyping rules (developed as part of this work) described in Appendix B. The comparison of ENT component representations works with parts of the `frame` declaration.

The *element* structure separates the instance semantic (*tags* part) and typing (*type* part) information. This results in slightly modified ENT-based subtyping rules for elements with non-empty *tags*. For SOFA CDL, this is the case of the `property` element in trait *properties*.

Tags

$access$: $readwrite <: readonly$

Element *property*

for properties p_1 and p_2:

$$p_2.tags['access'] <: p_1.tags['access'] \wedge \begin{cases} p_2.type <: p_1.type & \text{if } p_1.tags['access'] = readonly \\ \wedge p_2.type = p_1.type & \text{if } p_1.tags['access'] = readwrite \end{cases}$$

$$\Rightarrow$$

$$p_2 <: p_1$$

For elements in other traits, the rules are direct equivalents of subtyping defined for their meta-types.

3.4.2 Examples of Frame Comparison

In this section we show examples of some common differences between components that can be encountered in the real world. The examples use a simple addressbook application components; we omit protocol declarations here to keep the code examples short. The impact the differences have on component substitutability are discussed in Chapter 5 below.

First, a baseline set of declarations is specified. Then we introduce changes on the level of component, referenced types, or both. The differences discussed are, in the order given: a simple set of changes on the component level that lead to the generalisation difference, a contravariant (subtype compatible) change that includes changes in referenced types, and a subtype incompatible change.

Baseline Declarations

The following CDL source declares the necessary data types, and two interfaces provided by the `FAddressBook` component.

```
/** Data types */

typedef short PID;
typedef sequence <PID,1000> ListOfPID;

struct Person {
  PID Id;
  string Name;
};

struct Address {
  string Street;
```

```
    string City;
    string<10> Phone;
  };

  /**
   * R/W access to address book data
   */
  interface IAddressBook {
    PID addPerson(in Person data);
    void delPerson(in PID person);
    Person getPerson(in PID person);
    Address getAddr(in PID person);
  };

  /**
   * The component which encapsulates address book
   * manipulation on the data level.
   */
  frame FAddressBook {
  // [revision C1]
  readonly property long maxSize;
  provides:
    IAddressBook book;
  requires:
    ::sys::IFileAccess files;
  protocol:
    (?book.addPerson { !files.write } +
     ?book.delPerson { !files.write } +
     ?book.getPerson { !files.read } +
     ?book.getAddr { !files.read }  )*
  };
```

The representation of $C_1 =$ `FAddressBook` in the ENT data structures (the component element set) is $E_{C_1} = \{book, files, maxSize, nil(protocol)\}$ where

book

$name =$ book, $type =$ IAddressBook, $tags = \emptyset$,
$metatype =$ interface, $inh = \emptyset$,
classifier = $(\{feature\}, \{operational\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

files

$name =$ files, $type =$ ::sys::IFileAccess, $tags = \emptyset$,

$metatype$ = interface, $inh = \emptyset$,
$classifier = (\{feature\}, \{operational\}, \{required\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

maxSize
$name$ = maxSize, $type$ = long, $tags$ = ($access$,readonly),
$metatype$ = property, $inh = \emptyset$,
$classifier = (\{feature\}, \{data\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{all\})$

nil (protocol)
$name = nil$, $type$ = (repeat protocol definition), $tags = \emptyset$,
$metatype$ = protocol, $inh = \emptyset$,
$classifier = (\{quality\}, \{operational\}, \{provided, required\},$
$\{type\}, \{mandatory\}, \{na\},$
$\{development, assembly, runtime\})$

Example 1: Trivial Gen Change

The following CDL specifies a component which is a generalisation of C_1.

```
/**
 * The component which encapsulates address
 * book manipulation on the data level.
 */
frame FAddressBook {
// [revision C2]
                              // property maxSize removed
provides:
  IAddressBook book;
requires:
  ::sys::IFileAccess files;
  ::sys::IDbAccess db;      // added
protocol:
  (?book.addPerson { (!files.write + !db.insert) } +
   ?book.delPerson { (!files.write + !db.insert) } +
   ?book.getPerson { (!files.read + !db.select) } +
   ?book.getAddr { (!files.read + !db.select) }  )*
};
```

The ENT representation of the revision 2 of `FAddressBook` is $E_{C_2} = \{book, db, files, (protocol)\}$ where

book
: $name$ = book, $type$ = IAddressBook, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier$ = ($\{feature\}$, $\{operational\}$, $\{provided\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

db
: $name$ = files, $type$ = ::sys::IDbAccess, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier$ = ($\{feature\}$, $\{operational\}$, $\{required\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

files
: $name$ = files, $type$ = ::sys::IFileAccess, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier$ = ($\{feature\}$, $\{operational\}$, $\{required\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

nil (protocol)
: $name = nil$, $type$ = (repeat protocol definition), $tags = \emptyset$,
$metatype$ = protocol, $inh = \emptyset$,
$classifier$ = ($\{quality\}$, $\{operational\}$, $\{provided, required\}$,
$\{type\}$, $\{mandatory\}$, $\{na\}$,
$\{development, assembly, runtime\}$)

When compared to C_1 it is clear that in C_2 the property was removed, and a required interface was added. Regardless of types and tags of these two elements, such changes constitute a *generalization* difference (the types and tags of other elements as used in the comparison have not changed).

Example 2: Another Component

In the second example, the scenario is to compare the `FAddressBook` component with a different one C_d, possibly from another component provider.

```
typedef  short  TPersonID;
typedef  string TName;

struct TPersonData {
  TPersonID id;
  TName name;
};
```

```
struct THomeAddress {
  TName street;
  short number;
  TName city;
  string<10> phone;
};

typedef sequence <TPersonID,1000> ListOfIDs;

interface IAddrManagement {
  TPersonID addPerson(in TPersonData data,
    in THomeAddress addr);
  void updatePersonAddress(in TPersonID pid,
    in THomeAddress addr);
  void delPerson(in TPersonID person);
  TPersonData getPerson(in TPersonID person);
  THomeAddress getAddr(in TPersonID person);
};

frame CoAddrManagement {
// component Cd
provides:
  IAddrManagement book;
readonly property long maxSize;
protocol:
  (?book.addPerson + ?book.delPerson +
   ?book.updatePersonAddress +
   ?book.getPerson + ?book.getAddr )*
};
```

In this situation, the analysis of element sets E_{C_1} and E_{C_d} shows that the *Exports* category of C_d has the same number of elements as that of C_1, and that C_d needs less than C_1. This suggests possible $C_d \succ C_1$ and, as the names of exported elements are equal, we can proceed to the comparison of types.

The `maxSize` property is exactly the same. The comparison of the provided interfaces `IAddrManagement` and `IAddressBook` reveals, that the former is a subtype of the latter — it has an extra method and the types used in the rest are the equivalent (using SOFA CDL subtyping rules).

We can therefore conclude that $C_d \succ C_1$, or in terms of difference classification, $\mathit{diff}\,(C_1, C_d) = \mathit{specialization}$.

Example 3: Standard Evolution

This last example attempts to analyse a more realistic case of component evolution. The developers in this case corrected a few problems found in the data types, and added a new interface to provide a read-only access to the data. At the same time, the component implementation newly uses database to store the addressbook contents, therefore a new dependency (required interface) is declared.

```
typedef  short  PID;
typedef sequence <PID,1000> ListOfPID;

struct Person {
  PID Id;
  string FirstName;   // name changed
  string Surname;     // added
  string Nick;        // added
};

struct Address {
  string Email;
  string Street;
  short Number;
  string City;
  string Phone; // changed
};

interface IAddressBook {
  PID addPerson(in Person data);
  void delPerson(in PID person);
  Person getPerson(in PID person);
  Address getAddr(in PID person);
};

/* R/O and search access to address book data,
 * duplicates some of the IAddressBook methods.
 */
interface IAddressSearch {
  Person getPerson(in PID id);
  Address getAddr(in PID id);
  ListOfPID findByName(in string Name, in string Surname);
};
```

```
frame FAddressBook {
// [revision C3]
readonly property long maxSize;
provides:
  IAddressBook book;
  IAddressSearch find; // added
requires:
  ::sys::IFileAccess files;
  ::sys::IDbAccess db;
protocol:
  (?book.addPerson { (!files.write + !db.insert) } +
   ?book.delPerson { (!files.write + !db.insert) } +
   ?book.getPerson { (!files.read + !db.select) } +
   ?book.getAddr { (!files.read + !db.select) } +
   ?find.getPerson { (!files.read + !db.select) } +
   ?find.getAddr { (!files.read + !db.select) } +
   ?find.findByName { (!files.read + !db.select) })*
};
```

In this new revision of the `FAddressBook` (denoted C_3 here), the `Person` type was enriched by a missing field and the `Address` type was amended to accommodate international phone numbers. Additionally, a new interface is required because the component wants to store the data in a database.

The ENT representation of this revision of `FAddressBook` is $E_{C_3} = \{book, db, files, find, maxSize, (protocol)\}$ where

book
: $name$ = book, $type$ = IAddressBook, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier = (\{feature\}, \{operational\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

db
: $name$ = db, $type$ = ::sys::IDbAccess, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier = (\{feature\}, \{operational\}, \{required\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

files
: $name$ = files, $type$ = ::sys::IFileAccess, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier = (\{feature\}, \{operational\}, \{required\},$

$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

find

$name$ = search, $type$ = IAddressSearch, $tags = \emptyset$,
$metatype$ = interface, $inh = \emptyset$,
$classifier$ = ($\{feature\}$, $\{operational\}$, $\{provided\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

maxSize

$name$ = maxSize, $type$ = long, $tags$ = ($access$,readonly),
$metatype$ = property, $inh = \emptyset$,
$classifier$ = ($\{feature\}$, $\{data\}$, $\{provided\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{all\}$)

nil (protocol)

$name = nil$, $type$ = (repeat protocol definition), $tags = \emptyset$,
$metatype$ = protocol, $inh = \emptyset$,
$classifier$ = ($\{quality\}$, $\{operational\}$, $\{provided, required\}$,
$\{type\}$, $\{mandatory\}$, $\{na\}$,
$\{development, assembly, runtime\}$)

Because the `Address` type is used both as `in` and `out` parameters in the methods of `IAddressBook`, these natural changes lead to a mutation of the `IAddressBook` interface. Together with the addition of a required interface this means that $C_3 \triangle C_1$.

3.5 Discussion

We have described a method for comparing component interface specifications that fulfils the desired properties defined at the beginning of the chapter. Here we summarise its advantages, disadvantages and the issues that remain open.

3.5.1 Advantages

First of all, an important characteristics of the specification comparison method is its independence of the physical representations of the components compared. This is a key difference to the approach used by e.g. DCE interface compatibility, which compares the binary representations of interfaces, and DCOM, which does not allow any change in interface specifications.

Secondly, we believe our approach to component comparison provides an important contribution because it defines the foundational subtyping rules for *parts* of type. This is a difference to standard type systems which treat type as an unbreakable unit. The advantage of our approach is the possibility to interpret the subtyping results depending on context. It also allows us to omit parts of the component type from the subtyping comparison, should we wish so for some reason (see e.g. the notion of partial substitutability defined in Chapter 5).

Our method is designed to be open to extensions because it defines the comparison at a meta-model level. In particular, it is easy to incorporate new meta-types of elements in component specification — the method defines the component subtyping relation as a composite function, using the $role$ classification of elements in determining contravariance. Thus we believe we have in fact created a comparison *framework*, rather than a single-use method, that can be used on many component models. This is similar in approach to [SC00] who define standard contravariant subtyping for the ComponentJ language. Their definition of component type ($R \Rightarrow P$) in principle allows any type to be part of the R and P sets.

Using the ENT model, the method gives comparison results in a format that is easy to understand and interpret by humans — see Figure 3.6 on page 54. The straightforward classification of differences makes it easy to visualise them, e.g. in a tool that would show the side-by-side comparison of the specifications. Users can start at the coarse granularity of differences between categories, and dig into the lower levels to find details. This is important for quick and accurate interpretation of the comparison results.

On the more fundamental side, the results of the comparison are a basis for the ENT-based re-definition of the subtyping relation on components. The notion of strict compatibility defined in Chapter 5 effectively embodies this relation.

3.5.2 Disadvantages and Open Issues

The generality of our specification comparison method however points to its most important weakness: it depends on ENT model structures and subtyping rules being defined for the given specification language.

While it is usually not very difficult to develop these prerequisites even for specification languages that lack them, it adds to the amount of work needed to use the method. Additionally, there may be problems in defining the subtyping rules for some systems. The carrier language mapping problems discussed above show that this may not be a simple task in real world scenarios.

A possible weakness of our method is the use of the restriction operator in the comparison of the $Ties$ category. It may result in disregarding parts

of the component specification which are important for its environment. For example, let's consider the components with the specifications in Figure 3.7. Their comparison will determine that $C_2 \succ C_1$ but the required ordering of outgoing calls can be important for the components that bind to C_1's required interfaces. The standalone component comparison will in such case fail to prevent run-time errors when C_2 is used in place of C_1.

```
frame C1 {
  provides:
    InterfaceX x;
  requires:
    InterfaceA a;
    InterfaceB b;
  protocol:
    ?x.foo { !a.bar ; !b.qux }
};
```

```
frame C2 {
  provides:
    InterfaceX x;
  requires:
    InterfaceA a;
  protocol:
    ?x.foo { !a.bar }
};
```

Fig. 3.7: Frame specifications with different ties

Future specifications may include global rules for application consistency and semantics, to which a component must conform. Such rules will require that the comparison handle the interplay of the per-component semantics with these global rules. At present, the method described in this chapter does not consider global rules.

Of course, our comparison method depends on the existence of specifications — component features and properties that are not included in the specification cannot be compared even though they may be important. This issue is treated by the global simplifying assumptions of the work.

3.6 Summary

In this chapter we have presented a method for comparing ENT data structures derived from component specifications. Its design is driven by two key factors: to honour language type system so that the results can be reliably used in automated component management, and to reflect human viewpoint in presenting the differences found.

The comparison uses the typing rules for the given specification language in element comparison. The aggregation rules for trait and category creation then drive the comparison of these levels. On the level of whole components, the *Role* classification facet is used to distinguish elements with covariant and contravariant roles. The actual carrier language used to implement the components may however complicate the rules.

As an example, the implementation of the comparison for SOFA CDL was presented. This entailed the creation of so-far nonexistent subtyping

rules for the language together with ENT-related amendments. A prototype implementation of this comparison was created for both SOFA CDL and CORBA Component Model IDL3 specification languages. The prototypes allow to compare the contents of two specification files and display the differences found between the components declared in them.

The method presented in this chapter is a key part of the work of this thesis as it forms a basis for its main results presented in the following chapters: a consistent revision identification scheme for components (Chapter 4), and evaluation of their substitutability (Chapter 5. The use of the ENT model together with run-time component management structures creates some interesting possibilities compared to standard subtyping methods.

Chapter 4

Revision Identification Scheme for Components

Any piece of software that is successfully used for a period of time undergoes a series of changes — bug fixes, maintenance releases, enhancements. Software components are no exception. It is therefore important to be able to distinguish the resulting different versions of the same component.

This chapter describes a revision identification scheme suited for black-box software components. The need for such work is given by the fact that none of the current industrial strength component frameworks (CORBA Component Model, DCOM/COM+, EJB) as well as the various research frameworks (Darwin, Rapide, SOFA, Fractal) provide a systematic approach to component versioning. In addition, the version identification schemes used in version control tools are not standardised and are not suitable for determining the compatibility of subsequent revisions (see Figure 4.1). We believe these deficiencies already create problems in smooth upgrades of existing component-based software systems.

```
/**
* $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."*

Fig. 4.1: Meaning of revision identifiers in RCS-based systems

Our goal in this endeavour is to reconcile the sometimes conflicting needs of two kinds of players in the software component industry — the human developers and their automated tools. The developers are primarily interested in knowing what versions of an component are available and what has changed between any two of them; for this they need primarily a

concise, readable version identifiers and indication of the changes.

The automated configuration and deployment tools need to decide which component version to use for the desired configuration and how component changes impact the possibility to substitute one component version (in the given environment) by another one. This requires that the version and compatibility information be in a precise, standard, machine-readable format.

With these general needs in mind, we designed a versioning scheme that uses component specification as its subject. It is based on the generic ENT model view of software structure (Chapter 2) and on the classification of differences presented in the previous chapter. The main novel idea of our approach is that the revision numbers are derived automatically from the results of component type comparison. This leads to a clearer relation between (changes in) parts of the specification and parts of the revision identification, shown in Figure 4.2.

```
frame FAddressBook
[ @rev = 3.2.1 ]
{  ...
```

`@rev = 3.2.1` *means "since the first release (1.1.1) of this component, its provided parts have have changed two times, required once, and ties have been left unchanged."*

Fig. 4.2: Meaning of ENT-based revision identifiers

The text of the chapter describes first the motivation for a component versioning scheme. The main part covers the various levels of revision markings and their properties. Section 4.5 describes the implementation for the SOFA system, including the issues of branching and variant description. The chapter ends with a discussion of the strengths and weaknesses and a summary of the findings.

Overview of Versioning and SCM Terms

For a clarification of terms used, this paragraph provides an explanation of terms related to versioning. It is based on common understanding used by the software configuration management (SCM) community, contained in works by Tichy [Tic94], Conradi and Westfechtel [CW98] and others.

- *branch* is a separately identified line in the evolution of a software element, consisting of one or more successive revisions;
- *revisions* describe the evolution of a software element in time;

- *variants* denote alternate implementations of a particular revision with different properties (semantics, storage vs. speed requirements, etc.);
- a particular *version* (of a software element) in a SCM-specific meaning is its concrete realization that can be uniquely described by the combination of branch, revision and variant descriptions;
- *configuration* is a set of software elements (in concrete versions) that together forms an application or its part.
- *revision number* is an unsigned integer which is incremented whenever the related software element is changed.

There is one consequence of this understanding of terms — the branches, revisions and variants form a hierarchy. A component identified by name (which in effect denotes a version set) has one or more branches, one branch consists of one or more revisions, and one revision is a set of one or more variants.

This assumption is important for our work because the relation between revisions and variant reflects the relation between component specification and the implementation(s) derived from it. This allows us to define *specification-based* revisions in Section 4.2 below.

4.1 Issues in Component Versioning

There are several key considerations that a component versioning scheme must care for.

Scope Component versioning is concerned with assigning version identification to the whole software component. In particular, it is to be used for describing released components, not for versioning during development, and should therefore consider the needs of component trading, deployment and updating.

Granularity of versioned elements Components are often coarse grained elements that are developed, traded and used as atomic entities. However, they provide or require several interfaces and possibly other features. These features as well as related datatypes obviously undergo changes during component evolution.

It would be therefore useful if the versioning scheme would work at a finer granularity than that of a whole component. Ideally such scheme should also provide versioning of the individual interfaces and even structured data types.

Support for automation Perhaps the most important aspect of component versioning, from our point of view, is that it has to support automated processing. This means that it must be possible to create, query and match version descriptions in a way that is least obtrusive to users.

The component developers could be relieved from assigning version numbers because there is no technical reason for manual identification of revisions — the scheme should be analogous to e.g. RCS-based systems. This of course does not affect the provider's freedom to assign *marketing* version numbers.

Identifying branches and variants however will probably still remain largely a manual process as it is essentially based on classification according to pre-selected criteria.

Relation to substitutability A versioning scheme suitable for components should enable the tools to reliably detect whether there is a newer/older version of the component at hand (a typical role of traders) and whether it can substitute the currently used component.

That is, the link between versioning and update should indicate where are the differences between versions and how they affect substitutability. Additionally, there should be support for the common task of finding an appropriate version of a component that would match a currently existing architecture. Unless done manually, both of these tasks require the existence of version and change description with well-defined structure and contents.

Version families Due to the coarse grained nature of components we can expect that a given component will exist in relatively few revisions and variants. This means that from the users' point of view the version information need not cater for large version families. In particular, we expect that branching and merging should not be an important issue in component versioning.

Readability The scheme should not compromise readability despite its support for automation. The versioning should not be repelling to normal developers and should, and if need be, offer the possibility to create the version descriptions by hand. (One of the motivating aspects was to develop a system which does not require the use of unreadable UUIDs in the COM or DCE style.)

Additionally, because components are black-box units, their version description needs to be accessible outside of the component for the purposes of searching, trading and deployment. We thus need a suitable meta-data to contain the versioning information.

4.2 Specification-Based Revisions

In line with the fundamental positions of the thesis, we design our versioning scheme to use component specification as the primary object of versioning. At the same time, the specification provides the source data used to compute the revision identification.

This approach is motivated by two considerations. The first one is our general aim to use already existing data, rather than to introduce a need for manual entry. The second reason is the observation that component revisions are a result of changes that often manifest in the component specification.

4.2.1 Types of Changes Between Revisions

The changes between component revisions are incurred by the need to fix a bug, or to enhance the software in some way. In general, the change to a software component can be (in order of the increasing influence on its compatibility with the previous versions):

- internal (implementation) — no visible change in the interface or behaviour. Clients can use the new version without adaptation, except perhaps that they can trip over unspecified semantical properties.[1]
- semantic (behavioural) — the structural features on the interface are unchanged, the semantic properties differ (e.g. timing, state transitions, protocol). Whether a client can use the services of the element with the new behavioural characteristics is on its discretion.
- external (interface) — the component's interface is changed. Whether a client of the previous version can use the new one depends on the nature of changes, the probability is that if the new version is not a superset of the old one, problems will occur.

It can be seen that the latter types of changes subsume the former ones, i.e. that behavioural changes lead to internal changes, and that external changes lead to both behavioural and internal changes.

We observe that changes are often manifested in the software specification which is in particular true for software components. Our versioning relies almost completely on this fact. The internal changes may be reflected in the N category of specification traits, the external changes should be

[1] In this thesis, we assume a "nearly ideal world" — the software elements really do what they promise in the specification, and the specification reveals as much as possible about the component. In these assumptions we are inspired by existing languages like Eiffel or SOFA CDL, and by our hope that in future more such well-designed languages will be in use for the sake of software robustness.

reflected in the E category. The behavioural changes may show in the E or T categories provided the appropriate semantic specification is available.

The distinction between internal and behavioural change is thus based on the semantic specification available: if a change occurs in a semantic property captured by the specification (category T) then the change can be detected as behavioural by automated tools.

4.2.2 Relating Changes and Revision Identification

In the "real world", the changes classified above usually result in a change in the version (revision or variant) identifier of the affected software elements or components. Depending on the versioning system in use, the change in the version identifier may in some way hint the importance of the change. These paragraphs provide a digest of the current practice based on various sources — the DCE and Java systems, the RCS-based tools, the Debian Linux package versioning scheme, and assembly versioning in Microsoft's .NET framework [Rig02, Pet95a, Tic85, Ber90, J$^+$03, Cor02].

Most systems use a two- to four-number revision identification scheme, which we call here the "$M.m.\mu$" (major, minor, micro) scheme. Major enhancements (evolution and perfective maintenance, i.e. adding features) lead to external changes and are obviously accompanied by internal and behavioural changes. These enhancements are often indicated by changing the major version number(s) and/or giving the software a different marketing name.

Minor enhancements (preventive and adaptive maintenance, e.g. support for new hardware or media types) and bug fixes (i.e. corrective maintenance) lead primarily to internal changes in the implementation. However, in current practice there is no clear classification of what constitutes a "minor enhancement" with respect to changes in the component interface. These enhancements usually lead to some modifications of minor and/or micro revision number(s).

4.2.3 Our Approach: Specification-Based Revisions

The component versioning approach presented in this chapter attempts to change this fuzzy situation by linking the component revision identification to the information about exactly which parts of the specification are affected by the change between revisions.

This idea of using information obtained from syntactical analysis of software source for its versioning was used in the Gandalf system and its underlying SVCE versioning environment [HN86, KH83]. Compared to their approach we provide a scheme which is language-independent and which is well suited for coarse grained software elements.

Additionally, to provide an answer to the novel issues of component versioning, our scheme aims to be precise in the format and meaning of the revision identifiers. This lead us to the use of the ENT model and ENT-based specification comparison (described in the previous chapter) in the process of defining and practical implementation of the revision scheme.

In turn, this means that our revision identifications have a very close and well-defined relation to the structure of the specification it describes. We therefore call it specification-based revisions.

Definition 4.2.1 (Specification-based Revisions) *A* specification-based revision marker *is an ordered tuple* $(r_1, r_2, ..., r_n), r_i \in N$ *assigned to a single revision of component specification such that*

- *each* r_i *has a one-to-one relation to a well-defined part of the specification;*
- *the union of all* r_i *covers the whole component specification.*

A (specification-based) revision identifier *is a human readable form of revision marker. It has the form of a string "*$\mathtt{r}_1\mathtt{.r}_2\mathtt{.}\ \ldots\ \mathtt{.r}_n$*", for which a mapping between the positions in the marker and positions in the string is defined.*

The term revision identification refers to markers and identifiers collectively. ■

4.3 The ENT Revision Identification Scheme

This section describes the details of our scheme for component revision identification. It attempts to break new ground with its approach to the creation and meaning of the revision identifiers. At the same time it is intentionally very straightforward and strives to follow established conventions so that its results fit neatly in current mainstream versioning systems.

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.

We now present the individual levels of ENT-based revision identification, starting from the most detailed one. The revision number of a specification part is in the following text denoted $rev(\xi)$ where the ξ stands for either a trait, a category or the whole specification.

4.3.1 Detailed Revision Identification

The lowest level of revision markers is designed to provide the most detailed information about the evolution of the component specification. In

the ENT model, the finest grain is provided by the specification elements. However, to version each element separately would result in too much information which would be too volatile — elements come and disappear as often as they change. Element-based revision marker would thus have a variable number of elements, and thus would not be practical for identification purposes.

We therefore need to start with structures that are not too numerous, have fixed number of elements, yet are on a relatively low level of abstraction. The nearest such structure in the ENT model are specification traits. In the detailed revision marker, each specification trait is thus assigned a single revision number. The semantics of this number is that at first component release it has the value 1 (one), and at each subsequent release it is incremented if the trait has changed in some way.

Definition 4.3.1 (Detailed revision identification) *Let us have two immediatelly subsequent revisions of a component, C^1 and C^2, and traits $t_i \in Traits(C^1)$, $t_j \in Traits(C^2)$ such that $t_i.name = t_j.name$.*

The revision number $rev(t_j)$ *of the trait j is*

- $rev(t_j) = 1 \quad if \quad diff(t_i, t_j) = init$, *i.e.* C^1 *does not exist*[2]*;*
- $rev(t_j) = rev(t_i) \quad if \quad diff(t_i, t_j) = none;$
- $rev(t_j) = rev(t_i) + 1 \quad if \quad diff(t_i, t_j) \notin \{init, none\}$.

Detailed revision marker *of a component C^2 is a tuple $R_D = (r_1, \ldots, r_n)$, $r_i \in N$ such that $\forall t_j \in Traits(C^2)\ \exists r_j \in R_D : r_j = rev(t_j)$.* Detailed revision identifier *is a string "$r_1 . r_2 . \ \ldots \ . r_n$" where r_i are the corresponding elements of the marker.* ■

The definition stipulates that there must exist a mapping between traits and positions in the revision marker. This mapping is generated by an order defined on traits which is arbitrary as far as the definition is concerned — in practice it depends on the intended use of the detailed revision identification. For normal use we recommend lexicographical ordering, more elaborate schemes may utilise the ENT classification system as shown in the description of component marker level below.

The example in Figure 4.3 shows two revisions of a CDL specification. The detailed revision marker in the bottom row illustrate that their parts are closely tied to the specification: the changes in provided interfaces and properties of the component result in a change of two revision numbers of the revision marker. The order of traits — and thus of the marker elements — is the default (alphabetical) one: $properties$, $protocol$, $provides$, $requires$.

[2] The C^2 is the first revision

`frame FAddressBook {`
` requires:`
`  ::sys::IFileAccess files;`
` provides:`
`  IAddressBook book;`
` property short maxSize;`
` protocol:  // shortened`
`  (?book.addPerson`
`  { !files.write })*`
`};`
	`frame FAddressBook {`
	` requires:`
	`  ::sys::IFileAccess files;`
	` provides:`
	`  IAddressBook book;`
	`  IAddressSearch search;`
	` readonly property`
	`  short defaultSortOrder;`
	` property short maxSize;`
	` protocol:  // shortened`
	`  (?book.addPerson`
	`  { !files.write })*`
	`};`

$R_D = \{1, 1, 1, 1\}$	$R_D = \{2, 1, 2, 1\}$

Fig. 4.3: Two revisions of `AddressBook` frame CDL

4.3.2 Component Revision Identification

Since component specifications may contain a number of traits, detailed revision marker may consist of too many numbers to be practical for human reading and understanding. We therefore need a simpler description that more closely resembles the schemes in common use which have proven to be effective in practice, namely the "$M.m.\mu$" system.

At the same time, there should be a clear relation of this higher-level marker to the detailed one. This would create the desired correspondence between the marker and the specification, and reuse already computed data. We therefore need an aggregation mechanism to distill this higher-level markers from the trait-based ones.

Using the ENT model we find the concept of categories a natural vehicle for this purpose. It has two benefits with respect to versioning. First, it already aggregates the traits themselves and thus creating category revisions is straightforward. Second, equally important reason is the fixed number of categories (in a given category set) across different components but also different specification languages and component frameworks.

Therefore, each *category* of specification traits is assigned a revision number according to the changes in its traits. For the purpose of component revision identification we can in principle use any category set to obtain the aggregate revision description. However, to standardise the revision marker structure we use the key categories that we described in Chapter 2 — the E, N, T category set. The semantics of the resulting revision identification is the same as for trait-based identification.

Definition 4.3.2 (Component revision identification) *Let us have two imme-*

diatelly subsequent revisions of a component, C^1 *and* C^2*, and categories* K^c, K^r *such that* $K^c.name = K^r.name \wedge K^c.T \subseteq Traits(C^1), K^r.T \subseteq Traits(C^2)$*. The revision number of the category* K^r *is*

- $rev(K^r) = 1 \quad if \quad diff(K^c, K^r) = init;$
- $rev(K^r) = rev(K^c) \quad if \quad diff(K^c, K^r) = none$
- $rev(K^r) = rev(K^c) + 1 \quad if \quad diff(K^c, K^r) \notin \{init, none\}$

Component revision marker *of component* C^2 *is a triple* $R_C(C^2) = (r_E, r_N, r_T)$*, where* $r_E = rev(E^r)$*,* $r_N = rev(N^r)$ *and* $r_T = rev(T^r)$. Component revision identifier *is a string "*$r_E \,.\, r_N \,.\, r_T$*" where* r_ξ *are the corresponding elements of the marker.* ■

The component marker can be derived in two ways — from the detailed marker or by analysing differences between specification categories directly. Both methods are equivalent in terms of the revision marker produced, due to the method of aggregating elements into traits and categories. However we prefer to use the detailed revision marker as the source for component one because it does not increase the complexity of its computation.

Using the E, N, T category set the component revision number definition gives us a triple of numbers. This can be easily mapped to existing version data placeholders in current component frameworks (CORBA, COM+, .NET). As will be shown shortly, component revision identification has therefore all the properties required on the industry-standard revision identifiers.

4.3.3 Primitive Revision Identification

In certain cases, namely if user-defined data types need to be versioned, the component revision markers are still overly elaborate. One reason is that data types (including interfaces) are mostly homogeneous, meaning they consist of declarations that would be classified into a single trait and category. Another reason is that version numbers of types are pervasive in the specification code and the shorter they are the better for all users of the specification.

As the most abstract level we therefore define a single revision number that aggregates all the previously described revision information.

Definition 4.3.3 (Primitive revision identification) *Let us have two immediatelly subsequent revisions of a component,* C^1 *and* C^2*. The* primitive revision marker *of the latter component,* R_P *is a natural number*

- $R_P(C^2) = 1 \quad if \quad C^1$ *does not exist, i.e.* C^2 *is the first revision;*

- $R_P(C^2) = R_P(C^1)$ *if* $\forall t_i \in Traits(C^1), t_j \in Traits(C^2)$, $t_i.name = t_j.name : diff\,(t_i, t_j) = none$;
- $R_P(C^2) = R_P(C^1) + 1$ *if* $\exists t_k \in Traits(C^1), t_j \in Traits(C^2)$, $t_i.name = t_j.name : diff\,(t_i, t_j) \notin \{init, none\}$.

The primitive revision identifier *is a string representation "*R_P*" of the* R_P *number.* ■

4.3.4 Cascaded Derivation of Revision Markers

As mentioned above, each revision identification can be derived either directly from specification comparison or from the markers on the lower level. The definitions use the former approach, here we show the latter.

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, if the revision number belonging to one trait in the E category is incremented in the detailed revision identification, then the $rev(E)$ of the component is incremented.

Figure 4.4 describes schematically this hierarchy of revision marker levels and their dependencies. The notation "3,spec : 4" means "if 3 is the revision number of ξ^c and $diff\,(\xi^c, \xi^r) = spec$, then $rev(\xi^r)$ is 4".

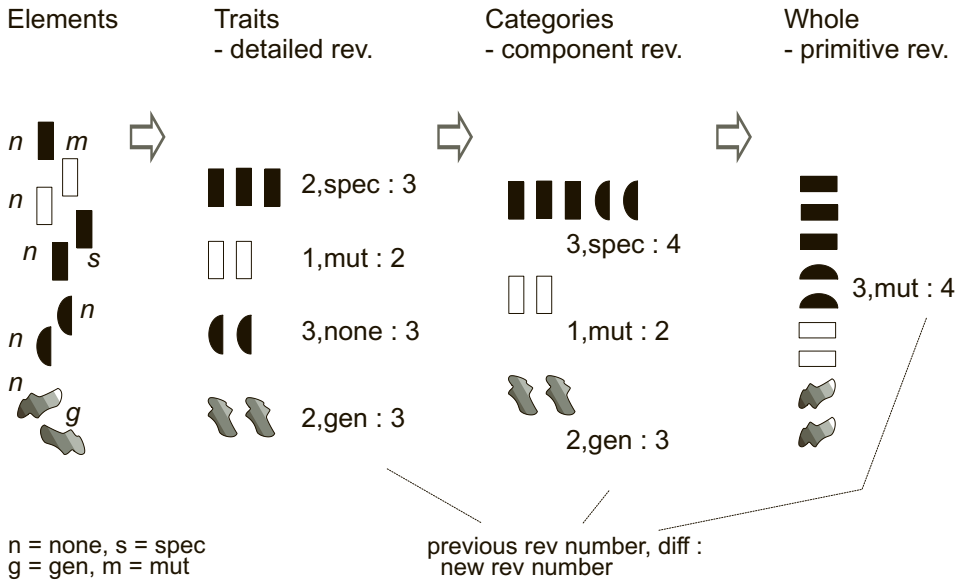

Fig. 4.4: Derivation of Revision Data

4.4 Properties of ENT Revision Identification

In the preceding paragraphs we described the scheme of revision identification suitable for components. It remains to show that its revision numbers actually behave correctly, mainly with respect to unique identification of revisions [IEE98, CW98][3].

This behaviour can be characterised by three properties. The first one, *consistency* says that the same change set, applied to a current revision, always results in a revision with the same revision marker. The second which we call *differentiation* means that the derived revision of a component cannot have the same revision marker as its predecessor. The third property is called *monotonicity* meaning that the markers preserve the time order of revision creation.

Consistency is needed for the stability of revision identifiers and their relation to the software they describe. Differentiation is fundamental for the identification role of our revision markers, while monotonicity for ordering components according to their markers.

Proof of Consistency

This proof is straightforward and intuitive, and is backed by the isomorphism of the $rev(\xi)$ functions. Application of a given set of changes will always result in the same *diff* values for the ENT representation of the revisions' comparison, at all levels of the model. Consequently, the revision marker of the second revision will always be the same.

Note that this property may not be satisfied in case of manually assigned revision identifiers, since different developers may interpret the applied changes in different ways and thus assign different revision identifiers to the second revision.

Proof of Differentiation

We show that our scheme differentiates revisions by contradiction. Assume we have a component revision C^p and a revision C^d of the same component for which C^p is the immediate revision ancestor, and that C^p and C^d differ in one element.

Assume further that the two components have equal revision markers: $R^p_D = (r^p_1, \ldots, r^p_N)$, $R^d_D = (r^d_1, \ldots, r^d_N)$, $R^p_D = R^d_D$.

The difference in one element between C^p and C^d must, by the rules described in Section 3.2 of the preceding chapter, lead to the same difference

[3] Although the following text explains these properties on the revision markers, the same holds for the revision identifiers.

in the traits which contain it, say t_i. Consequently, $diff(t_i^p, t_i^d) \neq none$.

This means that, using the definition of detailed revision marker, r_i^d must be equal to $r_i^p + 1$. We conclude that $R_D^d \neq R_D^p$, contradicting the assumptions.

Thus the differentiation property of our revision scheme is confirmed. It is apt to note that the scheme cannot guarantee *uniqueness* of revision markers: it is easy to show that two revisions C^1, C^2 derived from the same C^p can have the same revision markers even if $diff(C^1, C^2) \neq none$. The uniqueness of version identification must thus be handled by other means, e.g. using branch tags as described in Section 4.5 below.

Proof of Monotonicity

This proof uses induction on detailed revision markers. Assume that an order is defined on specification traits for the given language. A change in i-th trait of a component specification leads to its revision marker $R_D^1 = (r_1, r_2, ..., r_i, \ ..., r_n)$ (before change), $R_D^2 = (r_1, r_2, ..., r_i + 1, ..., r_n)$ (after change).

Using pair-wise comparison of numbers at the same positions results in $R_D^1 < R_D^2$. Thus later component revisions have greater revision identifiers.

4.5 Application of the Scheme in the SOFA Framework

In this section we show how the ENT specification-based revision identification scheme is used in the SOFA component framework. The interesting feature of our implementation is the use of revision identifiers as an integral part of the CDL where any user-defined type is versioned. This allows among other options the so called "versioned dependencies" — a component may explicitly declare that it requires a particular version of an interface.

Different types of clients prefer different formats of revision information: precise and rich descriptions are useful for tool use, while humans prefer conciseness and readability. Our implementation caters for both of these cases by providing different representation of the same revision marker in component meta-data and in the CDL specification.

The revision information is stored in two places: as part of the CDL specification of the component plus in a meta-data attached to its distribution form. The redundancy should not create the multiple maintenance problem as the data is handled by closely cooperating tools. The meta-data is described separately (Chapter 6); in this section we primarily describe the CDL versioning extensions.

4.5.1 All Types Have Revision IDs

The structures subject to the above 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.

For this type revision identification we use (recursive) type comparison, via the specification comparison method described in Chapter 3, to detect differences between types. Except for component types (`frame`s, which use component revision markers), primitive revision identification is then sufficient. To make the type versioning work, we have to use adornments on type declarations (to declare their revisions) and add version identifier when the type is referenced (to denote a unique type declaration). The adornments are described in subsection 4.5.4, the references in subsection 4.5.5 below.

Despite all the benefits, it would be a nuisance if the developers had to specify the version every time a type is used. And in many cases, the "ordinary" data types do not evolve. The SOFA implementation therefore uses default values to make type versioning simpler: when a reference to a type without version identification is encountered, the "trunk" branch and the most recent revision with respect to the referencing type is used.

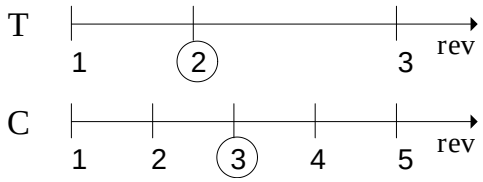

Fig. 4.5: Selecting default type revision

The use of default revisions is illustrated in Figure 4.5 where a component C references a type T. When all the revisions shown are available, revisions 3 and 4 of C use revision 2 of T (rather than the absolutely most recent revision 3, which would lead to type inconsistencies).

4.5.2 How Revision Identification is Derived

We have explained that SOFA revision identification is derived from the CDL specification of the data types. The tool which generates the markers uses comparison of normalised derivation trees to obtain the source differences.

Figure 4.6 on the facing page shows the process how the revision marker

is generated from CDL sources. The revision ID of the first revision is obtained either from its CDL source or from component meta-data, the description of changes is usually stored in the meta-data.

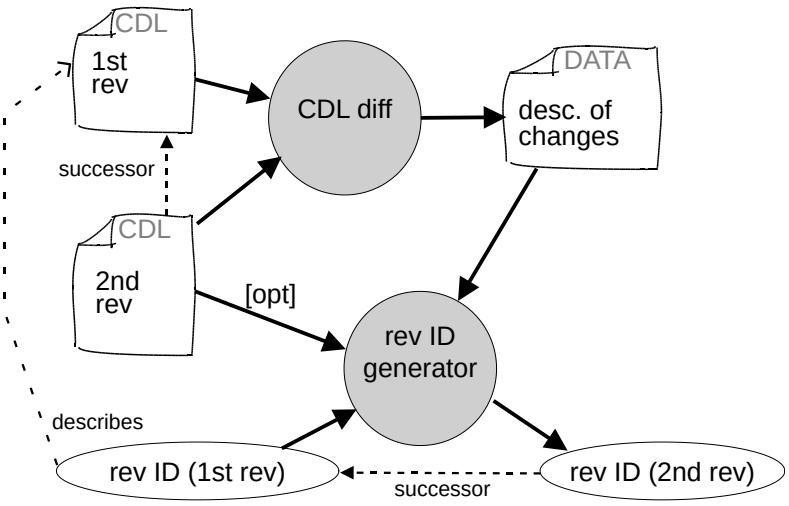

Fig. 4.6: Generating Revision Identification

4.5.3 Versioning Complete: Handling Branches and Variants

As was mentioned above, revisions alone are not sufficient for component identification — the development may occur on parallel branches, and each revision of a component specification may have several variants differing in the implementation.

While we do not explicitly deal with branching and variants in our work, we offer some thoughts on the topic. We expect that branching will be relatively rare due to the nature of components, which are coarse-grained market entities. Their providers therefore cannot change their versions (including branches) too often as this would adversely affect their business. It should therefore be sufficient to use simple branch tags which uniquely identify branches in the version family of a component.

The changes that do not manifest in the component interface (bug fixes and implementation changes, including different `architectures` of a frame) cannot be captured by the revision identification scheme. We therefore need to handle them as variants of a given revision.

For variant description we find it suitable to use boolean or description logic [Zel98] which provide means to specify variant properties in a sufficiently precise manner. They are also suitable to structured representations such as XML and URI schemes.

4.5.4 Version Data in CDL

Because the developers need to know the version information of the types (mainly interfaces and components) they use, we include revision identifiers in the CDL specification source. To this end we designed an extension to the original CDL grammar described below. The main emphasis of this form is on the conciseness and readability of the representation.

This representation uses component-level revision identifiers for frame revisions and primitive ones for all other types. The format follows the established convention, i.e. numbers separated by dots ("`3.2.12`" for component revisions, respectively "`3`" for primitive revisions). The conversion from the (r_E, r_N, r_T) triple uses the category ordering to map to positions in this $M.m.\mu$ format. The component revision identifiers thus preserve the benefit of showing the place of change in their structure.

The revision identifier of a type is placed in a meta-data section of its CDL declaration. The syntactical constructs are inspired by the attribute-based programming used e.g. in the C# language [ECM02]. An example of the version data section is given in Figure 4.7 on the next page, its grammar can be found in Appendix C. An extension of the standard CDL compiler was developed which generates and manipulates this data.

4.5.5 Identification of Versioned Types

An important goal of component naming is to identify components in remote repositories or run-time spaces. Distributed systems use name services for this purpose (e.g. CORBA Naming Service, Java Naming and Directory Services) which link logical names of components or server objects to their (remote) implementations.

In our work on the SOFA implementation we do not explicitly deal with these issues. Instead we provide a mechanism for component naming suitable for two different purposes which otherwise do not receive much attention. The first is the need to reference remote types and components in the CDL source, the second is a desire to support automated downloads of particular versions from remote traders.

The problem in this area stems from the need to extend the name by the complete information needed to uniquely identify a single component version. With independently deployed and traded components, this information has to include the provider, name, and version of the component. The latter consists of the branch, revision and variant identifiers (see above).

We therefore propose that type names in SOFA have the form of an URI (Uniform Resource Identifier, [BLFM98, Fie95]). This enables us to create structured identifiers which carry a lot of information yet remain human readable. The URI identifiers help us to maintain our position articulated

in the Introduction, that the system should support equally well tools and human developers — otherwise we could opt for numeric UUIDs used e.g. in COM.

```
frame FAddressBook
[  // meta-data section
  branch=freeze1;    // manually provided value
  @rev = 3.2.1;       // automatically generated
  @diff= (spec,mut,none);
]
{
  provides:
    IAddressBook book;
    sofa://com.notscape/ab/IAddressSearch  search;
  property short maxSize;
  requires:
    /sys/IFileAccess files;
    OfficeApps/IPhoneBook#rev=2  phone;
};
```

Fig. 4.7: Proposed revision identifiers in CDL source

A consequence of this design is that the extended CDL can reference any type — from different namespaces but also from different providers and/or in different version. This is a novel but quite natural extension of the ability to reference names in other namespaces (modules, packages, ...) known from modern programming languages. The illustration of the URI names is in Figure 4.7.

The format of the URI names is given by the grammar included in Appendix C (section C.2). It uses the fragment identifier notation ('#') to separate the component name (i.e. the version family identifier) from the identification of a particular version. The examples in Figures 4.7 and 4.8 on the following page show several names that can be used in the specification of a component frame and architecture as well as the use of URI identifiers in the SOFA CDL source.

In some cases, especially with complex variant descriptions, the URI format becomes too long and unwieldy. To preserve the readability and support for automation, a XML format of the identifier could be designed. This option is described in Chapter 6 where the component versioning information is merged with other related data to form a rich component meta-data.

As the examples given in Figure 4.8 on the next page show, the use of URI identifiers in CDL makes it possible to declare references to types

`sofa://com.borlund/school/Calcul`
: An incomplete absolute identifier, denoting a component in default branch and version.

`/school/Calcul#rev=3.2.4&var=architecture:3`
: Absolute scope name of a component in a concrete architecture.

`requires: sofa://com.borlund/school/ICalc#rev=3.3.7`
: It is possible to specify versioned dependencies in CDL, using the URI style of identifiers. Here we show the use of interface versioning, where a component frame can require a particular revision of an interface coming from a different provider.

Fig. 4.8: Examples of proposed SOFA URI identifiers

or components from different providers or in different versions. In this use, only the branch and revision parts of the version identification are significant.

Second potential use of the URI component names is during the downloads of components from traders or remote repositories. When a client node determines the complete identification of a required component, it can contact its repository with the component's URI (similarly to requesting a web page). This can make downloads and queries easily integrated into current WWW-based services.

4.6 Discussion

Version identification is a standard part of software engineering practice, ranging in granularity from approaches that version individual procedures and types [HN86, OMG02d] to application release numbers [Bai97]. Let us therefore discuss what are the merits and inadequacies of our approach.

4.6.1 Advantages

The results presented in this chapter provide a means to effectively create and manage revision identification of software components. Furthermore, it makes it possible to specify version information in component dependencies (described in Section 4.5 above). This improves the composeability and reliability of component-based configurations [BW98].

The greatest advantage of the ENT revision identification scheme is its blend of algorithmic predictability and comprehensibility of meaning. This

means firstly that the way revision markers and identifiers are derived ensures among others their consistency — the same set of changes will always result in the same marker. This is unlike schemes in e.g. RCS-based systems [Tic85, Ber90] or application management systems [Bai97].

Secondly, the ENT revision identifiers convey a clear meaning to humans (developers, component users). The correspondence between the place of change (in terms of category or trait of elements) and position in the identifier means that by comparing the revision identifiers, they can determine where the changes have occurred. Thus the identifiers help in assessing the impact of change in component interface. Again, this precise semantics is missing in current schemes.

These two aspects provide a formal backing for the standard "$M.m.\mu$" revision identificaion scheme. As this scheme is common in several industrial component frameworks, our approach can be readily used in practice.

We would also like to point out that it can be without modification applied to any user-defined type. In fact, the SOFA framework versioning uses it to version all user-defined data types as is shown in the next section. The only precondition for such use is the availability of the data type specification, which is by default satisfied in all IDL/ADL specification languages. The CORBA Component Model is similarly ready for such type versioning (see the discussion in Related work, Chapter 7).

4.6.2 Disadvantages and Issues

When compared to the "$M.m.\mu$" schemes, ENT revisions do not reflect the extent of differences between revisions. (In standard schemes, the changes in lower positions mean "small change" and correspondingly in the higher positions.) To provide such feature, we would need to augment the specification comparison method by some characterisation of the extent of differences, and use it in the revision marker.

Due to the ENT model structures, the revision marker abstracts away the source of differences — even the detailed marker does not point to the differing elements, just to their containing traits. This may make locating the differences somehow complicated. However, the detailed marker combined with the differences at trait level should provide sufficient information for the common cases.

4.7 Summary

In this chapter we have presented the first concrete result of the ENT-based component structuring and comparison — a scheme for component revision identification. It is novel in its use of grammar-based comparison

of component specifications for obtaining information from which the revision identification is derived. Consequently, this revision identification scheme provides semantically rich and structurally clear identifiers. The use of three kinds of revision markers with varying level of detail also helps to cater for the needs of both automation tools and human readers.

We have shown that our revision identifiers preserve the standard properties of distinguishing two revisions of the same component and expressing the time order in which the component revisions were created. On the practical side, they easily fit into mainstream versioning schemes, used e.g. by CORBA or COM+.

A novel aspect introduced by the implementation is the proposed usage of the URI identifiers in the SOFA CDL specification. Any identifier in the CDL grammar may in fact have the form of a URI which includes the revision identification of the denoted object. This makes it possible to declare versioned dependencies, i.e. references to types or components in different versions. A proof-of-concept implementation has been developed which creates revision markers from component comparison for the SOFA framework.

Chapter 5

Component Substitutability and Compatibility

The aim of software component technology is to provide means for easy creation and modification of software systems. A frequent kind of such modification is an *upgrade*, that is the replacement of an out-dated version of a component by a more current one. An upgrade is therefore a special case of component substitution.

The basic scenario is the standard one: we have an application consisting of several interconnected components. For whatever reason we want to substitute[1] one of these components by another one, be it a newer version or a component from a different provider. It is natural to require that the substitution be side-effect free, i.e. that after such substitution, the whole application must function correctly and its behaviour must be consistent with that before the change. When dealing specifically with upgrades, this property is usually called backward compatibility of the components.

Many approaches exist that attempt to ensure this, ranging from a-priori tests for behavioural subtyping [LW94] to intercepting incorrect functioning at run-time e.g. in fault-tolerant systems [Kop87]. The aim is usually to try all the options at hand before concluding, in the worst case, that the substitution is undesirable.

In this chapter, we present a formal underpinning of methods which test two components for substitutability a-priori. The goals of this part of our work can be summarised in two points. Firstly, the methods we design should be applicable in scenarios where the component substitution is fully automated (e.g. during remote or unattended updates of embedded component-based applications). Secondly, we need to find a suitable relation as the basis of substitutability definition — both practical experiences and related research [ZW97, VHT00, HL99] show that pure subtyping is often too restrictive.

[1] In this work we are not concerned whether the substitution occurs at design-time or at run-time (usually called “update”).

To some degree, these two goals are contradictory. Automation puts an emphasis on safety, which is best achieved by the subtype relation of the components. Yet we would like to soften this subtyping relation to gain some flexibility. Our approach to solving this dilemma is based on the fact that for components with specified dependencies (the *N* category), substitutability is the property of not just the two components but includes the environment in which they work.

The text of the chapter is structured as follows. We start by describing the issues and options in determining whether a component can be substituted by another one; this provides the context and motivations for our work. Section 5.2 on page 94 is the key part of the chapter – it contains the definitions of two primary and several derived kinds of black-box component substitutability. The results of previous chapters, mainly the ENT model and specification comparison, are extensively used in the definitions.

Section 5.4 describes some options in the practical use of our method of checking component substitutability in the SOFA component framework. Then the method is discussed and compared to other similar works, and the chapter is ended by a short summary.

5.1 Issues in Component Substitution

Before we describe the details of component substitutability and compatibility, let us consider several issues that affect our approach and the resulting definitions.

Definition of substitutability and compatibility. From the practical viewpoint, the *replacement* component is substitutable for a *current* component if it satisfies three general requirements:

1. Present the same operational interface to its environment.
2. Read and write the same data (with respect to the place and format) as the current one[2].
3. Conform to the semantics of the current component in all interactions in which it is engaged.

There are many definitions of substitutability that can be found in related work. Some emphasise the need for a holistic view, arguing that substitutability involves global integrity checking [Szy96]. While this is certainly right, the complexity of such checks may be prohibitive.

[2] The *input-output compatibility* in [LC99]

We therefore prefer the approaches that use local solutions, which are based on the comparison of just the components directly involved in the substitution. These approaches are mostly based on the principle of substitutability coined by Wegner and Zdonik [WZ88]: *a subtype (replacement) component should be usable whenever a supertype (the current one) was expected, without the client noticing it.* In our approach we adopt the same definition, and expand in some detail the aspect of linking clients to the component subject to substitution.

With respect to compatibility, there are two views on the meaning of this term. The "formal" one, represented e.g. by the work by Vallecillo at al [VHT00], understands it as a mutual correspondence of interfaces (to be) bound, so that their owners can interoperate. The "practical" one understands the term as the ability of a new version (of a software application or component) to safely substitute a previous one. This *backward compatibility* thus stresses the relation to versioning.

We position our work between these two view, considering compatibility of components to be a special case of substitutability applied to subsequent revisions of a software component.

Means of substitutability checking. Our work is concerned with black-box components, and therefore the checks for substitutability cannot be based on code analysis. Instead, we have two options. First, we can use a suite of compliance tests embedded with the component itself[3] as e.g. in the certified components research [M+01]. This may be flexible and cater for many practical issues in substitutability checking, but depends highly on the quality of the test suite. In reality, this approach would require non-trivial amount of work from the component developers, contrary to the goals of this thesis.

Alternatively, we can statically compare the available component specifications to determine whether they indicate substitutability. A number of methods exist that can be used for this purpose – signature matching [ZW97], comparison of pre- and post-conditions [Per87], protocol conformance [PV02], and also the component specification comparison method presented in the previous chapter.

The advantages of static checks are their fidelity — they precisely evaluate all of the declared aspects — and low computational demands (for signature and many semantic specifications, they are comparable to static type checking during compilation). In addition, this type of checking does not interfere with the (possibly running) applications. However, it is possible only so far as the adequate declaration of surface features exists. Thus static checks may not ensure complete configuration consistency.

[3] Software testing people would use the term *regression tests*.

In this work we use the static approach, which we prefer for its fidelity and automation possibilities.

Who should control substitution. The substitution can be initiated and controlled by a very diverse set of agents — component developers, system administrators, ordinary users, and increasingly also management software (automated and/or remote updates) [VHT00, Ore98]. The method of substitution used in any of these situations must therefore provide useful support to these agents, from reasonable default behaviour for fully automated updates to support for manual intervention.

This diversity suggests the need for precise and sensible definition of compatibility for substitution. This is the main topic of this chapter which in its approach is geared towards the unattended automatic methods.

Handling incompatibilities. Since black-box components don't provide access to their source code, there is little room for component adaptation. In the extreme cases (unattended updates, substitution run by non-programmers) we even can't write any adaptation code manually [Sta00]. Thus the substitutability checking methods should warn early about potential and real incompatibilities and provide hints when adaptation is attempted as a fix.

The use of adaptors or connectors [BP00] working on top of component interface is a possible step in this way. However, these issues are out of the scope of our work.

5.2 Substitutability of Components

In this section we define our notion of black-box component substitutability and consequently specialise it to define component compatibility. We begin with a discussion providing the rationale for our approach, and then present the definitions of two key compatibility levels, using the ENT model structures and their comparison described in the previous two chapters.

The principle of substitutability shows that this property does not concern just the two components in question. It tells us that we additionally need to take into account their use by clients. 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 replacement component's provided features should in most cases be at least the same as those of the current one, otherwise its clients will not be able to successfully communicate with it. However, it need not be a problem if the replacement component has different requirements because its environment may be able to satisfy them, i.e. to provide features to which the new dependencies will be bound.

The situation is unlike most programming languages where the type of the "replacement" object must be an exact subtype of the current type. The reason for this difference is the more complex and dynamic nature of inter-component relations: the bindings between the current, respectively replacement component and its surrounding components can themselves use subtyping relation, can be evaluated at replacement time (rather than statically by comparing just the two components, which would be equal to type checking in programming languages), and even adapted dynamically to overcome discrepancies.

This observation tells us that the commonly used notion "only subtypes are compatible" may be overly restrictive in the case of component substitution in a particular context, even when no adaptation is possible. This observation is supported by many works that attempt to provide more flexible notions of subtyping [ZW97, VHT00, Nie93, FW00].

We therefore define two kinds of component substitutability that deal with the extent to which the environment is considered: strict and contextual substitutability. The following subsections provide the definitions of these forms of component compatibility using the E, N and T categories of traits and their comparison.

5.2.1 Strict (Subtype) Substitutability

This type provides a notion of compatibility useful for the case of comparing two components alone, i.e. without any information about their actual use. The only data which we can use in this case are component specifications. This results in a usual definition of subtyping-based substitutability.

Definition 5.2.1 (Strict substitutability) *The replacement component with the* ENT *representation* $C^r = \{E^r, N^r, T^r\}$ *is* strictly substitutable *for the current one* $C^c = \{E^c, N^c, T^c\}$ *if* $C^r \succ C^c$, *that is* $E^r \succ E^c \wedge N^c \succ N^r \wedge T^r/A \succ T^c/A$ *where* $A = Names(C^r) \cap Names(C^c)$. ■

In terms of our difference classification system, this means that $diff(E^c, E^r) \in \{none, specialization\}$, $diff(N^c, N^r) \in \{none, generalization\}$ and $diff(T^c/A, T^r/A) \in \{none, specialization\}$.

The definition corresponds to the natural understanding: the replacement component provides at least the same, requires at most the same, and does not impose new semantic constraints. It is the same as the "common sense" notions used e.g. by Seco [SC00] or Vallecillo [VHT00]. Based on the standard contravariant subtyping between component types, this kind ensures substitutability "a-priori".

5.2.2 Deployment Context of a Component

We noted in the introductory paragraphs of this chapter that substitutability need not be a local property of the two components. In its evaluation, we can determine the actual run-time architectural environment of the application in which the component will be bound. This leads to a novel "architecture-aware" form of substitutability especially suited for black-box components.

We take into account two important aspects of the environment:

1. Which of the current component's provided features actually have bindings to particular required features of other components in the given application configuration (architecture).
2. Whether the environment provides features which the replacement component declares as required, not necessarily considering the requirements of the current component.

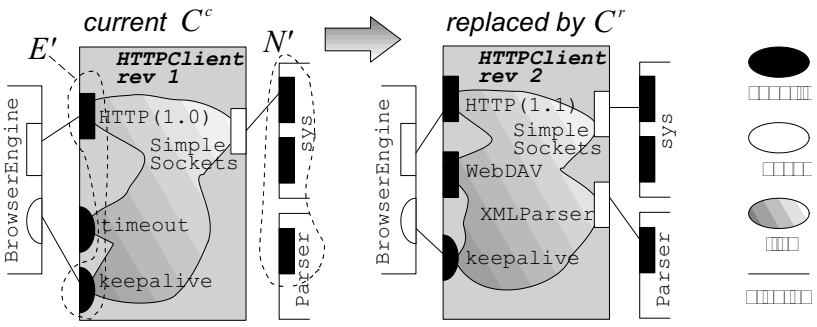

Fig. 5.1: Context and Component Substitution

This idea is illustrated by Figure 5.1: the new version of a `HTTPClient` component ("rev 2") provides both `HTTP` and `WebDAV` interfaces, and can be assembled in an application that does not use the WebDAV protocol, or a `HTTP 1.1` interface can be bound to a `HTTP 1.0` required one. On the opposite side, the new version requires an additional `XMLParser` interface that 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 follows.

Definition 5.2.2 (Deployment context) *Assume that C^c has been deployed and is correctly functioning in an environment which contains the sets of components L (clients) and S (servers). Let $E^S = \bigcup_{s \in S} Exports(s)$ be the set of all elements with*

$classifier.role = \{provided\}$, $N^L = \bigcup_{l \in L} Needs(l)$ be the set of all elements with $classifier.role = \{required\}$ that exist in the deployment environment.

The deployment context *of the current component C^c is a pseudo-component $Cx = E' \cup N' \cup T'$, where E', N' and T' are sets of elements such that*

- *$\forall e' \in E'$ it holds that there exists an element $n \in N^L$ bound to an element $e \in Exports(C^c)$, and $e'.name = e.name$, $e'.type = n.type$, $e'.tags = n.tags$ (to preserve type information of the context), $e'.metatype = n.metatype$, $e'.classifier = n.classifier$ and $e'.classifier[\text{“}role\text{”}] = \{provided\}$ (to preserve* ENT *semantics of C^c).*
- *$\forall n' \in N'$ it holds that there exists an element $e \in E^S$ (for which a binding to an $n \in Needs(C^c)$ may or may not exist) and we set $n'.name = n.name$ (if binding exists) or $n'.name = nil$ (otherwise), $n'.type = e.type$, $n'.tags = e.tags$, $n'.metatype = e.metatype$, $n'.classifier = e.classifier$ and $n'.classifier[\text{“}role\text{”}] = \{required\}$.*
- *$T' = T^c / Names(Cx)$ where $Names(Cx) = Names(E') \cup Names(N') \cup Names(T^c)$ is the set of names of elements relevant for the context.*

■

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 definition of context deserves a few explanations. The E' elements use the type $n.type$ which may be somehow unexpected. The purpose is to compare the replacement component's elements not to those of the current component, but to their actual counterparts in the bindings. This exploits the fact that the binding from n (required) to e (provided) is allowed only if $e.type <: n.type$.

It can be seen that multiple choices may exist for the elements in N' in case there are several elements (in one or more server components) which satisfy the subtyping relation for n. In order to create an unambiguous set of elements in N', a single element must be selected in such cases.

There are three strategies for this selection. The first one is to use an arbitrary element. Second, select the most specialised one if a total order can be created using the subtyping relation. Third, let the user choose if user interaction is possible. While we would prefer the second option, it may not be always possible (candidate elements may be mutually type incompatible) and thus either of the other strategies will be used depending on the circumstances.

5.2.3 Contextual Substitutability

Now we can define the kind of substitutability which uses the notion of deployment context.

Definition 5.2.3 (Contextual substitutability) *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 terms of difference classification, this means that $diff(E', E^r) \in \{none, specialization\}$, $diff(N', N^r) \in \{none, generalization\}$ and $diff(T'/A, T^r/A) \in \{none, specialization\}$.

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. Renaming of context elements is necessary for those in N' that were not bound to elements with $role = required$ in C^c and which need to be compared with the new elements of N^r.

Note that among other things the definition allows downgrading of provided features and extension in the required ones, through the definition of deployment context. For example, if C^c has its provided `HTTP 1.1` interface bound to a required `HTTP 1.0` then this feature of the replacement component can be downgraded to the latter one.

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

Proposition 5.1 (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 5.2.2 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 3.2.3 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. Only if no such indication is available the assessment of substitutability must be carried out at the assembly or update time.

5.2.4 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. For example, a client in a distributed system may be able to handle changes in call semantics (synchronous vs. asynchronous), or a user who only reads data generated by a component is not affected by changes in its operational interface.

We therefore should allow the definition of substitutability to disregard some aspects of the component or parts of its interface, in order to increase the chances of substitution. This approach of abstracting away details is in various modifications used in many works on interface matching, notably by Zaremski [ZW97].

Orthogonally to the two types defined above we therefore define a hierarchical system of partial substitutability levels. It gradually leaves out more of the "unimportant" information from the compatibility assessment. The levels were motivated by the work of Larsson [LC99]. In our definitions we take the advantage of using the classification system defined in Chapter 2 to specify the scope considered in the evaluation.

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. Therefore, the specification of qualities is disregarded under the assumption that clients will be able to adapt to changes in component semantics.

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. The motivation for this level is 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 $\forall e \in D : (\{feature\}, \{data\}) \in e.classifier$.

As is obvious from the above, these compatibility levels constitute a hierarchy: feature compatibility is a special case of full compatibility, and is a more general concept than data compatibility.

We would like to point out that we can use combinations of substitutability kinds and levels, due to their orthogonality. Thus we can require *full strict substitutability* to ensure plug-in replacement, *full contextual substitutability* for smooth upgrade of a given application, or for example *contextual data substitutability* if we know there are only a few operational bindings that we can adapt.

5.3 Backward Compatibility of Components

In the previous section we defined two kinds of component substitutability. As the reader has noted, the definition uses the specification comparison method described in Chapter 3. One of the consequences of the definitions is that the names of the components are not included in the comparison. This deliberate design allowed us to define *general substitutability* relation between any two 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. This means that both of them will have the same name. Substitutability between such two components is customarily called *backward compatibility*.

In this section, we therefore define this notion as a simple extension of the previous results. In particular, we combine the above substitutability relations with the approach to component revision identification described in Chapter 4.

Definition 5.3.1 (Strict backward compatibility) *Let us have two components, C^1 and C^2 with revision data assigned. We say that C^2 is* strictly backward compatible *with C^1 if*

1. $C^2.name = C^1.name$ *and*
2. $R_P(C^1) < R_P(C^2)$ *and*
3. C^2 *is strictly substitutable for* C^1.

■

That is, strict compatibility uses the comparison of primitive revision data of the components to establish the successor relationship.

Definition 5.3.2 (Contextual backward compatibility) *We say that the component C^2 is* contextually backward compatible *with the component C^1 if*

1. $C^2.name = C^1.name$ *and*
2. $R_P(C^1) < R_P(C^2)$ *and*
3. 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. On the other hand, in the sequence of revisions there may occur changes to component specification such that two distant revisions ($R_P(C^2) - R_P(C^1) > 1$) are compatible although intermediate revisions are incompatible.

5.3.1 Redefinition of Larsson's Compatibility Levels

Using our model of component interface, we can formalise the compatibility levels defined in some of the related works. We would in particular like to treat here the levels described by Larsson and Crnkovic in [LC99], in effect providing their "implementation definitions". The subject of their compatibility description are components in isolation, i.e. we are using strict substitutability.

Larsson's *behaviour compatibility* considers all component's characteristics, i.e. all its specification traits. It is therefore equivalent to our full strict substitutability as defined above.

The *interface compatibility* aims at preserving the interface but allows different implementations. It can be inferred from the article [LC99] that by "interface" the authors mean the standard IDL-like interfaces. This level is thus a special case of the feature substitutability defined above.

In our model this level would therefore be modelled by the strict feature substitutability, reducing further the subset of component elements considered by eliminating elements $e \in F : \ e.metatype \neq interface$.

Lastly, the *input/output compatibility* requires that only the format of data produced and read by the component to remain the same. In our approach this is modelled by the data substitutability.

Our substitutability levels were to a large degree inspired by these ones. However, the above analysis shows that — given the understanding of the term *interface* as an abstract class (like in OMG IDL, Java language, etc.) — Larsson's levels do not constitute a hierarchy. This is because the interface and input/output levels consider complementary sets of component

features. In case the component interface would include data descriptions, these three levels would become identical to our system.

5.4 Examples for SOFA Components

The SOFA component framework is an ideal platform for evaluating this approach to component substitutability checking. It uses black-box components with CDL specification of their surface, and provides means for automated swapping of components in applications (even at run-time). Such environment is a good model for real-world situations which require advanced checks to ensure application consistency across substitutions.

In this section we first show examples of strict and contextual substitutability using compatibility of SOFA frames (component declarations). Then we discuss the options in determining the context, and at last briefly treat use of our compatibility checking in component updates.

5.4.1 Compatibility of Frames

Strict Compatibility

We illustrate the strict compatibility (and consequently substitutability) on the `FAddressBook` SOFA component specifications. Please refer to Chapter 3 to the full CDL declarations.

```
frame FAddressBook {
[ @rev=1.1.1 ]
provides:
  IAddressBook book;
requires:
  ::sys::IFileAccess files;
protocol:
  (?book.addPerson { !files.write } +
   ?book.delPerson { !files.write } +
   ?book.getPerson { !files.read } +
   ?book.getAddr { !files.read }  )*
};

frame FAddressBook {
[ @rev=2.1.2 ]
provides:
  IAddressBook book;
  IAddressSearch find;
requires:
```

```
    ::sys::IFileAccess files;
  protocol:
    (?book.addPerson { !files.write } +
     ?book.delPerson { !files.write } +
     ?book.getPerson { !files.read } +
     ?book.getAddr { !files.read } +
     ?find.getPerson { !files.read } +
     ?find.getAddr { !files.read } +
     ?find.findByName { !files.read })*
  };
```

If revision 2.1.2 is the C^r and revision 1.1.1 the C^c, it is easy to see and prove that $E^r \succ E^c$ (the `find` element was added) and $N^r = N^c$. The $A = \{book, files\}$ and $T^r/A = T^c/A$ (the second revision's protocol contains that of the first one). The second revision is therefore strictly compatible with the first one.

```
frame FAddressBook {
[ @rev=3.2.3 ]
provides:
  IAddressBook book;
  IAddressSearch find;
requires:
  ::sys::IFileAccess files;
  ::sys::IDbAccess db;
protocol:
  (?book.addPerson { (!files.write + !db.insert) } +
   ?book.delPerson { (!files.write + !db.insert) } +
   ?book.getPerson { (!files.read + !db.select) } +
   ?book.getAddr { (!files.read + !db.select) } +
   ?find.getPerson { (!files.read + !db.select) } +
   ?find.getAddr { (!files.read + !db.select) } +
   ?find.findByName { (!files.read + !db.select) })*
};
```

In this case, the specialization difference in the *requires* trait (the added `db` interface) results in the loss of strict compatibility with both previous revisions.

Contextual Compatibility

To illustrate the method and effects of contextual compatibility checking, consider the latter two revisions of the `FAddressBook` component embedded in an contact-list editor.

```
frame FContactListEditor {
provides:
  IAddressBook contact;
requires:
  ::sys::IFileAccess files;
  ::sys::IDbAccess db;
protocol:
  // not important
};

architecture aProvider AContactListEditor
implements FContactListEditor {
  inst FAddressBook ab;
  subsume ab:files to files;
  delegate contact to ab:book;
};
```

Initially, the revision 2.1.2 of the address book is used as the `ab` instance in `AContactListEditor`. When an upgrade to the revision 3.2.3 is considered, we see that the `IAddressSearch` interface is not used (bound). Also, the `FContactListEditor` contains a declaration of a required `DbAccess` interface.

Thus the changes introduced by the third revision of the `FAddressBook` can be accommodated by the architecture – i.e. revision 3 is contextually compatible with revision 2 in this case.

5.4.2 Determining Context

The context of a component, needed for determining contextual compatibility, can in SOFA be determined in two places. The first option is an "a-priori" method which uses the description of architecture, i.e. the component interconnections on the first level of nesting. The second option is to use the run-time information about actual component structures held by the SOFA runtime system.

Context from Architecture

A SOFA application is (at the conceptual and declaration level) a hierarchical composition of components. Any component can be structured into a set of interconnected *subcomponents*, as described by its `architecture` declaration. Conversely, it can itself be a part of such structure of a *container* component. There are two exceptions: top-level components which represent whole applications are not part of any container component, and

primitive components at the leaves of the hierarchy are not decomposed into subcomponents.

```
frame C1 {
  provides:
    IProvided p;
  requires:
    IBetween br;
};

frame C2 {
  provides:
    IBetween bp;
  requiress:
    IRequired r;
};
```

```
frame A {
  provides:
    IProvided ap;
  requires:
    IRequired ar;
};

architecture aProvider aA
implements A {
  inst C1 c1;
  inst C2 c2;

  delegate ap to c1.p;
  bind c1.br to c2.bp;
  subsume c2.r to ar;
};
```

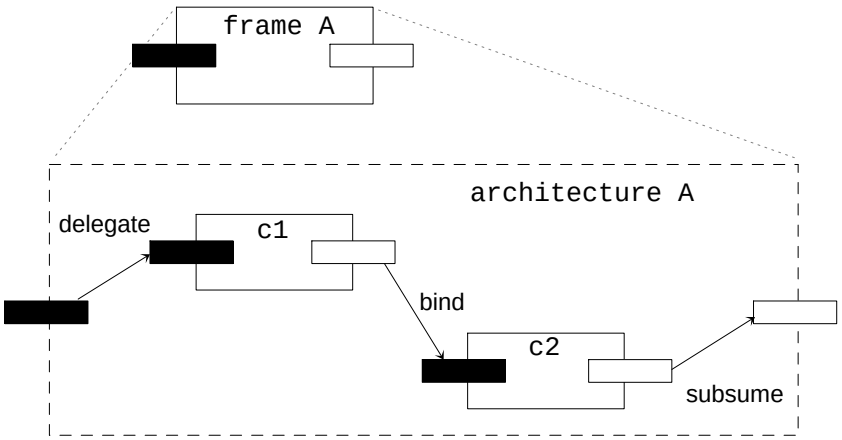

Fig. 5.2: An `architecture` declaration to illustrate context

The a-priori method of determining deployment context is suitable for components which are not the top-level ones, i.e. which can be contained in an architecture. For a (current) component C the method determines the *architectural* context from the architecture declaration A in which C is embedded. We use the knowledge of component interface bindings (the `bind`, `delegate` and `subsume` keywords) and the specifications of other frames

contained in the architecture (see the schematic example given in Figure 5.2 on the preceding page).

The E' part of the context is derived from two sources. First, the bindings within the architecture are examined — the `bind` keyword with a C's provided element on one side of the binding. Second, the delegations from the containing component's provisions are included — the `delegate` phrases which include C.

The N' also has two sources. First, it uses the provided features of other components contained in the architecture, i.e. all elements in $Exports(C_j)$ for all $C_j \neq C$. Second, the required interfaces of the container are included (to these the requirements of C^r can be subsumed). In the example in Figure 5.2, the context for c_1 is such that the $Elements(E'^{c1}) = \{ap\}$ and $Elements(N'^{c2}) = \{c2.bp, ar\}$.

The N' in architectural context may contain a superset of what may actually be needed by a particular C^r. This is because at time of its computation we don't know the actual C^r and thus rather take all the relevant elements available within architecture. We also use empty names in these elements. When a concrete C^r is supplied, the names are filled in according to its requirements. Because a component can be part of any number of architectures, the context pertains to the pair (C, A) — that is, a C can have multiple architectural contexts.

The main advantage of architectural context is that it can be determined in advance (when the architecture is created) and once for all future uses. This saves some computational time and resources as compared to the run-time method, for instance if used as part of component meta-data. A disadvantage is the limited scope of source information. If the component framework allows bindings across the boundaries of architectures (in the SOFA meaning) then this kind of context cannot include the parts of N' that are outside the boundary of A. Where this is a limitation, the method of determining context on-demand should be used.

Determining Context on Demand

This method uses the run-time data structures of the deployment environment called SOFA node (an analogy of "deployment dock" in [H+97] or "container" in EJB [Sun01a]).

The deployment environment maintains two kinds of data: the Run part maintains the list of active components (loaded in memory) and is a gateway to data about their bindings, in the form of component architectures. The component (or template) Repository contains all components available at the given SOFA node, including their CDL specifications. From this information the deployment context of a particular component can be reconstructed.

The use of this method is illustrated on Figure 5.1 on page 96. The replacement version of the `HTTPClient` component requires additional XML parsing interface and thus is not a subtype of the original one. However, the environment contains more components than those bound to the current version. Their provided features in this particular context contain the interfaces needed by the replacement version.

The primary property of this method is that the context is determined in a "just-on-time" manner: immediately before the attempted component substitutability check, at the node where the substitution should take place. This is both advantage and disadvantage — it provides the most complete and accurate data for the check, but compared to architectural context may take more time to compute. In case the component framework allows bindings across architectural boundaries, this method is probably the preferred one as discussed above.

5.4.3 Role in Component Updates

An update of a component means its replacement, at run-time, by another version. In a prototype implementation this replacement is guarded by substitutability checking mechanism which uses the results presented above.

The check for strict compatibility is fairly straightforward and easy to perform, and is especially suitable for the case of upgrading the component to a new revision. The system uses the component meta-data to obtain the base information for the check (see Chapter 6 for detailed description). It can then use the current architecture in which C^c is contained to establish the bindings of the replacement component.

When the context of the current component needs to be determined, the system may start from the architectural context if this is available. If it is not available, insufficient for the C^r, or if it is preferred to explore a maximum of choices, the current deployment context of C^c is determined. With the context created, it is compared against the specification of the replacement component, using contextual compatibility.

If C^r passes this check, the system must create a new architecture using the information from the context. In particular, it must specify the bindings between the elements of C^r and the elements in the context that match them. This ad-hoc architecture description is given as an input to the relevant component builders which do the bindings.

5.5 Discussion

Let us now consider the advantages and open issues of the presented definitions, as well as some practical aspects of component substitutability based

on specification comparison.

5.5.1 Advantages of our Method of Substitutability Checking

Checking substitutability of black-box components using the methods presented in this chapter 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.

Instead of a narrowly focused solution, we have created a generic framework for component substitutability and compatibility, applicable to different technologies and extensible to future developments (for instance in the area of behavioural subtyping [LW94, VHT00]). The use of traits and categories means the resulting substitutability evaluation will continue to work if we modify the specification language by adding new parts of specification, or devise new classification dimensions.

On the other hand, the approach allows us to leave out parts of the specification from the assessment. This feature can be used to loosen the requirements on substitutability if we know we are able to handle the resulting misalignments by adaptation. The result is an increased chance on substitution, i.e. a larger set of possible replacement components. For example if we can provide an adaptor modifying SOFA component protocol then the `protocol` trait can be left out of the assessment. The re-statement of Larsson's compatibility levels presented in this chapter another example.

The second use of leaving out traits in compatibility assessment is the application of the method in different phases of component lifecycle. In particular, design-time features or properties in the component specification will not be used at run-time. Thus they need not be considered in the case of component hot-swapping.

In addition to the advantages of static checking noted in the Introduction, the presented methods make it possible to compute the data for compatibility assessment in advance. This can be beneficial for components with extensive declarations, including the specification of semantics for which comparison algorithms sometimes have substantial computational complexity [PV02]. When used in conjunction with meta-data containing pre-computed compatibility data (see Chapter 6), this checking can be done in linear time.

5.5.2 Limitations of Our Method

Our notions of component substitutability is based purely on subtyping so some trivial changes (e.g. parameter swapping or element renaming) mark the components as non-compatible. This is mostly desirable since the me-

thod is targeted at fully automated updates with as little additional programming as possible. However, in many cases such non-compatible differences can be easily handled by software adaptation. Our methods can currently help in providing detailed information about the place and nature of the changes but do not otherwise count with adaptation.

Another possible deficiency of our approach lies in the handling of semantic aspects. First, such aspects often relate to the whole component and do not fit well in the contravariant scheme on which substitutability is based. Second, semantic and quality of service aspects often have close relation to global architectural rules for application consistency. Although our approach does not currently consider such global rules, it should not be difficult to extend the notion of context in this respect.

Obviously, our methods cannot discover incompatibilities which are due to implementation differences not reflected in the specifications. However, this issue is outside the scope of this work (as described in the Introduction); it can be from another viewpoint seen as a motivation for using suitable (read: sufficiently rich) specifications.

Use of contextual substitutability assumes an ability to set-up bindings automatically at component substitution. The key issue here is to automatically bind the new required elements of C^r to the available provisions found in the context. Although this requires suitable methods of finding and matching the elements to be implemented in the component runtime infrastructure, these issues are outside the scope of our work.

If the system has to determine the context and/or compare the component specifications directly (without pre-computed comparison results), the overhead of substitutability checking depends on the properties of specifications used. This may be substantial especially for behavioural quality attributes. For example, determining SOFA protocol conformance may lead to exponential complexity problems [PV02]. This is why we stress the possibility of pre-computing the comparison data (see the next chapter) with the net effect of linear time substitutability checking.

As a last point, we note that in many systems it is not possible to determine whether a component actually needs all the required features as declared. For example, if the `WebDAV` interface of the `HTTPClient` component is not used (see 5.1 on page 96), no calls will be issued on the required `XMLParser` interface. Such knowledge would be useful in the definition of context and in contextual compatibility assessment. A step in this direction is the work of Reussner on parametrised contracts [RS02] in which the relationship between provided and required elements can be established by source code analysis.

5.5.3 Compatibility and Real Life Development

We would like to conclude the discussion of our approach by giving an example of changes which can happen in realistic development scenarios. The example shows that there are situations which cannot be handled by any compatibility checks based on subtyping — as an antidote to the examples given at the end of previous chapter.

Consider again the following baseline declarations from a simple SOFA addressbook system.

```
typedef  short  PID;

struct Person {
  PID Id;
  string Name;
};

struct Address {
  string Street;
  short Number;
  string City;
  string<10> Phone;
};

typedef sequence <PID,1000> ListOfPID;

/**
 * R/W access to address book data
 */
interface IAddressBook {
  PID addPerson(in Person data);
  void delPerson(in PID person);
  void updateAddr(in PID person, in Address addr);
  Person getPerson(in PID person);
  Address getAddr(in PID person);
};
```

The following small changes, fairly natural and common in standard development process, make any component that uses the new `IAddressBook` interface incompatible with the old version.

```
struct Person {
  PID Id;
  string Name;
```

```
    string Nick;
  };

  struct Address {
    string Email;
    string Street;
    short Number;
    string City;
    string Phone;
  };
```

The reason is clear – the `Person` and `Address` types are used both as `in` and `out` parameters in `IAddressBook` methods. Therefore the changes make the interface type incompatible with the previous version.

When compatibility is more desirable than possible clarity of the interface, a partial remedy can be found: we create a separate type and corresponding methods for the data added to the `struct Person` as shown below. This will create a clean *specialization* difference in the interface; the new version will be a subtype of and thus compatible with the old one.

```
  struct Online {
    string Nick;
    string Email;
    string URL;
  };

  interface IAddressBook {
    // the old methods plus
    void updateOnline(in PID person, in Online data);
    void getOnline(in PID person, out Online data);
  };
```

Obviously this approach will not work in all circumstances, for example should we desire to generalise the `string<10> Phone` part of the `Person` type to `string Phone` (the second case here). Thus there will always be room for interface adaptation methods as well as a need for manual control over the compatibility assessment process.

5.6 Summary

In this chapter we presented a formal basis for checking substitutability of black-box components. It is provided by the definitions of component

substitutability and backward compatibility, based on our type-aware comparison of ENT representations of component specifications.

The strict substitutability provides a formulation of the subtyping relation on component specifications. It uses the standard notion of contravariance and applies it to the specific aspects of components — the existence of explicit declarations of the required elements and of the ties between the provided and required parts of the interface. The ENT model lets us also define partial substitutability levels, where the scope of the assessment is restricted to elements with selected classification properties.

The key contribution of this chapter is the notion of contextual substitutability, in which the knowledge of the deployment environment of the current component is used. This enables us to relax the usual requirement for a subtyping relation between the components, increasing the chances for successful substitution. The relaxation is possible because component programming involves the deployment phase in which component interconnections are [re]defined. This creates a chance to modify the subject of the subtyping assessment, using the actual and potential bindings of the to-be replaced component.

The practical results of the work include the redefinition of component compatibility levels described informally in other works, and a method for ensuring safe component updates in the SOFA framework. Also, an existing research prototype of the SOFA framework has been augmented to include substitutability checking. It uses the implementation of component declaration comparison mentioned in Chapter 3. (At the time of writing this text, the implementation is not fully complete though: support for obtaining the necessary component specifications from repository is yet to be provided.)

The discussion of the method emphasises its genericity which makes it platform-independent and able to accommodate future developments. On the other hand it confirms the standard weakness of subtyping, that even simple changes common in software development lead to component incompatibility; we therefore acknowledge the usefulness of on-demand interface adaptation.

Chapter 6

Compatibility and Versioning Related

In many software deployment and application installation systems the distribution packages contain meta-data which describes their purpose, version, dependencies and compatibility information. Such meta-data is used in what we call distribution activities — trading, installation, upgrades, deployment and configuration.

In our work we are particularly interested in the installation and upgrade phases, at the granularity of software components (rather than whole applications). It is clear that in particular, upgrade as a replacement of a current version by a newer one is an important activity in long-lasting systems.

One of the key conditions for the success of these actions is that the introduction of a new component (or a new version) does not lead to misbehaviour of the already existing application(s). Usually this is summed up in the requirement that "the new component must be compatible with the given environment". However, the common experience of system administrators is that very often upgrading means problems — broken dependencies, changes in data formats, differences in library interfaces, etc. Thus improvements in the automation and reliability of upgrades are highly desirable.

In this chapter we describe a step towards such improvement, using the results described in previous chapters. Our approach uses meta-data which combines revision data with indications of compatibility between component revisions. It is used prior to the upgrade to achieve reliable and at the same time fast substitutability checks. A key achievement is that this compatibility information is pre-computed once (on component release) and stored in a form which allows the checks to run (any number of times) in linear time.

The structure of the chapter is as follows. We first describe our motivation for and approach to combining revision and substitutability data of

components. In Section 6.2 we introduce the general scheme for the meta-data which embodies this combination. Next, a concrete realization which uses XML format of the meta-data for the SOFA system is outlined. The chapter ends with a discussion of its results and a summary of the achievements.

6.1 Relating Versioning and Compatibility

In long-lived systems, upgrades to new versions are the most likely forms of component substitution — introducing new functionality in downstream revisions, providing alternative variants, or patches with corrected bugs. Therefore compatibility is most often examined on components from one version family.

In this section we first look at the motivation for this view in more detail. Subsequently we describe an upgrade mechanism which uses the close relation betweeen version information and compatibility assessment.

6.1.1 Motivation

Component upgrade is a special case of component substitution, in which revision N is substituted by revision $N + 1$ (or $N + m$). The difference is that in upgrades, the new version has the same name and usually introduces just an incremental change in its interface or implementation. It is assumed that the upgrade does not break the consistency of the configuration.

This fact can be formulated as an intuitive rule, that "$rev(a) < rev(b)$ implies that b can substitute a but not vice versa." This rule expresses the tight relation between the compatibility of different versions of a configuration item and their version identifications. It is used (implicitly or explicitly) by many software installation systems to ensure system consistency.

In simple form, the rule leads to the plain comparison of revision numbers to determine substitutability. This approach, used e.g. by DCE or Java product versioning, relies entirely on the above interpretation of the rule and usually uses an informally defined interpretation of the revision ID parts.

A more complex approach used by software installation systems (Linux package management, DMI, etc.) relies on meta-data with information about the application's compatibility. In general, this data contains two kinds of information – compatibility (which versions of the application the current one can safely replace) and dependencies (which versions of other applications must be present so that the current one can function). This leads to greater reliability of upgrades as the data supports decisions in several dimensions of the problem.

However, the problem of both of these approaches is that the meaning of the revision IDs and meta-data is not formalised. Also, very often the data is created manually or semi-manually by the developers of the applications. These problems result in considerable manual effort, and introduce the potential for misinterpretation and errors.

Ensuring configuration consistency is a complex issue which involves both structural aspects (fitting interfaces) and semantic aspects (expectable behaviour). Reducing upgrade compatibility checks to a simple rule and using potencially errorneous data may be dangerous as it may result in false positives, allowing upgrades that should not happen. This issue is even more important in the area of component-based applications which should support automated, unattended upgrades.

6.1.2 Generic Mechanism of Upgrades

With the above motivation and previous results in mind, we can define a mechanism for component upgrading with rigorous compatibility checking. The mechanism is inspired by the above intuitive rule and the process used by the application istallation systems. Its subject are however the "smaller" software components which are the focus of our work.

The steps of the mechanism are as follows:

1. Decide which component in which version (C^c) should be upgraded.
2. Obtain the desired new version (C^r).
3. Check whether $rev(C^c) < rev(C^r)$.
4. (If yes) Check whether the new version can substitute the current one without breaking the consistency of the applications which use it.
   - First check for strict compatibility.
   - If it fails, obtain context and check contextual compatibility.
5. (If compatible) Perform the upgrade, i.e. replace the current version with the new version.

In this work we are interested in the core activities of this process. In step 3, the revision data of components as defined in Chapter 4 are used. Step 4 uses the compatibility definitions described in Chapter 5. Steps 1, 2 and 5 are outside the scope of our work.

More discussion of these decisions is desirable. Step 3 is included primarily as a sentry which prevents downgrading. In principle, we could use any revision identification scheme which has the monotonicity property (Section 4.4 on page 82). The ENT-based revision data however gives us

additional information apart from the time ordering of the versions: which parts of the component have changed between the two revisions. Thus we may leave out the unchanged parts from the checks in step 4, leading to improved efficiency of the process.

In the checks of step 4, the current research and industrial systems use either hand-written compatibility data, or attempt formal methods in the checks. The approach we propose is an attempt to get the best of both worlds. We believe it is important to use the type-based comparison because it is the only way to ensure run-time safety. Nevertheless, decisions based on meta-data are appealing in their simplicity.

As we are able to represent the results of type-based comparison by classification values (Chapter 3, Section 3.2 on page 47) we do not have to make compromises. In our approach, the comparison is automated and performed in advance, at the release of the given version of the component by the provider. Its results are stored in meta-data distributed together with the component itself, and used by the installation system in the compatibility checks.

In the following two sections, we present first the contents of the meta-data in generic terms, and then a concrete implementation for the SOFA framework.

6.2 Meta-data: The Integrating Element

Several current component systems [Sun01a, vdHW02] and software packaging systems [Hes03] use meta-data that contain information necessary in trading, assembly, configuration and deployment of components. Neither of these systems however supports component versioning in the way we require, and consequently the available meta-data specifications do not include corresponding provisions.

As we have shown in the previous section, there are compelling reasons for such provisions. In particular it is advantageous to merge versioning data with indications of compatibility, as both are useful during upgrades. Component meta-data is a natural place to put such indications.

6.2.1 What the Meta-Data Should Contain

We propose that the meta-data of a given component version, C^v, contain primarily the elements described below. Of course, there are many other pieces of information that should or could be included. Here we will concentrate only on those directly related to the problem of reliable upgrades.

- Identification of the component, i.e. the name of the provider, the name of the component, and the namespace in which it exists.

- Identification of the version, i.e. the branch, revision and variant description of C^v.
- Difference of this revision against the previous one C^{v-1}, i.e. the base data for determining their compatibility.

Secondly, there are some data which we suggest should be included in the component's meta-data although it is not strictly necessary:

- The revision history, i.e. the path from the first release revision (C^1) to the current one.
- Differences between the subsequent versions listed in the history.
- Pairwise differences between the current version and each revision listed in the history.

These optional sets of data can be easily reconstructed from the meta-data of the intermediate revisions. However, these revisions may not always be available at the place of upgrade. In such case the data would have to be obtained from the provider on request which would unnecessarily delay the upgrade.

Sometimes it would be beneficial to increase the accuracy of compatibility checks between distant revisions (i.e. where the revision history between the current and the new versions contains at least one revision). For this purpose, the meta-data should additionally contain the differences of the current version and each historical one.

The following paragraphs describe the structure and role of these meta-data elements in more detail.

Revision and Diference Data In the motivation part (Section 6.1 above), we stated that it is beneficial to put the revision and compatibility data together.

The proposed meta-data therefore contains two tuples, d_{ENT} and d_T. The first represents coarse-grained data: $d_{ENT} = (R_C, D_C)$; R_C is component revision data, $D_C = (d_E, d_N, d_T)$ where $d_\xi = \mathit{diff}(\xi^v, \xi^{v-1})$ is the result of comparison of the E, N, T categories in the given and preceding revision.

In other words, this is the description of revision and difference at the category level, using the key category set E-N-T. Its primary purposes are (1) to provide human readable revision identification, and (2) to provide source data for strict compatibility assessment.

The second tuple, $d_T = (R_D, D_T)$; R_D is detailed revision data, $D_T = (d_1, \ldots, d_n)$ contains corresponding difference information at the granularity of traits (assuming the specification language of C has n traits). This

detailed data is included in the meta-data to provide additional information about the place and nature of changes from the previous revision.

Component Revision History The revision history is an ordered set $H = \{d_{ENT,1}, d_{ENT,2}, \ldots, d_{ENT,v-1}\}$. It contains the category-level data of all preceding revisions of the component.

The purpose of the history is to speed up compatibility checks of distant revisions by computing their difference from the *diff* values of the intermediate revisions. This is done by a fairly straightforward combination of the result of previous computation with the current value.

Let K_u, K_v, $u < v$ be the contents of category K in the specifications of two revisions of a component, and $\{d_i^K, u \leq i < v\}$ be the differences in the intermediate revisions K_i. Then the value of d_v^K is

- $none$ if $\forall i, u \leq i < v, d_i^K = none$;
- $specialisation$ if $\forall i, u \leq i < v, d_i^K \in \{none, specialisation\}$;
- $geneneralisation$ if $\forall i, u \leq i < v, d_i^K \in \{none, generalisation\}$;
- $mutation$ otherwise, i.e. if $\exists j, u \leq j < v, d_j^K = mutation \vee$ $\exists i, j; u \leq i, j < v : d_i^K = specialisation \wedge d_j^K = generalisation.$

Pairwise Differences The method of comparison via revision history described above suffers from one systematic deficiency: a combination of the *specialization* and *generalization* differences — which in reality may still result in a subtype relation — is always flagged as $mutation$. For instance, if C^m adds interfaces I_1, I_2 to its $Exports$, and then C^{m+1} removes I_2, then it is still a subtype of C^{m-1}. It would however be marked as type incompatible, using revision history data alone.

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. Its computation is straightforward, and the data can be e.g. stored as an extension to the revision history.

The reason for the optionality of the pairwise differences is the time penalty it incurs. If a component has a large revision history, the computation of this data may significantly delay the release of the component. The provider can therefore choose not to include it in the meta-data.

6.3 Use in the SOFA Framework

The previous section describes the general design of meta-data which embodies the combination of component revision and compatibility informa-

tion. This section describes the prototype implementation in the SOFA framework. In this case, the meta-data is automatically derived by comparison of the CDL component specifications. It is used to enable the co-existence of multiple versions in a system, and to ensure safe upgrades including the case of dynamic update via the DCUP mechanism.

This meta-data contains mainly version and difference data, but obviously also component name, description, references to key implementation objects in the binary, etc. Since the primary users of this form are various tools, the emphasis is on completeness and an easily parsed structure.

6.3.1 Metadata Formats

In SOFA, the meta-data is present in two places. The revision data is made part of the CDL component specification by extending the grammar of the language and its compiler. The complete data in XML format is stored in the component distribution package.

CDL Extension We observe that declarative aspects of computer programs have become more important in recent years. Additionally, versioning is a key aspect of development and it is customary that programmers put some kind of version information as a visible part of program source.

We therefore extend the SOFA CDL specification language to include version data description blocks for each user-defined type. Thus an important part of the meta-data is accessible to the developers and users directly in the component specification. Details of this CDL extension are described in Chapter 4 of this thesis and in Appendix C.

Persistent XML Format For distribution with the components, we use a XML format for meta-data (and consequently revision data) representation. It can describe fairly complex structured data, is easibly extensible and automated manipulation is supported by many tools. With respect to version data, it allows us to put all levels of revision data in one place.

This representation of revision data is illustrated in Appendix C together with the complete document type definition (DTD) for the format. The example in Section C.4 shows how the different levels of revision data can be merged with corresponding differences between the appropriate specification parts. Moreover, the chosen XML markup results in very readable meta-data description.

The data contains also the revision history of the component. For complete representation of the path in the revision graph, both the branch and revision identifiers must be used. As an compromise between size of the data and its expressiveness, we include only the component revision and

difference data in the history. This is sufficient for basic identification and compatibility assessment, and detailed checks can be done on easily obtainable detailed data or CDL specifications.

6.3.2 Repository for Versioned Components

When versioning is enabled for components, it is necessary to enhace the repository where they are stored. In current component systems, the repositories (as in CORBA) or containers (in EJB) are not implemented to contain more versions of the component with the same name.

Based on the above described meta-data, we have developed a prototype implementation of SOFA Template Repository. It uses a hierarchical organization of the storage which corresponds to the tree-shaped namespace of SOFA IDL types including their branch and revision identification. The meta-data included in the component distribution package is read to insert the component into an appropriate place.

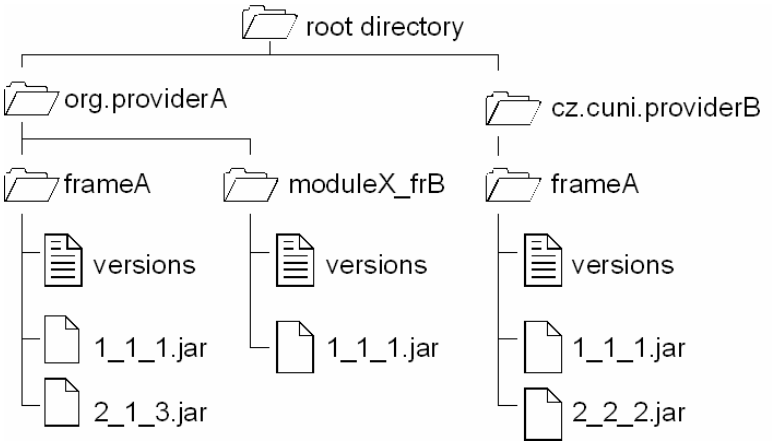

Fig. 6.1: The structure of the Template Repository contents.

For its various uses, the repository provides several interfaces including `In2TR`, `Run2TR` and `QueryTR`. The first one contains methods to insert component distribution package into the repository, the second to obtain parts of the component (the component manager and builder classes, for example), and the last one to search repository contents to find a particular (version of a) component.

To identify versioned components at runtime (e.g. in `Run2TR` methods), a data structure called `ComponentDescriptor` is used. It contains the provider, namespace and component name, and its version data. The contents of this structure is read from the XML meta-data via the `QueryTR` methods.

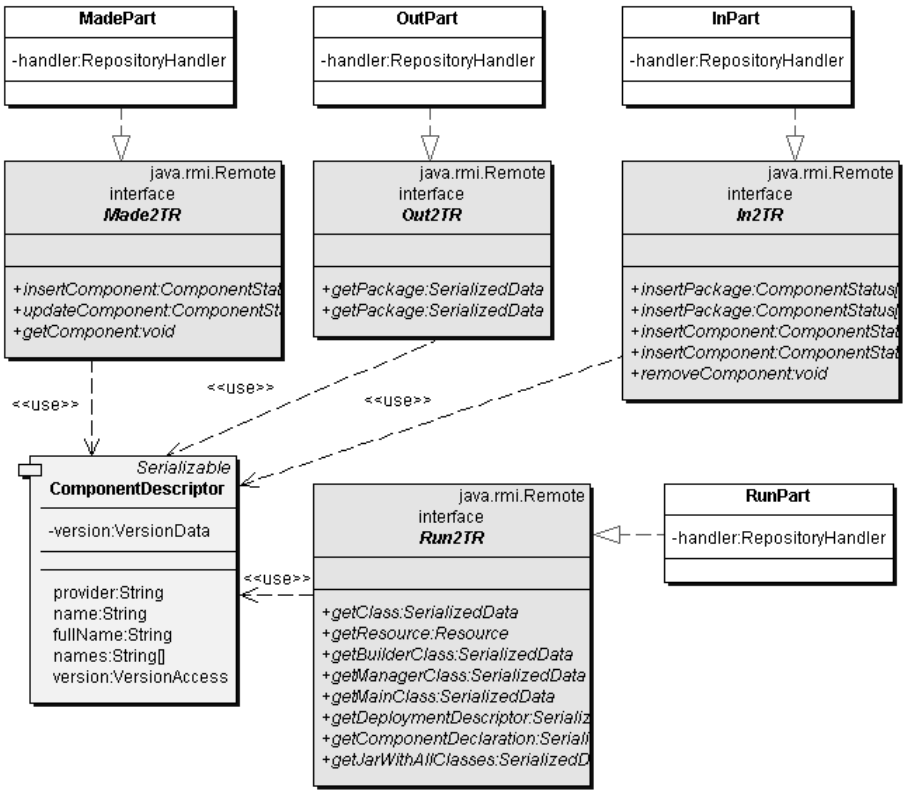

Fig. 6.2: Key interfaces of Versioned Template Repository.

6.3.3 Component Updates with Versioning

The update mechanism which is used in the prototype SOFA implementation is also easily extensible by the compatibility checks. When a user requests the update[1] of a selected component, the SOFAnode ensures its viability.

This is done by running the strict compatibility checks on the current and replacement components' version data, prior to the actual update. Only if subtype relation is confirmed (i.e. the update acually means an upgrade), the replacement is allowed. The present implemenation does not yet include the contextual compatibility assessment.

The `VersionData` structure is the run-time representation of the corresponding `<version>` part of the XML meta-data. An important operation which realizes the compatibility check is `compareTo()`; it uses the revision and difference parts of the meta-data. It is therefore here that the results

[1] The SOFA term for run-time replacement

of this chapter — the close relation of component versioning and compatibility — is used at run-time.

6.4 Summary and Discussion

In this chapter, we have presented how the key results of our work — the specification-based revision identification and compatibility assessment — can be merged into a coherent whole. The result is a conceptual proposal for a meta-data scheme (plus a sketch of its practical implementation) used in checking substitutability during component upgrades.

The key achievement is the realization that meta-data can contain the base data for compatibility checks and which can be created once upon component release. This leads to reduced computational complexity of these checks and consequently shortened time needed for the upgrade.

A prototype implementation of SOFA component repository has been developed. It uses the XML form of component meta-data, can store multiple revisions of a component and allows advanced querying of the repository contents, including the possibility to specify revision number ranges in the search.

6.4.1 Advantages

The essential aspect of our approach is the close and well-defined relation between revision and compatibility data. This data is automatically derived from the component specification (which is also used to generate its implementation skeleton). In effect, the system proposed here makes component upgrades more reliable. Its rules ensure that only type-compatible downstream revisions will be replaced, which reduces the probability of run-time errors.

The automated meta-data creation relieves the developers of a work which is usually percieved as administrative and therefore tends to be neglected. Even if the data were not included with the distributed component, the target systems may be able to reconstruct it without human intervention, using available component specifications.

The bundling of the meta-data with the component itself has several other advantages. First, it makes it easier to maintain consistency of the whole component distribution system. Is also allows the coexistence of multiple versions of a component in a repository, prevents duplicate downloads of existing versions, etc. Last but not least, the clear relation of the structure of the data to the parts of the specification is helpful for humans who need to evaluate the information.

6.4.2 Issues

The downside of our approach is the need to design and implement the algorithms for parsing and comparing the specification, needed to generate the revision and change data. This may not be a trivial work especially for more complex semantic specifications.

As mentioned earlier, the pre-computed differences stored in meta-data reduces the time complexity of the compatibility checks. However, if the system has to determine the context and/or compare the component specifications directly, this advantage of the distributed meta-data is lost.

This may easily happen in the case of upgrading from an earlier revision (rather than from an immediate predecessor) as then the meta-data contains only basic difference information. Also, when there is no path from the current to the replacement component in the version graph (e.g. parallel branches), no compatibility information is available in the meta-data.

Chapter 7

Overall Evaluation and Related Work

The work described in this thesis spans several areas that in literature tend to be treated separately; as this thesis shows this is needless or even disadvantageous. In this chapter we will therefore consider each area in turn, discussing works that are influential at large or were inspirative for us in some particular point. We will also note the points of contact between these areas.

The key purpose of this chapter is to compare our work with these related research efforts, in order to show its merits, differences and possible weaknesses. We would further like to stress its originality in several respects:

- In our approach to component meta-modelling, we aim at generality and relevance for human users. Existing meta-models tend to be limited in their modeling capabilities and oriented towards the technical "wiring standards".
- Substitutability of components is a key issue in a longer term, to which our work provides a flexible, well-founded and practically applicable method. Related works are mostly concerned with foundational issues and techniques, neglecting somehow the overall picture.
- Version identification that results from rigorous modeling and provides more than just discrimination tags is another novel aspect of our work. The research in versioning resulted in many achievements but still does not provide good answers to the challenges of automated version identification, selection and composition.

7.1 Component and Interface Meta-Models

During the last several years, dozens of component models and several meta-models have emerged in both research and industry. In this section

we provide an overview of the *meta*-modelling efforts, because of their direct relation to our ENT model and their higher importance for the developments in component programming.

7.1.1 Distilling Commonalities from Component Models

The component models implicit in the existing component frameworks [M+95, S+95, Sun97, PBJ98, Han98, ACN02b, C+02, OMG02f], though defined at different levels of abstraction and formalism, use a fairly consistent set of concepts and vocabulary. This makes it possible to extract common modelling features into a meta-model. Meta-models consequently allow to design components in a platform-independent manner (then generate their implementation code for the selected framework) and to integrate the tools and technologies that use them.

Despite these compelling reasons there are few works that attempt at creating such meta-models, or even "just" unifying specification languages (like ACME [GMW97]) or classification systems. What we are interested in here are abstractions at the M3 (meta-metamodel) layer of the metadata architecture described in OMG's Meta Object Facility specification [OMG02g, Section 2.2].

In a good survey of early ADLs and component models by Medvidovic [MT00], a classification system is defined for the purpose of model comparison. It separates component features into the syntactical structures ("interface"), semantics, and non-functional properties. This is similar to the ENT classification system, with two differences.

First, Medvidovic does not differentiate operational and data features (the $Kind$ facet in our classification); we see this as an shortcoming since the distinction between data and operational features is an important one from both practical and theoretical point of view. Second, his semantics and non-functional properties are in the ENT model both modeled as quality attributes (in the $Nature$ facet). While we admit their separation would be a more accurate approach, the current selection of terms in the $Nature$ facet does not limit the use of the ENT classification system in applications encountered so far.

The Vienna Component Framework (VCF) [OGJ02] is a unified component model (in fact, a meta-model) that aims to support interoperability of components from different component frameworks. Its design is based on an analysis of current industrial component models — Microsoft COM, Enterprise JavaBeans, CORBA distributed objects, and JavaBeans.

Despite interesting achievements on the implementation and interoperability sides, the design of the VCF meta-model does not offer advanced or novel modelling possibilities. The predefined *feature*s in a VCF component interface are a method, a property, and an event; all of them are on a

too fine granularity for component modelling and development. The framework does not use any IDL, relying on naming conventions and its own introspection mechanism in its Java implementation. As discussed at length in the JavaBeans case study (Section A.3), we consider this a shortcoming in view of analysis and modelling purposes.

Rastofer [Ras02] has developed a simple meta-model which is derived directly by extracting common basic features of ADLs and major industrial component frameworks. A component can have *port*s (which denote service points, provided and required) and properties. Ports have name, type and multiplicity, properties have name and type. The meta-model also includes connectors and constructs for describing composite components. An interesting aspects from our point of view is that the author defines the conformance relation, i.e. component subtyping, on the meta-model level, similarly to our approach (Chapter 5). The motivation is also similar — the resulting generality of the relation.

The meta-model itself is reasonable, generic in the intent, but rather poor in expressiveness. The only two constructs to describe features on component interface — ports and properties — are consequently used for rather different notions, e.g. a required port can denote a method (JavaBeans), receptacle interface (CCM), or event sink (CCM). We prefer to separate these notions in the meta-model more clearly; to this end we use the rich ENT classification system and suggest separate language keywords for separately classified traits.

We believe Rastofer's model also has an incorrect understanding of the provided-required roles. In his model, role assignment is based on the direction of control flow. This leads to the classification of CCM event sources as provided, and sinks (as well as JavaBean methods) as required features. It is the exact reverse of our classification which is based on the role of the element in inter-component dependencies and supported by the standard usage in many component models including the CCM.

The consequence in Rastofer's model is that a component event sink (its "server" port) would require at least one event source to be bound to, and event source (on which the component emits events at its own will) does not need a binding to an event sink. This is contrary to both reality — it would mean that a server needs at least one client to function at all — and author's own definition of the provided and required roles.

7.1.2 Meta-Models Defined as Such

The component meta-models defined as such (the "a-priori" models) are created as a prerequisite of or a means to defining concrete component model(s). The OMG Meta Object Facility [OMG02g] is the primary industrial meta-modelling framework used today. It defines by default a four-

layer ($M0$ to $M3$) meta-data architecture which is important as a generic context of our ENT modelling research. Language-independent notations like UML and IDL are positioned on the $M2$ layer, and since the ENT model generalizes from these, it is positioned on the $M3$ (topmost) level of the MOF framework.

For our purposes the MOF as such is insufficient because it is an *object*, not component meta-modelling framework. (The elements of the framework are basically a generalization of the features offered by UML). In particular, MOF does not provide first-class constructs for components (with separation of provided and required element roles), behavioural specification or structured data interface elements.

The UML Profile for Enterprise Distributed Object Computing Specification (EDOC) [OMG02h] describes the model-driven architecture approach to specifying enterprise distributed applications. In particular, Chapter 3 defines the Component Collaboration Architecture, a UML profile for component-based modelling. It provides good modelling features and flexibility in terms of current industrial standards — the component can have ports (operational or data access poinst) of several kinds, typed properties and also state-machine based behavioural semantics (the "choreography" part of port protocol). A sketch of a graphical notation is included in the specification; far from complete, this notation example can be understood as a supporting evidence of the close link between meta-modelling and graphical representations (cf. Section 2.5).

Despite its flexibility and elaborated features, we find several problems in the EDOC profile. To start with, the term "component" is very loosely defined in the specification ("something that is composable" [OMG02h, Section 3.3.3]) which makes it difficult to interpret its meaning and relate meta-model's structures to concrete models. This is why in our work we prefer to restrict ourselves to the black-box components as discussed in the Introduction.

Concerning port types, EDOC is interesting in the distinction of three kinds of component interfaces which include mixed in-out interfaces ("protocol ports", similar to ArchJava ports) and data-oriented interfaces ("flow ports"). While this may add features to models conforming to EDOC, we believe our ENT classification system embodies a more general approach.

Our biggest technical concern with ports is their unclear (if not completely missing) relation to type systems. Since in the meta-model, ports of any kind do not refer to their types by any identifier or association, they have to define the type explicitly ever again by enumerating their contents. Even the "interface" classifier as a special kind of protocol does not have to have a name and thus cannot be referenced. This is really impractical for component design and implementation; it even does not map to some important component frameworks, most notably the CORBA component

model.

The mixing of provided and required roles in protocol ports makes subtyping difficult, resulting in complications when checking component substitutability in concrete models that use this feature of the EDOC profile. These ports additionally allow to mix the specification of syntax (operations) and semantics (choreography) without distinction by identifiers, associations or language constructs. We believe that a clear separation of these concepts on the meta-model level is beneficial for component modelling, implementation and analysis. In our work, we solve this issue by the *Nature* and *Role* classification facets, which lead to separate component traits and consequently to related grammar rules/keywords of a concrete models' specification languages.

As a last point in this subject, we note that the EDOC meta-model allows recursive composition of interfaces. This feature adds flexibility but we have doubts about the usefulness of such abstraction, and feel that recursively defined ports are overly complex to understand, model and analyse. Since none of the current research or industrial component frameworks supports such feature, we have not included recursion in the definition of the ENT model's "specification element" term.

The Fractal framework [BCS02, C+02] defines both a meta-model (the authors call it a "general model") and one concrete model which is its specialisation. The goal of the Fractal team is to create a component model more general than the current industry ones. To this end, the components (called *kell*s) in the Fractal meta-model contain a "membrane" which can control and adjust kell's behaviour and allows component sharing and hierarchical composition. A notable characteristics of Fractal is the possibility to pass kells through interfaces, to allow dynamic evolution of component applications.

Concerning the component specification elements supported, the meta-model is quite poor — a component interface consists of only interfaces (sets of signals) and one kind of behavioural specification. Missing are for example configuration properties and "illities" (although the related ADL for the Fractal concrete model [Fra03] uses attributes in component templates). On the other hand, Fractal as the only component framework known to us makes explicit the optionality of elements at run-time by its "contingency" tag on component interfaces. Our *Presence* classification facet was directly inspired by this notion.

Finally, the meta-model described by Seyler and Aniorte [SA02] is based on a work in the area of system engineering using reusable components. The meta-model is unique in the separation of the data and control flow in component description. The component interface is split into functional (control) and data (information) parts, and orthogonally into the standard required and provided roles. The control part contains function invocation,

synchronization and resource access points; the information part contains information input and output points.

This meta-model provides features we think the commonly known models are lacking, and supports our position that data elements should be specified on component interface. On the other hand, its notion of information points is a very general concept which needs more concrete mapping on real objects – files, data streams, tables etc. A formal description of the meta-model is missing, as well as a mapping to an interface specification language or to an implementation form, which makes the concepts found in the meta-model stand on a rather weak ground.

7.1.3 Summary

Should we summarise the topic, we would do so by observing that in the end, all of the above meta-models explicitly enumerate the possible kinds of component specification elements (i.e. the possible traits). The approach we chose is rather to enumerate the *properties* of such elements, and let the ENT model user create its own set of traits, forming a concrete component model. We believe this results in much increased flexibility, openness as well as better chances to create mappings to present and future component models and frameworks.

7.2 Component Comparison and Substitutability

The research and practical use of comparing specifications (often called specification *matching*) and evaluating component substitutability has been going on since 1980s. Yet at present, there seem to be few works targeted at the needs of black-box components.

In this section we will first discuss the approaches to specification comparison. The following parts will treat subtyping-based approaches to ensuring substitutability and then approaches specific to black-box components.

7.2.1 Specification Comparison and Matching

There exist two fundamental approaches to the comparison of software and interface specifications: text-based differentiating and grammar-based methods. The first ones are mainly used in versioning systems, the latter ones in systems for software search/retrieval and in relation to language typing systems.

The text-based methods use a line-by-line comparison of (mostly plain ASCII) texts. This approach is most prominently represented by the `diff`

tool found in UNIX systems and used e.g. in the `rcs`-based version management systems [Tic85, Ber90]. While it is very common in practice, its results are inappropriate for the purposes of component substitutability evaluation because they do not reflect the semantics of the text compared (e.g. changing a coding style in C source code may show as a bigger difference than more substantial changes like changing method signatures).

Our approach belongs to the grammar-based methods that use syntactical analysis to extract information from the specification or source code and then compare the results. In a similar approach, though for a different purpose of library function retrieval, Zaremski [ZW97] as well as Hemer and Lindsay [HL99] use matching of specifications (function signatures plus simple semantic descriptions) based on their syntactical structures and type rules. Zaremski provides relaxed matches (generalized and specialized) that are based on an idea similar to our specialization and generalization differences.

Loosely related to these issues are also development environments that provide syntax highlighting and/or syntax-driven editing of source code. Although they do not provide means of handling changes or differences, they share our view that the information available in the syntactical structure of the code should be utilised to a greater degree.

7.2.2 Subtyping-based Substitutability

The assessment of component substitutability and compatibility uses either grammar- and typing-based specification matching, or external, in-advance provided compatibility data. The latter approach is discussed in the following subsection; here we treat the first group to which most research efforts (including this thesis) belong. Their focus usually revolves around the need to provide more flexible substitutability relation than the strict (contravariant) subtyping.

In the Inscape environment, Perry [Per87] has defined strict and upward compatibility for functions with pre- and post-condition semantics that uses separate handling (subset comparison) of parts of the specification. Specification matching of Zaremski and Wing [ZW97] in fact also defines substitutability for functions and modules, through the goal of finding components that are similar to, and therefore can be substituted for or bound to, a query component.

Although highly relevant for their algorithms for change analysis, these approaches work with declarations of too fine granularity (usually method signatures). To indicate changes at a level useful for our goals, the data which they produce would need to be aggregated using our notions of traits and categories.

Substitutability is often formally expressed by the subtype relation, de-

fined in (or on top of) a given language's type system. Most industrial specification languages (e.g. CORBA IDL) provide component specification on the signature level, for which standard contravariant subtyping is usually used [Car97, SC00]. Some works mentioned above (Zaremski and Wing, Perry) relax the subtyping rules in various aspects to increase the chances of substitution.

More importantly, there is a growing body of work on *behavioural subtyping* [LW94] which deals with specifications enhanced by semantic descriptions. Most approaches relevant to our work define subtyping rules for specifications with operation pre- and post-conditions plus object/component invariant [Per87, Mey92], interaction trace ("protocol") descriptions [PV02, VHT00, RS02] or variations of finite-state machine models [Nie93, FW00]. Such information increases the reliability of substitution as the replacement component matches the behaviour of the current one more closely. We should add that trace/protocol descriptions are examples of interface elements that fall into the (so far scarcely populated) "Ties" category of the ENT model.

Partially related to semantic descriptions are quality of service (QoS) specifications. They have as yet received less attention in the research community, despite the signs that practice would benefit from such specifications [Lav02, Han99a]. The QML (Quality of service Modeling Language) by Frolund and Koistinen [FK98] makes it possible to describe QoS properties of CORBA interfaces, which makes it an ideal vehicle for this purpose. In addition, the authors define a conformance relation for QoS profiles and contracts which can be used in our framework as the subtype relation. The only problem in their proposal is the lack of information on how to link QML specifications with the IDL syntactical structures.

Richer and more precise semantic specifications (e.g. formal notations like B [Wor96]) receive less attention in both component research and practice. The likely reason is that they require increased effort in writing (unsubstantiated by comparative increase in substitution reliability in mainstream practical situations) and their comparison tends to have high computational complexity.

An example of practically used, though not formally specified, substitutability based on subtyping is the Java binary compatibility specification [GJS96]. It defines the rules for changes in interface description, resp. class implementation that are considered compatible with respect to already existing compiled code. Java runtime then uses code reflection upon class loading and linking to prohibit incompatible code bindings. The problem from component-based point of view is that some of the rules are very low-level (including e.g. the treatment of changes in names of method formal parameters) and provide little help for substitutability of black-box components.

We would like to add two notes to the subject of subtyping-based substitutability. First, the concept of reduction operation on interface elements and categories defined in this thesis (chapters 2 and 5) is a generalization of trace projection defined by Fischer and Wehrheim [FW00] and protocol restriction defined by Plášil and Višnovský [PV02]. While its realization for other kinds of elements (interfaces, boolean expressions) may be complicated, these sources provide good substantiation for introducing such operation.

Second, we have encountered only a handful of works that would link subtyping with extensible specifications or meta-models, in order to create a more general scheme similar to ours. In one such work, Seco et al [SC00] use standard contravariant notion of component subtyping with the property that their definition of component type ($R \Rightarrow P$) in principle allows any metatype to be included in the R (requires) and P (provides) parts.

7.2.3 Component Substitutability and Compatibility

Substitutability of black-box components as such is the topic of several (though not many) important research works. In a broader view, Szypersky [Szy96] discusses the issue from the point of view of independent system extensibility. Among other things he argues that interface specifications are needed to enable independent extensibility.

Vallecillo et al [VHT00] is a good survey of issues in the area of substitutability of components. The authors define the terms *substitutability* and *compatibility* in a very useful way which has influenced our view. The latter term nevertheless is understood as a mutual correspondence of interfaces (to be) bound, so that their owners can interoperate — this is different from our use in Chapter 5 where the motivation is to express the ability to upgrade a particular component version.

Crnkovic and Larsson [LC99] define substitutability (termed "compatibility" in their papers) specifically for black-box components, providing several levels of relaxation: *behaviour* compatibility preserves the interface as well as non-functional characteristics of the component, *interface* compatibility preserves only the functional interface, and *input and output* compatibility considers just the data which the component exchanges with its environment. This is the first approach we know about to include the data aspect of component interface in compatibility assessment.

However, the authors do not mention any means of determining whether a component satisfies the given compatibility level. In addition, the levels actually do not constitute a hierarchy of gradually looser requirements — in Chapter 5 we show a formal redefinition of these levels and correct this deficiency.

As mentioned above, Rastofer in [Ras02] defines component confor-

mance relation at the meta-model level as an expression of substitutability; this relation is consequently equal to component subtyping. Similarly to us, he handles subtyping of the "tags" (in our terms), for instance port multiplicity. However, due to the limited generality of his meta-model the conformance definition lists the "tags" explicitly which would require changes in case the model is extended.

Besides subtyping, there is also another approach to component substitutability which uses externally provided compatibility data (instead of employing specification matching) and emphasize the role of an update manager. As it turns out, this is usually the approach of routinely used industrial systems.

The DCE environment [Pet95b] and Solaris/Linux libraries [BR00b] define the rules for changes in server or library interface that are considered compatible with respect to existing clients. Both systems use a version numbering scheme with a mapping to these rules to express interface compatibility and evaluate it easily at run-time: if a client requires a server or library and the required version number is bigger than that available (meaning that client's interface is a kind of subtype of the server's), the binding is prohibited to prevent invocations of unknown operations.

Although this approach has its merits and practical use (the idea of relating versioning and compatibility was particularly inspirative for our work), it is unsuitable for current state of art in component-based software. The key problem is that the objects of comparison and versioning are binary images, which consequently leads to overly strict and counter-intuitive compatibility rules — the standard example is DCE's susceptibility to reordering of interface operations.

It is surprising that Object Management Group (OMG) seems to pursue little activity in the area of object and component substitutability. While the Interface Repository [OMG02d] makes provision for using type version identifiers for rudimentary compatibility checking (roughly along the lines of the DCE rules but even less formal), this seems to be rarely used. The recent Online upgrades Request for proposals and available reponses [OMG01, OMG02a] deal with updates of object implementations only, i.e. do not allow interface evolution.

In his research on systems with update managers, Oreizy [Ore98] describes the use of an evolution manager to allow flexible 3rd party component software evolution. The manager uses a description of architectural rules and evaluates attempted changes (adding a new component, restructuring, ...) against them. Our approach is not directly related to this one, but we have noted that to make contextual substitutability operational the system needs to extract the context information. Thus a component similar to Oreizy's evolution manager is assumed to exist.

Some industrial software installation systems ensure the compatibility of applications with respect to prospective upgrade – e.g. the DMI initiative [Des98] or various Linux packaging systems [Bai97, J+03]. They use structured description of the software packages which include explicit information about required application/library versions and about compatibility with older versions of itself. Their main disadvantage from our point of view is that these descriptions must be supplied manually and thus may contain errors or not reflect correctly the real properties of the application implementation.

7.2.4 Summary

Except for several attempts, the current research of component substitutability is scattered and insufficient. Type-based or behavioural subtyping of one kind is usually assumed, and little has been done to use the advantages of defining component substitutability on the meta-model level or considering component environment in its assessment.

In our work we try to blend the strength of proven approaches (subtyping, formally defined meta-models) with the unique opportunities provided by component technology (the knowledge of the bindings and deployment environment, the roles of interface elements). We believe the resulting contextual substitutability is an important step towards methods of substitution which provide both reliability and flexibility.

7.3 Component Versioning

The situation with versioning targeted specifically for software components is probably even worse than with substitutability assessment. Despite early calls for action [BW98, Szy98], there seem to be only a few research works on this topic and just rudimentary support in industrial systems. In the following paragraphs we will look at the state of the art, considering also the link between versioning and substitutability, and use of meta-data.

The survey by Conradi and Westfechtel [CW98] provide the current vocabulary for the area of version management models and systems. Should we classify our ENT-based component revision identification according to Conradi, it would be an example of state-based extensional versioning, and its combination with hierarchical component composition (as e.g. in SOFA) leads to version-first selection in a total versioning scheme. Deltas are not supported as such, rather they are abstracted using the difference classification system.

A second survey by Estublier et al [E+02] reviews the state of the practice in the field. Its findings, among others, support our position that

RCS-based revision identification scheme (by which our scheme is partly inspired) is still by far the most acceptable to the software engineering community.

7.3.1 Industrial Frameworks

Version support in current industrial component frameworks can be characterised as "rudimentary, if any". Perhaps the most advanced use is found in Microsoft's .NET framework [Cor02] at the level of assemblies (not .NET components). Each assembly may declare a 4-level version "number" (the real datatype is string) used by the Common Language Runtime when binding digitally signed assemblies. The version number is effectivelly part of assembly identification, and it is also recorded in assembly dependencies (stored by compiler). With respect to relation to substitutability, by default only the assembly with exactly the same version number as recored can be bound; however this policy can be changed.

The CORBA Component Model [OMG02f] builds on and extends the CORBA IDL [OMG02c] and Interface Repository [OMG02d] specifications. In the Interface Repository, any `Contained` element has attribute `version` which means that components (as well as all user-defined types) can be versioned. However, the specification provides no information about version number structure and semantics.

Additionally, the specifications are unclear about the relation between this repository version attribute and the `pragma version` IDL directive: the value of the pragma is used in repository IDs but the specification does not mandate that it be inserted into the repository. In effect, version identifications in IDL and in the repository may diverge and is practically unusable for both version separation [SV01] and substitutability.

In the Koala component model [vO01] used by Philips, a package is the entity closest to our understanding of the term component. Packages are versioned, using a "M.m.p" scheme for release version identification set by the developer. However, the papers mention just hints, not formal rules about the meaning of the release numbers. In particular, there is no relation to package compatibility as this is described in terms of informal rules related to application build process.

In our opinion, the Koala model would benefit from a formalization of its version identification. If the build system supported such formalization, it could decide on the Koala's "golden" and "silver" rules for component compatibility without building the component-based applications.

JavaBeans and EJBs can use Java product versioning [Rig02] and versioning of serializable objects [Sun01c]. Both options provide a link between version identification and package/class compatibility, which is a step forward. However, the lack of any formal definition of terms and algo-

rithms leads again to the situation that Java versioning is at present not suitable for any robust automated reasoning. (A more detailed discussion can be found in [Bra00].)

7.3.2 Research in Component Versioning

Research component frameworks are usually concerned with fundamental component modelling features and omit versioning issues. We are aware of two research works that are directly related to our efforts.

The Ragnarok architectural configuration management model by Christensen [Chr98] relates the version numbers of components to those of the whole configuration (total versioning in [CW98]). This includes the need to reflect architectural changes in the version identification. Our approach uses the separation of component interface and architectural implementation, versioning only the interface part. Christensen's work can be used for versioning of the component implementation, e.g. via the SOFA `architecture` construct.

The configuration management for component-based systems described in [LC99] is concerned mainly with component dependencies and lack of their explicit declarations in Microsoft COM. With respect to component versioning, Larsson proposes a (standard) COM interface `IVersion` that could be used to obtain version information at runtime. While this is a workable solution, we find it insufficient as this information is equally important at design and deployment time to set component interconnections properly. On the other hand, we agree with the author that "configuration management functions should provide information about the changes on the interface level" — Chapter 5 of our thesis provides a method that suits this need.

Lastly, Hoek [vdH01, vdH+02] articulates the need for component versioning related to deployment. In the second paper he describes a prototype distributed version repository that could be used for such purposes. Because his work is concerned mainly with the issues of release management and changes in configuration, it does not treat versioning in much detail.

7.3.3 Syntactical Analysis and Meta-Data in Versioning

Finally we would like to mention works that propose or use syntactical analysis for versioning and deploy meta-data for similar purposes as we do.

The SVCE and the Gandalf system [KH83, HN86] were among the first to use information obtained from syntactical analysis of software source for its versioning (the language-based approach to versioning according to [CW98]) and also provided syntax-based editing environment. The prob-

lem of Gandalf from our point of view is the environment was bound to a concrete programming language. Also, it worked on elements of rather fine granularity as compared to components and did not provide any means of aggregating the results.

Among the industrial systems that use meta-data for component versioning are Enterprise JavaBeans [Sun97, Sun01a] and software package descriptions [Bai97, J+03, Des98]. Especially this latter group deploys meta-data that contains rich information about software dependencies including version descriptions. As noted before, one of their problems from our point of view is potential incorrectness resulting from manual creation.

In component research there are several works that hint or emphasize the need for meta-data. For instance, in [vdHW02] it is used to track component dependencies and support downloading the transitive closure. Unfortunately the description of meta-data in this paper is very vague and (again) assumes manual work.

Last but not least, the WebDAV standard [WW98] by the World Wide Web Consortium uses XML-based data about the managed documents. Together with the URI notation [BLFM98] it provides an elegant way to uniquely identify objects on a global scale while retaining human readability of the identifiers.

7.3.4 Summary

Current component frameworks treat versioning more as a marketing than a technical issue — version numbers are assumed to be created and interpreted by humans (developers, users) rather than tools. In our work we challenge this approach since we believe that components are highly technical artefacts and thus need adequate automated support, including the configuration management issues. We add to the small body of component versioning research a unique approach to creating and interpreting revision numbers. It is based on the rigorous notion of subtyping which results in precise semantics of revision identifiers; at the same time, their structure allows them to be easily incorporated in currently existing versioning schemes.

Chapter 8

Conclusion

When software component technology came on the scene after extensive research in the 1990s, much hype arose about its potential. The hype has gradually died down, the technology has matured (at least in some aspects), but its forecasted widespread use ([Szy98], section 2.3) did not happen. The work presented in this thesis is based on the hypothesis that part of this lack of success is due to inadequate component versioning and compatibility evaluation methods. Its results provide solutions to some of the open issues in these areas.

8.1 Summary of Our Work

Our work is founded on the ENT meta-model of component specification. It captures the common characteristics of component models in a way which makes it easier for humans to analyse component properties. The elements constituting the component specification are classified using a faceted scheme derived from an analysis of important component models and modular programming languages. The two-level hierarchy of the meta-model's aggregate structures — traits and categories — facilitates both human understanding and automated analyses of component interface specification.

The presented specification-based versioning scheme provides component revision identification with several unique features: multiple levels of detail, well-defined relation to component's structure and place of change, and automatic derivation from results of component specification comparison. Its key advantages are the ability to denote compatibility between revisions, to version dependencies and to become part of component naming. At the same time, its representation should fit well into version placeholders present in current component frameworks.

The proposed definitions of component substitutability and compatibility target the main aspect of component substitution — the fact that it takes place in an architecture composed of interrelated components. This is expressed by the notion of a context, utilized by the novel concept of con-

textual compatibility which enables substitution even if strict compatibility (using pure component subtyping) would prevent it. This gives the developers more freedom in making changes to component specification during its evolution.

The explicit linking of component compatibility with versioning, achieved by the use of meta-data with versioning and compatibility information, is another important result of our work. It puts on a firm ground the implicit version semantics used by software package installation systems, and thus provides a way to fully automatic and reliable component upgrades.

On the fundamental level, we believe that the biggest contribution of this work is the fact that the above achievements are defined at the meta-level rather than for a selected component model only. Our component comparison (on which the revision identification and substitutability checking are based) is defined on the ENT structures, i.e. at the meta-model level. It thus makes the revision identification and substitutability assessment applicable to any framework fitting the model and moreover open to new developments in the area of specification languages. This work can therefore be also considered as a contribution to the component standardization efforts.

To provide a path to practical use of these results, we have strived to give the methods the "non-functional" properties listed in the Introduction. Apart from branch and variant identification (which embody human understanding of the software element at hand), the key results are designed so that their implementation can run without user intervention. The main instrument is the comparison of types (present in already existing source code) using subtyping rules, which clearly can be done algorithmically.

We believe that our proposal is well suited to some widely used systems, in particular to the CORBA Component Model. Our confidence in the presented methods is also supported by several proof-of-concept implementations for SOFA and CORBA Component Model that were developed while working on the proposed ideas.

8.2 Lessons Learned

During the development of the ideas and methods covered in this thesis we have learned several lessons. We summarise them in this section; details can be found in the appropriate chapters of this thesis.

Component modelling It is fairly easy to extract model information from IDL or ADL sources, but doing so for component models implemented directly in (standard) programming languages is very difficult. An example is the JavaBeans component model and its event handling, discussed as

a case study in Appendix A.3. At present, only the CORBA Component Model IDL3 specification language and most research ADLs or component programming languages offer good support for modelling efforts.

The lesson of this experience is that component programming and modelling greatly benefits from languages with a direct support of its key abstractions; in other words, through the lack of such languages we learned about their importance.

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. As discussed in Section 2.6, interfaces and properties are commonly available but data elements or behavioural specifications are rare.

Through our research on the ENT meta-model we realised that component languages could (and should) be much richer in their repertoire, which would bring the pleasant consequences of better usability of components and improved reliability of their substitution.

Carrier language impact on substitutability Where specification language is used as a basis of component comparison, care must be taken to deal with the effects of implementation in concrete carrier language(s). As our studies of IDL3 vs. Java (Section 3.3) show, the carrier language and the resulting binary form can have its own (sub)typing rules which may modify the rules of the specification language.

The lesson of our studies is that with separate specification and carrier language, the mappings between the two should be defined with much care and should consider not just translation of language and data structures but also of the associated semantics and (sub)typing rules.

Specification-based versioning gets to the bone Until now, component versioning has been understood as simple technical or marketing tagging. However, our approach reveals that the nature of components — at the same time design abstractions, language constructs and tradeable items — requires to acknowledge that versioning needs to be integrated into the component (meta-)models and related languages. Otherwise we cannot achieve both a high composability of individual components that evolve and a high reliability of the applications that use them.

8.3 Open Issues

While we believe that our work has achieved its goals, a work of this scope can hardly cover all of the open issues. Let us therefore mention at least

few of them.

An issue with the ENT revision identification scheme is that it cannot capture the extent of differences, a feature that is at least naively handled by many "M.m.μ" industrial schemes. We consider this shortcoming an important one, as providing the extent of difference is useful for developers. Such feature would neatly fit into our aims; however, it is not easy to implement it without additional data structures and methods that were not in scope of the presented work.

Our versioning scheme assumes a tight integration into the interface specification language. It thus can be incorporated into IDL-based systems (e.g. CORBA Component Model) by simple modifications of the specification language syntax, and would blend easily into the .NET framework languages which support declarative attributes. However, to apply the proposed revision identification scheme to systems that rely on implementation code only (Enterprise JavaBeans and similar) would require considerable modifications to the implementation language syntax.

In the area of compatibility checking, our methods ensure type safety but cannot (therefore) cope with some simple changes like changing an order of method parameters. They would benefit from the inclusion of the achievements in specification matching and automated interface adaptation to solve this issue. Also, we noted that compatibility in the component world can include compliance with global (architectural) properties. In our approach they can be expressed as part of the context but an explicit extension of contextual compatibility would certainly be a cleaner solution.

The biggest problem in the practical use of our results is the application of the ENT model, and consequently the implementation of our versioning and compatibility methods, in component systems which do not use separate interface specification. At present, this is especially the case of frameworks like Enterprise JavaBeans and ArchJava which use naming conventions, design patterns and Java language extensions to represent component-based design. This requires additional effort (in the case of EJBs a substantial one) to extract from these sources the component interface description needed for our work.

8.4 Future Work

Many ideas have therefore come to our mind about "where next". While it is generally difficult to give any assurance about future research directions of the field as well as of the author[1], some challenges open to interested successors are hinted below.

[1] Worrying about future is vain effort anyway, see the proof by J. Nazarene cited e.g. by Matthew in [Wan90, part 6.34].

Our experience shows how important it is to have a well-designed interface specification language. Most current IDLs concentrate on the functional aspects of interfaces and components, neglecting the data interface and non-functional properties — which sometimes have far greater impact on software usability [BV02]. We would like to see more effort put into research into, and then industrial use of, the next-generation means of component interface specification that increase the role of declarative programming in tackling software complexity.

Because the introduction of component versioning affects component repositories, it would be useful to contribute some work on the design of these repositories. In practice they would certainly benefit from suitably merged meta-data of all contained components to facilitate search and retrieval, and also from more efficient storage of code — e.g. by storing only one copy of the parts of code which do not differ between versions.

Variant description partly overlaps with specifications of quality of service. The research in this area has been going on for several years but we have not noticed any attempt to derive any meta-data or variant description from QoS declarations (or code). Such effort would pay off by contributing to automated derivation of information useful in component trading and composition.

Our research of related systems makes it clear that creating a decent component model is a difficult task, as it needs to encompass many orthogonal aspects in an elegant way. Our work provides versioning *on top of* existing component models — for practical implementations it would be more efficient to build it into a more elaborate component model.

Finally, the importance of visual modeling for large software applications is clear. There however is still a lack of suitable notation targeted at black-box components [MAV02] which slows component adoption. Research in this area, especially such that propagates into standardization efforts, is needed.

Appendix A

ENT Model Definitions for Primary Component Frameworks

A.1 The ENT Model for SOFA Components

The SOFA Component Model was the reference framework used in the development of the ENT model. This section contains the reference definitions of traits in the SOFA component model.

properties

metatype = $property$,
classifier = ($\{feature\}$, $\{data\}$, $\{provided\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{all\}$)

protocol

metatype = $protocol$,
classifier = ($\{quality\}$, $\{operational\}$, $\{provided, required\}$,
$\{type\}$, $\{mandatory\}$, $\{na\}$,
$\{development, assembly, runtime\}$)

provides

metatype = $interface$,
classifier = ($\{feature\}$, $\{operational\}$, $\{provided\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

requires

metatype = $interface$,
classifier = ($\{feature\}$, $\{operational\}$, $\{required\}$,
$\{instance\}$, $\{mandatory\}$, $\{multiple\}$,
$\{development, assembly, runtime\}$)

A.2 The ENT Model for CORBA Components

The CORBA Component Model uses fairly rich component specifications, which provide both method- and event-based communication.

A.2.1 Trait Definitions

attributes
: metatype = *attribute*,
classifier = $(\{feature\}, \{data\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{na\},$
$\{development, assembly, deployment, runtime\})$

emitters
: metatype = *event*,
classifier = $(\{feature\}, \{operational\}, \{required\},$
$\{instance\}, \{mandatory\}, \{single\},$
$\{development, assembly, runtime\})$

facets
: metatype = *interface*,
classifier = $(\{feature\}, \{operational\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

publishers
: metatype = *event*,
classifier = $(\{feature\}, \{operational\}, \{required\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

receptacles
: metatype = *interface*,
classifier = $(\{feature\}, \{operational\}, \{required\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

sinks
: metatype = *event*,
classifier = $(\{feature\}, \{operational\}, \{provided\},$
$\{instance\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

supports
: metatype = *interface*,
classifier = $(\{feature\}, \{operational\}, \{provided\},$

$\{type\}, \{mandatory\}, \{multiple\},$
$\{development, assembly, runtime\})$

A.2.2 Example: The `Parking` Component Source

The source (from OpenCCM [MMV01] examples):

```
// the parking.
component Parking
{
   // parking states.
   readonly attribute string description;
   readonly attribute ParkingState state;
   readonly attribute PlaceNumber capacity;
   readonly attribute PlaceNumber free;
   // parking facets.
   provides ParkingAccess for_barriers;
   provides ModifyState for_admin;
   // parking events ports.
   publishes ChangeState state_notify;
};
```

A.2.3 Example: The `Parking` Component in ENT

The representation of the `Parking` component in traits is as follows, ommitting empty traits and element classifiers.

---

attributes $= \{(description, string, \emptyset, \{readonly\}$,*atribute*$, (\dots)),$
$(state, ParkingState, \emptyset, \{readonly\}$,*attribute*$, (\dots)),$
$(capacity, PlaceNumber, \emptyset, \{readonly\}$,*attribute*$, (\dots)),$
$(free, PlaceNumber, \emptyset, \{readonly\}$,*attribute*$, (\dots))\}$

facets $= \{(for\_barriers, ParkingAccess, \emptyset, \emptyset$,*interface*$, (\dots)),$
$(for\_admin, ModifyState, \emptyset, \emptyset$,*interface*$, (\dots))\}$

publishers $= \{(state\_notify, ChangeState, \emptyset, \emptyset$,*event*$, (\dots))\}$

A.3 The ENT Model for JavaBeans

It is not easy to create the ENT meta-model (i.e. mainly the trait definitions) for JavaBean [Sun97] components; the difficulty of developing formal models of JavaBeans and Enterprise JavaBeans has been also noted by Sousa [SG00]. The primary reason is the way the component model is

defined — that is, using excessively close links to the Java language type system and to a set of name conventions[1].

First, to find JavaBean's properties, syntactical analysis of method signatures must be complemented by an appropriate lexical analysis of their names. This is because properties are implemented as pairs of accessor and mutator methods that are named according to a convention. For example, to define a property `int property`, a JavaBean must contain methods `int getProperty()` for reading plus the dual `void setProperty(int value)` for modifying the property.

Next, the JavaBean model contains an event-handling mechanism using the publish-subscribe design pattern. However, this is realized by JavaBean classes implementing listener interfaces which group event declarations. If we were to define a trait for the events a component can react to, we would have to refer to the contents of such interfaces which is obscure at the conceptual level and difficult for the implementation.

The most we could do is to rely on another name convention as listener interface names should end in `Listener` – but this method is highly unreliable because the name convention is not mandatory. We could also try to detect event-handling methods in component interface – but the "design pattern" by which they are described in the specification (`void <eventOccurenceMethodName> (<EventStateObjectType> evt);` in the Section 6.4 of [Sun97]) is clearly so general that it is useless for any automated analysis.

Finally, the lifecycle applicability of bean's elements (design-time or runtime) is defined by testing the `isDesignMode()` dynamic property (from `java.beans.DesignMode` interface) inside method bodies. Unless a sophisticated analysis of method body code is used, it is impossible to correctly set the *lifecycle* classification property of each method.

We thus conclude that the JavaBeans framework belongs to systems that do not lend themselves easily to formal, in our case ENT-based modelling. We also consider this a supporting case for our call to the development and use of specification languages with clearer separation from component implementation.

A.3.1 Trait Definitions

The JavaBean framework therefore does not fit cleanly into the ENT classification system. The following trait definitions are a compromise result of what can be resonably obtained from a beans' source code.

Creating an ENT model of JavaBeans that corresponds fully to the ideas described in the frameworks' specifications would be a complicated issue in terms of its implementation (i.e. it would be difficult to develop a

[1] The specification refers to these as "design patterns" which is obviuosly a misnomer.

suitable parser to extract such ENT model representation).

cimport
: metatype = *class*,
classifier = $(\{feature\}, \{operational, data\}, \{required\}, \{type\}, \{mandatory\}, \{single\}, \{development, deployment\})$

implements
: metatype = *interface*,
classifier = $(\{feature\}, \{operational\}, \{provided\}, \{type\}, \{mandatory\}, \{multiple\}, \{all\})$

methods
: metatype = *method*,
classifier = $(\{feature\}, \{operational\}, \{provided\}, \{instance\}, \{mandatory\}, \{multiple\}, \{runtime\})$

pimport
: metatype = *package*,
classifier = $(\{feature\}, \{operational, data\}, \{required\}, \{type\}, \{mandatory\}, \{single\}, \{development, deployment\})$

properties
: metatype = *property*,
classifier = $(\{feature\}, \{data\}, \{provided\}, \{instance\}, \{mandatory\}, \{multiple\}, \{assembly, runtime\})$

A.3.2 Example: The `MyJuggler` JavaBean Source

The following code is an shortened and modified version of the standard "example" bean `sunw.demo.Juggler`. The method bodies are removed due to the fact they are not primarily interesting for interface specification. Several specification elements are highlighted in the code.

```
package cz.cuni.mff.ent.demo;

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.net.URL;
```

```
import java.beans.*;
import java.beans.DesignMode.*;

public class MyJuggler
 extends
   Applet
 implements
   PropertyChangeListener, DesignMode
{
  /** design time methods */
  public void setDebug( boolean debug) { /* ... */ }

  /** property: int animationRate */
  public int getAnimationRate() { /* ... */ }
  public void setAnimationRate(int x) { /* ... */ }

  /** Juggler methods */
  public synchronized void startJuggling() { /* ... */ }
  public synchronized void stopJuggling() { /* ... */ }
  public void startJuggling(ActionEvent x) { /* ... */ }
  public void stopJuggling(ActionEvent x) { /* ... */ }
  public boolean isJuggling() { /* ... */ }

  /** administrative methods */
  public void setDesignTime(boolean dmode) { /* ... */ }
  public boolean isDesignTime() { /* ... */ }
  public boolean isDebug() { /* ... */ }
}
```

A.3.3 Example: The `MyJuggler` JavaBean in ENT

The representation of the `MyJuggler` JavaBean in traits is as follows, omitting the metatypes and classifiers.

---

pimport $= \{(nil, java.awt, \emptyset, \emptyset, \ldots),$
$(nil, java.awt.event, \emptyset, \emptyset, \ldots),$
$(nil, java.awt.iamge, \emptyset, \emptyset, \ldots),$
$(nil, java.beans, \emptyset, \emptyset, \ldots),$
$(nil, java.beans.DesignMode, \emptyset, \emptyset, \ldots)\}$

cimport $= \{(nil, java.net.URL, \emptyset, \emptyset, \ldots)\}$

extends $= \{(nil, Applet, \emptyset, \emptyset, \ldots)\}$

implements $= \{(nil, PropertyChangeListener, \emptyset, \emptyset, \ldots),$
$(nil, DesignMode, \emptyset, \emptyset, \ldots)\}$

methods $= \{(setDebug, boolean \rightarrow void, \emptyset, \emptyset, \ldots),$
$(startJuggling, void \rightarrow void, \emptyset, \{synchronized\}, \ldots),$
$(stopJuggling, void \rightarrow void, \emptyset, \{synchronized\}, \ldots),$
$(startJuggling, ActionEvent \rightarrow void, \emptyset, \emptyset, \ldots),$
$(stopJuggling, ActionEvent \rightarrow void, \emptyset, \emptyset, \ldots),$
$(isJuggling, void \rightarrow boolean, \emptyset, \emptyset, \ldots),$
$(setDesignTime, boolean \rightarrow void, \emptyset, \emptyset, \ldots),$
$(isDesignTime, void \rightarrow boolean, \emptyset, \emptyset, \ldots),$
$(isDebug, void \rightarrow boolean, \emptyset, \emptyset, \ldots)\}$

properties $= \{(animationRate, int, \emptyset, \emptyset, \ldots)\}$

Appendix B

SOFA CDL Subtyping Rules

B.1 The Rules for the SOFA CDL

To support the definitions of component comparison (Chapter 3), we describe here the subtyping rules for SOFA CDL.

We use the fact that the subtype relation is reflective and transitive [Car97]. For base types we define rules for type coercion, i.e. conversions between simple types. From template types onwards, full subtyping is defined. The abstract syntax shown for each metatype aims to present the structure of the data, rather than to exactly correspond to CDL grammar rules.

A note on the presentation style: we do not use the standard "bar" notation for the typing rules (premises above the bar, consequences below). Rather, we present the rules as standard boolean expressions, using logical connectives and implication. The reason is the complexity of some rules for structured types, combined with a desire to use the names found in the CDL syntactical structures to enhance the readability of the rules.

A. Simple Types

base types

Abstract syntax:
base ::= *short* | *unsigned short* | *long* | *unsigned long* | *long long* |
unsigned long long | *float* | *double* | *long double* | *fixed* |
char | *wchar* | *octet* | *boolean* | *any*

Subtyping rules:
unsigned short <: *long*
unsigned long <: *long long*
float <: *double* <: *long double*
long double <: *fixed*
char <: *wchar*

Notes:

- Noteworthy facts on non-subtype relations: *boolean*, *octet* and *any* — no coercion, *short* $\neq$ *unsigned short*, *long* $\neq$ *unsigned long*, *long long* $\neq$ *unsigned long long*.
- Type coercion between *fixed* and *double* is based transitively on the long double coercion, which in turn is based on the Java mapping used in the CDL compiler implementation — for `java.math.BigDecimal`, there exists a constructor with a *double* floating point parameter, and a `double doubleValue()` method which can return the constants `DOUBLE.*_INFINITY` if the fixed number is too big (see the JDK 1.3 API documentation in [Sun01b]). The relation of these types is otherwise not clearly defined in SOFA CDL or CORBA IDL specifications.
- The relation between signed-unsigned type pairs is schizofrenic when it comes to actual carrier language binding: they should be type incompatible (the value sets are not in the subset relation) but all Java mappings use the *short* type for both, so in practice (for Java implementations) they are type identical. The rules we define here follow from the CORBA IDL definitions of these types (see [OMG02c]).

sequence

Abstract syntax:

s_1, s_2 ::= *sequence*
sequence ::= ***sequence***< *type*,*length* >
length ::= *number* | ε

Subtyping rule:

$s_2.type <: s_1.type \quad \wedge \quad s_2.length \geq s_1.length$
$\Rightarrow$
$s_2 <: s_1$

string, wstring

Abstract syntax:

s_1, s_2 ::= *string*
string ::= ***string***< *length* > | ***wstring***< *length* >
length ::= *number* | ε

Subtyping rules:

$length_2 \geq length_1 \quad \vee \quad length_2 = \varepsilon$
$\Rightarrow$
$s_2 <: s_1$

B. Constructed Types

enum

Abstract syntax:

e_1, e_2 ::= *enum*
enum ::= ***enum*** *name* '{' *contents* '}'
contents ::= *identifier* (',' *identifier*)*

Subtyping rule:

$e_2.contents \supset e_1.contents$
$\Rightarrow$
$e_2 \ <: \ e_1$

union

Abstract syntax:

u_1, u_2 ::= *union*
union ::= ***union*** *name* ***switch*** *('type')* '{' *contents* '}'
contents ::= (***case*** *const: type name;*)* ***[default****: type name ';']*

Subtyping rule:

$|u_2.contents| \geq |u_1.contents|$
$\wedge \ \forall e_1 \in u_1.contents \ \exists e_2 \in u_2.contents :$
$e_2.const = e_1.const \ \wedge \ e_2.name = e_1.name \ \wedge \ e_2.type <: e_1.type$
$\Rightarrow$
$u_2 \ <: \ u_1$

struct, exception

Abstract syntax:

s_1, s_2 ::= *struct*
struct ::= ***struct*** *name* '{' *contents* '}'
e_1, e_2 ::= *exception*
exception ::= ***exception*** *name* '{' *contents* '}'
contents ::= (*type name* ';')*

Subtyping rules:

$|s_2.contents| \geq |s_1.contents|$
$\wedge \ \forall c_1 \in s_1.contents \ \exists c_2 \in s_2.contents :$
$c_1.name = c_2.name \ \wedge \ c_2.type <: c_1.type$
$\Rightarrow$
$s_2 \ <: \ s_1$

$|e_2.contents| \geq |e_1.contents|$
$\wedge \ \forall c_1 \in e_1.contents \ \exists c_2 \in e_2.contents :$
$c_1.name = c_2.name \ \wedge \ c_2.type <: c_1.type$
$\Rightarrow$
$e_2 \ <: \ e_1$

C. Interface Types

function

Abstract syntax:

f_1, f_2 *::=* *function*
function ::= *invoc rettype fname '(' params ')' mtraises ';'*
invoc ::= ε | ***oneway***
*params ::= ([dir] type pname)**
mtraises ::= ε | ***mtraises*** '{' *(mtexc)+* '}'
dir ::= ε | ***in*** | ***out*** | ***inout***

Subtyping rule:

$$
\begin{aligned}
& f_2.invoc = f_1.invoc \\
\wedge\ & f_2.rettype <: f_1.rettype \\
\wedge\ & \forall exc_{1,i} \in f_1.raises\ \exists exc_{2,j} \in f_2.raises : \\
& \quad exc_{2,j} <: exc_{1,i} \\
\wedge\ & |f_2.params| = |f_1.params| \\
\wedge\ & \forall p_{1,i} \in f_1.params\ \exists p_{2,i} \in f_2.params : \\
& \quad p_{2,i}.dir \supseteq p_{1,i}.dir \\
& \quad \wedge\ \Big(p_{2,i}.dir \in \{\varepsilon, in\} \Rightarrow p_{2,i}.type :> p_{1,i}.type \\
& \qquad \vee\ p_{2,i}.dir = out\ \Rightarrow\ p_{2,i}.type <: p_{1,i}.type \\
& \qquad \vee\ p_{2,i}.dir = inout\ \Rightarrow\ p_{2,i}.type = p_{1,i}.type\Big) \\
\Rightarrow & \\
f_2\ & <:\ f_1
\end{aligned}
$$

attribute

Abstract syntax:

a_1, a_2 *::=* *attr*
attr ::= *ro* ***attribute*** *type (name)+ ';'*
ro ::= ε | ***readonly***

Subtyping rule:

$$
\begin{aligned}
& a_2.ro \subseteq a_1.ro\ \wedge\ (a_2.ro = readonly\ \wedge\ a_2.type <: a_1.type) \\
\Rightarrow & \\
& a_2 <: a_1
\end{aligned}
$$

constant

Abstract syntax:

c_1, c_2 *::=* *const*
const ::= ***const*** *type name '=' value ';'*

Subtyping rule:

$$
\begin{aligned}
& c_2.type <: c_1.type \\
\Rightarrow & \\
& c_2 <: c_1
\end{aligned}
$$

interface

Abstract syntax:

i_1, i_2 ::= interface
interface ::= ***interface*** *name inherits '{' types elems '};*
inherits ::= ε | ':' (inh)+
*elems ::= (const | attr | function)**
*types ::= (typedef | struct | union | enum)**
typedef ::= ***typedef*** *typespec name ';'*
typespec ::= base | sequence | string | identifier | enum | union | struct

where *inh* is an identifier that denotes an *interface* type.

Subtyping rule:

$$
\begin{aligned}
&\forall inh_{1,i} \in i_1.inherits\ \exists inh_{2,j} \in i_2.inherits : \\
&\quad inh_{2,j} <: inh_{1,i} \\
\wedge\ &\forall e_{1,j} \in i_1.elems\ \exists e_{2,j} \in i_2.elems : \\
&\quad e_{1,i}.name = e_{2,j}.name\ \wedge\ e_{2,j}.type <: e_{1,i}.type \\
\wedge\ &\forall t_{1,j} \in i_1.types\ \exists t_{2,j} \in i_2.types : \\
&\quad t_{2,j} <: t_{1,i} \\
\Rightarrow\ & \\
&i_2 <: i_1
\end{aligned}
$$

Notes:

- For *attribute*s, subtyping is allowed only if the attribute is read-only: the fact that read-write attributes exhibit both covariant and contravariant behaviour necessitates type equality.
- For *const*ants, correct typing of the value is handled by the appropriate typing rule — subtyping cannot deal with changes of the value in any way.
- For *function*s, the `context` term is not used in the CDL compilers and its role is not defined; it is therefore left out from the type rules.
- In the standard scenarios of our use, only functions with the same name can be compared for subtyping (the function name is its "unique identifier" in function call so comparing differently named functions does not make sense). While this could be dealt with in the rule for *function* by stipulating that $f_1.name = f_2.name$, it is left to a higher level (namely interface subtype rule) in order not to create an exception in the general pattern of subtyping rules.

D. Component Types

frame

Abstract syntax:

f_1, f_2 ::= *frame*

frame ::= ***frame*** *name* '{' *provisions requirements properties protocol* '}'

provisions ::= ***provides***: (*type name*)* ';'

requirements ::= ***requires***: (*type name*)* ';'

properties ::= (*[ro]* ***property*** *type name* ';')*

ro ::= ε | ***readonly***

protocol ::= ***protocol***: *prot*

where *prot* is behaviour protocol expression as per [PV02].

Subtyping rule:

$$
\begin{aligned}
&\forall p_{1,i} \in f_1.provisions\ \exists p_{2,j} \in f_2.provisions : \\
&\quad p_{2,j}.name = p_{1,i}.name\ \wedge\ p_{2,j}.type <: p_{1,i}.type \\
\wedge\ &\forall p_{1,i} \in f_1.properties\ \exists p_{2,j} \in f_2.properties : \\
&\quad p_{2,j}.name = p_{1,i}.name \\
&\quad \wedge\, p_{2,j}.ro \subseteq p_{1,i}.ro \\
&\quad \wedge\, p_{2.j}.ro = readonly \\
&\quad \wedge \begin{cases} p_{2,j}.type <: p_{1,i}.type & \text{if } p_{1.j}.ro = readonly \\ p_{2,j}.type = p_{1,i}.type & \text{if } p_{1.j}.ro = \varepsilon \end{cases} \\
\wedge\ &\forall r_{1,i} \in f_1.requirements\ \exists r_{2,j} \in f_2.requirements : \\
&\quad r_{2,j}.name = r_{1,i}.name\ \wedge\ r_{2,j}.type :> r_{1,i}.type \\
\wedge\ &f_2.prot\ is\ compliant\ with\ f_1.protonNames(f_2) \\
\Rightarrow\ & \\
&f_2 <: f_1
\end{aligned}
$$

Notes:

- For *property*, subtyping is allowed only if the attribute is read-only: read-write properties exhibit both covariant and contravariant behaviour which necessitates type equality.
- Protocol comparison is based on the protocol compliance relation defined in [PV02, Section 3.4.2].
- Architecture (the `architecture` construct) has no typing rules for a clear reason — it is not a type definition and therefore is not subject to comparison.

Appendix C

The Specifications of SOFA Component Meta-Data

C.1 CDL Meta-Data Section Grammar

```
<type_id>  ::= <identifier> <metainfo>
<metainfo> ::= '[' <metaelem> [ <metaelem>* ] ']'
<metaelem> ::= <metaid> '=' <metaval> ';'
<metaid>   ::= '@'<identifier> | <identifier>
<metaval>  ::=  '"' <arbitrary-string> '"' |
                <string-without-whitespace>
<identifier>  ::=  /* standard identifier */
```

The following meta-information elements (the `<metaid>`s) are known:

@rev Revision identification of the type. Generated. String, the format is "`N`" or "`N.N.N`" where N is a natural non-zero number. For the `frame_dcl`, the component versioning (with three numbers, for instance `rev=3.2.1`) is used; for all other constructs, the primitive versioning (with one number, i.e. `rev=1`) is the only possibility.

@diff Difference indication of the revision. Generated. String, the format is "`(D.D.D)`" where D from the set {`none`,`spec`,`gen`,`mut`} denotes the difference classifier ("none", "specialisation", "generalisation", "mutation" respectively) of each ENT category . Used only for components, i.e. in the `frame_dcl` meta-data section.

@time Timestamp of the revision. Generated. String, the format is an ISO-compliant, human readable datetime string; the default format is `YYYY-MM-DD hh:mm:ss.ddd GMT[+|-]hh:mm`.

branch Name of branch in the version graph of the type. Manually inserted. String without whitespace, default is "trunk".

tag Human readable moniker of the given revision. Manually inserted. String without whitespace, no default.

description|desc Human readable description. Manually inserted. An arbitrary string, default is empty.

C.2 Grammar of the URI form of SOFA identifiers

```
<scoped_name> ::= <sofauri>
<sofauri>  ::= [<scheme>":"]<path><identifier>["#"<version>]
<scheme>   ::= "sofa"
  // identifies SOFA URI-style naming
<path>  ::= ["//"<provider>]["/"]{<module>"/"}*
  // '/' follows URI syntax rules, i.e. separates
  // hierarchical levels (SOFA modules);
  // module name ".." means the containing module

<version>  ::= ["branch="<branch>"&"]["rev="<revision>
                  ["&var="<variant>]]
  // '&' follows URL conventions,
  // separating one attribute from another

<revision> ::= <number>"."<number>"."<number>
  // uses component revision identifiers, e.g. "3.3.4"

<variant>  ::= <term>{<op><term>}*
<term>      ::= <key>":"<val> | "("<variant>")"
<op>       ::= "," | ";"
  // uses description logic terms, <op>erators are
  //  "," for AND and ";" for OR
  // <key> is variant property name, <val> is its value,
  // using "/" to separate levels in hierarchical attributes.

<provider>   ::= <reversed-DNS-name>
<module>   ::= <identifier>
<branch>   ::= <identifier>
<key>      ::= <identifier>
<val>      ::= <string-without-whitespace>
```

C.3 XML Meta-Data Document Type Definition

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!--  ===== SOFA component metadata DTD ===== -->
```

```
<!-- Pseudo-type declarations;
     e.* for elements, a.* for attributes -->

<!-- element content type that does not allow whitespace -->
<!ENTITY % e.identifier "#PCDATA">
<!-- parameter content that does not allow whitespace -->
<!ENTITY % a.identifier "NMTOKEN"> <!-- should be ID -->
<!-- numberic element content -->
<!ENTITY % e.number "#PCDATA">
<!-- numeric parameter content -->
<!ENTITY % a.number "NMTOKEN">
<!-- revision id string (num.num.num...) parameter content -->
<!ENTITY % e.revid "#PCDATA">
<!-- revision data/id level -->
<!ENTITY % a.revlevel "primitive|component|detailed">
<!-- difference classifiers: should be init|none|spec|gen|mut -->
<!ENTITY % e.diff "#PCDATA">

<!-- Top-level elements, component info -->

<!ELEMENT compdata
  (provider,namespace,name,about?,version,resources) >
  <!-- which component framework is used -->
  <!ATTLIST compdata
    system %a.identifier; #REQUIRED
    >
<!-- reversed DNS of provider -->
<!ELEMENT provider (#PCDATA)>
<!-- hierarchy of modules delimited by / -->
<!ELEMENT namespace (#PCDATA)>
<!-- name of the component -->
<!ELEMENT name (%e.identifier;)>

<!-- administrative info -->
<!ELEMENT about (moniker?, date, description?)>
<!-- alternative marketing name -->
<!ELEMENT moniker (%e.identifier;)>
<!-- release date in mm.dd.yyyy format -->
<!ELEMENT date (#PCDATA)>
<!-- human-readable description -->
<!ELEMENT description (#PCDATA)>

<!-- Version information -->

<!-- version data -->
<!ELEMENT version (tag?,branch?,revision,history?,variant)>
<!-- marketing tag of the version -->
<!ELEMENT tag (%e.identifier;) >
```

```
<!-- name of development branch -->
<!ELEMENT branch (%e.identifier;) >
<!-- revision ID in string format -->
<!ELEMENT revid (%e.revid;) >
<!ATTLIST revid
  level (%a.revlevel;) "component" >

<!-- Revision data -->

<!-- revision data -->
<!-- seq: sequence number of revision, used in history -->
<!ELEMENT revision (tag?,date,parent,data+) >
<!ATTLIST revision
  seq %a.number; #IMPLIED >
<!-- immediate predecessor in history;
     if empty, means no predecessor -->
<!ELEMENT parent ((provider,namespace,name)?,branch?,revid?) >
<!-- individual elements of revision data -->
<!-- level: of rev data -->
<!ELEMENT data (part)* >
<!ATTLIST data
  level (%a.revlevel;) "component" >
<!-- one revision data element -->
<!-- name: of the rev data part,
     value denotes trait or category -->
<!ELEMENT part (revnum,diff) >
<!ATTLIST part
  name %a.identifier; #REQUIRED >
<!-- revision number of the part -->
<!ELEMENT revnum (%e.number;) >
<!-- difference classifier -->
<!ELEMENT diff (%e.diff;) >

<!-- Revision history -->

<!ELEMENT history (branch,data)* >
  <!-- list of previous revisions; only component level data
       are used, sequence (a path from root in version graph)
       is assumed -->

<!-- Variant data -->

<!ELEMENT variant (term|op)* >
<!ATTLIST variant
  exprtype %a.identifier; #IMPLIED >
  <!-- variant description -->
  <!-- exprtype: how to interpret variant expr,
       possible values "and","boolean","feature";
       default "and" -->
```

```
<!ELEMENT op (#PCDATA) >
  <!-- operator joining expressions,
       values for exprtype="boolean" are
       "and","or","(",")" -->
<!ELEMENT term ((key|dim),value) >
  <!-- a term in variant expression -->
<!ELEMENT key (%e.identifier;) >
<!ELEMENT dim (%e.identifier;) >
  <!-- first part of variant expr term;
       key denotes ordinal attributes,
       dim denotes hierarchical attributes -->
<!ELEMENT value (#PCDATA) >
  <!-- second part of the key,value pair -->

<!-- Resource descriptions -->

<!-- resources used by the component during lifecycle -->
<!ELEMENT resources (item)*  >
<!ELEMENT item (key,value) >
<!ATTLIST item
  type %a.identifier; #IMPLIED
  mime CDATA #IMPLIED >
  <!-- parts of resource description -->
  <!-- type: how to interpret the value part,
       valid values "string","int","boolean","float","file","class",
       default "string" -->
  <!-- mime: MIME type of referenced content for type="file",
       default "application/octet-stream" -->
```

C.4 An Example of Complete Meta-Data

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE compdata SYSTEM "compdata.dtd">
<compdata system="sofa">

 <provider>cz.zcu.kiv</provider>
 <namespace></namespace>
 <name>FAddressBook</name>

 <about>
   <moniker>Address Book</moniker>
   <date>13.4.2001</date>
 </about>

 <version>
   <branch>FixDate</branch>

   <revision>
```

```
    <tag>rev3</tag> <date>13.4.2001</date>
    <parent> <revid level="component">3.1.1</revid> </parent>
    <data level="component">
     <part name="E"> <revnum>4</revnum> <diff>spec</diff></part>
     <part name="N"> <revnum>2</revnum> <diff>spec</diff></part>
     <part name="T"> <revnum>1</revnum> <diff>none</diff></part>
    </data>
    <data level="detailed">
     <part name="provisions">
      <revnum>4</revnum>
      <diff>spec</diff></part>
     <part name="dependencies">
      <revnum>2</revnum>
      <diff>spec</diff></part>
     <part name="properties">
      <revnum>2</revnum>
      <diff>none</diff></part>
     <part name="protocol">
      <revnum>1</revnum>
      <diff>none</diff></part>
    </data>
  </revision>

  <history>
    <branch>trunk</branch>
    <data level="component">
      <part name="E"> <revnum>1</revnum> <diff>init</diff></part>
      <part name="N"> <revnum>1</revnum> <diff>init</diff></part>
      <part name="T"> <revnum>1</revnum> <diff>init</diff></part>
    </data>
    <branch>trunk</branch>
    <data level="component">
      <part name="E"> <revnum>2</revnum> <diff>spec</diff></part>
      <part name="N"> <revnum>1</revnum> <diff>none</diff></part>
      <part name="T"> <revnum>1</revnum> <diff>none</diff></part>
    </data>
    <branch>FixDate</branch>
    <data level="component">
      <part name="E"> <revnum>3</revnum> <diff>mut</diff></part>
      <part name="N"> <revnum>1</revnum> <diff>none</diff></part>
      <part name="T"> <revnum>1</revnum> <diff>none</diff></part>
    </data>
  </history>

  <variant exprtype="and">
   <term><key>OS</key><value>JDK1.3</value></term>
  </variant>

</version>
```

```
 <resources>
  <item type="string"> <!-- class vraci soubor-->
   <key>ManagerClassName</key>
    <value>sofa.test.calcdemo.CalcDemo1CM</value>
  </item>
  <item type="file">
   <key>DeploymentDescriptor</key>
    <value>foo.cdl</value>
  </item>
 </resources>

</compdata>
```

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.

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. Technical Report UW-CSE-02-04-01, University of Washington, April 2002.

[ACN02b] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, May 2002. ACM Press.

[Ada95] International Organization for Standardization. *Ada 95 Reference Manual: Language and Standard Libraries*, 1995. International Standard ISO/IEC 8652:1995(E), Version 6.0.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, 1998.

[aJP00] María José Presso. Declarative descriptions of component models as a generic support for software composition. In *Workshop on Component-Oriented Programming (WCOP'00)*, Nice, France, June 2000. Position Paper.

[Bab86] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.

[Bai97] Ed Bailey. *Maximum RPM*. Sams, 1997.

[BCS02] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with

sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP 2002)*, 2002. Available at http://research.microsoft.com/~cszypers/events/WCOP2002/.

[Ber90] Brian Berliner. CVS II: parallelizing software development. In *Proceedings of the USENIX 1990 conference*, 1990.

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

[Bör95] Juergen Börstler. Feature-oriented classifciation for software reuse. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pages 204–211, Rockville, MD, USA, June 1995.

[BP00] Dušan Bálek and František Plášil. Software connectors: a hierarchical model. Technical report, Charles University, Faculty of Mathematics and Physics, 2000.

[BP01] Dušan Bálek and František Plášil. Software connectors and their role in component deployment. In *Proceedings of DAIS'01*, Krakow, Poland, September 2001. Kluwer.

[BR00a] 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.

[BR00b] David J. Brown and Karl Runge. Library interface versioning in Solaris and Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, 2000. USENIX.

[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, Department 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.

[Bro95] Frederick Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, 20th anniversary edition, 1975, 1995.

[BV02] Manuel F. Bertoa and Antonio Vallecillo. Quality attributes for COTS components. *I+D Computación*, 1(2):128–144, 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.

[CCF00] Denis Conan, Michel Coriat, and Nicolas Farcet. A software component development meta-model for product lines. In *Workshop on Component-Oriented Programming (WCOP'00)*, Nice, France, 2000. Position Paper.

[Chr98] H. B. Christensen. Experiences with architectural software configuration management in Ragnarok. In *Proceedings of SCM-8 Workshop, ECOOP 1998*. Springer-Verlag, 1998.

[Cor02] Microsoft Corp. *Inside the .NET Framework*, MSDN library edition, 2002. http://msdn.microsoft.com/library/en-us/cpguide/html/.

[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.

[E+02] Jacky Estublier (Editor) et al. Impact of the research community on the field of software configuration management: Summary of an impact project report. *Software Engineering Notes*, 27(5), September 2002.

[ECM02] ECMA International. *C# Language Specification*, 2002. ECMA Standard 334, 2nd edition.

[Fie95] Roy Fielding. Relative uniform resource locators. RFC 1808, IETF, 1995.

[FK98] Svend Frolund and Jari Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[Fra03] The Fractal Project team. *Fractal ADL Tutorial*, 2003. Available at http://fractal.objectweb.org/current/doc/tutorials/adl/.

[FW00] Clemens Fischer and Heike Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In *Proceedings of AMAST 2000: International Conference on Algebraic Methodology And Software Technology*, volume 1816 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[GJS96] J. Gosling, B. Joy, and G. Steele. *Java Language Specification*. Sun Microsystems, Inc., 1996. Chapter 13 (Java Binary Compatibility).

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: an architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[H+97] R.S. Hall et al. An architecture for post-development configuration management in a wide-area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, USA, May 1997. IEEE Computer Society Press.

[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.

[Han99a] Jun Han. An approach to software component specification. In *Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE'99*, May 1999.

[Han99b] Jun Han. Semantic and usage packaging for software components. In *Object Interoperability: ECOOP'99 Workshop on Object Interoperability*, pages 25–34, Lisbon, Portugal, June 1999.

[Hes03] Joey Hess. *Comparing Linux/UNIX Binary Package Formats*, July 2003. Available at http://www.kitenet.net/ joey/pkg-comp/.

[HL99] David Hemer and Peter Lindsay. Specification-based retrieval strategies for module reuse. Technical Report 99-11, University of Queensland, Queensland, Australia, July 1999.

[HN86] Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12), December 1986.

[IEE98] IEEE Computer Society. *IEEE Standard for Software Configuration Management Plans (828-1998)*, 1998.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

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

[KH83] Gaile E. Kaiser and Nico Habermann. An environment for system version control. In *Proceedings of 26th IEEE Computer Society Conference*, San Francisco, CA, 1983. IEEE Computer Society Press.

[Kop87] Herrmann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1987.

[Lav02] Ronan Mac Laverty. Robocop: Robust open component based software architecture for configurable devices project. Initial specification, ITEA PROJECT 00001 Deliverable, 2002.

[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.

[LR01a] Chris Lüer and David S. Rosenblum. UML Component Diagrams and Software Architecture – Experiences from the Wren Project. In *1st ICSE Workshop on Describing Software Architecture with UML*, pages 79–82, Toronto, Canada, 2001.

[LR01b] Chris Lüer and David S. Rosenblum. WREN—An Environment for Component-Based Development. In Volker Gruhn, editor, *Proceedings of ESEC/FSE 2001*, pages 207–217, Vienna, Austria, 2001. ACM Press.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[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.

[M+95] J. Magee et al. Specifying distributed software architectures. In *Proceedings of ESEC'95*, Barcelona, Spain, 1995.

[M+01] John Morris et al. Software component certification. *IEEE Computer*, September 2001.

[MAV02] Raúl Monge, Carina Alves, and Antonio Vallecillo. A graphical representation of COTS-based software architectures. In *Proceedings of IDEAS 2002*, pages 126–137, La Habana, Cuba, April 2002.

[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.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mic95] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, October 1995.

[Mic03] Microsoft corp. *COM SDK Documentation*, February 2003. MSDN Library, Component Development.

[MMV01] R. Marvie, P. Merle, and M. Vadet. The OpenCCM platform. http://corbaweb.lifl.fr/OpenCCM/index.html, 2001.

[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.

[OGJ02] J. Oberleitner, T. Gschwind, and M. Jazayeri. Vienna component framework: enabling composition across component models. Technical report TUV-1841-2002-48, Technical University of Vienna, September 2002.

[OMG96] Object Management Group. *ORB Interface Type Versioning Management*, January 1996. OMG Request for Proposals orb/96-01-06.

[OMG01] Object Management Group. *Online Upgrades*, September 2001. OMG Request for Proposals orbos/2001-09-10.

[OMG02a] Eternal Systems and others. *Online Upgrades*, May 2002. OMG Revised Joint Proposed Specification mars/2002-05-01.

[OMG02b] Object Management Group. *Issue 2227: Versioning needed for CORBA Core*, 1998-2002. Available at http://www.omg.org/-issues/components-ftf.html.

[OMG02c] Object Management Group. *Common Object Request Broker Architecture (CORBA): IDL Syntax and Semantics*, July 2002. OMG Specification formal/02-06-39.

[OMG02d] Object Management Group. *Common Object Request Broker Architecture (CORBA), The Interface Repository*, July 2002. OMG Specification formal/02-06-46.

[OMG02e] Object Management Group. *The Common Object Request Broker: Core Specification (Version 3.0)*, December 2002. OMG Specification formal/02-12-06.

[OMG02f] Object Management Group. *CORBA Components*, June 2002. Version 3.0. OMG Specification formal/02-06-65.

[OMG02g] Object Management Group. *Meta Object Facility (MOF) Specification*, 2002. Version 1.4. OMG Specification formal/02-04-03.

[OMG02h] 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.

[Par72] David L Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, December 1972.

[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.

[PDF87] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 18(1), January 1987.

[PDH99] Allen Parish, Brandon Dixon, and David Hale. Component based software engineering: A broad based model is needed. Technical report, The University of Alabama, Tuscaloosa, AL, USA, April 1999.

[PDN86] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), November 1986.

[Per87] Dewayne Perry. Version control in the Inscape environment. In *Proceedings of ICSE'87*, Monterey, CA, 1987.

[Pet95a] M. Peterson. *DCE: A Guide to Developing Portable Applications*, chapter 17: UUID and Version attributes. McGraw-Hill, 1995.

[Pet95b] M. Peterson. *DCE: A Guide to Developing Portable Applications*. McGraw-Hill, 1995.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.

[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.

[Rog97] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

[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/.

[SC00] J. C. Seco and L. Caires. A basic model of typed components. In E. Bertino, editor, *Proceedings of ECOOP*, number 1850 in Lecture Notes in Computer Science, pages 108–128. Springer Verlag, 2000.

[SEI97] Software Engineering Institute, Carnegie Mellon University. *Module Interconnection Languages (Software Technology Review)*, 1997. Available at http://www.sei.cmu.edu/str/descriptions/mil_body.html.

[SG00] João Pedro Sousa and David Garlan. Formal modeling of the enterprise javabeans component integration framework. Technical Report CMU-CS-00-162, Carnegie Mellon university, September 2000.

[Sof01] Borland Software Corporation. *Delphi 6 Developer's Guide*, 2001.

[Sta00] Erlend Stav. Component based environments for non-programmers: Two case studies. In *Proceedings of the Workshop on Component Oriented Programming (WCOP'00)*, 2000.

[Sun97] Sun Microsystems, Inc. *JavaBeans API Specification (version 1.01)*, 1997. Available at http://java.sun.com/products/javabeans/docs/spec.html.

[Sun01a] Sun Microsystems, Inc. *Enterprise JavaBeans(TM) Specification (Version 2.0)*, August 2001. Available at http://java.sun.com/-products/ejb/docs.html.

[Sun01b] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition, v 1.3: API Specification*, 2001. Available at http://java.sun.com/j2se/-1.3/docs/api/index.html.

[Sun01c] Sun Microsystems, Inc. *Java Object Serialization Specification: Versioning of Serializable Objects*, 2001. Available at http://java.sun.com/j2se/1.4.2/guide/serialization/-spec/version.doc.html.

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

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

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

[T+96] Richard N. Taylor et al. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[Tic85] Walter F. Tichy. RCS – a system for version control. *Software – Practice and Experience*, 15(7):637–654, 1985.

[Tic94] Walter Tichy, editor. *Configuration Management*. Trends in Software series. Wiley, 1994. ISBN 0-471-94245-6.

[vdH01] André van der Hoek. Integrating configuration management and software deployment. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, Brisbane, Australia, December 2001.

[vdH+02] André van der Hoek et al. A testbed for configuration management policy programming. *IEEE Transactions on Software Engineering*, 28(1), January 2002.

[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.

[vO01] Rob van Ommering. Configuration management in component based product populations. In *Proceedings of 10th International Workshop on Software Configuration Management (part of ICSE 2001)*, Toronto, Canada, May 2001.

[Wan90] Henry Wansbrough, editor. *The New Jerusalem Bible*. Darton, Longman and Todd Ltd, 1990.

[Wor96] John Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

[WW98] James Whitehead and Meredith Wiggins. WEBDAV: IETF standard for collaborative authoring on the Web. *IEEE Internet Computing*, pages 34–40, September-October 1998.

[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.

[YAM99] Sherif Yacoub, Hany Ammar, and Ali Mili. Characterizing a software component. In *Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE'99*, May 1999.

[Zel98] Andreas Zeller. Versioning system models through description logic. In *Proceedings of System Configuration Management: ECOOP'98 SCM-8 Symposium*, volume 1439 of *LNCS*, Brussels, Belgium, July 1998. Springer-Verlag.

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

The manuscript of this thesis was written in the `vim` (Vi Improved) editor, the images were created in Zoner Callisto 4 and Corel Draw 8. Typeset into the Portable Document Format using the pdfLaTeX system with a slightly modified `thesis.cls` document class (originally by Wenzel Matiaske). The typefaces used are 11pt Baskerville for body text, Helvetica for headings, and Courier for source code samples.